FLOW: A MUTITIMBRAL, POLYPHONIC, MODULAR, ADDITIVE MUSIC SYNTHESIZER

Sean LUKE[1]

[1]*Department of Computer Science*, **George Mason University**, Washington, DC USA

ABSTRACT

Flow is an open source additive synthesizer that is very capable, but is specifically designed to be programmed by a musician without requiring the extraordinary number of parameters commonly demanded by additive synthesis. To achieve this, Flow works largely like a modular synthesizer, except that modules pass arrays of partials to one another rather than audio. Flow is polyphonic and multitimbral, supports MPE and microtonality, and easily extended and hacked. This paper describes Flow, its motivation, and its architecture, and discusses some of the lessons learned in its development and use as an instrument.

1. INTRODUCTION

Flow is a large additive software synthesizer with an unusual combination of features. It is polyphonic and multitimbral. It has full support for MIDI features including MPE and microtonality. It allows for up to 256 sine waves (partials) with both real-valued frequency and amplitude, but not phase. But most importantly, it is structured as a fully modular synthesizer, including cyclic patch connections, and patches can have any number of modules. Flow has over 70 modules available. Patches may contain special modules which encapsulate entire other patches (so-called *macros*). Flow is written in Java and is free open source, available at https://github.com/eclab/flow

I began Flow in 2018 with a specific experimental hypothesis in mind: is it possible to make a sophisticated additive synthesizer which does not overwhelm the musician with a massive number of required parameters? I think the answer has proven to be yes.

An additive synthesizer commonly changes a large number N of sine waves dynamically in real time, and each of these waves may have some M parameters defining how it changes over time, and so the total number of parameters can get large very quickly. Thus additive synthesis is notorious for its high number of parameters and tedious programming.

Rather than control each wave independently, Flow instead treats the entire set of waves as a group and passes them through a graph structure of modules. Each module sculpts the waves in different ways, ultimately outputting the result. You might recognize this architecture: it is the architecture commonly used by subtractive modular synthesizers. However, Flow's modules are rapidly passing arrays of partials to one another rather than sound waves, and are operating on the sound solely in the frequency domain, not the time domain. In essence, Flow is applying a classic modular subtractive interface to additive synthesis.

2. HISTORY AND RELATED WORK

Additive synthesis is very old. Indeed, organ ranks may be considered mechanical additive synthesizers, with each rank contributing new partials to the final sound.

Early additive synthesizers were electromechanical. Among the very most important early synthesizers was Thaddeus Cahill's extraordinary Telharmonium, a massive additive synthesizer circa 1896. The Telharmonium used rotating tonewheels and magnetic dynamos (pickups) to sum harmonics and broadcast the resulting sound to remote listeners. Figure 1 illustrates the tonewheels. Many of these concepts have since found their way into the Hammond Organ, which even now applies tonewheels and pickups to produce its sound.

In 1964 James Beauchamp developed the Harmonic Tone Generator [1] at the University of Illinois, Urbana-Champaign. This instrument used analog electronic circuits to synthesize a tone of six harmonics. Much electronic additive synthesis since then has been digital, and has often been associated with the analysis of sounds in addition to their synthesis. Starting in 1967, Jean-Claude Risset used Bell Labs's Music V software [2] to analyze and reproduce trumpet and later bell sounds [3]. The phase vocoder [4], also developed at Bell Labs, has likewise served as an analysis and additive resynthesis tool [5].

Notable early examples of digital additive synthesis for music production included the RMI Harmonic Synthesizer, the Fairlight Qasar M98 and CMI, and the New England Digital Synclavier II. In the 1980s and 1990s, Teisco/Kawai was perhaps the biggest producer of additive synthesizers, including the K3, K5, and K5000 series of synthesizers, with increasingly powerful additive synthesis capabilities. For further discussion of these and other examples, see [6].

Nowadays additive synthesis is less common and largely relegated to software. There are still many examples however, including Native Instruments's Razor, Image-Line's Harmor, Camel Audio / Apple's Alchemy, and AIR Music Technology's Loom. Additive synthesis is also popular in the open source community, perhaps because simple versions of the technique are easy to implement.

Many software synthesizers take a classic approach to additive synthesis, attaching modulators to each of the individual harmonics or partials. But a few, including some of the recent software synthesizers mentioned earlier, take an approximately modular approach which to some degree resembles Flow. Probably the most similar to Flow is AIR's Loom [7], in which a fixed number of processing modules are arranged in series. Each module receives an array of partials, modifies them in some way, and passes them down the line to the next module. Thus unlike Flow (as discussed later) there is only a total ordering among the modules. Each of these modules may then be modulated by a fixed set of separate modulation functions.

3. ADDITIVE SYNTHESIS IS HARD TO USE

Additive synthesizers usually produce sound by continuously changing the features of some N sine waves, or *partials*, and outputting their sum. These sine waves may differ in frequency, amplitude, and phase, and all three of these characteristics can change dynamically with time. In theory, if N is sufficiently large, and the change over time is sufficiently rapid, an additive synthesizer can produce any sound with adequate fidelity.

From an algorithmic standpoint, additive synthesis is trivial: we're just adding sine waves. But from the *musician's perspective,* additive synthesis can be very hard, as it can present a huge parameter space. This is because in its general form, additive synthesis requires at least $3N$ time-varying functions (frequency, amplitude, and phase for each of its N partials), each with sufficient parameters to describe the change over time of a partial when played.

Some simple additive synthesizers require only a few partials, but to make rich or complex sounds it is not uncommon for N to be large, often in the hundreds, particularly to adequately describe low tones. This can result in a very large number of parameters to program. For example, a single patch for the Kawai K5000 series — which were well regarded additive synthesizers — could have up to 6000 parameters.

There is also the computational cost to consider. These N sine wave functions must be calculated in real time per voice, which can be expensive, though in certain cases depending on the nature of the architecture, and the limits on the sound generated, this cost can be reduced by using a Short-Time Fourier Transform.

I think that these two challenges are the main reasons why additive synthesis has not been as popular as other methods. And these challenges are so dominant that many additive synthesizer designs are forced to reduce the parameter space in one way or another. Additive synthesizers generally use one or more of three strategies to do this.

First, it is very common for additive synthesizers to constrain the frequency of each sine wave to be an integer multiple of the fundamental. That is, they work not on partials but on arrays of *harmonics.* This simplification is justified under the assumption that sounds perceived as tonal are often largely composed of harmonics: noise or anharmonic sounds might be added in after the fact in a traditional subtractive manner.

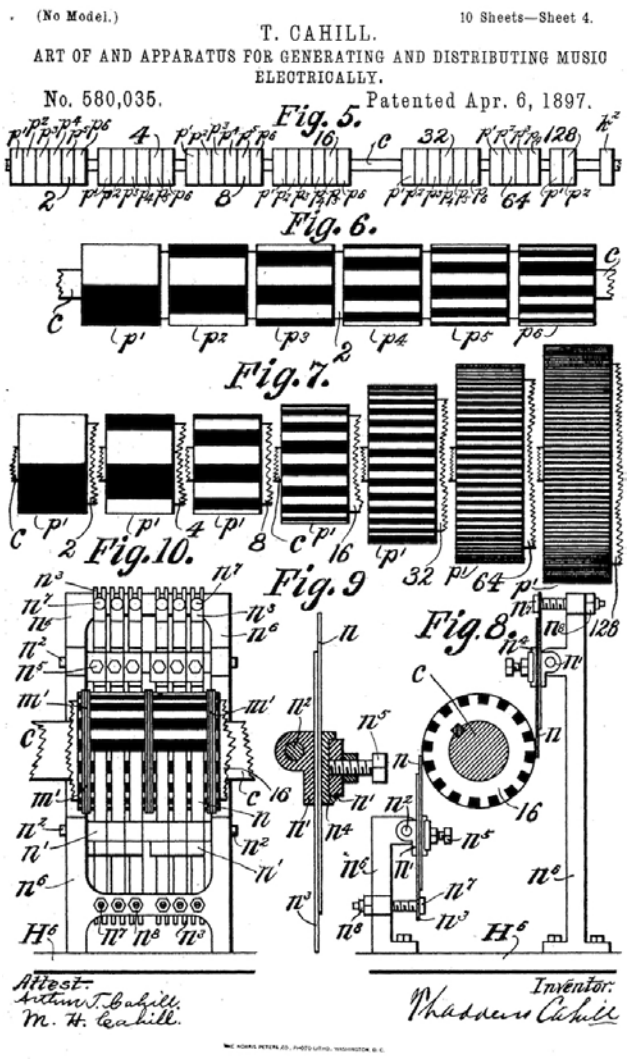

Figure 1. Illustration from 1897 Telharmonium patent filing, showing additive tonewheels.

Second, many additive synthesizers discard the phase of each sine wave in a voice, resetting it to 0 each time a new note is played. This simplification is justified under the presumption that, at least for the higher frequency sounds, it can be difficult to distinguish between two sounds which only differ in the (arbitrary) phases of their respective partials. Furthermore, if synthesizers are constrained to just produce harmonics, they cannot get out of sync with one another in phase, and so starting at 0 is reasonable.

Third, synthesizers might reduce the total number of partials, or skimp on the complexity of the modulation used to manipulate each of them. For example, let us restrict ourselves to just the Kawai synthesizers: while the Kawai K5 had 126 harmonics per voice, the K5000 series only had 64, though it could overlay multiple sets of harmonics in some situations. The K3 had just 32 harmonics, which is not atypical for many simple additive synthesizers. On the other hand, while the K5000's harmonics each had their own envelopes, the K5's harmonics had to share just four of them.

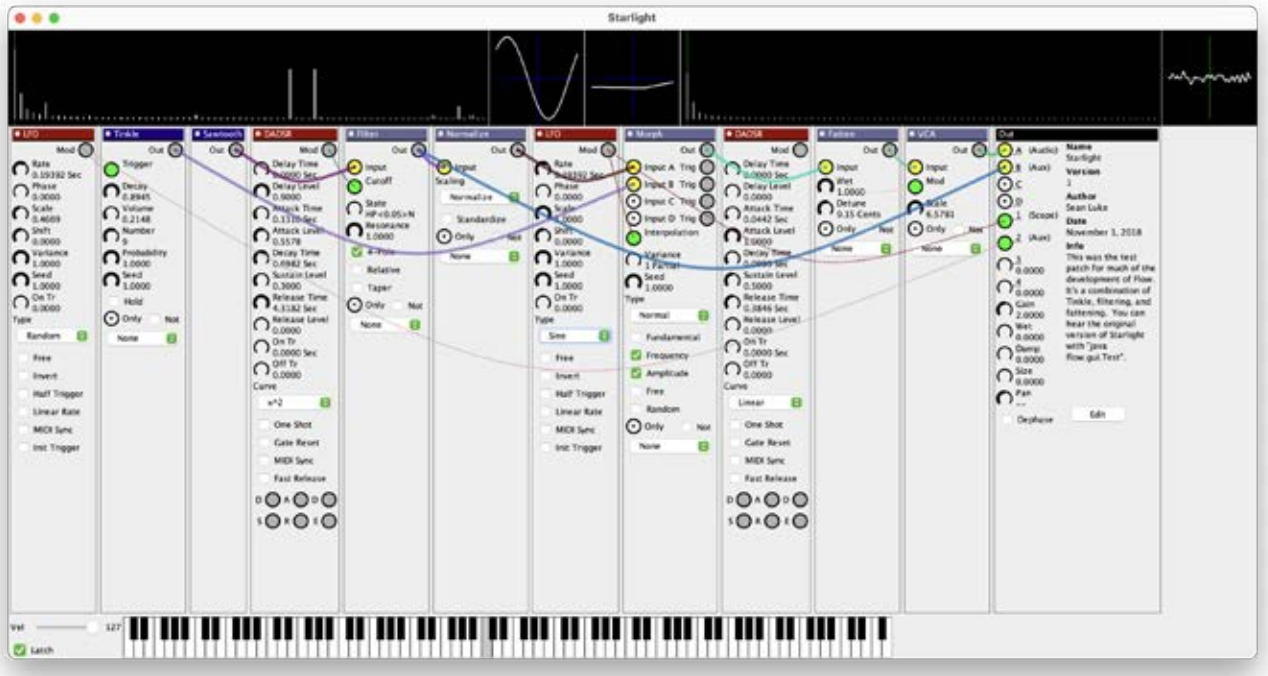

Figure 2. Flow interface, showing the partials displays, oscilloscopes, and sound output at top, the modules of a patch at center, and the optional keyboard at bottom.

4. FLOW

Flow takes a different approach. Rather than modulate the parameters of individual partials over time (though it can do that), Flow passes whole sets of partials through modular functions which manipulate them as a group. Flow is very much like a classic modular synthesizer, except that the modules work in the frequency rather than time domain: they do not pass sound to one another, but rather arrays of partials. Modules can be connected in any way the musician sees fit. Flow presently supports:

- Up to 256 partials, user-selectable.
- Partials with floating-point frequency and amplitude. Phase is discarded.
- Any number of modules per patch.
- 1–32 voice polyphony, user-selectable.
- Full multitimbrality. Only one timbre at a time may be manipulated in real-time in the UI.
- Sampling rates up to 48KHz, at 16 bit depth.
- MIDI Polyphonic Expression (MPE) [8], Clock Sync, Channel and Polyphonic Aftertouch, Pitch Bend, NRPN, RPN, and CC.
- Microtonal architectures via Scala files [9].

The idea is that, rather than manipulate each partial, the musician can select a small set of functions to manipulate the partials in a predictable way, which presents a much simpler and more intuitive interface. However, this combination is also computationally expensive. However it also presents a much easier environment for the musician to construct additive patches.

5. INTERFACE

Flow presents itself as a modular software synthesizer in a fashion similar to VCV Rack and similar [10]. An example of Flow's interface is shown in Figure 2. At the top of the window are two (long) displays for showing the frequency and amplitude of partials in a sound: the one at left displays the partials of the sound being generated. Also, (centered) are two oscilloscopes for modulation signals, and (far right) one oscilloscope for the final sound output. These displays are optional. Also optional is the small keyboard at bottom.

A Flow patch consists of some *N modules*, each connected via virtual patch cables. In the center of the window (Figure 2) are the patch's modules, arranged left-to-right first-to-last. Modules may be added, removed, and changed in order relative to one another.

5.1 Patch Cables and Patching

There are two types of patch cables, and they connect only to input and output sockets of their type. First there are *sound cables* (drawn with thick lines) along which sound data is transferred in the form of arrays of 256 partials. Each partial consists of a frequency, an amplitude, and an *ordering*, a unique number 0...255 which represents the sine wave generator in the Output's generator bank associated with that partial (see Section 8.4). Partials in the array

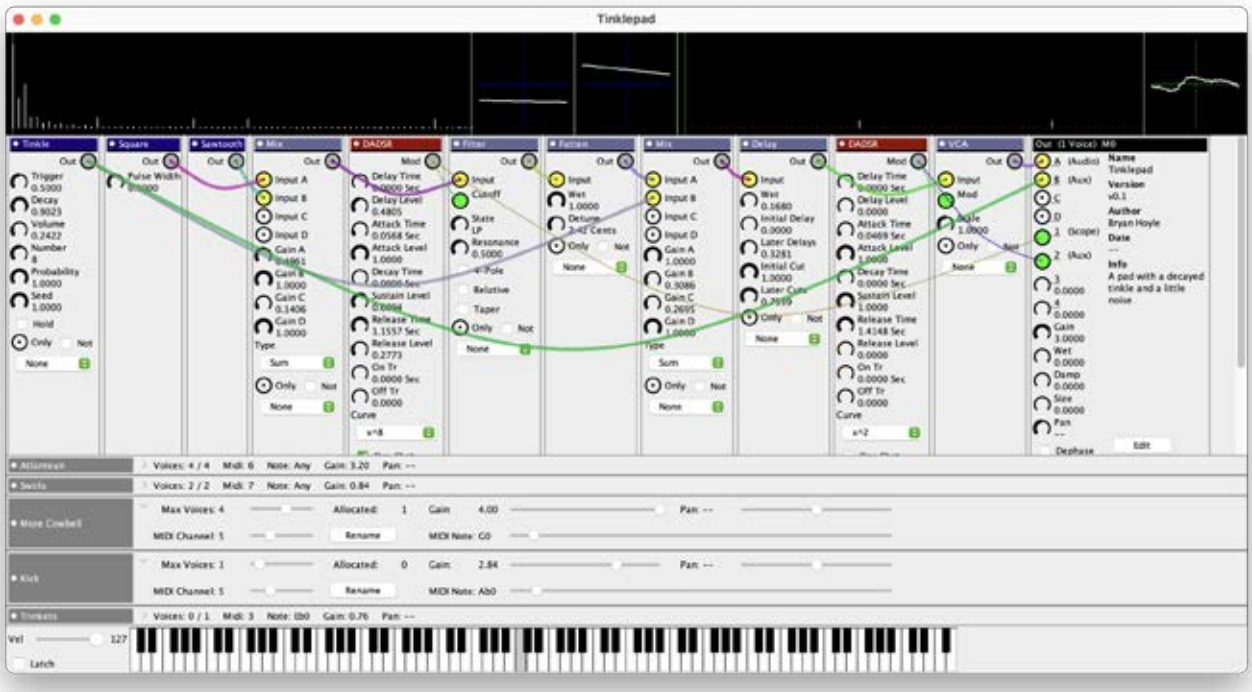

Figure 3. Flow interface, showing the modules of the primary patch and five multitimbral subpatches below.

are always sorted by frequency, as the large majority of modules' algorithms benefit from frequency ordering. Flow pulses each of its modules in order left to right, and when a module is pulsed, it retrieves the latest incoming partials from modules connected to it, then produces a new array of partials for other modules to request from it. Flow attempts to pulse all the modules as quickly as possible.

Second, there are *modulation cables* (drawn with thin dashed lines), along which modulation — essentially control voltage — data is passed from module to module. Flow offers two twists on the traditional modulation cable. First, these cables are not plugged into jacks per se, but directly into the modulation dials on each module. For example, a Filter module might have a cutoff frequency dial, either settable directly by the musician, or pluggable with a cable to control it instead. If it is useful for a parameter to be both modulated and user-controlled, the module simply provides two dials. Second, modulation cables carry not only modulation signals but also one or more typed *triggers* which signal events which have occurred. Triggers are used for many purposes. For example, an LFO module might output along its cable not only the LFO signal itself, but also a trigger fired each time the LFO has completed one period: this could then be attached to a sequencer to clock it. Modules request data along modulation cables in the same manner as is done for partials.

Flow rarely uses gates. This is because all modules directly receive cooked MIDI event information, including notes, control change and pitch bend information, etc. The most common use of gates in a modular synthesizer is to provide note on/off, which Flow's modules already know about.

Normally cables are forward-patched, meaning that there is a partial order of data flow among the modules from early (leftmost) modules to later (rightmost) ones. However nothing precludes a musician from back-patching: connecting a cable to flow data from a module to an earlier one to its left. A common use of back-patching is to provide delayed feedback loops. Thus Flow's patching structure can accommodate any cyclic directed graph. Back-patching is handled in the obvious way, namely that when the upstream earlier module requires partials or modulation data from the downstream module, it receives the previous data from the *last* iteration.

Flow has one special module called *Out*, of which exactly one copy always exists in a patch. Out is a sink which receives partials data and hands it off to the system to process into sound.

Though the musician only perceives one set of modules, in fact Flow maintains an independent set of modules for each voice to provide for polyphony.

6. MULTITIMBRALITY

Flow is not only polyphonic but also multitimbral. In a classic multitimbral synthesizer there exist both *single-mode* and *multimode* patches. A single-mode patch describes a single kind of sound. A multimode patch contains multiple different single-mode patches, or references to them, and states how voices and other performance elements are to be allocated to the various single-mode patches.

Flow's approach is a little different. Flow patch files consist of a *primary* (single-mode) patch, and zero or more (single-mode) *subsidiary patches*. Both the primary and

subsidiary patches have voices, MIDI data, and the like allocated to them, but only the primary patch can be edited and manipulated by the musician during performance. However the musician can, at any time, swap the primary patch with any subsidiary patch, so the subsidiary patch can now be manipulated instead. Subsidiary patches can be loaded or discarded from the main patch at any time.

7. MACROS

One of Flow's most unusual features is its *Macro* facility. After you have created a patch and saved it to disk, it is possible to load that patch and encapsulate it into a single module (a Macro), to be used among other modules in another patch.

The Out module contains not just a single jack to receive output partials, but three more auxiliary jacks of this kind, as well as four modulation jacks. All eight of these jacks can be named, and so when a patch is loaded as a Macro, the Out jacks connected in the patch are exposed as the Macro's outputs. Furthermore, the patch may have one or more copies of an optional module called *In* which has eight nameable input partials jacks and eight nameable input modulation jacks, plus default modulation values for them. If connected, these then appear as inputs to the Macro.

Macros are pulsed in a depth-first fashion: when the Macro module is pulsed, it loads the needed data for its *In* module, then recursively pulses all of its internal modules left-to-right once, including recursive nested Macros, then prepares resulting data from its *Out* module to provide to downstream outer modules.

Flow comes with a number of examples of Macros as useful utilities. These include formant voices, sound fatteners, brick wall filters, random drift utilities, bad attempts at supersaws, etc.

8. MODULES

Excepting Macros, Flow at present has 71 modules divided into five categories. The title bar color of each module reflects its category (see Figure 2). It is straightforward to add new modules to Flow, and the manual has step-by-step tutorials explaining how to do it.

I will not here subject the reader to the tedium of enumerating all these modules, but will highlight a few as examples.

8.1 Partials Generators

These are the rough equivalent of oscillators in a typical synthesizer. Flow has 20 partials generators, producing arrays of partials from common waves (like sawtooth), or from modulatable wavetables and vocoders, from musician-specified partials or harmonics, from time-varying functions that introduce and remove partials, and so on. Flow also sports modules producing partials derived from the waves or harmonics of other historic synthesizers, namely the Casio CZ series, Kawai K3 and K5, Ensoniq SQ-80, and Sequential Prophet VS, as well as those from the AdventureKid Waveform collection. [11]. Note that though the original data for some of these modules is derived from time-domain waves, these modules are all purely additive and frequency-domain.

8.2 Partials Modifiers

Flow has 28 modules which take partials as input, modify them, and output the result. These include a variety of time-varying filters, amplifiers, and mixers or combiners, but also many other modifiers special to Flow's additive nature.

Filters are particularly interesting. Among its other filters, Flow has modules which model N-pole filters, but does not have to restrict itself to Z-domain style digital filters. It can model a Laplace-domain (analog) filter clear to the Nyquist frequency, simply by applying the filter's amplitude response function to each partial. The phase effects of filters are, however, ignored.

Flow also has several modules dedicated to taking multiple additive partials streams and combining them in some modulatable fashion. This includes everything from cross-fading to morphing the frequencies of partials to partial-by-partial dissolves.

Combination is nontrivial because Flow has a fixed number of total partials, and so when two partials streams are combined, we must decide which partials don't make the cut. The combiners vary largely in the strategy used to select partials, or to gradually introduce partials and remove others under modulation. Additionally, when a partial is removed and another is introduced, the corresponding sine wave generator is lurched from the frequency and amplitude the first partial to the second, and in so doing combiners must employ schemes to prevent these changes from producing pops and other unwanted artifacts.

Flow also contains modules peculiar to the frequency domain, such as skeletonization and dilation, fattening (duplicating and detuning specific partials), adding frequency or amplitude jitter to them, shifting and rotating select partials, as well as delays and transition smoothers, and so on.

Nearly every partials modifier module can be *constrained*, that is, restricted as to *which* partials it is permitted to operate on. In many cases this is as simple as performing a function on every partial, then restoring certain partials to their previous values. In many other situations, notably combination modules, constraints are more complex.

8.3 Modulation Generators and Modifiers

Flow has 17 modules dedicated to generating, or modifying, modulation signals. These include the usual suspects such as LFOs, several envelope generators, sequencers, joysticks, sample and hold, and the like. They also include modules that expose a variety of MIDI features and event data not used by modules by default, including MPE modulation, NRPN, CCs, etc.

8.4 Utility Modules

The remaining modules are prosaic: *Out* and *In* as described before, plus modules for stretching patch points, adding textual notes, locking sounds to specific pitches, and so on.

9. THREAD ARCHITECTURE AND SOUND GENERATION

Flow consists largely of two parts: voice management and additive output. For each and every voice, the voice manager maintains a separate copy of modules in order to produce the latest partials of the sound. These partials are then handed to the additive output, which maintains its own independent bank of threads to push the partials though sine wave functions, add them up, and add effects.

Given that Flow is multitimbral and polyphonic, it should not then be surprising that Flow is very heavily multithreaded. As shown in Figure 4, Flow consists of a *MIDI Input Receiver* thread, which receives and processes all incoming MIDI data. This is passed to the *Voice Sync* thread, which manages per-voice processing of each of the modules. Each voice has its own set of modules and works with a (possibly shared) thread to usher partials from one module the next, and ultimately to the *Out* module. The Out module serves as the interface between the Voice Sync thread and the *Sound Output* thread. Once the latest partials from all voices have all arrived, the Sound Output thread takes the partials and updates the sine waves being used to produce the sound. This update is done with a leaky integrator so as to not produce artifacts stemming from a sudden change in frequency or amplitude. The Sound Output thread maintains a bank of output threads which divvy up the current sine waves and continually produce the latest sine output values. The sum of these values is then pushed through a Freeverb reverb implementation and output as sound.

The GUI also has its own separate event thread: this thread largely interacts with the Voice Sync thread to change module parameters and architecture; though at times it does work with the MIDI Input Receiver and Sound Output threads.

The Voice Sync threads are asynchronous from the Sound output threads: even if the patch's modules take a long time to run, and so the partials are slow to arrive, the output threads still work in the background to keep the sound buffer filled. Flow can be asked to try to maintain a rough ratio of output samples to voice sync partial updates. By default this is normally around 16 to 32 samples generated per partials update.

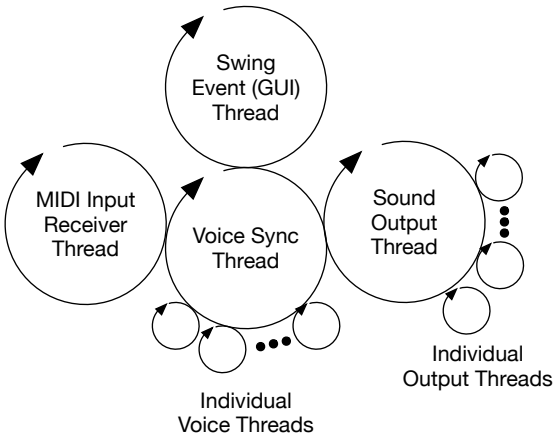

Figure 4. Flow's primary threads. Touching circles implies locking interaction.

10. DISCUSSION

I think Flow has been very successful as a demonstration of how to reduce the parameter complexity of additive synthesis. I and others have long used it as a performance and sound design tool, but it has also been particularly useful to me in an educational context, illustrating how sound synthesis works in the frequency domain.

Development and subsequent use of Flow has nonetheless brought up interesting issues worth discussing here.

10.1 Consequences of Disregarding Phase

Like many additive synthesizers, Flow discards phase as a computational and interface simplification measure. However while additive synthesizers constrained to harmonics do not need to concern themselves with changes in relative phase, Flow's partials can change in frequency in real-time. Consider two sine waves of the same frequency and phase, playing in unison. Flow modifies the frequency of one of them by some arbitrary amount, waits for a moment, then returns the frequency to its previous value. Now the two waves are out of phase.

For higher frequency notes, such as above middle C, the difference in sound can be negligible. But this is not the case for lower notes. The ability of humans to discern the difference between in-phase and randomized-phase partials grows dramatically with lower frequency [12]. Particularly in the case of very low frequencies, highly synchronized sounds such as square waves sound very different when their respective partials are out of phase with one another, not the least of which is because of the sudden and large jump in the sound wave (such as the vertical portion of a sawtooth) when all the phases are in sync.

Flow cannot compensate for patches which shift the phases of their underlying partials: the change in sound will be part of the nature of the patch's timbre. However it does reset their phases when a new note is played. Furthermore, Flow patches have an option ("Dephase") to add to each partial a specific deterministic random phase amount, and this masks most perceived phase changes when individual partials' frequencies are modified. In my experience, dephasing a low-frequency sawtooth or square wave also makes it sound "smoother", less harsh, and somehow more electric. This may or may not be desirable.

Though Flow gets along fine without phase, sometimes phase would be helpful, particularly in sampling sounds and reproducing them, such as with a phase vocoder. Flow's *Wavetable* module can record sounds (speech, say), chop them up into individual harmonics, and reproduce them, but as it lacks phase information it can only do so in a robotic fashion reminiscent of old-style vocoders.

10.2 Technical Issues

Flow's architecture brings up a number of technical issues. Here are three.

10.2.1 Number of Partials

Flow at present has only 256 partials, which is fewer than commercial additive synthesizers (for example, 384 in Harmor or 512 in Loom). This is not due to a technical limitation. The only reason for this is that Flow uses a single byte as the ordering ID for a partial, that is, the tag that identifies which sine wave generator is assigned to the partial. In the future Flow may move to a short or int, which would allow far more partials.

10.2.2 Partial Update Rate

Flow's per-voice sync threads are typically set to be slower than its output threads: as mentioned before, it's common for about 16 to 32 samples to be generated per partials update, though this can be changed if you have a fast enough computer. Furthermore, after new partials arrive the current partials are gradually adjusted to match them using a leaky integrator. This introduces latency.

This disconnect also has curious effects on extremely short sounds, particularly bursts of noise such as for a click or snap. Because the onset of the click is not synced to *when* the next set of partials is updated, clicks will not be consistent in amplitude. Additionally, because partials are not updated 1:1 with samples, modeling FM synthesis etc. would be difficult.

10.2.3 Use of Java

Flow is written in Java, and this allows Flow to be easily ported to a variety of platforms. While Java is a very fast language, one of its slowest areas is array access, and Flow relies heavily on array access for its partials. Nonetheless I have found Java more than sufficient to handle Flow's computational requirements.

Java is a garbage collected language and does not provide much control over how or when it happens. Java's traditional garbage collectors may occasionally produce a glitch in the sound output as Flow is unable to fill the sound buffer while being garbage collected. However Java's new low-latency collectors, such as its Z collector, entirely eliminate this issue.

The primary disadvantage of Java is that it is generally incompatible with the VST API; as a result Flow will likely remain a standalone application rather than embeddable in a DAW.

10.3 Specific Value of Additive Synthesis

Flow has a many useful modules which are peculiar to additive synthesis. For example, *Dissolve* will cross-fade from one sound to another but will do so by replacing random partials one-by-one in one sound with the other. *Jitter* will add random noise to the amplitude and/or frequency of select partials. *Shift* will, among other things, change the amplitude of each partial to that of the partial immediately below it in frequency. And so on.

However it is also true that Flow contains other modules which are essentially additive versions of subtractive classics, such as filters, amplifiers, mixers, and oscillators producing traditional waves. And I have found that, in my own patch development, I do often use these modules. The open question is why. Is it because I am used to these sounds historically and so sometimes gravitate to them? Is it because they are easy to understand and integrate? Or is it because many designed-for-additive modules can result in significant anharmonicity or manipulation of high frequencies?

Flow was developed partly in the hope that additive synthesis would offer sounds or programming approaches which were fundamentally different from subtractive synthesis designs. I strongly believe it does. But the question as to what degree this is true is worth examining further.

11. CONCLUSION AND FUTURE WORK

Flow was originally conceived as part of an effort to develop an additive synthesizer with sophisticated patching, modulation, and partial-manipulation capabilities, but which did not impose on the musician the tedium of programming a large number of parameters. To this end, Flow has adopted a modular synthesis approach, transferring entire sets of partials from module to module to be processed. While this pipeline approach has been adopted occasionally by other software, Flow goes rather further along this route, providing arbitrary connection graph structures including ones with cycles, any number of modules, and recursive module development in the form of macros.

Even though Flow does not require large numbers of parameters, it still has a very large array of options, because it has many modules and the musician can arrange and connect any number of them in any order. This makes attractive the notion of automating the patch-development process in order to search the space of patches more effectively.

I have previous work in automated patch exploration and co-creative tools. I have developed a large patch editor tool for many synthesizers, called Edisyn [13]. Part of Edisyn is a facility which uses evolutionary computation methods (such as the Genetic Algorithm) to work with the musician to explore the space of patches without having to program them. In this approach, known as *interactive evolution*, the program auditions some N patches to the musician, who selects those he prefers. Based on this information, the program mixes and matches, then slightly mutates, these patches to create N new patches in the vicinity of the musician's selections, then auditions them to the musician, and so on. Essentially, the program is co-creatively working with the musician to wander through the space of patches towards those he likes best.

This technique exploits the fact that the large majority of synthesizer patches may described as fixed-length arrays of numbers. But Flow's patches are arbitrary graph structures, which presents a large and non-metric space without a proper distance measure. Such spaces are difficult for evolutionary computation and other stochastic optimization methods to tackle efficiently, though there are certain approaches, such as genetic programming, which might be applied [14]. As future work I am interested in asking how these methods might be used, and to what degree, to simplify the task of programming additive synthesizers even more.

Acknowledgments

Microtonality and MPE were added to Flow with help from Vi Hoyle. Thanks to Giorgio Presti and Federico Avanzani at LIM (U. Milano) for their support and feedback in the development of this paper.

12. REFERENCES

[1] J. W. Beauchamp, "Additive synthesis of harmonic musical tones," *Journal of the Audio Engineering Society*, vol. 14, no. 4, pp. 332–342, October 1966.

[2] M. V. Matthews, *The Technology of Computer Music*. MIT Press, 1969.

[3] J.-C. Resset, "Computer music experiments, 1964–..." *Computer Music Journal*, vol. 9, no. 1, pp. 11–18, Spring 1985.

[4] J. L. Flanagan and R. M. Golden, "Phase vocoder," *Bell System Technical Journal*, vol. 45, no. 9, pp. 1493–1509, 1966.

[5] J. A. Moorer, "Signal processing aspects of computer music — a survey," *Computer Music Journal*, vol. 1, no. 1, pp. 4–37, February 1977.

[6] S. Luke, *Computational Music Synthesis*, 1st ed., 2021, available for free at http://cs.gmu.edu/~sean/book/synthesis/.

[7] Air Music Technology, "Loom II," Available as https://www.airmusictech.com/virtual-instruments/loom-ii.html, as of 1/1/2024.

[8] *MIDI Polyphonic Expression (Version 1.0)*, MIDI Manufacturers Association, 2018.

[9] M. O. de Coul, "Scala," Available at https://www.huygens-fokker.org/scala/, as of 1/1/2024.

[10] "VCV Rack Modular SoftSynth," Available at https://vcvrack.com/, as of 1/1/2024.

[11] K. Ekstrand, "Adventure Kid Waveforms (AKWF)," Available at https://www.adventurekid.se, as of 1/1/2024.

[12] M.-V. Laitinen, S. Disch, and V. Pulkki, "Sensitivity of human hearing to changes in phase spectrum," *Journal of the Audio Engineering Society*, November 2013.

[13] S. Luke, "Stochastic synthesizer patch exploration in Edisyn," in *International Conference on Computational Intelligence in Music, Sound, Art and Design (EvoMUSART)*, 2019.

[14] ——, *Essentials of Metaheuristics*, 2nd ed. Lulu, 2013, available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.