

Flow

An Open-source, Fully Modular, Multitimbral, Polyphonic, Additive Synthesizer

Version 11

<https://github.com/eclab/flow>

By Sean Luke

Department of Computer Science

George Mason University

sean@cs.gmu.edu

Contents

1	Introduction	3
1.1	A Bit on Additive Synthesis	3
1.2	Fully Modular Additive Synthesis	4
1.3	Specifications	5
2	Using Flow	6
2.1	Tuning Parameters	6
2.2	Modules	7
2.3	Partial and Modulation Signals	8
2.4	Setting Values in Dials	9
2.5	Triggers	9
2.6	Responding to Notes	9
2.7	Options and Constraints	10
2.8	Partials Displays and Oscilloscopes	10
2.9	Playing a Patch	11
2.9.1	The Play Menu	12
2.9.2	Microtuning	12
2.9.3	Using MPE	12
2.10	Erasing, Loading, and Saving Patches	13
2.10.1	Macros	13
2.10.2	Naming Patches	13
2.11	Connecting to a Digital Audio Workstation	13
2.11.1	Multitimbral Patches	15
2.11.2	Digital Audio Workstation Integration Strategies	18
3	The Modules	20
3.1	Modulation Sources	20
3.2	Modulation Shapers	24
3.3	Partials Sources	27
3.4	Partials Shapers	30
3.5	Special Modules	36
3.5.1	The <i>Out</i> Module in Basic Use	37
3.5.2	Creating a Macro with the <i>In</i> , <i>Out</i> , and <i>Macro</i> Modules	38
3.5.3	Using a Macro	39
3.6	Strategies for Merging Partial Sources	39

4	Developing Modules	41
4.1	Building a Modulation	46
4.2	Building a Unit	55

1 Introduction

Flow is a **fully modular, polyphonic and multitimbral, additive software synthesizer**. It is free open source and written in Java, and designed to be extended with new, easy to write modules. You can also create your own patches, then store them as modules to be used in larger patches.

There have been many additive synthesizers, even semi-modular additive synths, but a fully-modular additive synthesizer is not very common, much less one that is multitimbral. It's been fun to write this. Hopefully this software will inspire others to design similar systems.

Flow is an academic research experiment and a personal and amateur effort, not a professional product. Synthesis of the kind attempted here is very costly, and furthermore Flow tends to emphasize ease of development over efficiency. This means it'll keep your laptop quite warm and will have lots of odd ends and strings poking out here and there. Absolutely no guarantees are made with regard to its stability or correctness. If you find problems with the software, repeatable bugs, or general irritations, send me mail (sean@cs.gmu.edu).

Flow was written in Java because it is highly portable and still pretty fast: perhaps using this synthesizer will convince you of this. You can run a great many modules at once in a patch as necessary. But because Flow is in Java, it must be a standalone application, not a VST or AU. The software should run fine on Windows, OS X, and Linux, though it was developed for OS X and has not been tuned or tested much on the other platforms to date (especially not Windows).

1.1 A Bit on Additive Synthesis

An additive synthesizer is a synthesizer which produces sounds by *adding up* multiple sine waves of different frequencies, amplitudes, and phases. Each such sine wave is called a **partial**, because it is in some sense a *part* of the final wave.

Consider Figure 1, which shows three sine waves in green, orange, and blue. These sine waves are differing in their *frequency* (how fast they go up and down), and in their *amplitude* (how far they go up and down). If you add the three sine waves up, you get the wave shown in bold. In fact, it's easily shown that in theory you can synthesize *any wave* by adding up the right combination of sine waves.

Notice we didn't mention *phase*. Phase is the horizontal shift in the sine wave, as shown in Figure 2. As it turns out, humans often can't easily distinguish between sounds whose partials only differ in phase relative to one another: we just don't have the hardware built-in to do it.¹ As a result, most (but not all!) additive synthesizers, including Flow, don't consider phase in order to simplify matters.

If we only consider frequency and amplitude, we can chart our three sine waves as shown in Figure 3: plotting the amplitude of each partial by frequency. Notice that the Y axis is still Amplitude, but the X axis has changed from *time* to *frequency*. For this reason, thinking of a sound in this fashion is known as casting the sound into the **frequency domain**; whereas thinking of a sound as a wave in the traditional way (Figure 1) is known as considering it in the **time domain**. While we have been trained to think of sound waves in the time domain: in fact our brains receive sound information in the frequency domain: hardware in our ears first breaks out the sounds into high and low frequencies and their respective amplitudes before our brains even get them.

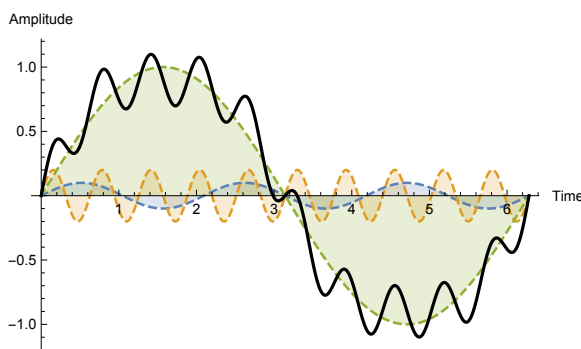


Figure 1: Three sine waves (colored) and the result of adding them up (in bold).

¹While this is certainly true for high- and mid-pitch sounds, it isn't really true for bass sounds: a low-frequency square wave (for example) sounds pretty different — buzzy — than one whose partials have randomly-assigned phases.

Figure 3 is more or less the same basic kind of data that sound modules in Flow send to one another: a list of partials, where each partial has a frequency and an amplitude. At the end of the day, the final list of partials is sent to the output facility to have their sine waves added up and emitted as a sound wave.

The lowest partial is commonly known as the **fundamental**,² and the other partials are often known as **overtones**. It so happens that a large number of physical things which we perceive as “musical sounding” (such as strings or flutes) produce partials which, by and large, have a certain interesting feature: they are all integer multiples of the fundamental. That is, if the fundamental is at frequency f the next partial will be at frequency $2f$, the next partial will have frequency $3f$, and so on (and conveniently in Figure 3 this was the case). Partial which fall along these integers are known as **harmonics**.

Many early (and current!) additive synthesizers, such as the Kawai K5 and K5000, simply assumed that all sounds of interest would be generated using harmonics, not arbitrary partials. Flow does not make this assumption: you can have partials at any frequency you like. But you will see sounds consisting of only harmonics in many places, as they are musical and useful.

In Flow, if you adjust partials so that they are all harmonics, with no gaps among the integers, this procedure is called **standardizing** them. Another term you’ll see: if you adjust the partials so that their amplitudes add to 1.0, this is called **normalizing** them. Some modules can do standardization or normalization as an option.

(Useful) partials can have any frequency from 0 up to the **Nyquist limit** of a digital sound wave. The Nyquist limit is the highest frequency a digital representation is capable of producing: it is half the sampling rate. So if the audio rate is 44100 Hz (as is the case for Flow, and of course, for CDs), then the highest possible frequency is 22,050.

1.2 Fully Modular Additive Synthesis

Additive synthesis is viewed as a sort of holy grail of synthesis: because you can generate all sounds in an additive fashion in theory, additive synthesis is in some sense an “optimal” way to doing synthesis: but it has a problem which prevents it from being easily used in reality. And that problem is: the classical approach to additive synthesis involves a very, very large number of tedious parameters. Sound waves have many hundreds or thousands of partials, each with a frequency and amplitude and maybe phase: and as sounds change and evolve over time, so do all those values. And so a synthesizer in which you set all the partials independently will have a great many parameters to set. Consider: the Kawai K5000 had 126 harmonics,

²This is kind of a lie. Technically the fundamental is the lowest or “first” partial: but the term *fundamental* is also often used to describe the *loudest* partial, or the one whose frequency we think of as the pitch of the sound. These three things usually coincide, but for some musical instruments they’re not the same. For example, in organs the fundamental is the tone we associate with the pitch: but there may exist one or two partials *lower* than it. This is also the case for bells, where the fundamental (also called the *strike tone*) is usually higher than at least one partial known as the *hum tone*. In a bell, there are often partials louder than the fundamental as well, and in fact the fundamental might not even exist — we think we’re hearing a note, but in fact there’s no partial there! Bells are weird. For simplicity, when Flow refers to the fundamental, it means the lowest partial.

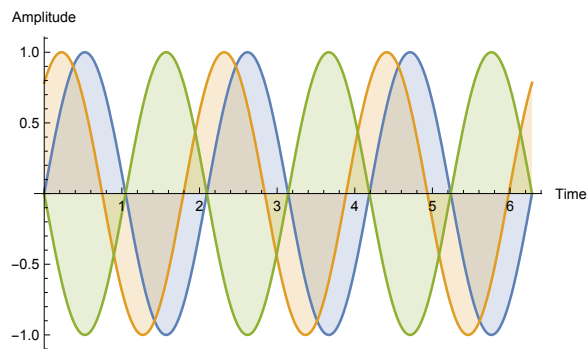


Figure 2: Three sine waves which differ only in phase. Note that the green and blue waves differ by so much that they have become inverses.

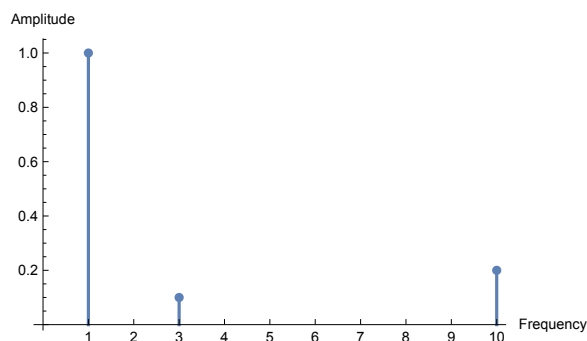


Figure 3: Three sine waves (colored) and the result of adding them up (in bold). Compare to Figure 1.

each with a base amplitude and *its own envelope* which specified how it changed over time. This meant that the Kawai K5000's patches had on the order of a thousand parameters.³

An alternative approach, taken by software tools such as Harmor/Harmless, Razor, and Loom,⁴ is to create a series of modules which either generate sets of partials or modify existing partials, and then to arrange them in an assembly line, routing partials from one module to the next, until a sound is emitted at the end. The idea is that instead of directly manipulating partials and envelopes, you're using operations (the modules) designed to sculpt away at the partials. But because you can't wire up the modules in any fashion, these tools generally use a kind of **semi-modular synthesis** approach.

Flow is in contrast closer to a **fully modular synthesis** approach: you can organize as many modules as you like, and wire them up however you like, just like a hardware modular synthesizer. Since they're not in a line, modules can be fed from, and feed into, *multiple* other modules. There's no reason you can't have modules wired up in cycles or other patterns. And just like in a hardware modular synthesizer, if you don't wire the modules up at all, you don't get any sound.

Note however that unlike a classical fully-modular synthesizer, a software synthesizer like Flow can easily be (and is) **polyphonic** and **multitimbral**, and you can save and load patches. And of course, unlike a traditional modular synthesizer, Flow is **additive**: the stuff moving through wires isn't sound waves but arrays of partials.

Flow has another powerful trick up its sleeve: after you have saved your collection of wired modules out as a **patch**, you then have the option of loading that patch to operate as a **self-contained module in another patch!** When a patch is loaded in this way, it is called a **macro**. And yes, you can have macros which contain macros which contain macros.

1.3 Specifications

At present Flow has the following specifications. They may change:

Number of Partial	64, 128, or 256, user-selectable. Phase is disregarded.
Polyphony	1–32, user-selectable.
Multitimbrality	Total. Up to 32 unique sounds.
Sampling Rate	44,100 Hz
Audio Bit Depth	16 bit
Channels	1 (Mono) or 2 (Stereo)
MIDI Features	MPE, Clock, Channel and Polyphonic Aftertouch, Pitch Bend, NRPN, RPN, CC
Microtuning	Scala files

³I'm not denigrating the K5000! The K5000 was one of the most incredible and awe-inspiring synthesizers in history.

⁴Harmor and Harmless are trademarks of Image-Line Software. See <https://www.image-line.com/plugins/Synths/Harmor/> and <https://www.image-line.com/plugins/Synths/Harmless/> Razor is a trademark of Native Instruments, Inc. You can find Razor at <https://www.native-instruments.com/en/products/komplete/synths/razor/> And Loom is a trademark of inMusic Brands. You can find Loom at <http://www.airmusictech.com/product/loom-ii>

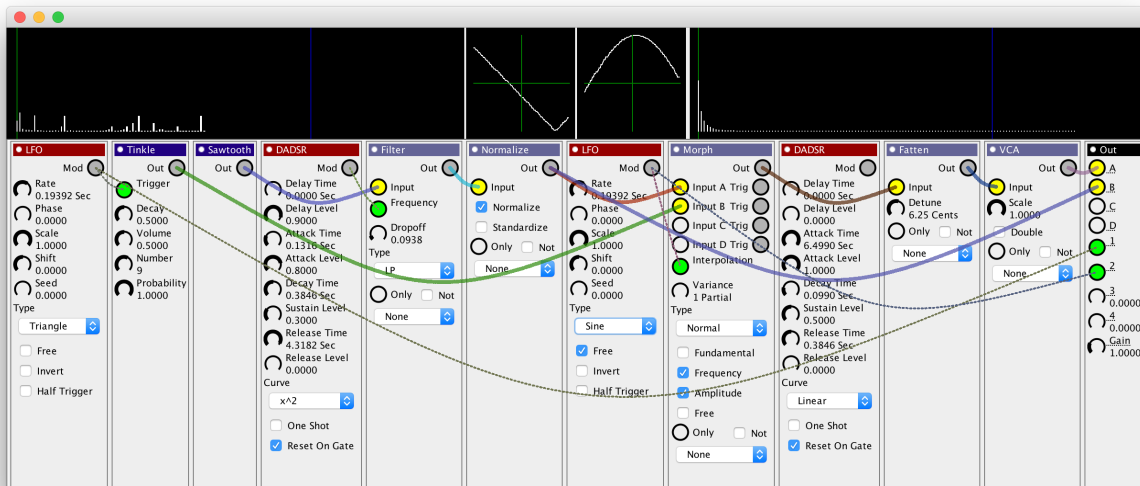


Figure 4: An Example Patch

2 Using Flow

After you fire up Flow, you're presented with a near-empty **rack** (except for the **Out** module) and can start adding **modules** to it. Then you tweak the parameters of modules and wire them up by dragging from output jacks to input jacks or to dials. The result might look something like the example patch in Figure 4.

If you look carefully at Figure 4, you'll see that there are not only twelve modules, but also two wide black rectangular regions called **displays**, and two small squarish black regions called **oscilloscopes**. Displays show partials in the same manner as Figure 3, and oscilloscopes display modulation signals.

2.1 Tuning Parameters

First things first. Flow is very, very computationally expensive and benefits from tuning to your computer speed. If your computer can't keep up with Flow's demands, you'll hear a **glitch**. This is a pop or a buzz in the audio stream, and will be accompanied by the **Out** module (the top-right module shown in Figure 4) coloring its black title bar red and saying **Glitch**. A glitch occurs when Flow can't keep the audio buffer filled fast enough, so your computer is forced to start filling it with zeros. You don't want that. Some of these parameters may also have an effect on **latency**, especially MIDI input.

Glitches often settle down after running Flow a minute or so. But if not, how should we tamp them down? By playing with a few parameters. If you select **Tuning Parameters** from the **Options** menu, you can at present change six parameters. Pressing *Okay* or *Reset* (which resets to defaults) will change the parameters for the *next* time you run Flow, not the current session.

You may not yet understand some of the terminology below: in this case, read the rest of the manual, try out the synth, and return to the Tuning Parameters later.

- **Polyphony** (1–32, default: 8) This is the number of voices the synthesizer can play at once. More voices incurs a higher load.
- **Buffer Size Per Channel** (default: 1152, which is crummy, but the best Java on the Mac can do) This is the size of the audio buffer (for each channel) that Flow keeps filled for your operating system to tap into to generate sounds. You want a small buffer if you can (much less latency); but a bigger buffer makes it easier for Flow to keep up. Some operating systems require a large buffer size: for example,

OS X typically needs at least 1152 . On Linux you can get away with much smaller buffers: see the Linux portion of Section 2.11 for more information. I can't help you much regarding Windows.

- **Partials** (64–256, default: 256) This is the number of partials passed around from module to module. More partials will produce a richer sound, particularly in bass notes, but incurs a much higher load.
- **Voices Per Thread** (1, 2, 4, 8, 16, default: 8) This is the number of voices which will be assigned to a single CPU thread in the modular partials-updating stage of synthesis. More voices per thread means fewer threads. You'd like as many voices assigned to a given thread as possible, because switching threads incurs significant computational overhead. However if you have too many voices on a thread, then they cannot take advantage of the parallelism afforded by your laptop's nifty multi-core CPU and you won't have fast partials updates. Also if you have very few voices per thread, then you'll have many more threads in the partials-updating stage and this may nudge out the critical output threads, resulting in more glitches. You have to find a balance. I have found 8 to be a good default.
- **Outputs Per Thread** (1, 2, 4, 8, 16, default: 2) This is the number of voices which will be assigned to a single CPU thread in the sound-output stage of synthesis. More voices per thread means fewer threads. You'd like as many voices assigned to a given thread as possible, because switching threads incurs significant computational overhead. However if you have too many voices on a thread, then they cannot take advantage of the parallelism afforded by your laptop's multi-core CPU. You have to find a balance. I have found 2 to be a good default.
- **Samples Per Partial Update** (1, 2, 4, 8, 16, 32, 64, default: 16) The output facility repeatedly grabs the latest partials information from the voices, then produces some N samples. *Samples Per Partial Update* is the value N .

To make sound, Flow runs two procedures in parallel. The *voice facility* repeatedly gets the latest MIDI data, pumps each of the modules in each of the voices, and produces and posts the latest set of partials. Simultaneously the *output facility* repeatedly grabs the most recently posted partials and generates some N mono or stereo samples from them.

If the output facility hasn't grabbed the most recently posted partials, and the voice facility has even newer partials to give it, it will wait until the output facility grabs the older ones before posting the newer ones. This means that if N is *high*, then the voice facility will wait for longer times. This causes two bad things. First, it means that the voice facility will take longer to respond to new MIDI messages, resulting in a noticeable latency. Second, it means that sounds will change partials more slowly: you may or may not notice this second effect (though if $N = 1$, the partials update is so fast that you can effectively do FM with the LFOs). So why not keep this value low? Because low values of N mean that the output facility will constantly be grabbing the latest partials, which incurs a significant computational load.

The default value is $N = 16$, but if the lag is too high, try a bit lower. Be warned that if you have a large and complex patch, then the amount of time can easily be high enough that the MIDI latency and partials update speed will be poor anyway.

- **Stereo** If this is unchecked, then Flow will output mono sound regardless of the setting of the pan knob or slider. This might incur a somewhat lower overhead, but it's likely to be minor.

Okay, we're done with tuning, so let's make a patch. Quit Flow, fire it up again (so its tuning parameters take effect), and start adding modules...

2.2 Modules

A Module in Flow is very similar to one in a modular synthesizer: it's a device which takes modulation signals or incoming audio (in the form of partials), and outputs modulation signals or audio (again as partials). Modules which mostly *produce* partials or modulation signals are known as Modulation or Partial

Sources. Modules which largely *modify* incoming partials or modulation signals, and output the results, are known as **Shapers**.

Select a module! The **Sources** menu contains partials sources, and the **Shapers** menu contains partials shapers. The **Modulation** menu contains modulation sources at top and partials sources at bottom. Finally, the **Other** module contains some special modules which don't fit in with the others.

You add a module to the rack by choosing it from the menu. Once it is in the rack, you can move a module to another location by dragging its title bar. You can also copy a module to another location by dragging its title bar while holding the Control key (on the Mac, you can also hold the Option key). You can delete a module by clicking on its close box: the little white dot to the left of its title. Finally, if you double-click on a module, it will expand to show some **help documentation**. Double-click again to shrink it back.

By default, modules are added to the beginning of the rack. You might find this annoying. If you would prefer modules to be added to the end of the rack (just before the Out module), choose **Add Modules At End** from the **Options** menu.

2.3 Partial and Modulation Signals

Modules can send and receive two kinds of signals. First, a module can send or receive **sets of partials** to/from other modules. Since this is an additive synthesizer, a set of partials is essentially a sound being emitted from a module. Additionally, a module can send or receive a **modulation signal**. This is a single number, ranging from 0 to 1, which can change over time. Modulation signals are used to change parameters in modules in real time.

Modules can send any number of partials and modulation signals, and they can receive any number of the same. But some modules are intended *primarily* to **create modulation signals** (their title bars are red), to **shape incoming modulation signals and emit new versions** (light red title bars), to **create partials** (blue), or to **shape incoming partials and emit new versions** (light blue). A few system-level or special modules, such as the **Out** module, have black title bars. Macros (see Section 2.10.1) have green title bars.

Figure 5 shows a very simple patch: a Sawtooth wave run through a low pass filter whose cutoff frequency is modulated by a sine-wave LFO. Notice the special module **Out**. If you attach a module to the jack labelled *A (Audio)* on this module, then the module's partials will be used to produce the final output sound. We've attached the jack *Out* on the module **Filter** to the jack *A (Audio)* in the module **Out**, so the Filter's output is being routed directly to audio. For more information on the capabilities of the **Out** module, see Section 3.5.1.

The Filter is also receiving partials (which it's filtering). These partials are coming from the **Sawtooth** module. Its *Out* jack is attached to the **Filter**'s *Input* jack. Notice that the wires connecting Partial input and output jacks are thick, solid (random) colors. You make these wires by just dragging from an output jack to your chosen input jack.

In contrast, modulation wires are thin and dashed. Note the **LFO** has two such wires coming out of its *Mod* jack and leading to the **Filter**'s *Frequency* jack. We have set up the LFO to **modulate** the cutoff frequency of the filter. The jack is also connected to a jack in **Out** called *1 (Scope)*: this will cause the modulation to appear on the primary oscilloscope.

While the input jack for partials is a circle, what is the input jack for modulations? The answer is: it's a dial. If you haven't connected it, it appears as a dial and you can simply set its value. If you connect it, it turns into a connected circle and receives its values as modulation.

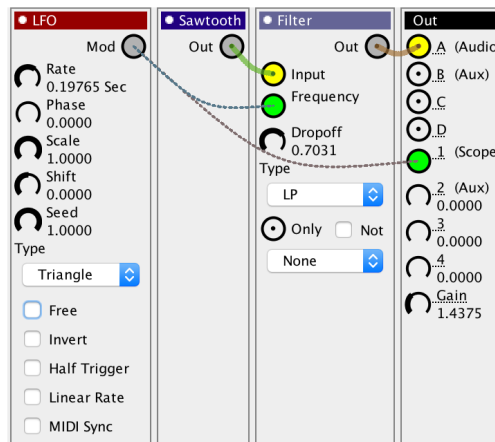


Figure 5: A Simple, and Terrible, Patch

You disconnect a wire by clicking on the input jack it's connected to. And you change the color of a wire by clicking on its output jack.

Patching Backwards Typically you'd wire a module to send information on to another module further to its right in the Rack. But there's no reason you can't wire a module to send to another module to its *left*, and there are often good reasons to do so. However you should be aware of one minor consequence of this. Every time the Rack wants to get the latest partials to send the audio output, it does by pulsing modules left-to-right. A module can extract information from other modules only when it's being pulsed. Normally this is fine: if module X gets information from module Y to its left, by the time X is extracting information from Y, Y has already been pulsed and so is ready to provide the latest information. But if module X is getting information from module Z to its *right*, Z has not been pulsed yet and so its information will be old. How old? By default, it'll be typically about 32 samples, or about 0.725 ms old. If you chain a bunch of modules backwards like this in a line, this delay starts multiplying. With about 5 modules wired backwards in a chain like this, you'd be over 3 ms, which is getting into the realm of human detection.⁵

2.4 Setting Values in Dials

If an input jack for a modulation isn't connected, it appears as a dial: you can fix its value to anything between 0.0 and 1.0 inclusive by dragging the dial up or down. Some dials won't show values between 0.0 and 1.0, but rather give a more informative value display like "3.23 sec" or "2023 Hz". But underneath they're always just values from 0 to 1.

There are some other options available on setting these dials.

- If you double-click on the dial, a menu will pop up with common settings for you to choose from. Importantly, if you double-click on rate or time dials in LFOs or various Envelopes, these settings will correspond to note speeds if you were playing at 120 BPM (or whatever rate the MIDI Clock is running at).
- If you hold the **Alt** or **Option** key down and then drag, your resolution will be 4x.
- If you hold the **Control** key down and then drag, your resolution will be 16x.
- If you hold the **Control** and the **Alt** or **Option** keys down and then drag, your resolution will be 64x.
- If you double-click on the dial while holding down the **Shift** key, or if you double-click on the dial using the right mouse button, a dialog box will pop up which allows you to enter a precise value.

2.5 Triggers

Modulation signals not only carry a time-varying number, but also can send a **trigger event**. Various modules use incoming modulation triggers to pulse them. When a trigger occurs depends on the modulating module: for example an LFO sends a trigger every time it completes a single wave.

In a classic modular synthesizer, control signals take the form of either gates or control voltage (CV), hence the term "gate/CV". In a very real sense, modulation and triggers together perform the same function as gate/CV, albeit combined into a single wire.

2.6 Responding to Notes

Flow differs from a classic modular synthesizer design in that you don't feed modules the current pitch or volume information in the form of Gate/CV signals. Rather, every module is automatically informed when a

⁵Perhaps it's the effect you were aiming for!

note has been pressed and it has been released. There's no reason to send a gate or CV signal to an envelope, for example, as it already knows the note has been pressed.

It's best to think of the Rack not as a single Rack, but as N parallel copies of that Rack, where N is the polyphony of the synthesizer. Each time a note is pressed, one Rack copy is assigned to that note, and the modules are all told that a note has been pressed (and later that it has been released). Modules can query other information as appropriate, such as aftertouch or pitch bend. At the end of pipeline, the *Out* module produces a set of partials with frequencies (normally starting at 1.0) and amplitudes. These frequencies are each multiplied against the note pitch, and the amplitudes are multiplied against the note volume: sine waves are produced and added up, and this becomes the final sound.

Though modules such as envelopes (or LFOs) already know that notes have been played, it's occasionally useful to trigger them manually for other purposes rather than having them trigger themselves automatically; there are certain additional options available to do this.

2.7 Options and Constraints

In addition to input and output jacks and dials, modules also sport options in the form of pop-up combobox menus and check boxes. For example, on the LFO you can choose a wave *Type* (currently set to "Sine"). You can also click a checkbox to *Invert* the wave among other features. Another checkbox option on the LFO, *Half Trigger*, tells it to send triggers every half of a wave rather than every full wave.

Some modules are outfitted with **Constraints**: for example the **Filter** module has a jack called *Only*, a checkbox called *Not*, and an unlabelled combobox (presently set to "None"). The constraints facility allows you to constrain which partials are being modified by the module and which are just passed through unchanged.

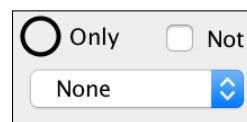


Figure 6: Constraints

2.8 Partials Displays and Oscilloscopes

There are two partials displays and three oscilloscopes above the modules in the Rack. The left partials display will show the partials of anything connected to the *A* (*Audio*) jack in **Out**, that is, the partials being played as a sound. The right display will show the partials of anything connected to the *B* (*Aux*) jack. You can use this display as an auxiliary display.

Figure 7 shows a typical partials display. Note that the display has a green vertical line: this indicates the position where the fundamental would be if it was exactly $1 \times f$ frequency: typically this marks the fundamental. There may also be a blue line: this indicates where the Nyquist frequency cutoff is. Partials to the right of this line are not included in the final sound, and appear light gray on the display. If you're confused, review the discussion of harmonics in Section 1.1. When a partial is 0.0 amplitude or higher than 1.0 amplitude, it turns red.

If you hover over a display with your mouse pointer, the display will highlight the nearest partial in orange, and it'll display some text, in the form:

Partial *partial* Amp: *amp* Freq: *absolute-freq* (*relative-freq*)
Mouse Amp: *amp* Freq: *absolute-freq* (*relative-freq*)

This says that the nearest partial to the mouse pointer (the one in orange) is partial number *partial*, and it has a given amplitude and relative frequency (the fundamental would have frequency 1.0). Multiplied against the current note pitch gives the provided absolute frequency. Moreover, the mouse pointer is currently pointing to a spot with a certain amplitude, relative frequency, and absolute frequency as well.

Optional Display Features By default, partial frequencies are displayed linearly. You can change this to a logarithmic plot by choosing **Log-Axis Display** in the **Options** menu. There are two scaling options as well.



Figure 7: A Partials Display. Note the partials to the right of the blue line. These are beyond the Nyquist limit and are not audible.

You can instruct Flow to provide enough room to display up to some N number of harmonics to the right of the fundamental by choosing the appropriate value under **Max Displayed Harmonic** in the **Options** menu. The default is 150. If you're doing a logarithmic plot (this won't affect a linear display), you can also select how much room should be provided to the left of the fundamental by selecting **Min Displayed Harmonic** from the **Options** menu. The default is 1/16.

Displays can also be set to *Waterfall Mode*, where the partials become dots (brighter dots are higher amplitude) and scroll by in time. To try this out, choose **Waterfall Display** in the **Options** menu. Finally, you can hide (or show) the displays with the menu option *Show Displays*: hiding uses less computer power.

Oscilloscopes The far right oscilloscope shows the output audio. When clipping, the clipped portions are shown in red. If you press on it, you can freeze the image.

The middle two oscilloscopes instead display modulation signals of your choice. The left oscilloscope shows the modulation signal of anything connected to the 1 (*Scope*) jack in **Out**. Notice for example, that in Figure 5, the LFO is wired up to modulate the 1 (*Scope*) jack: this will cause it to be displayed in the left oscilloscope display. Similarly, the right oscilloscope shows the modulation signal of anything connected to the 2 (*Aux*) jack. Oscilloscopes have axes (the horizontal axis passes through the 0.5 position). These axes change colors from blue to green (or from green to blue) any time the modulation triggers. Oscilloscopes don't currently display any text when you hover over them. Maybe they should!

A warning about the middle oscilloscopes: like displays, they show the data of the most recently pressed note. This can be confusing if (for example) you're using an LFO. You've just reset the LFO with a note-on, but instead of resetting from its current position, it magically jumps to something much higher and resets from there. This is because the voice being allocated for your note has its LFO at that position at the moment, *not the voice you've just been watching*, and the oscilloscope suddenly jumped to it.

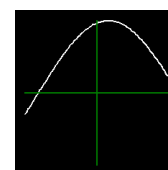


Figure 8: An Oscilloscope

2.9 Playing a Patch

In order to hear a sound you need to first specify your **Audio Device** from the window which pops up when you select **MIDI and Audio Options** from the **Options** menu. I suggest setting it to your computer's speaker.

Now, if you directly connect a partials source (**Sawtooth**, say) to the **A (Audio)** input of your **Out** module, it'll start playing. See Figure 9 at right. In fact, **all voices** might start playing: there's nothing to stop them! It won't sound good. At this point you're just testing your patch and may want to hear just a single voice. To do this, select the menu option **Monophonic** in the **Play** menu. The word "(Mono)" will appear in the **Out** module.⁶

You might also choose **Monophonic** if you'd like a monophonic synth. Flow does *last note priority*, meaning that if, while playing monophonically, you hold down note A, then hold down note B, then hold down note C, then let go of C, then B will begin to sound, and if you let go of B, then A will begin to sound.

Ultimately you'll probably set things up with a **VCA** (see the last entry in Section 3.4) modulated by an **Envelope** (typically a **DADSR Envelope**: see Section 3.1) which won't play the sound unless a key is pressed. See Figure 10 at right. That way different voices are played when you play different keys. At that point you'll want to uncheck **Monophonic** so you can play more than one voice.

So how do you play a using keyboard? You attach it to Flow by selecting **MIDI and Audio Preferences** from the **Options** menu, then choosing the appropriate **MIDI Device** and **MIDI Channel**. You'll notice an **Auxiliary MIDI Device** as an option: this lets you connect two MIDI devices to Flow. It's not useful to use this unless you're in multitimbral mode (discussed in Section 2.11.1).

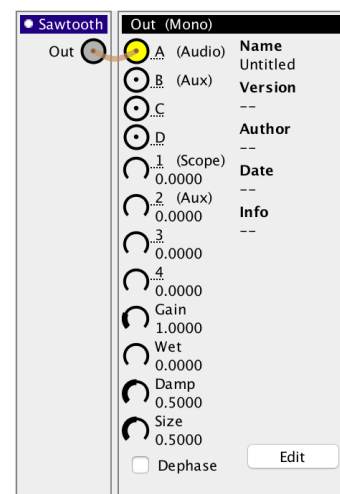


Figure 9: Playing a Saw wave

⁶Note that *Mono* here means monophonic in the sense of a monophonic (single-voice) synthesizer: the output could still be in stereo.

If Flow is playing too loudly its sound will **clip**. The title bar of the **Out** module will briefly turn yellow and say **Clip**. You may also hear an audible buzz or click, and you'll see red portions of the wave on the audio output oscilloscope. To prevent this in the future, you can turn down the sound's **Gain** (in the **Out** module). You can also change the **Master Gain** in the **MIDI and Audio Preferences**, but this is better reserved for multi-timbral mode instead.

2.9.1 The Play Menu

Reset This option completely resets the synthesizer's oscillators, modulators, and timing. Is a MIDI key stuck? Have the oscillators drifted out of sync? Press reset.

Monophonic As discussed earlier in section 2.9, this toggles between monophonic and polyphonic mode. You'd want monophonic if you want to test out a patch, play in mono legato, or reduce CPU utilization.

Responds to Bend and Bend Octaves This toggles whether Flow responds to pitch bend information, and how sensitive it is to pitch bend. You might wish to turn off pitch bend if your Roli Seaboard's incessant pitch bend is driving you nuts, for example.

Velocity Sensitive This toggles whether Flow responds to velocity information at Note On, or if it always plays at full volume.

Synced to MIDI Clock This toggles whether Flow allows modules to define their timing according to an incoming MIDI clock rather than an internal clock. Just because you have selected this doesn't mean a module uses the incoming MIDI clock: you often also have to set that in the module itself.

2.9.2 Microtuning

If you load a microtuning file (a *Scala* or .scl file), via **Load Microtuning File...** then Flow will use it to map standard MIDI notes to the tunings defined in the file. You'll also be asked to stipulate a root note corresponding to some MIDI note (such as middle C). Once you have things set up, you can then toggle microtuning on and off by selecting **Microtuning**.

2.9.3 Using MPE

To set up a MIDI Polyphonic Expression (MPE) device, go to **MIDI and Audio Preferences** in the **Play** menu, and change the MIDI Channel to either the MPE Lower Zone or the MPE Upper Zone, as determined by your MPE Controller. You can then specify the number of MPE channels in the Zone. Flow also responds to an MPE Configuration Message (or MCM) sent over MIDI from your controller.

MPE devices are often designed to link certain behaviors to attack velocity, pitch bend, aftertouch, CC 74, and release velocity. All of these are accessible via the **MIDI In** module. However, pitch bend and attack velocity also do specific things in Flow. If you wanted Flow to respond in some *non-bend* way when

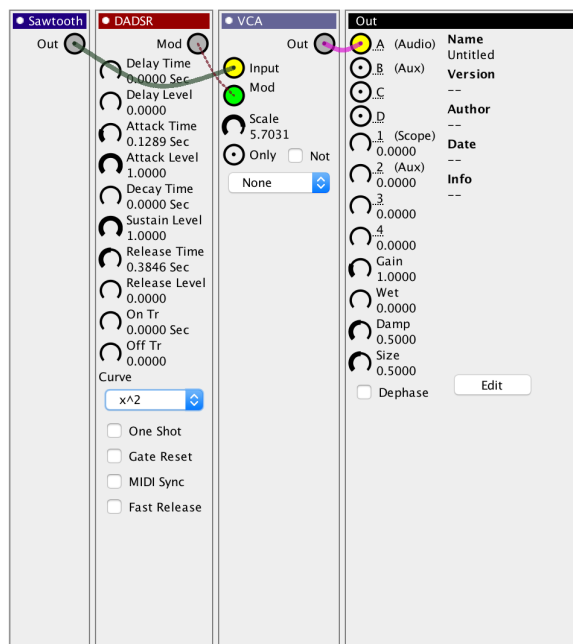


Figure 10: Playing a Saw wave modulated by an ADSR envelope and a VCA

you perform the bend behavior on a Roli Seaboard (say), you need a way to disable bend. To do this, you can deselect **Responds to Bend** in the **Play** menu. Similarly, you can disable **Velocity Sensitive** in the **Play** menu. The Bend and Attack Velocity signals still will be sent to the MIDI In module, where you can tap into them to make your patch do what you like.

2.10 Erasing, Loading, and Saving Patches

You can clear the Rack to prepare for a new patch by choosing **New Patch** from the **File** menu. Once you have created and tested your patch masterpiece, you can save it with **Save Patch As...** You'll be prompted to give the patch a name, and then save it as a file. You can resave your patch at any time with **Save Patch**. Later on, you can load a patch again with **Load Patch...**⁷

When you save a patch, if you have not already named the patch, the name of the patch will be set to the filename (minus its extension). Thus if you save it to a file called *Foo.patch*, then the patch will be named *Foo*. Once a file has been named — either after your first save, because you named it by choosing *Name Patch...* in the *Play* menu — its name won't change again even if you save it under a different file name. To rename the patch, just select *Name Patch...* again.

There's only one Rack: you can't create or load multiple patches at a time. If you load a patch, or clear it, the previous one will be deleted.

2.10.1 Macros

One of the particularly interesting features of Flow is its ability to load a saved patch as a **Macro**. A Macro encapsulates the patch and allows you to use it as a module in a larger patch. Macro modules appear with green title bars and are named the name of the patch.

A patch is most useful as a Macro if it's been wired up to send and receive modulation and partials through its module. To do this, you'll need to prepare the patch first. See Section 3.5.2 for an extensive discussion of the process.

2.10.2 Naming Patches

Patches have four pieces of auxiliary text information stored in them: a *patch name*, a *patch author*, a *patch date*, a *patch version*, and some *patch info*. You can put whatever you like in these four text boxes: just choose **Patch Info...** from the **Play** menu.

A patch doesn't need to have an author, date, version, or info: but it ought to have a name. This is because if you load a patch as a Macro, the Macro's module will be titled with the patch name. If you save a patch that has no name (yet), Flow will set the patch name to the filename (minus the extension).

2.11 Connecting to a Digital Audio Workstation

The primary challenge to connecting Flow to a DAW is that both Flow and your DAW are designed to connect to *MIDI Devices* and to *Audio Devices*, yet neither of them *is* a MIDI or an Audio Device! Thus neither of them can connect to each other! This is a stupid mistake in how MIDI and Audio routing is implemented in operating systems, but there you have it.

To get Flow and a DAW to send MIDI and Audio data to each other, you need to create a *MIDI Loopback* device and an *Audio Loopback* device. A Loopback device is a tunnel which presents itself as a device at each end: by connecting an application at one end and an application at the other end, each thinks it's talking to a device, but in fact they are talking to each other.

The problem with loopback devices is that they are slow: they introduce a great deal of latency both in MIDI and audio. This is the biggest problem with using Flow with a DAW. Flow has certain features which help mitigate this.

⁷If you're launching Flow from the command line, you can have it open a patch like this: `java flow.Flow patchname.flow`

Routing MIDI from a Digital Audio Workstation To enable your DAW to send MIDI to Flow, you need to make a *MIDI loopback*. This is where you create two *virtual devices* which are connected to one another. Flow and your DAW can both see these devices. Consider Figure 11. If your DAW outputs to Virtual Device X (say) and Flow is set up to *input* from Virtual Device X, then it will receive what the DAW outputs.

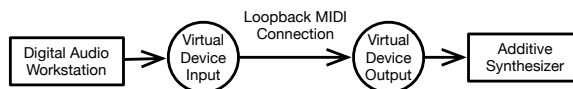


Figure 11: A MIDI loopback connecting a Digital Audio Workstation to Flow. Compare to Figure 12.

Making a MIDI loopback device varies depending on your operating system:

- **On the Mac** First, open the application /Applications/Utilities/Audio MIDI Setup, and choose “Show MIDI Studio” from the Window menu. Next, click on the “IAC Driver” icon to open the “IAC Driver Properties” window. Add a new port, named whatever you like (I named mine “From Ableton”). Check the box “Device is Online”. This new port will appear to your DAW and to Flow as the loopback device.
- **On Windows** There is no way to do this in Windows directly: instead you’ll need to run a program which provides this service. Programs include loopMIDI, loopBe1, MIDIOx’s MidiYoke, and so on. Googling for “loopback MIDI Windows” will get you there.
- **On Linux** In most flavors of Linux, to get virtual devices running you’ll first need to type the command `sudo modprobe snd-virmidi` and then type in your password. If you’re using something like Gentoo or any other distro that does not come with this kernel module, you’ll need to custom compile your kernel to get it.

This procedure will create a bunch of of virtual devices with names like VirMIDI [hw:2,0,0] or VirMIDI [hw:2,1,8]. Select a device whose third number is 0 (such as VirMIDI [hw:2,0,0] or VirMIDI [hw:3,1,0], but not VirMIDI [hw:2,1,1]). Have the DAW send to this device and have Flow listen from the same device.

Routing Audio from a Digital Audio Workstation To send audio from Flow to your DAW to be recorded, you’ll need a loopback device for audio similar to the one for MIDI. Once again how you’d make an Audio Loopback device depends on your operating system:

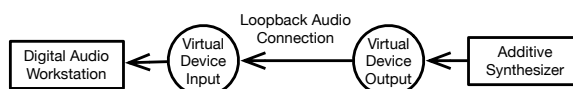


Figure 12: An Audio loopback connecting Flow to a Digital Audio Workstation. Compare to Figure 11.

- **On the Mac** Unlike for MIDI, there’s no built-in audio loopback facility on the Mac. You’ll need to install a program which does audio loopbacks. Here are some options:

– An older open-source driver called *SoundFlower*⁸ which works well though with high latency.

⁸<https://github.com/mattgall/soundflower/releases/> Note that the GUI installer fails on High Sierra even if you have given permission per the instructions on the website. But installing via the Terminal seems to work. Do the following:

1. Unpack the Soundflower-2.0b2.dmg disk image.

2. In the terminal, type:

```
cd /Volumes/Soundflower-2.0b2
sudo installer -pkg Soundflower.pkg -target /
```

Then press RETURN

Then press RETURN

3. Enter your password and press RETURN.

4. When you’re told that this has failed, go to “Security and Privacy” in System Preferences, and permit the software to load. You may need to click “Allow...”, then select “MATT INGALLS” (the developer) in the list and approve it. Then do steps 2 and 3 again.

5. Wait a long time. It takes quite a while to install for no good reason.

6. At this point, SoundFlower 2ch and 64ch will appear as audio devices. You want SoundFlower 2ch. You can configure it further in Audio MIDI Setup if you like, or just use it straight.

- A new open-source driver called *BlackHole*⁹ which promises low latency. I have not tested it.
 - The commercial product *Loopback* by Rogue Amoeba, which provides support and good user experience.¹⁰ It works very well, and if you're poor you can try it in unfettered trial mode for about 20 minutes, quit it, and start it again for another 20, etc.
 - **Do not** use the *Jack* framework or any tools using it. Jack breaks all Java Audio applications: you won't be able to run Flow at all after installing Jack. And furthermore, Jack's *uninstaller* won't remove its files properly and you'll be permanently unable to run anything that needs Java Audio (including Flow). **Do not install Jack on OS X.** You will regret it.¹¹
- **On Windows** I can't help you here. I'm sure there are many ways to do this, but I don't know them.
 - **On Linux** You'll want to use Jack 2 and Cadence. A good solution with minimal friction is to enable the ALSA and Pulseaudio bridges from within the Cadence GUI. Set the buffer size for your sound card in Cadence to something small, like 128 samples (ideally, the smallest you can get it without glitches): this allows you to minimize the application buffer size, reducing latency. After running Cadence and starting the jack daemon, run flow and set the sound output to the jack bridge in Pulseaudio via Pavucontrol (or an equivalent audio configurator). This will allow you to route audio from Pulseaudio into jack, which will most likely be able to interface with your DAW and other audio applications.

2.11.1 Multitimbral Patches

Flow can be set up to play more than one patch at a time. This is particularly useful because Flow is a very computationally costly: if you have attached Flow to a DAW and you want to play more than one Flow patch through the DAW, you'd probably rather not run multiple copies of the application, each tuned with a small number of voices.

In most synthesizers, a multitimbral patch is different from a "single" (one-sound) patch. In synthesizers like this, you'd make a bunch of single patches, then make one multitimbral patch which references all of them. Flow's multitimbral patch model is different. In Flow, you load a single patch, which we'll call the *primary patch*, then you attach one or more additional single patches to it as *subpatches*. You can save this combined bundle of patches out to a standard file and load it again. An example of a primary patch and three subpatches is shown in Figure 13.

Subpatches are added to the primary patch by choosing **Load Subpatch...** in the **File** menu. Only the primary patch can be edited: the subpatches will appear as thin rows beneath its modules. For each subpatch you can make the following settings:

- **Voices Requested** Here you stipulate how many voices you'd like to allocate to this subpatch. Voices allocated to a subpatch are taken out of voices available to the main patch.
- **Voices Allocated** Obviously not every subpatch can have a maximal number of voices. Here you can see how many voices were *actually* allocated. The allocation strategy is simple: for each subpatch from top to bottom, Flow attempts to give that subpatch every voice it asked for. Flow must reserve at least one voice for the main patch. If Flow runs out of voices, a subpatch may get fewer voices than it asked for, and later subpatches may get no voices at all.

One confusing situation arises when you have selected *Monophonic* from the *Play* menu. Here Flow only uses a single voice, and that voice must go to the primary patch, so the subpatches will all have zero voices. To make this situation clear, the subpatches will show **[M]** (for "monophonic") for the number of voices allocated, and "(Mono)" will appear in the **Out** module of the primary patch.

⁹<https://github.com/ExistentialAudio/BlackHole>

¹⁰<https://rogueamoeba.com/loopback/>

¹¹In fact, after I reported this, Jack's administrators seem to be leaning towards eliminating Jack on OS X entirely rather than fix things. Oh well. If you screwed up and installed Jack on OS X, the following URL contains a list of all the files you'll need to remove manually (as administrator): <https://github.com/jackaudio/jack2/issues/379>

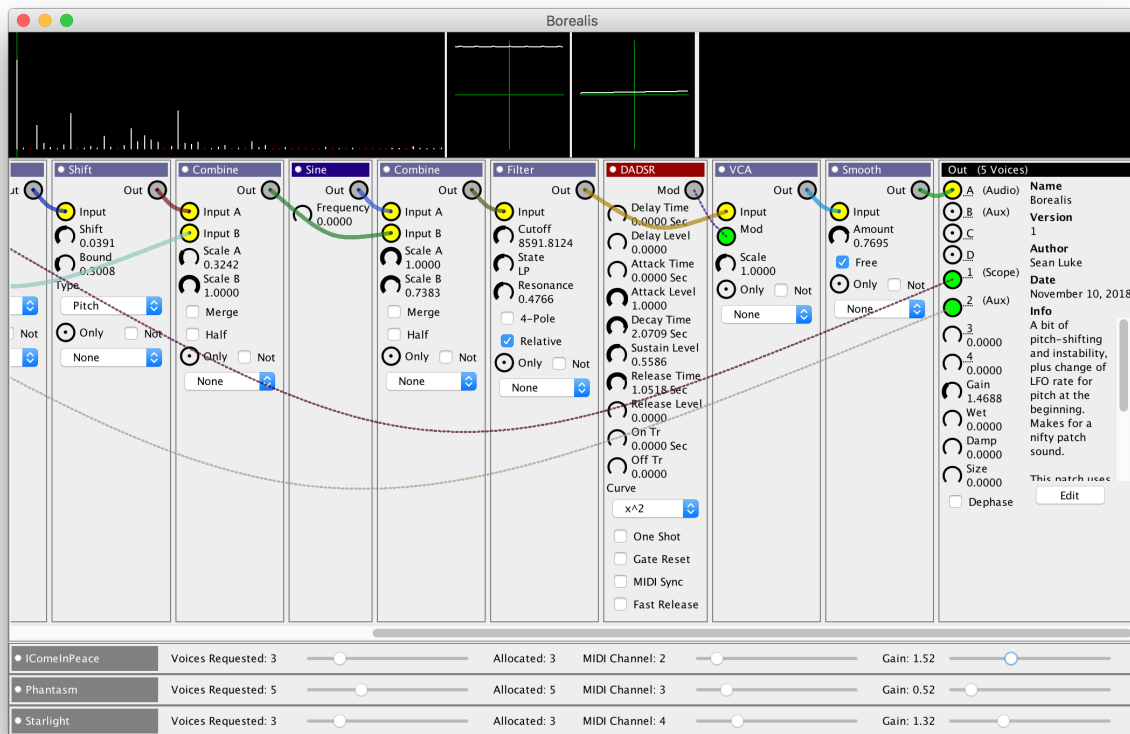


Figure 13: Primary patch and three subpatches.

- MIDI Channel** This specifies which MIDI channel is used by the subpatch (or, if no MIDI channel, “Off”). If multiple subpatches request the same MIDI Channel, the earlier (higher) one will prevail. Subpatches also get MIDI channel priority over the primary patch, so if a subpatch and the primary patch are using the same MIDI channel, the subpatch gets it.

This has interesting implications when the primary patch is set to “Any Channel”, “MPE Lower Zone”, or “MPE Upper Zone”. Let’s start with “Any Channel”. If your primary patch is set to this, then it responds to any channel *except* for the ones allocated to the subpatches.

MPE is more complicated. Recall that if you are using MPE, then you also specify the number of MPE channels. If you have chosen MPE Lower Zone with n channels, then normally the primary patch would be allocated MIDI channel 1 for its *MPE Global Channel* and channels 2 ... $n + 1$ for the requested MIDI channels. However if a subpatch has allocated one of those channels, it will take it from the primary patch. This will disrupt MPE.

Thus if you have chosen MPE Lower Zone with n channels, you want your subpatches to request channels somewhere in the range $n + 2$... 16. For example, if you have 3 subpatches, you might set up MPE Lower Zone with $n = 12$ (so MPE needs channels 1 ... 13), and then use channels 14, 15, and 16 for the subpatches.

If you have instead chosen the MPE Upper Zone with n channels, then the MPE Global Channel is 16 and the MPE channels go down from there, that is, they’re assigned to 16 - n ... 15. Once again, you want your subpatches to stay out of this region, so you’d assign to subpatches channels in the region 1 ... 16 - $n - 1$. For example, if you have 3 subpatches, you might once again set up MPE Upper Zone with $n = 12$ (so MPE needs channels 4 ... 16), and then use channels 1, 2, and 3 for the subpatches.

Subpatches cannot use MPE, nor can they be assigned to “Any Channel”. Only the primary patch can.

- **MIDI Note** You can restrict a sound to only respond to a given note: any other notes on the MIDI channel will be passed on to later sounds. This is useful for saving MIDI channels while building a drum kit: have all of your percussion sounds respond to the same MIDI channel but to different notes. It’s also helpful when your DAW has a restricted number of tracks, since the entire drum kit can be on the same track.
- **Gain** This sets the overall volume of the subpatch in the final mix. This is exactly the same as the *Gain* knob on the **Out** module of the primary patch, mentioned earlier in Section 2.9. And in fact, the Gain knob can be modulated: if your subpatch was originally modulating its Gain knob before you loaded it as a subpatch, changing this slider will override it and fix it to a constant value. You can reset this by making the patch primary and then secondary again.

If you want to change the overall gain of the primary patch and all the subpatches, you can do this by changing the *Master Gain* option in **MIDI and Audio Preferences**. Note that Master Gain is not saved with patches and is reset when you restart Flow.

- **Pan** This sets the overall pan of the subpatch in the final mix. As is the case for Gain, this is exactly the same as the *Pan* knob on the **Out** module of the primary patch, mentioned earlier in Section 2.9. The Pan knob can be modulated: if your subpatch was originally modulating its Pan knob before you loaded it as a subpatch, changing this slider will override it and fix it to a constant value. You can reset this by making the patch primary and then secondary again.
- **Rename** You can rename the subpatch name.

Flow mixes the primary patch and subpatch voices as follows: each voice is amplified according to its the gain of its patch or subpatch, and then the voices are added together. Finally all the voices are pushed through the reverb. This means that **reverb is global**: the settings you make in the primary patch will affect all voices, even ones in subpatches. We may change that in the future.

You can remove a subpatch by clicking on its close box, just like removing a module. You can also drag subpatches to reorder them relative to one another, just like modules.

Primary Patch Parameter Display Subpatches take priority over the primary patch both in allocated voices and MIDI channels, so how you set them seriously affects the primary patch’s resources. You can see what those resources are by looking at the title bar of the **Out** module:

- **Voices** If the title just says *Out*, then it has full voices. Otherwise it’ll say something like *Out (13 Voices)*.
- **Monophonic** If you are playing in monophonic mode, the Primary patch gets the one and only voice. In this case, the title will say *Out (Mono)*.
- **MIDI Channel** If you have any subpatches, the title will display the MIDI channel of the primary patch. For example, if the channel is 13, the title will say something like *M13*. If another subpatch has *also* chosen channel 13, then the title will say *M13?*. Whether the subpatch has fully overridden the primary patch depends on whether the subpatch is restricted to a single MIDI Note (and it’s different from the primary patch of course).
- **MIDI Note** Speaking of the MIDI note: normally you can’t set the MIDI note in a primary patch—however if you set it in a subpatch, then swap the subpatch to the primary patch, then it will have a MIDI note (mostly under the assumption that you’re doing this temporarily and will swap it back). In this case, the note will be displayed, something like *Ab2*.

Changing Subpatch Order As mentioned before, the order of the subpatches affects which patches get allocated voices and priority over MIDI channels. You change this order by dragging the titlebar of a subpatch and dropping it before or after another subpatch.

Swapping a Subpatch with the Primary Patch If you drag the titlebar of a subpatch clear into the region of the *primary patch*, it will **swap** the subpatch with the primary patch. Flow will preserve the reverb settings from the previous patch. This is useful for making a subpatch the Primary Patch temporarily in order to tweak it.

Swapping the subpatch comes with some MIDI channel complications. First, only the Primary patch can perform MPE: so if you are swapping a primary patch that uses MPE, when it becomes a subpatch its MIDI channel will become “No Channel”. Similarly, only subpatches can have “No Channel” as a MIDI channel option,¹² so if one of them is swapped to a primary path, its MIDI channel will change to “All Channels”.

Another thing you need to watch out for is swapping the Primary patch whose Gain is under the control of a modulation such as an LFO. When it becomes a subpatch, the subpatch’s Gain slider will be set to 0 (and it won’t work anyway). In short: don’t control Gain from a modulation.

You can do a different kind of subpatch swap if you do the swap while holding down the Control key (on the Mac, you can also hold the Option key).¹³ This will swap the two patches as before, but the MIDI channel, MIDI Note, and Number of Requested Sounds won’t be swapped: they’ll stay as they were. Why would you want to do this? If you are auditioning a bunch of different sounds, you may have them lined up as subpatches and are dragging them into the primary patch to hear them play a given part (which has its own MIDI channel).

Loading, Clearing, and Saving the Primary Patch Obviously you could clear the primary patch or load a new one by choosing *Load Patch...* or *New Patch* respectively, but this would clear out the subpatches as well. If you just want to clear just the primary patch alone, choose *New Primary Patch*. You’ll be asked if you wanted to optionally *demote* the primary patch, that is, to turn it into a subpatch rather than eliminate it.

Similarly, if you choose *Load Primary Patch...*, you will load a new primary patch, replacing the original, but keep the subpatches. You will likewise be asked if you would like to optionally *demote* the primary patch just as before. If you’d like to save the primary patch to a file (but not its subpatches), you can choose *Export Primary Patch...* This sometimes happens when you’ve modified a subpatch in a multitimbral setting and would like to keep it: swap it to the primary patch, export it, then swap back.

2.11.2 Digital Audio Workstation Integration Strategies

Here’s a common scenario. You have so far recorded MIDI for several Flow tracks in your DAW, and you’re currently recording MIDI for a new track while playing the old ones. This new track is connected to Flow’s Primary Patch, while the several old tracks are connected to subpatches. When you have finished recording the new track, you load a new patch into Flow’s Primary Patch, demoting the current Primary Patch to a subpatch, and you begin anew.

To get this scenario working, you assign each Flow subpatch a unique MIDI channel. The Primary patch also gets its own MIDI channel. Your keyboard is connected to the DAW’s recording track, and the DAW is rerouting its MIDI data to the primary patch in Flow. Flow’s sound output is connected to the DAW in anticipation of eventually recording the audio: for now the DAW is just routing it to the speakers.

Latency There is a big problem with this scenario: high latency. Your MIDI keyboard is routed to the DAW, which is then *rerouting* it to Flow, which is then sending audio to the DAW, which is then *rerouting* the audio to your speakers.

¹²This may change in the future.

¹³Yes, I know this is the behavior for “copying” while drag and drop, and in fact you’ll get a copy icon. But there’s no “alternative drag” option in Java, so I had to make do.

We can reduce the MIDI latency by eliminating the rerouting: the DAW doesn't reroute the keyboard to Flow, but only records it into the track. Instead, we set up the keyboard as a *second MIDI device* in Flow: it's going directly to Flow's Primary patch. The DAW is also sending MIDI to Flow (for the subpatches). To do this, you can set the keyboard as the **Aux MIDI Device** in the **MIDI and Audio Options** window. Make sure that the DAW isn't routing the keyboard to Flow as well or you'll get two copies of the same MIDI data.

The best way to reduce the Audio latency is to not send audio to the DAW until you absolutely have to (to record an audio track). Instead, just have Flow play directly to your speakers.

Computational Cost Flow uses a lot of CPU. *A lot of CPU.* Your objective is to eek out as many voices as possible to make your song before it begins to glitch. I can usually get 20–24 voices on my laptop. How is this done?

- Turn off Flow's partials and oscilloscope displays. That saves a lot.
- For each subpatch, reduce the voices requested to the minimum number needed before you hear the artifacts of voice stealing.
- If necessary, temporarily reduce the number of partials being used while recording and editing the MIDI in tracks. Then bring the partials back up to (say) 256 for the final audio recording. This is an extreme measure but it works, especially in combination with...
- You can significantly increase the audio buffer size in the final recording, which will reduce glitching by quite a lot even with 256 partials and a maxed-out voice count: the downside is heavily increased latency, but that wouldn't matter for the final audio recording and can be shifted in post.
- Keep in mind that some patches are much more costly per-voice than others. Keep their voice counts minimized.
- Changing to Mono from Stereo might help a tiny bit.
- Shut off other applications: especially web browsers and other CPU-hungry stuff.

Compressing Tracks I am cheap and don't own a full-featured copy of Ableton: I just have Ableton Live Lite. The problem with this is that I am restricted to 8 tracks total, including the audio track (I record multiple MIDI tracks, then play them together with subpatches and record a final audio track). This means that I can have at most 7 MIDI channels going into Flow. How then can I play (say) 12 different sounds?

It's often the case that many of my sounds are percussion. Percussion sounds are typically played with a single note each — indeed it doesn't matter what the note is because I use the **Fix** module (Section 3.5) to force them to play at a certain pitch anyway. For this reason, I often assign each percussion sound to a certain note: the bass kick to C4, the high hat to C5, etc., and have them all recorded like a drumset on a single track.

Flow can handle this without any difficulty: each percussion subpatch has the exact same MIDI channel, but is assigned a different note via the subpatch's **MIDI Note** restriction. The kick gets assigned to C6 etc.¹⁴ This lets me play multiple Flow percussion sounds from a single track!

Master Gain At the end of the day I can get my whole song to play without glitching, but it still clips here and there. Rather than go through and adjust the gain of each subpatch separately, I just pull down the master gain (see the **MIDI and Audio Options** window). Note that Master Gain is not saved with patches and is reset when you restart Flow.

¹⁴Note that Ableton's octaves go from -2 to 8 in old Yamaha tradition, while Flow's go from 0 to 10. I just add 2 to convert to Flow.

3 The Modules

The Modules are divided into five categories:

- **Modulation Sources** Are meant to be sources of modulation information. Example: a low frequency oscillator (LFO). These appear in the top portion of the **Modulation** menu.
- **Modulation Shapers** Are meant to modify modulation information. Example: a module which takes an incoming signal and does a Sample and Hold on it. These appear in the bottom portion of the **Modulation** menu.
- **Partials Sources** Are meant to be sources of, well, partials. Example: a sawtooth wave. These appear in the **Sources** menu.
- **Partials Shapers** Are meant to modify partials, producing new ones. Example: a low pass filter. These appear in the **Shapers** menu.
- **Special** Various system modules, such as In, Out, Choice, Fix, Note, and Macro. Just because these are at the end doesn't mean they're not important (they're probably the *most* important!). These appear in the **Other** menu, along with the option of loading an entire patch as a Macro module.

3.1 Modulation Sources

AHR A simple Attack-Hold-Release (or is you like Attack-Hold-Decay) envelope. As you play a note, this envelope starts at the Start Level, ramps up to the Attack Level over the course of the Attack Time, then holds at that level for the Hold Time. It then releases (decays) back to the Start Level over the Release Time. If you let go of the key early, it immediately starts releasing for the given Release time (unless it's already releasing, in which case it just continues to do so). AHR is much simpler than the DADSR envelope module, but it does have one feature that DADSR does not: you can specify the attack and release curves independently. There are many curves available, as shown in Figure 14. The envelope can be one shot, meaning it continues until it is finished even if you have released. The envelope triggers on completion of each of its stages. You can request that the envelope sync to MIDI when the synthesizer is set to MIDI (otherwise it's based on time as usual). If you double-click on any time dial, a pop-up menu will appear which lets you set various note speeds corresponding to playing at 120BPM. If the MIDI clock is running, these note speeds will correspond to the MIDI clock's rate.

AHR can be configured to instead act as a *ramp* function: when a Note-On happens, the envelope restarts at the Start Level, then ramps to the Attack Level over the course of the Attack Time, and *holds there indefinitely*. A Note-Off simply stops the ramping and it holds wherever it stopped. If the envelope is one-shot, then a Note-Off has no effect at all.

The *On Tr* option restarts the envelope whenever it receives a trigger (as if a Note-On happened), and the *Off Tr* option starts the release stage whenever it receives a trigger (as if a Note-Off happened). These are auxiliary triggers: don't use them to trigger on a note-on/note-off directly (as in from the MIDIIn module).

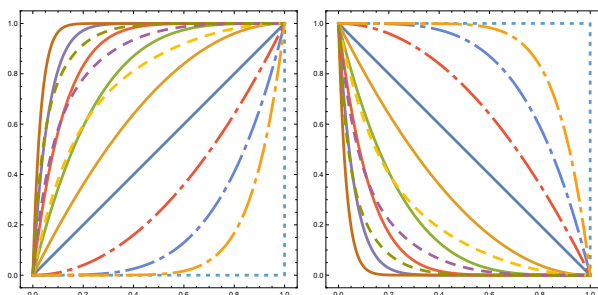


Figure 14: Envelope modulation curves, shown in rising and falling scenarios. Solid lines are (right to left) *Linear*, x^2 , x^4 , x^8 , x^{16} , and x^{32} . If you're looking for an exponential-sounding curve, x^4 works pretty well. Dashed lines are (right to left) $(x^2 + x^8)/2$, $(x^4 + x^{16})/2$, and $(x^8 + x^{32})/2$ (denoted $x^{2,8}$, $x^{4,16}$, and $x^{8,32}$ in the popup menu). Dot-dash lines are (left to right) $1 - (1 - x)^2$, $1 - (1 - x)^4$, and $1 - (1 - x)^8$ (denoted *Inv* x^2 , *Inv* x^4 , and *Inv* x^8 in the popup menu). The dotted line is *Step*, which stays put until the very end, then suddenly jumps to the ultimate value.

The envelope does that by default already. Furthermore, if you connect to the On Tr port, then the envelope will *only* restart when a trigger occurs: it will *not* restart on a Note On. Similarly, if you connect to the Off Tr port, then the envelope will *only* release when a trigger occurs: it will *not* release on a Note Off.

At the bottom are outputs which trigger when the envelope starts to Hold or Release, or when it at its End. Mod will trigger in all these cases. See also **DADSR** and **Envelope**.

DADSR A standard Delay-Attack-Decay-Sustain-Release envelope.¹⁵ You can select curves from linear to pseudo-exponential, plus “Step”, which immediately steps to the next level. Figure 14 on page 20 shows available curves. The envelope can be one shot (it doesn’t stop at sustain), and it can be set to reset to 0 on a Gate (Note On), as opposed to just starting from its current position. The envelope triggers on completion of each of its stages. You can request that the envelope sync to MIDI when the synthesizer is set to MIDI (otherwise it’s based on time as usual). Finally, *Fast Release* changes how release and Decay interact. If Fast Release is true, then on Note Off, the envelope will start the release stage immediately; but if Fast Release is false (the default), then if you do a Note Off during the Delay stage, the Delay stage will be completed fully before the release starts. No other stages are affected: for example, if a Note Off occurs during Attack, release immediately starts no matter what.

If you double-click on any time dial, a pop-up menu will appear which lets you set various note speeds corresponding to playing at 120BPM. If the MIDI clock is running, these note speeds will correspond to the MIDI clock’s rate.

The *On Tr* option restarts the envelope whenever it receives a trigger (as if a Note-On happened), and the *Off Tr* option starts the release stage whenever it receives a trigger (as if a Note-Off happened). These are auxiliary triggers: don’t use them to trigger on a note-on/note-off directly (as in from the MIDIIn module). The envelope does that by default already. Furthermore, if you connect to the On Tr port, then the envelope will *only* restart when a trigger occurs: it will *not* restart on a Note On. Similarly, if you connect to the Off Tr port, then the envelope will *only* release when a trigger occurs: it will *not* release on a Note Off.

At the bottom are outputs which trigger when the envelope starts to Attack, Decay, Sustain, or Release, or when it at its End. Mod will trigger in all these cases. See also **AHR** and **Envelope**.

Envelope An eight-stage envelope with looping. The envelope has three stages prior to sustain, then three sustain stages, then two stages at release. Each stage has a time to completion, and a level at which it will have reached when completed. If the time is set to 0.0, then the stage is ignored entirely.¹⁶ There is also a *Start* option which specifies the initial level. The envelope triggers on completion of any of its stages. You can select curves from linear to pseudo-exponential, plus “Step”, which immediately steps to the next level. Figure 14 on page 20 shows available curves. You can request that the envelope sync to MIDI when the synthesizer is set to MIDI (otherwise it’s based on time as usual).

The Envelope has several types. *Sustain* works like a regular envelope: it works its way up through the final sustain state and then stays there, and if you do a note-off, it starts immediately on the first release state and continues from there. *Sustain Loop* works like Sustain, but once the final sustain state has been completed, it loops back through the sustain states rather than staying at the last one. *One Shot* works its way through all the states until it has completed them, regardless of note off. *Play Through* is like *One Shot*, except that it jumps to the first release stage on note-off; thus Play Through is like Sustain but without waiting at the last sustain stage. *Loop w/Rel* (Loop with Release) loops through all the states over and over again, and stops looping on note-off. *Loop* loops through all the states over and over again forever; but is reset on note-on like all other envelope types.

If you double-click on any time dial, a pop-up menu will appear which lets you set various note speeds corresponding to playing at 120BPM. If the MIDI clock is running, these note speeds will correspond to the MIDI clock’s rate.

¹⁵Though in fact the “Delay” is really just a first-stage attack: it’s only effectively a delay if its level is set to 0. Indeed, if you treated it as an attack, and then set Attack’s level to the same as Delay’s level, then you’d have an “ADHSR” (“H” for “Hold”, as in E-mu products and the Moog One).

¹⁶This means that if you attach modulation to your envelope time values, you might want to make sure your modulation doesn’t quite hit 0.0, or things might not work exactly as expected.

The *On Tr* option restarts the envelope whenever it receives a trigger (as if a Note-On happened), and the *Off Tr* option starts the release stage whenever it receives a trigger (as if a Note-Off happened). These are auxiliary triggers: don't use them to trigger on a note-on/note-off directly (as in from the MIDIIn module). The envelope does that by default already. Furthermore, if you connect to the *On Tr* port, then the envelope will *only* restart when a trigger occurs: it will *not* restart on a Note On. Similarly, if you connect to the *Off Tr* port, then the envelope will *only* release when a trigger occurs: it will *not* release on a Note Off.

At the bottom are outputs which trigger when the envelope starts its Pre 1, Pre 2, Pre 3, Sus 1, Sus 2, Sus 3, Rel 1, or Rel 2 stages, or when it at its End. Mod will trigger in all these cases. See also **DADSR** and **AHR**.

Joystick A controller that lets you simultaneously change six different modulation outputs. Click *Show* to reveal the Joystick. As you change the joystick, you modify the modulation outputs in the following way:

- The → and ↑ outputs vary as you drag the joystick horizontally and vertically respectively. These are useful if you want to control two different modulations completely independently of one another.
- The (↖ ↗ ↘ ↙) outputs are each associated with a corner the value of each output is based on how close the joystick is to its corner (closer = higher value).¹⁷ All four modulations are at 0.5 in the center. This is more useful if you want to “blend together” four different modulation sources, and is the classic vector synthesis arrangement.

True to vector synthesis tradition, you can also *record* a drag sequence, and once you are finished, the recording will be played back each time you press a key, affecting that voice. You record a sequence by clicking on the *Record* button, which arms it. The joystick turns green and the button now says *Cancel*, and clicking on it again will (of course) cancel the recording. Recording begins as soon as you grab the joystick and start moving it: the button will then say *Recording* and the joystick will turn red as usual. A point labelled *Start* will indicate where you started (helpful for closing loops, see below). When you let go, the joystick turns blue again, and the button now says *Clear*, indicating that you have made a recording. Playing notes will now trigger the prerecorded movements (independent of one another). If you now click the button, you will erase the recording and go back to *Record*. You can always click the little triangle to hide the joystick: the recording remains (and is saved with your patch).

The *Loop* checkbox determines whether the recorded trajectory, when played back, does so in an infinite loop. You can also start (or restart) playback of a recorded trajectory by sending a trigger to the *On Tr* dial. If the *On Tr* trigger is connected, then a Note On will no longer start the playback: only the trigger will.

Joystick is particularly useful with the **Mix** or **Combine** modules, where you can hook up each corner to the Gain or Scale of a different input to get those classic vector synthesis vibes.

LFO A low-frequency oscillator with a rate and phase, a shift of its zero point vertically, and vertical scaling of the wave. If you double-click on the rate dial, a pop-up menu will appear which lets you set various note speeds corresponding to playing at 120BPM. If the MIDI clock is running, these note speeds will correspond to the MIDI clock's rate.

The oscillator can run free, you can invert the wave. The LFO triggers on completion of one full wave, or one-half wave if *Half Trigger* is set. If *Init Trigger* is set, then the LFO also triggers on the start of the very first wave. The waves can be sine, triangle, saw up, square, random, and random sample and hold. You can request that the LFO sync to MIDI when the synthesizer is set to MIDI (otherwise it's based on time as usual). And finally, you can indicate that the rate dial should be linear and not exponential (which is useful when modulating an LFO with another LFO).

Random works as follows: a random new target Y position is set and the LFO spends *rate* time to get there. Once it reaches the point, it picks a new random target Y position and repeats. The sample and hold version is similar except that the wave does not move to the target Y position, but rather immediately jumps

¹⁷Technically the value of a modulation is $1.0 - w$, where w is the maximum of the horizontal and vertical distances from the corner: the joystick has a width and height of 1.0.

to it and stays there for *rate* time. *Variance* indicates how big a typical random jump is: this is also scaled by *Scale*. If the seed is set to *Free* (0), then the random waves are fully random; but if the seed is $\neq 0$, then every time a gate occurs (assuming the LFO is not free), the LFO's random behavior will be exactly the same as before, so you can make a predictable "random" sound.

The *On Tr* option restarts the LFO whenever it receives a trigger (as if a Note-On happened). This is an auxiliary trigger: don't use it to trigger on a note-on directly (as in from the MIDIIn module). The LFO does that by default already. Furthermore, if you connect to the On Tr port, then the LFO will *only* restart when a trigger occurs: it will *not* restart on a Note On.

MIDIIn A source of MIDI information. Modulation sources include:

- A gate: 0.0 when NOTE OFF, and 1.0 when NOTE ON. The gate modulation triggers on NOTE ON.
- The most recent MIDI Note: values 0...127 map to 0.0...1.0. This modulation triggers on NOTE ON.
- The most recent attack velocity value, drawn from NOTE ON. This modulation triggers on NOTE ON.
- The most recent release velocity value, drawn from NOTE OFF. The velocity triggers on NOTE OFF.
- The current pressure (aftertouch) value. This responds to both channel and polyphonic pressure.
- The pitch bend value. This value is the actual MIDI bend signal, so extreme bend low is 0.0, extreme bend high is 1.0, and no bend is 0.5.
- The MIDI clock. This triggers every MIDI clock pulse. Additionally, when the clock is running, the value is 1.0, else 0.0. When a MIDI START message is received, the clock will also go to 0.0 prior to the first pulse (at which point the clock is considered running, so it goes to 1.0). You will probably need to divide the MIDI clock pulse to something more useful for your purposes: for example getting a trigger every beat (every 24 pulses). To do something like this, try using the *Trigger 2A* through *Trigger 192A* options in the **Mod Math** module.
- Eight CCs. You can specify the CC parameter. The resulting value (0...127) is mapped to 0.0 ... 1.0. If a CC changes, its modulation will trigger; thus you can set up buttons on your controller to trigger events. The popup menu for each CC dial shows common CC parameter settings. The first option on this menu says "Learn Most Recent CC": if you select this, then the CC parameter will be set to the most recent CC that Flow has received. That way you can just turn a knob on your controller, then select Learn Most Recent CC, and you're done.

Note that CC values 6, 38, 98, 99, 100, and 101 are invalid: they are used by Flow for NRPN. If you set to them, you won't get any resulting modulation. Additionally note that Flow responds to CC parameters 64, 120, and 123 (sustain pedal, all-notes-off, and all-sounds-off respectively), so you should probably leave those alone. The first CC value is set to 74 by default, which is convenient for MPE users, and the second CC is set to 1 by default, which corresponds to your controller's Mod Wheel. You can change these of course. If you need more than 8 CCs, just create another MIDIIn module.

Keep in mind that gate, velocity, pitch bend, note, and clock information are already passed to all the modules internally; so this is really auxiliary stuff that might be useful for you to hook up. See also the **NRPN** module, next.

MPE A source of Multidimensional Polyphonic Expression (MPE) information. Sources include:

- The most recent attack velocity value, drawn from NOTE ON. This modulation triggers on NOTE ON.
- The most recent release velocity value, drawn from NOTE OFF. The velocity triggers on NOTE OFF.

- The pitch bend value. This value is the actual MIDI bend signal, so extreme bend low is 0.0, extreme bend high is 1.0, and no bend is 0.5.
- The current pressure (aftertouch) value, mapped from 0...127 to 0.0...1.0, then modified by the offset and variance below, then bounded to between 0.0 and 1.0. This responds to both channel and polyphonic pressure.
- Pressure offset: this value is added to the pressure.
- Pressure variance: this value is multiplied by the pressure (prior to offset). Also, if the variance is negative, the final pressure value is inverted (subtracted from 1.0).
- The current “Y controller” (CC 74) value, mapped from 0...127 to 0.0...1.0, then modified by the offset and variance below, then bounded to between 0.0 and 1.0.
- Y controller offset: this value is added to the pressure.
- Y controller variance: this value is multiplied by the pressure (prior to offset). Also, if the variance is negative, the final pressure value is inverted (subtracted from 1.0).

The purpose of the offsets and variances is that different controllers issue quite different values for pressure and Y. Notably the Roli Seaboard has fairly poor offsets for pressure and Y.

NRPN A source of MIDI NRPN information. NRPN works similarly to CC in that it has parameters, each of which can be set to values. The parameters range from 0...16383 and are set using two numbers: the *Most Significant Byte* (MSB) and the *Least Significant Byte* (LSB). Each ranges 0...127. The parameter number $P = \text{MSB} \times 128 + \text{LSB}$. When you set the MSB and LSB to specific values, to the right of the LSB you’ll see the resulting parameter number. Some synthesizers only use the MSB or the LSB.

Like parameters, values also can be in the range 0...16383. Flow always expects incoming NRPN data in “Fine” or “14-bit” mode: that is, the whole value range 0...16383 is mapped to modulation values 0.0 ... 1.0. If you need more than 6 NRPNs, just create another NRPN module.¹⁸ See also **MIDIIn**, previous.

Random A source of random modulation values. Random changes its current value on three events: resetting Flow, on an incoming trigger to Trigger, and (if Note On is set) on note on. You could attach a Note-On Random module to Output’s Pan to create a pan spread, for example. The random values emitted are chosen from the range between Low and High. If the seed is set to *Free* (0), then the random values are; but if the seed is $\neq 0$, then on reset the random sequence will be the same for a given seed.

3.2 Modulation Shapers

Either/Or A switch which chooses among up to four outputs. You choose the number of outputs with *Options*. Thereafter, whichever output is currently being referenced by the value in *Choice* will receive the value of *Yes*. All other outputs will receive the value of *No*. Either/Or is particularly useful with **Mix**, where it can be used to zero out the amplitudes of all the mix inputs except for one, allowing you to choose from among several inputs. See also **Mix**.

¹⁸A technical note. NRPN data comes in chunks of four different types: MSB, LSB, Increment, and Decrement. Flow’s NRPN module doesn’t update its value when it receives an MSB chunk. It waits until an LSB, Increment, or Decrement shows up.

Geiger A source of randomness in timing. When it receives an input *Trigger*, Geiger outputs a trigger (and switches from modulation of 0.0 to 1.0 or vice versa) only with a certain random probability. If *Trials* is 1, then the probability is simple: it's just the probability specified by the *Prob* dial. If *Trials* is > 1 , then instead of outputting a trigger, Geiger instead increments a counter with the given probability. When that counter is $\geq \text{Trials}$, only then does Geiger output a trigger, and the counter is reset to 0. The effect of this is that as *Trials* is increased, the effective variance between fast and slow intervals is reduced: but you'll also need to increase the rate at which you provide incoming triggers to keep things at about the same speed.¹⁹ Geiger also has a *Seed* for its randomness. If the *Seed* is *Free* (0), then Geiger uses a nondeterministic random source: otherwise the random sequence is always the same every time a note is played, and that sequence is specified by the *Seed*. Finally, if *Pulsing* is true, then after rising, Geiger always drops on the next incoming pulse and does not issue a trigger when doing so.

Map Takes an incoming modulation signal and two values, *A* and *B*. If *Inverse* is not checked, the signal is then stretched such that what used to be 0.0 is now *A*, and what used to be 1.0 is now *B*. That is, if the incoming value is x , then the outgoing value $y = A + (B - A)x$. If *Inverse* is checked, it's the opposite mapping: what used to be *A* (or lower) is now 0.0, and what used to be *B* (or higher) is now 1.0. The point of *Inverse* is that you can use a *Map* to map a modulation value, then use another *Map* to *unmap* that modulation value back to its original range. You also have the option of *Flipping* x first, that is, setting it to $1 - x$ (if *Inverse* is checked, *Flipping* happens at the end). *Map* has a final option called *Center*. If this is checked, then instead of directly specifying *A* and *B* as bound for your range, you instead specify a *Center* and a *Variance* of the range, which is often more convenient. The bounds are thus defined as $A = \min(0, \text{Center} - \text{Variance})$ and $B = \max(1, \text{Center} + \text{Variance})$.

Mod Math This module performs a variety of operations on incoming modulation signals *A* and *B*, then bounds the result to be between 0 and 1, and finally outputs it. It also can do some logic on the incoming trigger.

Operations can be *unipolar* (the default) or *bipolar*. A unipolar operation treats the inputs and output (0...1) as they are. A bipolar operation treats them as $(-1/2 \dots +1/2)$, and so the midline (1/2) is treated as if it were 0. This is done by first subtracting 1/2 from *A* and *B*, then performing the operation, then adding 1/2 to the result, and finally bounding the result to between 0 and 1 before returning it.

Operations:

- **+** Does $A + B$
- **-** Does $A - B$. By setting $A = 0$ (when unipolar), you can also use this to negate *B*.
- **×** Does $A \times B$
- **min** Does $\min(A, B)$
- **max** Does $\max(A, B)$
- **square** Does $A^2 + B$
- **sqrt** *Unipolar*: does $B + \sqrt{A}$ *Bipolar*: if $A \geq 0$, does $B + 1/2\sqrt{2|A|}$, else $B - 1/2\sqrt{2|A|}$
- **cube** Does $A^3 + B$
- **discretize** Discretizes *A*, using *B* to set the amount of discretization (from 1 to 128 chunks). This operation is always unipolar.

¹⁹In case you were interested, Geiger's random number events are using a *Negative Binomial Distribution*. Following https://en.wikipedia.org/wiki/Negative_binomial_distribution Geiger's *Trials* dial specifies the r parameter, and the *Prob* dial specifies $1 - p$. When *Trials* is 1, then this simply the *Geometric Distribution*, and following https://en.wikipedia.org/wiki/Geometric_distribution the *Prob* dial specifies the value p .

- **map hi** Does $A \times (1 - B) + B$. This “inverts” A under B.
- **average** Does $(A + B)/2$
- **threshold** If $A \geq B$, returns 1, else returns 0.
- **switch** If B more recently triggered than A, then B, else A

Additionally, there are a number of operators on triggers.

- **A** Triggers on A’s trigger only.
- **A or B** Triggers on A’s trigger or on B’s trigger.
- **A and B** Triggers only when A’s trigger and B’s trigger coincide.
- **A and not B** Triggers only when A’s trigger occurs but B’s trigger does not.
- **2 A through 192 A** A rate divider. Triggers every N triggers of A. This is particularly useful in conjunction with *MIDI Clock* information from the **MIDI In** module.²⁰
- **dir** Triggers when A’s value changes direction.
- **center** Triggers when A’s value crosses the center line (0.5).

S & H Sample and Hold. Takes an incoming modulation signal and a trigger. If the type is *S & H*, then on receiving the trigger, it samples the signal and outputs that value until the next trigger, when it samples again, and so on. If the type is *T & H* or *T & H 2*, then we are in *Track and Hold* mode. Here, the trigger *toggles* between sampling and holding the value, or letting it pass through normally. The difference between the two Track and Hold modes is phase: in the first version, when a gate is received, we are in sample-and-hold mode, where as in the second version, when a gate is received, we are in pass-through mode.

Seq A step sequencer of up to 32 steps, as specified by *Steps*. Each step is a modulation value which you can set. On note-on, the sequencer resets itself to step 1, and begins incrementing each time the *Trigger* is fired. If *Stop on Release* then on note-off the sequencer stops, otherwise note-off is ignored. If *Free* is true, then the sequencer is free-running and ignores both note-on and note-off.

The value of the current step is outputted as *Mod*. When *Sample* is true, then *Mod* outputs the value of the current step as sampled immediately upon transitioning to that step; but if *Sample* is false, then the value of the current step may be changed in real-time, and *Mod* will output whatever its current value is. *Mod* also outputs a trigger each time the step is incremented: you can use this to reset a sound to its start.

Seq doesn’t just read trigger values from *Trigger*: it also reads modulation values. This can be used in two ways. First, you can use the modulation value to slide the sequencer from one step value to another. Specifically, *Change* specifies how the sequencer interpolates from one step to the next: the default is *Step*, and other curves are shown in Figure 14. Interpolation is based on the incoming modulation value: if you’re using an LFO, try setting it to Sawtooth. You can get the opposite interpolation waves by setting the LFO to an inverted Sawtooth. Second, if *Guided* is true, then the *Trigger* not only specifies when a step occurs, but its modulation value also specifies *which* step is performed, rather than incrementing the step as normal.

²⁰Note that if you’re dividing a MIDI Clock pulse, some of these divisions might be useful for specifying note lengths:

1 Eighth Triplet (Triplet 64th Note)	12 Eighth Note	48 Half Note
2 Quarter Triplet (Triplet 32nd Note)	16 Quarter Note Triplet	64 Whole Note Triplet
3 Thirty-Second Note	18 Dotted Eighth Note	72 Dotted Half Note
4 Half Triplet (Triplet 16th Note)	24 Quarter Note	96 Whole Note
6 Sixteenth Note	32 Half Note Triplet	144 Dotted Whole Note
8 Triplet	36 Dotted Quarter Note	192 Double Whole Note

In addition to sequencing modulation signals, Seq can also be used with **Shift** to sequence note values. To do this, attach *Mod* to Shift's *Shift* port, and set Shift's *Bound* to 1/3 and its *Type* to *Pitch*. Feed the sound into Shift's *Input*, and the result in Shift's *Out*. In this arrangement, each of Seq's modulation dials ranges from two octaves below to two octaves above. To make things easier, if you double-click on a modulation dial, you'll get a pop-up menu with a keyboard, from two octaves below to two octaves above. "Middle C" is 0.5 (the keyboard is keyed in C, but it'll be relative to whatever the current pitch is). See also **Shift**.

At the bottom are outputs which trigger when the step sequencer reaches any single step.

Soften A modulation signal smoother.²¹ Takes an incoming modulation signal and a degree with which to smooth it. You can also specify whether or not Soften is free-running or always resets on gate. Because softening will often significantly reduce the amplitude of a signal, Soften has an additional modulation, *Scale*, which lets you scale it back up.

User A tool for user control over modulation signals, mostly for testing. User contains four modulation inputs, each of which is directly connected to a modulation output. Thus if you connect the modulation output to multiple items, you can change all of them by changing the single input knob. Additionally there are four trigger buttons: pressing a button will issue a trigger to the associated modulation output.

3.3 Partial Sources

AKWF A collection of harmonics drawn from the Adventure Kid Waveform single-cycle wave collection.²² Waves are grouped into categories: pick a category first, then dial in the wave. You can *mix* two different sources and have the additional option of *Normalizing* the partials. See also **Waves** and **KHarmonics**.

All Outputs all harmonics, standardized, at full amplitude. Useful for later shaping or for testing.

Audio In This takes snippets of sound from the Input Device (which you specify from *MIDI and Audio options* under the *Options* menu), extracts their harmonics in real time, and organizes them as harmonics for the current note. Because the partials are relatively few and are organized as harmonics, this has the effect of making the input sound like a robot. This is exactly the same process used to load samples into the *Wavetable* module. The sampling is designed to remove some of the choppiness from reading an arbitrary sound, but it won't remove all of it. You can eliminate most of the rest by running the partials through a *Smooth* module set to about 0.5. Also beware of audio feedback during testing! You might be able to tamp some of that down with a filter. See also **Wavetable**.

Constraints A module which makes it easy to create more complex constraints to feed into the constraints facility of other modules (Section 2.7). You set up three constraints definitions and lump their partials together (perform a union), then whittle down the results to only those partials which are also in a fourth constraints definitions. The resulting output can be fed into the *Only* jack of a module's constraints facility.

Draw Lets you draw partials, which are then output. You can constrain the partials being drawn to a variety of subsets. When you click on the *Menu* button, you can also save your scribbles and load them again, clear them, maximize, normalize, and standardize them, and (importantly) take a snapshot of partials from an incoming source. The module can be collapsed to a smaller size if you wish.

Draw also allows you to load *single-cycle waveforms*, that is, sound files which start and end at zero and represent a single cycle of a sound wave. You can do this by choosing *Load Wave...* from the *Menu* button. These must be WAV files, 16-bit signed integer monophonic, and cannot be very long (presently at most 2048 samples). After loading, Flow converts them (naturally) into harmonics.

²¹So why isn't this called Smooth? Because there's another module called Smooth: see later in Section 3.4

²²The AKWF collection is public domain at <https://www.adventurekid.se>, but the author asks that you consider making a small Paypal donation to kristoffer.ekstrand@gmail.com.

See also **Waves**, which contains harmonics of single-cycle waves from certain well-known synthesizers, as well as **AKWF**. See also **KHarmonics**, a similar collection drawn from various Kawai K5 harmonics.

Drawbars Produces the sine wave equivalents of the drawbars of a tone wheel organ. You can also set the percussion type, decay, and volume. There are three different tuning options: the “classic” tuning found on drawbar organs, the equal temperament tuning which the classic tuning tries to approximate, and the true harmonic ratio tuning (integer harmonics).²³ Finally, there are a number of preset options for the drawbars.

Harmonic Lab Allows you to describe a sound as a combination of three mathematical functions operating on your harmonics. For each of the three, you can define the function type, a function parameter, the gain of the function, and the harmonics the function manipulates. When multiple functions manipulate the same harmonic, the earlier function takes precedence. **Important Note 1** Unlike standard constraints menus, PartialLab’s constraints menus include “All” at the end. **Important Note 2** This is a costly module if you are modulating any of these values in real time, perhaps by an LFO (there are many calls to x^y): you can do it, but it’ll keep your laptop warm.

Harmonics Outputs the first 32 harmonics, whose amplitude (but not frequency) you can define independently. You optionally can load these harmonics from a source, compress them (moving zero-amplitude partials to the end), and mutate them. See also **Partials**.

KHarmonics Produces a wide range of partials drawn from patches from the Kawai K5 synthesizer, including both factory patches (with Kawai’s permission) and various patches in the public domain. You can *Mix* two different sources and have the additional option of *Normalizing* the partials. Note that among the harmonics are ones purporting to be drawn from the Kawai K3. These are just wrong. However the **Waves** module has the correct ones.

See also **Draw**, which allows you to load single-cycle waves: the manual section on Draw has suggestions for sources of these waves. See also **Waves**, a similar collection drawn from various other synthesizers, as well as **AKWF**.

Noise Random noise, hiss, etc., as best as an additive synthesizer can do it. This is done by creating and immediately destroying random partials independent of the pitch. You can specify the number of partials being used (using fewer partials is less computationally costly and you can get away with surprisingly few to make a decent hiss). To change the color of the partials: you can also specify the high and low frequency bounds for the partials; you can create a ramp in amplitude from the low to high partials; and you can specify the degree of variance in amplitude among the random partials. You can also set the average volume (the gain) of the hiss. Finally, if the *Seed* is *Free* (0), then the noise will be completely random and vary from note to note; if it’s non-zero, then the randomness will be reset from on Note on and Reset, meaning that the pattern of randomness will be the same from note to note: the seed specifies which pattern it is.

This module is designed to merge with other sources to add hiss to them: most commonly, you would use it as the primary source in the **Fill** module. To make this convenient, by default the partials used are the highest numbered ones: the lower ones are all set to amplitude 0 and frequency 0. If you use this in combination with a merging option like **Combine** which steals from the high partials, or other adjusters like **Fatten**, you’d want the low partials to be used. By setting *Top* to false, you can cause the low partials to be used for the hiss, with the remainder set to amplitude 0 and a frequency beyond the high frequency bound.

You will find that very short bursts of Noise vary in amplitude, which makes it annoying to use for (for example) percussion. This might be a bug; but I think it’s more likely due to the random number generator’s interaction with Flow’s output thread, which only updates partials every 32 or so samples.

²³See <https://electricdruid.net/technical-aspects-of-the-hammond-organ/> for more information on these tunings.

Partials Outputs 16 partials whose amplitude and frequency you can define independently. You optionally can load these partials from a source, compress them (moving zero-amplitude partials to the end), and mutate them. See also **Harmonics**.

Rectified The harmonics of a rectified sine wave, that is, $|\sin(x)|$.

Sawtooth The harmonics of a sawtooth wave.

Square The harmonics of a pulse wave. You can specify the pulse width modulation.

Sine A single harmonic at 1.0 amplitude. The harmonic can be any partial frequency from 1.0 to 128.0.

Tinkle Produces random partials, creating a tinkling effect. Some *number* of new partials are created whenever the *trigger* occurs, and at the given *volume*. The partials *decay* at some rate. The *probability* specifies the likelihood that partials will be created when a trigger occurs. If the *seed* is *Free* (0), then the tinkles will be completely random and vary from note to note; if it's non-zero, then the pattern of randomness will be the same for each note: the seed specifies which pattern it is. Finally, if *Hold* is true, then the decay of the most recent tinkles won't start until the next tinkles are entered. Furthermore, *Hold* causes *decay* to behave differently when set to 0: normally this would be a very fast decay, but if *Hold* is true, then it's an *immediate* decay — the previous tinkles go straight to zero amplitude. Try turning on *Hold* and setting decay to 0 to get a kind of sample and hold effect.

Tinkle behaves differently than other modules when it comes to constraints. Whereas in most modules the constraints *undo* the action of the module for certain partials, in Tinkle the constraints *bound* its actions. For example, if you had 4 partials tinkling and you restricted the constraints to the low 32 harmonics, then you'd still be guaranteed to get 4 tinkles each time: they'd just be within 0...31. Indeed, setting the constraints to bound to the low N harmonics is a useful way to guarantee that tinkles won't occur beyond Nyquist.

Triangle A triangle wave. Note that normally triangle waves include negative amplitude harmonics, that is, ones with inverted phases. As Flow does not shift partial phases, these harmonics become all positive (you likely won't hear a difference).

Waves A collection of harmonics drawn from the single-cycle waves of several major synthesizers:

- **Kawai K3** This is the full complement of Kawai K3 waves, except for waves 31 (user-defined additive harmonics) and 32 (white noise).²⁴
- **Ensoniq SQ-80** This collection is incomplete: my original source had some errors.²⁵ To combat antialiasing, the SQ-80 had different oversampled versions of the same wave depending on the octave of the note played on the keyboard. The harmonics here are taken from the lowest octaves (with the most harmonics).
- **Sequential Prophet VS** The Prophet VS traditionally had waves 32–127, and these more or less line up with the harmonics provided by Flow: but you'll notice that Flow is missing a few because my original source is missing them.²⁶ Note that some additional waves are labelled *VS Extra*: these were a few additional waves in the original sources originally with numbers outside the VS range. I asked John Bowen about them, and he thinks that they might be a few additional waves that were on the

²⁴The K3 harmonics are from http://www.deepsonic.ch/deep/audio.samples/kawai_k3-_samples._33_waveforms.mp3 by deep!sonic (<http://www.deepsonic.ch>)

²⁵The SQ-80 harmonics are drawn from the waves in http://www.buchty.net/ensoniq/files/the_waves.zip by //christian.

²⁶The Prophet VS harmonics are drawn from <https://www.dropbox.com/sh/ja0tc8wn6cnudzj/AACYqe2mjrWU2CeXIitm6MhOa> as discussed in the forum thread <http://www.muffwiggler.com/forum/viewtopic.php?t=33918&sid=9e0b9554f0ac483b62a88c074c6136c3>

development hard drive at Sequential but which didn't find their way into the VS proper. So I'm including them.

- **Casio CZ Series** These are many of the CZ's standard waves. Casio's waves are formed by concatenating two simpler wave shapes and then applying a window function. The wave names in the menu are of the form "*CZ Number Wave1 Wave2 WindowFunction*". The basic wave shapes are Sawtooth (Saw), Square (Sqr), Pulse (Puls), Null (---), Sine-Pulse (Sinp), Saw-Pulse (Sawp), Multi-Sine (Msin), and Pulse 2 (Pul2). The window shapes are None (---), Sawtooth (Saw), Triangle (Tri), Trapezoid (Trap), Pulse (Puls), and Double-Saw (2Saw). You'll note that not all combinations are here: effective duplicates have been removed.²⁷ Some wave names start with "CZ#" rather than "CZ". This denotes that a wave is one of the classic original waves directly accessible from the CZ front panel (many CZ waves could only be generated via sysex).

Obviously morphing from a Sine wave to one of these waves in Flow won't be anything like applying Phase Distortion on a CZ: but there you go.

See also **Draw**, which can load single-cycle waves: the manual section on Draw has suggestions for other single-cycle wave sources. See also **KHarmonics**, a similar collection drawn from Kawai K5 harmonics, as well as **AKWF**.

Wavetable Lets you load wavetables from WaveEditOnline (<https://waveeditonline.com/>), or Waldorf Blofeld wavetable sysex files, and then wander through them using the provided modulation. You can turn off interpolation between waves. Wavetables are loaded (as ".WAV" files for WaveEditOnline, or as ".SYX" files in the Blofeld case) using the *Wavetable* button. If you're interested, you can edit wavetables directly using WaveEdit (<https://synthtech.com/waveedit>).

You can also load an arbitrary sound directly and Flow will chop it up into a wavetable for you by pressing the *Sample* button. Your sound file must be a 16-bit signed integer WAV file, mono only. The Sample button is different from the Wavetable button in that it is designed to remove some of the choppiness from reading an arbitrary sound: but it won't remove all of it. You can eliminate most of the rest by running the partials through a Smooth module set to about 0.5. Note: this is only for sampling arbitrary sounds: you don't want to do any of this stuff (the sample option, Smooth) if you're reading a prepared wavetable. The sampled option uses the same basic procedure as Audio In uses to sample its real-time input. See also **Audio In**.

3.4 Partial Shapers

All of the shapers have the ability of being **constrained**. This means that you can tell the shaper to *only* perform its operation on certain partials; the others should be left as they are.²⁸ The constraint facility (Figure 6) is at the bottom of the shapers and consists of three widgets. First, there is a pop-up menu (combobox) where you can specify common constraints. Above it is a checkbox labelled **Not**, where you can optionally flip things: that is, specify that you want to perform the operation on everything *but* the partials in question. Finally, there is a *constraint partials source*, labelled **Only**. If you attach a unit's output to this, then its incoming partials indicate what the constraint should be: specifically, the partials that will be constrained will be those whose positions match up with non-zero amplitude partials coming into the constraint partials source. For example if the constraint partials source has non-zero amplitudes for partials 4, 7, and 9, then this tells the shaper to constrain its outputted partials 4, 7, and 9. The constraint partials source overrides the pop-up menu. Okay, here we go.

²⁷See <https://www.kasploosh.com/cz/11800-spelunking/> for an extended discussion of these waves, including pictures. The original waves are from the project *Casio CZ-1 Spelunking REMIXED* by John Thompson (<https://www.facebook.com/Hardwood01>) and Michael Rickard (<https://www.kasploosh.com>). Many thanks to them for giving me permission to extract the harmonics from these waves. If you'd like to get the wave sources, just buy them for \$10 from the project home page at https://www.kasploosh.com/cz/16293-cz_waves_for_your_mix/

²⁸If a shaper has multiple partials sources, then "left as they are" means that they should be reverted to whatever values they had in the first partials source. For example, in Amp Math the first partials source is A.

Amp Math This shaper modifies the partials from two sources A and B by putting every pair of them $\langle A_i, B_i \rangle$ through the same function, along with a modulation source M . The operations are:

- **+** Does $A_i + B_i + M$
- **-** Does $\max(A_i - B_i - M, 0)$
- **\times** Does $A_i \times B_i$
- **inv \times** Does $A_i \times (1 - B_i)$
- **compress** Does $1 - (1 - A_i) \times (1 - B_i)$
- **average** Does $M \times A_i + (1 - M) \times B_i$
- **min** Does $\min(A_i \times (1 - M), B_i)$
- **max** Does $\max(A_i \times (1 - M), B_i)$
- **filter** If $B_i < M$ then returns 0, else A_i
- **filter not** If $B_i \geq M$ then returns 0, else A_i
- **fill** If $A_i > M$ then returns A_i , else B_i
- **threshold** If $A_i > M$ then returns 1 else 0
- **scaledown** Does $A_i \times M + B_i$
(Scales in the range 0...1)
- **scaleup** Does $A_i \times (M + 1) + B_i$
(Scales in the range 1...2)
- **scalefar** Does $A_i \times (M + 9) + B_i$
(Scales in the range 1...10)
- **clampdown** Does $\min(A_i, M) + B_i$
- **clampup** Does $\max(A_i, M) + B_i$

You have the option of normalizing the resulting partials. Be sure to read “Strategies for Merging Partial Sources”, Section 3.6. See also **Combine** and **Fill** and **Mix** and **Dissolve** and **Morph**.

One common task is to set the amplitudes of a source to 0 if they’re below a certain limit. To do this, just set to *filter*, plug your source into *both* A and B , and use M as your limit.

Buffer This is a sort of Sample and Hold for partials. Every time Buffer receives an incoming trigger, it takes a snapshot of its input and uses that snapshot for its output. Buffer can be constrained, which means that it outputs its snapshot for certain partials, while outputting the current source partials for the remainder.

Chord Reassigns high frequency partials in order to form an interval or chord. The volume of additional chord notes (beyond the original root) is controlled by *Gain*. You can *Align* the fifth to be exactly a harmonic (normally it’s slightly off of a harmonic, which can produce beating). Your *Chord* options are:

- **None** Do nothing (just pass the sound through)
- **m2** Minor Second Interval
- **M2** Major Second Interval
- **m3** Minor Third Interval
- **M3** Major Third Interval
- **4** Perfect Fourth Interval
- **TT** Tri Tone Interval
- **5** Perfect Fifth Interval
- **m6** Minor Sixth Interval
- **M6** Major Sixth Interval
- **m7** Minor Seventh Interval
- **M7** Major Seventh Interval
- **Oct** Octave Interval
- **Oct+m3** Minor Third Interval above an Octave
- **Oct+M3** Major Third Interval above an Octave
- **Oct+5** Perfect Fifth Interval above an Octave
- **2 Oct** 2 Octave Interval
- **min** Minor Triad
- **min-1** Minor Triad (First Inversion)
- **min-2** Minor Triad (Second Inversion)
- **Maj** Major Triad
- **Maj-1** Major Triad (First Inversion)
- **Maj-2** Major Triad (Second Inversion)
- **7** Dominant 7
- **min7** Minor 7
- **Maj7** Major 7
- **dim7** Diminished 7
- **min+Oct** Minor Triad plus Octave
- **Maj+Oct** Major Triad plus Octave

Combine The partials from up to four sources are merged by throwing together all their lowest partials. This works as follows. We first divide the number of partials evenly among the sources, then take that number of the lowest partials from each source. The remaining (higher frequency) partials are discarded. We

then toss together all those lower partials to form the output. You also can scale (down) the amplitude of each of the incoming sources. Be sure to read “Strategies for Merging Partial Sources”, Section 3.6, particularly if you’re hearing pops. See also **Amp Math** and **Fill** and **Mix** and **Morph** and **Dissolve** and especially **Joystick**.

Compress The amplitude of every partial is raised to a certain power, controlled by the *Scale* modulation. If *Scale* is greater than 0.5 (the mid-point), then the amplitude is raised and compressed more and more, resulting in them getting louder and with less difference among them. If *Scale* is less than 0.5, then the opposite happens: the amplitudes are *expanded*, resulting in a larger difference among them, and with them all getting quieter.

Delay Partials are added to an initial one-shot delay, then a repeated delay. Each delay has a length (the “delay”) and a cut (how much to cut down the delayed sound). You can also specify the wet/dry ratio.

Dilate If a partial has a neighbor partial which is *larger* than it, then its amplitude is increased towards its neighbor: this change trickles down to its further (smaller) neighbors as well. This has the effect of boosting neighboring partials until ultimately all partials are the same amplitude. *Boost* is the amount to increase. You can specify whether only lower neighbors, higher neighbors, or both are considered. See also **Skeletonize** (which is almost, but not quite, the opposite concept).

Dissolve Given two incoming sources *A* and *B*, this module one by one replaces each of *A*’s partials with the equivalent partial in *B* one at a time. The amount of replacement is specified by *Dissolve*. A partial isn’t suddenly replaced: it’s smoothly morphed with its corresponding partial. If you want an even smoother transition, you might try hooking Dissolve up to **Smooth**. By default the order in which this replacement is done is at random, determined by the *Seed*. If *Seed* is *Free* (0), the replacement order will be randomly different with every new note: otherwise the replacement order will be the same from note to note. Alternatively you can perform a *Wipe*: replacing the partials in order top to bottom or bottom to top. This is done by selecting the appropriate *Type* option. See also **Either/Or** and **Amp Math** and **Combine** and **Fill** and **Morph**.

Fatten The top half partials are discarded and reused to create detuned versions of the lower half partials. The detuning amount is specified as well as the wet/dry ratio.

Fill Given two incoming sources *A* and *B*, this module first adds all of the non-zero-partial of *A* to the output; any remaining partials are filled from non-zero amplitude partials in *B*, starting at its lowest frequency partials and working up. You can scale (down) the amplitude of each of the incoming sources. This module is particularly useful with **Noise**. Be sure to read “Strategies for Merging Partial Sources”, Section 3.6. See also **Amp Math** and **Combine** and **Mix** and **Morph** and **Dissolve**.

Filter This is a resonant state-variable 2- or 4-pole filter. You can sweep through High Pass (at 0.0) to Notch (at 0.25) to Low Pass (the default, at 0.5), to Band Pass (at 0.75) and finally to High Pass again (at 1.0). Resonance values currently range from no resonance (that is, resonance $Q = \frac{1}{\sqrt{2}}$) at 0.0 to ten times that amount ($Q = \frac{10}{\sqrt{2}}$) at 1.0. And you can, of course, set the cutoff frequency.

Filter can be set to be *Relative*. This means that the filter cutoff frequency is set to be relative to Middle C. If you’re playing a higher note, the cutoff frequency will automatically be set higher to match; similarly if you’re playing a lower note, the cutoff frequency will be automatically lower.

Filter emulates an analog filter: its cut-down function goes out to infinity. This means that at the Nyquist limit, the partials can still have a significant amplitude, particularly for high notes. You may not want this, and instead prefer to gradually *Taper* the curve, starting at the cutoff frequency and dropping to zero by the time you reach Nyquist. This probably only makes sense for low-pass filters.

See also **Ladder Filter**.

Flange Filter This filter simulates various flanging, phasing, and chorusing effects via rough approximations of comb filters. You can select what kind of comb filter shape you'd like ("spike down" is the classic for flanging and chorusing etc.). You can also stipulate whether the filter shifts to follow the pitch of your note or is fixed in the frequency location of its lobes. The flange filter has several confusing dials. *OffsetMod* lets you modulate the lobe positions. *Stretch* lets you specify the size of the lobes. *StretchMod* is similar but meant to have a modulation source attach to it. You can also specify the wet/dry ratio. **Hint:** the *degree* of *OffsetMod* or *StretchMod* is very high, but you can reduce this by changing your LFO's Scale value. The LFO's shift is also interesting to play with.

Formant Filter This filter simulates human vowels using up to five formants. You can specify up to eight vowels and interpolate from the first to the last. Interpolation is not done by crossfading between different filters but rather by directly shifting the formant frequencies, bandwidths, and peak amplitudes themselves, thus morphing from one vowel to the next. *Gain* lets you boost the output volume of the filter. Formants are created using a resonant bandpass filter. You can change how peaky and narrow each formant in the vowel is using *Resonance*. By default the filter is two-pole, but you can also set it to *Four Pole*.

Some usage hints. I think a mixture of Square (at 75% pulse width) and Triangle sounds good as an input to the filter. *Rectified* works well too. If you'd like to control how long interpolation goes from one vowel to the next, you might try using a multi-stage **Envelope** slowly rising from 0 to 1 though stages of various length. Also if you'd like to *rotate* through a set of vowels, try this. Let's say your vowels are A, I, and O, and you want to rotate A I O A I O A I O etc. Set up four vowels A, I, O, A. Then attach an LFO with a sawtooth up to the interpolation.

Jitter You can add jitter to either the frequencies or the amplitudes (or both) of your partials. The amount of noise is controlled by *Freq Var* and *Amp Var* respectively. The *Trigger* informs the module to add noise to its current jitter configuration. You can also specify that Jitter should *only* apply to non-zero partials this is useful when using Jitter along with Fill. Additionally, with *Amp Ratio* you can specify that Jitter should add noise *proportionally* to the amplitude, or instead just directly insert noise. Finally, if the *Seed* is *Free* (0), then the jitter will be completely random and vary from note to note; if it's non-zero, then the pattern of randomness will be the same for each note: the seed specifies which pattern it is.

Ladder Filter This is a resonant state-variable 3-, 4-, 6-, or 8-pole low pass ladder filter, plus a non-resonant 1-pole low pass filter (from which ladder filters are built). Resonance values currently range from no resonance at 0.0 to a near-maximum possible value at 1.0. And you can, of course, set the cutoff frequency.

Filter can be set to be *Relative*. This means that the filter cutoff frequency is set to be relative to Middle C. If you're playing a higher note, the cutoff frequency will automatically be set higher to match; similarly if you're playing a lower note, the cutoff frequency will be automatically lower.

Filter emulates an analog filter: its dropoff stretches out to infinity. This means that at the Nyquist limit, the partials can still have a significant amplitude, particularly for high notes. You may not want this, and instead prefer to gradually *Taper* the curve, starting at the cutoff frequency and dropping to zero by the time you reach Nyquist.

The bass in Ladder filters tends to bottom out with increasing resonance. You might be happier with the filter used in **Filter** in this case.

Linear Filter This is a filter where you can specify up to 8 nodes (points of change) in terms of frequency and gain. The filter then just draws lines attaching the nodes, and this becomes the filter frequency function. Nodes are always sorted first by frequency. A *Base* frequency is added to the frequencies of all the nodes. *Num Nodes* is the number of nodes being used.

Linear Filter can be *Relative*. This means that the filter cutoff frequency is set to be relative to Middle C. If you're playing a higher note, the cutoff frequency will automatically be set higher to match; similarly if you're playing a lower note, the cutoff frequency will be automatically lower. See also **Partial Filter**.

Mix This is a straightforward mixer: the amplitudes of the partials of four different inputs can be mixed, with their respective gains. The *frequencies* of the partials are set to those of the first input whose amplitudes are non-zero. In many cases, it's likely that all four inputs have partials with the same frequency, so it wouldn't matter. But this strategy is also useful because it allows Mix to be used in combination with **Either/Or** to select a single output among the four mix inputs. Be sure to also read "Strategies for Merging Partial Sources", Section 3.6. See also **Either/Or** and **Amp Math** and **Combine** and **Fill** and **Morph** and **Dissolve** and especially **Joystick**.

Morph This module morphs (interpolates) between two or more inputs. In the most basic configuration, when two signals are morphed together, the amplitudes of their partials are weighted and averaged, and the frequencies of their partials are *also* weighted and averaged. The weighting constant α is controlled by the *Interpolation* knob. For example, consider Inputs X and Y. For each pair of partials $\langle p_i^X, p_i^Y \rangle$ in X and Y respectively, with frequencies $\langle F_i^X, F_i^Y \rangle$ and amplitudes $\langle A_i^X, A_i^Y \rangle$, we produce a final partial P_i whose frequency $F_i = (1 - \alpha)F_i^X + \alpha F_i^Y$ and whose amplitude $A_i = (1 - \alpha)A_i^X + \alpha A_i^Y$.

You have the option of turning on or off morphing of the frequency or amplitude. You can also tell Morph to *not* morph the fundamental.

That's the simple case. But there are several other bells and whistles in Morph. First, you can have more than two inputs. In this case, when Input A has finished morphing into Input B, Morph will then switch to morphing Input B into Input C, then C into D, and then D into A, looping back around again. Morph decides to switch when the direction of the interpolation signal changes. For example, suppose you had a triangle LFO attached to the Interpolation. As the LFO goes up, A is morphed into B. Then as the LFO goes down, B is morphed into C. As the LFO goes up again, C is morphed into D, and as it goes down again, D is morphed to A.

Some inputs change over time — when Morph switches to that input, we'd like a way to inform the input source to reset itself to prepare to be included in the Morph. To do this, Morph sets the trigger for that Input (labelled *Trig*), which you can use as a preparation signal. It also sets the *Trig* output to the current interpolation value for that sound. You can use the triggers to reset a sound as if you had just played Note On by feeding Morph's trigger output into the "On Tr" input of the sound's envelopes or LFOs; or you could just save the sound as a patch, then load it as a Macro and feed the trigger output into the "On Tr" input of the macro.

Normally every Partial i in the first Input is paired up with the same Partial i in the other input to morph to it. But you can change what the pairing is with the *Type* popup. These pairs include:

- **Normal** Just the standard pairing.
- **Random** Partials are paired up at random with "nearby" partials. The definition of how far away nearby could be is specified by the *Variance* knob. If the *Seed* is *Free* (0), then the pairings will be different for every note; if it's non-zero, then the random pairing pattern will always be the same: the seed specifies what pattern it is.
- **2-Pair, 4-Pair, 8-Pair** Partials are paired up in groups with other partials 2, 4, or 8 away from them.
- **Increasing** The first partials are paired. Then the next two partials are paired in 2-pair fashion. Then the next 4 partials are paired in 4-pair fashion, and so on.

If *Free* is set, then Morph doesn't reset to Inputs A and B every time you press a note — it keeps morphing from wherever it left off. Finally, if *Random* is set, then Morph chooses random new inputs each time it transitions, rather than rotating through them in turn. The random seed is (you guessed it) *Seed*, reset every note-on, except if *Seed* is 0, in which case it's fully random.

Be sure to read "Strategies for Merging Partial Sources", Section 3.6. See also **Switch**, **Amp Math**, **Combine**, **Fill**, **Dissolve**, and **Mix**.

Normalize This *normalizes* the input (scales it so that the amplitudes all sum to 1.0, creating a relatively consistent volume) or *maximizes* it (scaling it so that the highest amplitude equals 1.0). Optionally it also *standardizes* it (sets all the frequencies to standard harmonic values).

Partial Filter This is a filter based on the partials of an incoming signal. The partials form nodes in the filter, and the filter function interpolates between them. Try drawing partials with **Draw** and feeding them into Partial Filter to filter some sound to get the idea.

Partial Filter can be set to be *Relative*. When relative, the partials are simply filtering their equivalent frequencies in the incoming sound. When *not* relative, then the partial frequencies are fixed as follows. A partial at position 1.0 (where the fundamental normally is), or indeed ≤ 1.0 , would represent 0Hz, a partial at 2.0 would represent 100Hz, a partial at 3.0 would represent 200Hz, and so on. Typical standardized partials (going from 1 to 128) would thus cover 0Hz to 12.7KHz. That may not be far enough for you; but you can easily *double* this scale, so 1.0 is 0Hz, 2.0 is 200Hz, etc., clear up to 128.0 being 25.4KHz.²⁹ See also **Linear Filter**.

Rotate This takes all the partials within some frequency range from *A* to *B* certain range and shifts all their frequencies by some value *X* (where $A \leq X \leq B$). Partial shifted to beyond *B* are wrapped around to be back in-range, so this has the effect of rotating all the partials. Rotation is usually modulated using an LFO set to a sawtooth or inverse sawtooth.

There are a number of options. First, you can *stretch* the partials to a non-linear scaling, such as x^2 or x^8 . Second, you can *window* the amplitudes so that the partials at either end are scaled down in volume. This windowing can have its center either in the middle of the range (if *center* is true) or can be stretched to match the stretched partials. You have the option of *soloing* the rotation: muting all the partials other than those involved in rotation. Finally, you can *thin* out some number of partials; this often creates a better sounding tone for rotation.

Scale This scales the frequency of all the partials but keeps the pitch the same: it “stretches” the partials out away from frequency 1.0, or compacts them towards it. A *Scale* value of 0.5 is no scaling; larger values stretch the partials, and smaller values compact them. See also **Pitch Shift** and **Partials Shift**.

Shift This shifts all the partials in your input in a certain way. *Shift* specifies by how much (this is easily modulated), and *Bound* scales the shifting. There are three ways you can shift. First you can shift the *Pitch*: this is essentially multiplying the partials by a certain amount. Second, you can shift the *Frequency*: this *adds* a certain amount to the partials, which creates an inharmonic shift effect. Third, you can shift the *Partials*: this shifts the whole partial amplitude array up or down. A warning about partials shifting: this can make abrupt changes in amplitude, and you may hear static-like clicks or pops, especially in low pitches. You can clean this up by running things through **Smooth** set to about 0.9. See also **Seq**. Finally, *Vibrato* is the same as *Pitch*, but with a much smaller bound: at maximum, the bound is one whole step (major second) in each direction. This makes it easier to dial in precise vibrato amounts.

Skeletonize If a partial has a neighbor partial which is *larger* than it, then its amplitude is cut by a percentage. The percentage is a power of the distance of the partial to the peak partial on the left or right, whichever is further. This has the effect of stripping partials out and only leaving the locally highest partials. *Cut* is the base percentage. You can specify whether only lower neighbors, higher neighbors, or both are considered. See also **Dilate**, which is almost, but not quite, the opposite concept.³⁰

²⁹You could have stretched this with **Scale** or **Shift** too of course.

³⁰Odd names? Skeletonize and Dilate are both the names of similar operations on 2D images in computer graphics.

Smooth This module smooths over the frequency and amplitude differences between successive sets of partials. The degree of smoothing is specified by *Amount*. Specifically, it maintains an internal set of Partial *P*. At every timestep, it takes the current incoming Partial *Q*, and sets each amplitude A_i^P of *P* to $A_i^P = (1 - \alpha)A_i^P + \alpha A_i^Q$. Likewise it sets each frequency $F_i^P = (1 - \alpha)F_i^P + \alpha F_i^Q$, where α is the Amount. Smooth has to deal with one unusual case: when the incoming partials of *Q* start wandering about in frequency and eventually cross one another. This creates discontinuities in the smoothing. Smooth handles this by trying to remember which partials were which; this strategy will work fine if Smooth is fed partials from the *first* partials output of any module. No guarantees are made regarding other partials outputs. Modules rarely have more than one output, anyway.

Sub Adds up to four partials at octave spacing below the fundamental, eliminating the top four partials. You can specify the amplitude of the partials relative to the fundamental.

Switch This module switches among eight different inputs. You select an input by sending a trigger to its corresponding dial. Switch will then interpolate the amplitudes and frequencies between the old input and the new one until the new one has completely taken over: the rate of interpolation is given by *Rate*. The purpose of non-zero Rate values is to prevent popping from sudden changes.

Initially, Switch starts at Input A. If *Free* is set, then Switch doesn't reset to Input A every time you press a note — it keeps morphing from wherever it left off. You can also send a trigger to *Next* instead of to one of the inputs, and it will automatically select the next input.

Switch is useful for playing sequences of sounds: you can do this by attaching an envelope of some sort to it, and routing each of its individual stage triggers into Switch's trigger inputs (or routing its modulation output to *Next*). When Switch switches to a new sound in the sequence, you probably will want to reset the sound as if it had been just played with Note On. This is most easily done by also feeding the modulation to the "On Tr" input of the sound's envelopes or LFOs; or you could just save the sound as a patch, then load it as a Macro and feed the output into the "On Tr" input of the macro. See also **Morph**.

VCA This is just what it says on the tin: an amplifier (VCA is the classic term). A VCA is very commonly used in conjunction with an envelope. It takes an input and multiplies all of its amplitudes by a value. This value is determined by multiplying *mod* (which goes 0–1) by *scale* (which goes 0–8), and then adding *add* (which goes 0–8). The *mod* modulation is designed for you to attach LFOs or envelopes etc. The *scale* modulation is not linear and so is just meant for you to dial in a maximum scale value.

The *add* modulation, normally zero, shifts the overall amplification up or down, and is useful in creating tremolo (slight variation in volume), because without it the VCA would just range between a given volume and *silence*. For example, normally if your scaling was 1.0, then the volume would go between silence (0.0) and normal volume (1.0). If you instead set *add* to 1.0, then the volume would go between 0.5 and 1.5.

3.5 Special Modules

There are currently six special modules: **In**, **Out**, **Choice**, **Fix**, **Note**, **Patch**, and **Macro**. The **Out** module is critical in every patch, and is discussed below in Section 3.5.1. **In** and **Macro** enable *macros*— and **Out** helps out here as well — as discussed below in Section 3.5.2. Here are the others:

Choice The Choice module is an odd duck: it doesn't produce any sounds or modulation at all. Instead, it enables you to change a module's checkboxes and comboboxes using a modulation signal. To do this, note that Choice has a modulation input and a sound input port both called *Target*. Plug one of your target module's modulation or sound outputs into the appropriate Target port: this tells Choice what target module it should be modifying. Don't attach to both of them, or Choice will just use the sound input port.

After this is done, you will find you can select an option (a combobox or a checkbox) via the *Option* dial.³¹ Once this is done, you can change the value of the selected option via the *Value* dial. This might be particularly interesting to do via an LFO.

Note that when you change the value, it's not immediately reflected (or reflected at all!) in the target module's GUI. This is because updating the GUI in real time like this would be very computationally costly. But it's been changed! You can convince yourself of this by dragging the module and moving it. Similarly, if you click on the option combo box or check box, the result is not directly reflected in the Option dial.

Note This module also doesn't process sound at all. Instead, it lets you leave a note or comment. You can change the width of the Note panel with two arrow-like buttons at the bottom. You can have as many Note modules as you like; they'll each save their comments independently as part of your patch. If your goal is to document your patch for later users, Note will work great, but you ought to first edit **Out**'s *Info* field (and fill out its other fields as well). See also **Out**.

Fix This module likewise doesn't process any sound. Instead, it lets you fix the note (pitch) being played, and/or the velocity being played, to specific values. After Fix is called, it changes the pitch and volume, and all later modules will use these values: thus you want Fix early in your modules, ideally as the very first one. I use Fix in percussion patches to force the patches to always make the same pitched sound regardless of the note being played.

Patch This module is simply a patchbay: each input is routed directly to its corresponding output with no changes and that's it. Patch is useful in a few contexts. For one, you can use Patch to break up long connections in your patch that are hard to connect or understand given scroll restrictions. You could also use Patch to easily swap a module routed to many others. To do this, hook the first module to a Patch input, and all the downstream modules to the corresponding Patch output: now you can change the first module without reattaching everything.

3.5.1 The Out Module in Basic Use

Every useful patch has to have an Out module, since it provides the facility for outputting your final partials to the audio system. Out does several important basic things:



Audio Generation Out is responsible for converting your partials into sound. To do this, connect your final partials to the port on Out normally labelled *A*: it's the topmost port in the module. You can adjust the volume of this port by changing the *Gain* knob. Similarly you can adjust the pan via the *Pan* knob.³²

Reverb Flow sports a simple implementation of the Freeverb algorithm.³³ This algorithm has three parameters. *Wet* is the dry/wet knob for the algorithm (0 is fully dry). *Damp* is the damping: basically a low-pass filter applied to the reflected signals. *Size* is the room size, which translates roughly into the reverb decay. Note only a single sound (voice #0) controls the modulation of Wet, Damp, and Size. This means that if you attach an Envelope etc. to any of these modules, it may not respond in exactly the way you think. It's probably best to keep them as knobs rather than attaching a modulation source.

³¹At present, the constraints facility is not part of the available options. This might change in the future.

³²Note that changing Pan manually will cause clicks due to the sudden change in relative volume of the two channels. Attaching a square, sample and hold, or sawtooth LFO will cause pops for the same reason: while sine and triangle and random will sound fine. If you want a square, sample and hold, sawtooth to control Pan, consider Softening them very slightly first.

³³A well regarded reverb algorithm in the public domain by "Jezar at Dreampoint".

Dephase By default Flow tries to keep all of its partials consistent with their original phases, so when you output a Sawtooth it'll look like this:  However this may sound too buzzy in low pitch sounds, so selecting this button will cause Flow to add a specific fixed set of random values to the phases of its partials. This might change the Sawtooth wave to look like this  This sure doesn't look like a Sawtooth wave, but if you play it in the high- and mid- pitch notes, you'll find it *sounds* exactly the same! For low-frequency notes the randomly-phased Sawtooth will sound less buzzy; you may or may not prefer this. It is the case that Dephase gives Flow the opportunity to do some tricks which reduce computational cost by a bit.

Partials Visualization When you connect to the *A* port, the resulting partials are also displayed on the left partials display in your rack. The *B* (Aux) port is also useful for displaying partials (it shows them on the right partials display).

The displays have certain special colors you should be aware of: the *Green* line is where your frequency 1.0 partial is located (if you have one!). The *Blue* line, if displayed, shows the Nyquist limit: partials beyond this line (in *Gray*) are too high frequency to be represented in the digital sound signal and will be discarded. Partial with zero amplitude are displayed in *Red*. Similarly, partials with more than 1.0 amplitude are also displayed in *Red*.

Modulation Visualization The Out module also lets you display modulation signals: the first two dials, normally labelled 1 and 2, correspond to the left and right oscilloscope displays respectively.

In addition to the modulation signal, the oscilloscope also draws a helpful set of axes. The color of these axes can be either *Blue* or *Green*: every time the signal produces a *trigger*, the oscilloscope will change the color of the axes.

There's no reason you can't insert multiple Out modules into your patch: but only the last one will do anything.

Documentation If you press the *Edit* button, you can edit your patch's *Author Name*, *Creation Date*, *Version*, and *Info* (an area for general documentation). If you load a patch as a Macro, you can view its Info string in the resulting module, so Info is useful to fill out with instructions for use.

3.5.2 Creating a Macro with the *In*, *Out*, and *Macro* Modules

Once you have saved your patch, you can reload it as a **Macro** in a higher-level patch of your design: it will appear as a module like any other. The partials fed into the (*A*) port of your Out module, normally used to produce audio, will be available to feed into other modules. When used in a macro, Out's Reverb and Gain are disabled. The underlying patch's information (author, etc.) are available by clicking on a disclosure triangle labelled *Info*.

But that's not all. Your patch can in fact output *four* partials streams, by default called *A*, *B*, *C*, and *D*. And you can also output *four* modulation signals, called 1, 2, 3, and 4. Just connect to them in the Out module of your original patch, and when you load it as a Macro, they're all available as outputs in the Macro module to feed into other modules.

Input to a Macro: the *In* Module So far so good. But how about feeding partials and modulation signals *into* your Macro? This is accomplished with the **In** module. The In module has eight partials outputs and eight modulation outputs, conveniently called *A ... H* and *1 ... 8*. If, before you save your patch, you wire up some modules to listen in on these ports, then when you load the patch as a Macro, you can attach modules to feed into the Macro at those ports and their partials and modulation signals will be handed the modules inside the Macro. Unlike Out, you can have as many In modules as you like: all of them will happily provide (the same) signals.

When a Macro has Inputs, it will also sport Constraints just like other partials shapers.

Modulation Defaults The In module also contains some modulation inputs (dials) with the same name as its modulation outputs. The primary purpose of these inputs is to allow you to set default values for the Macro. For example, if you set the first dial to 0.25, then the Macro will use 0.25 by default. If while editing the patch you change these dials, then the In module will also temporarily send the same value out its equivalent modulation output. This makes it a bit easier to figure out what values to set by default. However this also means that you can also hook modulation signals to these inputs and the same signal will be output: but when you load the patch as a macro, these modulation signals will be ignored and the default will be set to 0.0. In short: you should only set the dials manually, not via modulation signal: it'd be confusing otherwise.

Naming Things You can rename the input and output partials and modulation ports of your Macro. This is accomplished by renaming their compatriots in your In and Out modules. Note that, unlike in other modules, the ports in your In and Out modules have *underlined labels*. Just click on a label and you'll be asked to give it a new name.

Note that although you can have as many In modules as you like, and they will all work, only the ports of the first one will be used to rename ports in the Macro. (Of course, as mentioned before, while you can have multiple Out modules, only the last one will function at all, and this includes naming ports).

3.5.3 Using a Macro

You can load a Macro using *Load Patch as Macro...* in the *Other* menu. A Macro will appear with exactly the modulation inputs and outputs specified by your In and Out modules; and you can expand the *Info...* disclosure button to show the original Patch information (version, author, date, info).

A Macro also contains an *On Tr* and an *Off Tr* modulation dial. If you send a trigger to *On Tr*, then a Gate (Note On) message will be sent to all the underlying modules in the Macro. This results in the Macro resetting itself as if a new note had been played. This is particularly useful in combination with **Switch** or **Morph**: see those modules for further information. Likewise, sending a trigger to *Off Tr* will send a Release (Note Off) message to the underlying modules.

Macros have one more trick up their sleeves. The *Pause* button changes the behavior of the *On Tr* modulation dial. If you turn on *Pause*, then while *On Tr* is set to zero (0.0), the underlying modules in the Macro will not be stepped at all: they're effectively put to sleep. This can be used in combination with **Switch** or **Morph** again to entirely shut off Macro sounds while they're not being used, saving a bit of computational power. This is because the modulation outputs of **Switch** and **Morph** both go to zero when they are not using the corresponding sound. With great power comes great responsibility: be careful that you know what you're doing here.

3.6 Strategies for Merging Partial Sources

Ordinarily if you wanted to mix two sources into one stream, you'd imagine you could just lump all the partials together and be done with it. But there are two catches. First, Flow has a fixed number of partials: so if you lump the partials from two sources together, half of them must be eliminated. Second, partials are each associated with a sine wave oscillator in the output (that's the partial's *order*): when two sets of partials are mixed, you'll often have the situation where partials from each set are competing for the same oscillator, and in the end only one can have it. If this results in a sudden shift of control, the oscillator's frequency or amplitude may suddenly shift, producing a pop. So all this is a little more complex than you'd think.

Flow has several modules which implement different strategies for mixing, depending on your needs: *Amp Math*, *Morph*, *Mix*, *Combine*, and *Fill*. It's helpful to understand how these strategies differ from one another.

First some terminology. Let the two sets of partials from incoming sounds be *A* and *B*, and the resulting partials outputted by the mixing module be *O*. Sets of partials are actually arrays (lists). Thus each partial has a *position* in the list: Flow requires that lower-frequency partials have lower positions than higher-frequency

partials. Partial O_i is the i th partial in the list for output O (things work similarly for A_i and B_i). Each partial has an *amplitude*, a *frequency*, and an *order* (the oscillator ultimately used to output the partial). We'll call these $\text{amp}(O_i)$, $\text{freq}(O_i)$, and $\text{ord}(O_i)$ (and similarly for A and B). The order is a unique integer ID, and it's not the same thing as the position in the list i : as partials change their frequencies, they can get rearranged in a list but maintain their respective order tags.

- **Amp Math** and **Mix** both just line up the corresponding partials and perform a function on their amplitudes. The frequencies of the first set of partials are used. That is, $\text{amp}(O_i) = f(\text{amp}(A_i), \text{amp}(B_i))$ for some function $f(\dots)$, but $\text{freq}(O_i) = \text{freq}(A_i)$ and $\text{ord}(O_i) = \text{ord}(A_i)$.

This can produce some surprises. Let's say you made a sawtooth wave and detuned it in some way, then mixed it (as source B) with another plain old sawtooth. All of your careful detuning will disappear! This is because the frequencies of the outgoing partials are set to those of A : in essence the partials of B will have their frequencies lined back up with A before the mix occurs.

- **Combine** takes about the same number of lowest frequency partials from each source and puts them together to form the output. Combine is good for many things, but not if you want to mix one source with lots of partials with another source with only one or two partials: you might use **Fill** instead.
- **Fill** adds to O all the non-zero amplitude partials of A . Any remaining partials come from the lowest non-zero partials of B . Fill is particularly useful when mixing in Noise from a limited number of partials: set A to the noise. Fill also works great if both sources (or at least A) have lots of empty partials. Note that orders come from A : orders in B are reassigned. This means that if the non-zero-amplitude partials in A change in number or location, this could potentially create a pop or buzz as a partial in O suddenly gets reassigned from A to B or vice versa.
- **Morph** lines up partials in A and B according to some map and interpolates their frequencies and amplitudes. Orders come from A . Because of interpolation, this is unlikely to create pops or buzzes in most situations.
- **Dissolve** one by one replaces the partials in A with their equivalents in B in some specified order. This is also unlikely to create pops or buzzes.

4 Developing Modules

The architecture behind Flow is not particularly complex. The top-level class, of which there is only a single instance, is **Output**. This class is responsible for managing most of the threads in the system and for maintaining the facility which builds and emits samples to Java's audio system. Output also contains a single **Input** object which manages incoming MIDI data and sends it to the right places. Input does this by working with **MIDI**, an untidy class borrowed from Edisyn which provides a wrapper facility around Java's midi subsystem. Last, Output contains a single **AudioInput** class which samples sound from a microphone or other input source. AudioInput is at present only used by the *Audio In* module.

Output also contains some N **Sound** objects, one per voice. Sound objects are *registered* with Output. Output, and not the Sound objects themselves, handles the spawning of voice threads associated with the Sound objects. Each Sound object contains some number of *modules* which are attached to one another and pass partials or modulation signals to one another. One module is designated as the module which *emits* a final set of partials to the Sound on request: Sounds then hand off their partials to the Output which uses them to produce samples to give to the audio system.

Each Sound object belongs to a **Group**, and there are up to G Groups stored in Output. Each Group is really a patch (the Primary patch or a subpatch), and all Sounds associated with a Group have the same organization of module because they produce the same kind of sound. Groups have names, MIDI channels, etc. Each Group also has a user-requested number of voices (Sounds); the number of Sounds allocated to a Group will be no more than this number. The Primary patch is Group 0, and it always contains at least one Sound (number 0). Other Groups can have as few as zero Sounds. You will rarely access a Sound's Group when developing a module.

The top-level abstract superclass for modules is **Modulation**, which designates a module capable of receiving modulation signals from some M sources, and likewise some P different modulation signals which other modules may subscribe to. A Modulation which the user will think of as primarily producing modulations, rather than filtering modulations, may be designated a **ModSource**, which gives it a specific color in the GUI. Examples of ModSources might include LFOs or envelopes; though they do receive modulation (such as LFO's rate), their primary function is to provide modulation. A Sample and Hold module, which receives an incoming signal and produces a stepped version of the same, would not be a ModSource.

Modulation signals are single numbers (ranging from 0...1) passed from module to module; these numbers typically change over time. There are many Modulation subclasses, and you can create your own. Generally Modulations are registered with Sounds, which will pulse them as necessary to produce timely modulation values. But one special Modulation, **Constant**, simply provides a single constant number all the time. Constants are not registered with Sounds and primarily serve to fill in incoming modulation slots for which you do not have a Modulation class hooked up. In the GUI, modulation dials which aren't wired to anything are represented by Constants. null cannot be used to fill an incoming modulation source slot: if it's not filled by some other Modulation, it must be filled by a Constant.

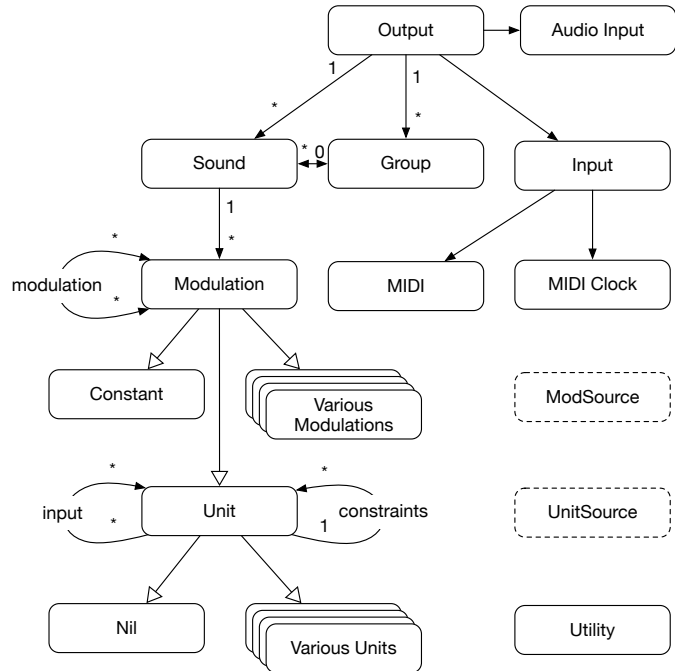


Figure 15: Core Classes.

A special kind of Modulation is a **Unit**. This is the abstract superclass for modules which, in addition to (possibly) using and/or emitting modulation signals, also use and/or emit arrays of partials for additive synthesis. Units provide timely partial arrays to other Units as necessary. The module designated to *emit* partials to a Sound must be a Unit. A Unit which primarily exists to produce partials may be designated to be a **UnitSource** so as to give it a special recognizable color in the GUI. A Unit which simply provides a Sawtooth wave is an example of a UnitSource; whereas a low-pass filter would not be.

A single partial consists of a real-valued *frequency* > 0 , a real-valued *amplitude* ≥ 0 , and an integer (byte) tag called, confusingly, an *order*. Frequencies are relative to a base pitch: thus the lowest frequency partial typically has a frequency of 1.0: this would normally be the *fundamental*. However frequencies can be lower than this. The *order* tag is a unique ID for this partial; when partials are moved around in the array, we can use the order tag to keep track of which partial is which. This is useful for some modules (notably *Smooth*), but it is particularly important when building the final sound, as each sine wave generator, and its current position in the sine wave, is based on the order.

Non-UnitSource Units typically filter or otherwise modify incoming partials from one or more sources and then send them on to other Units. Before they sent them on, these Units typically will optionally *constrain* the partials by frequency. This means that they will select some user-defined set of partials to reset back to the original values they held when they were received from the Unit's sources in the first place. This allows the user, for example, to stipulate that he wants to apply a low-pass filter on all partials except the ones forming perfect fifths. The user can also provide another Unit as a *constraint source*: the index numbers incoming partials for which it has nonzero amplitudes would indicate which partials the original Unit should preserve.

Like Modulation, you can create your own Unit subclasses, and many already exist. However one special Unit class is **Nil**, which provides a single array of partials with zero-level amplitudes. If a Unit's incoming partials source slot is not filled by another Unit, it will be filled by Nil. Source slots may not be null.

The final class in the core package is **Utility**, which simply provides various utility functions, notably mathematical approximations to various transcendental functions.

Threads These classes spawn and are manipulated by several threads, as shown in Figure 16. First and foremost is the **voice sync thread**. This thread can either be spawned to run on its own, or it can simply be the application's `main(...)` thread, calling `output.go()` in an infinite loop. The voice sync thread is responsible for getting the latest MIDI data from the **MIDI input receiver thread**, then syncing all of the **individual voice threads** and having them produce a new set of partials, one per Sound. If there is only one voice, the voice sync thread has the option of handling that voice itself rather than spawning with and negotiating with an individual voice thread. Finally, it hands these partials to the **sound output thread** which converts them into a sound wave and emits it. The sound output thread does this by splitting the task per-voice into separate output threads to add up the respective partials.

The sound output thread and MIDI input receiver threads run constantly in the background; but the voice sync thread and the individual voice threads can all be paused by locking on Output's **sound lock**. The recommended pattern is:³⁴

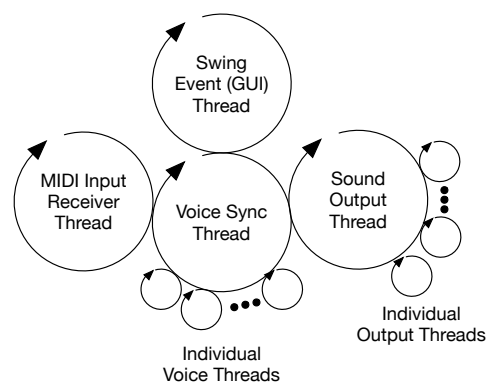


Figure 16: The threads and their interactions.

³⁴This doesn't use the `synchronized` keyword because the lock must be a "fair" (non-barging) recurrent lock internally. The locks used in `synchronized` are (unfortunately) unfair, so we can't use them.

```

Output output = ... ;
output.lock();
try
{
    ... safely read or modify the Sounds and their various Modulations and Units here ...
}
finally
{
    output.unlock();
}

```

And indeed this is how the Swing event thread (if you have a GUI) interacts with the system: by acquiring the sound lock to read information to draw, or to modify things in response to GUI events.

When the threads are spawned The MIDI input receiver thread and the output sound thread are both spawned when you call `new Output()`. The voice sync thread is not necessarily spawned: if you call `output.go()` in a tight loop, then you are the voice sync thread. However you can spawn a voice sync thread to do this for you by calling `output.startPrimaryVoiceThread()`. The individual voice threads are spawned (if at all) as soon as the voice sync thread calls `output.go()` for the first time. This means that you must have constructed all the Sound objects and registered them with the Output prior to this happening.

Performance tuning parameters You should be aware of several tunable parameters in `Output.java` which impact on the performance of these threads and ultimately the performance of the application. Some of these are constants, others are variables which `Output.java` loads from Java preferences in its constructor, and are available to change in the GUI.

- **SAMPLING_RATE** This value, normally 44100, could of course be reduced (perhaps halved) to improve performance, but at a dramatic cost in sound quality. I would not do that.
- **bufferSize** Specified by the user, by default, `DEFAULT_BUFFER_SIZE = 1152`. The size of the sound buffer being fed by the sound output thread. Java's audio system grabs data from this buffer in fairly random gulps: on the Mac they're 128 bytes at a time and can happen in spurts. The sound output thread feeds the buffer two bytes (one 16-bit sample) at a time. If the audio system cannot grab bytes from the buffer without reducing it to zero, then you will hear a **glitch** (an audible pop or zap). We don't want those. We fix this by keeping the buffer pretty big. The disadvantage of a large buffer is that if we have filled it, we're creating latency, and we don't want that either. At an absolute minimum the buffer must be at least 1024 (the apparent minimum on the Mac to even work properly); but in reality we need it larger than that to prevent glitches. 1152 seems adequate but potentially can make a large latency (about 38 milliseconds worst-case).
- **SKIP** We don't necessarily produce a new set of partials for every single sample; if your computer isn't fast enough, this isn't reasonable. Instead we generate N samples for each partials set. This is the value of `SKIP`. The default value is 32. Upon receiving a new sample set, the sound output thread linearly interpolates between the previous sample set and the next one over 32 samples. This also induces a different kind of lag: if you generate a sound, you'll have to wait up to 32 samples before it begins to be produced, and up to 64 samples before it is fully present. Ideally `skip` would be 1, which would produce the best quality sound, as well as produce true FM synthesis effects. But there you have it.
- **PARTIALS_INTERPOLATION_ALPHA** Because partials don't arrive every sample (see `DEFAULT_SKIP` above), we'd like to smoothly interpolate between them. This is done by repeatedly multiplying in a small amount of the new partials amplitudes until they're dominant. The amount multiplied in is `PARTIALS_INTERPOLATION_ALPHA`, by default set to 0.05. If you set this to a larger value, you'll start

hearing pops or hisses when you make abrupt changes in volume in your partials (try a Drawbars patch to test). But smaller values mean that new changes will take a longer time to take over—essentially a kind of attack rate. At 0.05, partials are about 95% takeover after about 100 samples (roughly three `DEFAULT_SKIP` intervals), or approximately 2ms, which seems reasonable. At any rate, if you change `DEFAULT_SKIP`, this may have an impact on what setting you want to use for `PARTIALS_INTERPOLATION_ALPHA` too.

- `numVoices` is the maximum number of voices that the system will generate. Specified by the user, can be no more than `MAX_VOICES`. It is by default set to `DEFAULT_NUM_VOICES = 8`. More voices of course means more computing power required.
- `numVoicesPerThread` Specified by the user, by default `DEFAULT_NUM_VOICES_PER_THREAD = 8`. If you have a high thread context switching overhead, you can assign more than one voice to a single thread. For example, if `numVoicesPerThread` was equal to 2, then you'd have 4 threads handling your 8 voices. If you don't have 8 cores, you might wish to do this. However keep in mind that the total number of modules your voice can have is bounded by a single core: if you have two voices on a thread, they will share that core and your total number will be halved.
- `numOutputsPerThread` Specified by the user, by default `DEFAULT_NUM_OUTPUTS_PER_THREAD = 2`. Affects the number of output threads generated to convert the partial results into the sound. Fewer threads, and you've got more work being done by a single processor (which may not be able to take the load); but more threads means more total work given the thread context switching.
- `MINIMUM_VOLUME` If a sound is below this value, Flow doesn't bother adding it into the sum, which saves a little bit of CPU.

Finally, there's one constant in `Unit.java` of interest:

- `NUM_PARTIALS` is the number of partials being handed from module to module. This is by default 128, but can be changed to 64 or 128 in the Preferences.

The GUI The GUI facility is (of course) structurally more complex than the core code, but it's not too bad: see Figure 17. But here is the gist of it. Flow contains a single **Rack** which is associated with the Output and which holds some N instances of **ModulePanel** in a `JScrollPane`. Each `ModulePanel` represents a *module* in the user's patch, either a Modulation or a Unit, and lets the user manipulate that module. Recall that in fact there is one such module for every Sound: the `ModulePanel` will associate itself with the "official" copy of that module, namely the copy stored in `Sound 0`.

There are several parts of a module that the GUI must display to the user inside a `ModulePanel`:

- Ports for incoming Modulutions to attach, or for the user to manipulate as if they were dials. These are displayed using **InputModulation**.
- Ports for incoming Units to attach. These are displayed using **InputUnit**.
- Ports to attach to outgoing Modulutions. These are displayed using **OutputModulation**.
- Ports to attach to outgoing Units. These are displayed using **OutputUnit**.
- A **ConstraintsChooser**. This is the gizmo attached to `ModulePanels` for many (but not all) Units which allows you to constrain which partials are being modified. It contains a combo box, a checkbox, and a port for an incoming Unit (which is why it's a subclass of `InputUnit`).
- **OptionsChoosers**. These allow the user to select an option in a module: they're presented either as check boxes or combo boxes.

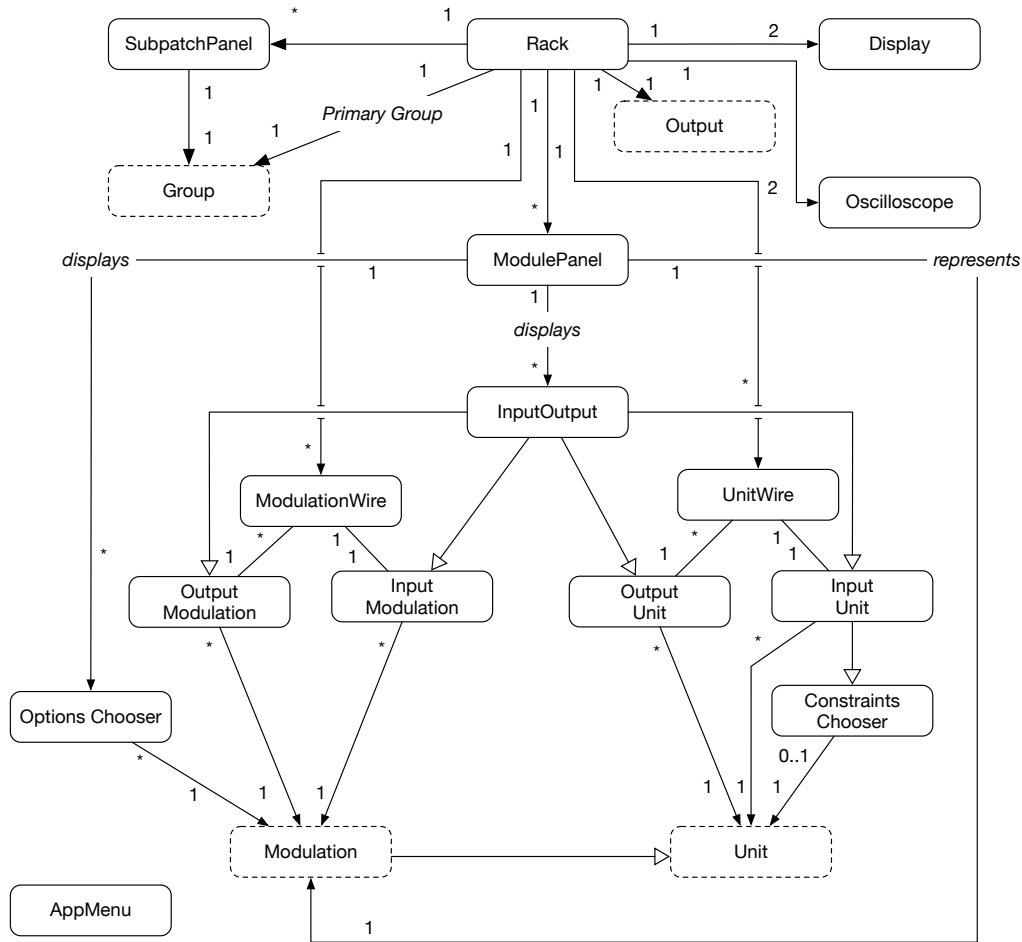


Figure 17: The GUI Facility.

InputUnits, OutputUnits, InputModulations, and OutputModulations (and ConstraintsChoosers) share a lot of code in common, and so belong to a common abstract superclass called **InputOutput**. No, it's not a good name.

InputUnits and OutputUnits are connected together by the user with a **UnitWire**. Similarly, InputModulations and OutputModulations are connected together by the user with a **ModulationWire**.

In addition to holding all ModulePanels, the Rack also holds the list of all InputWires and ModulationWires in the patch. The Rack also contains two **Displays** which draw partials for the user. The Rack is stored in a JFrame and is associated with the synth's menu. The menu is produced via various static methods in **AppMenu**. AppMenu also stores the list of current available Modules (for now).

There are a few more utility GUI widget classes, but that's the bulk of it.

Macros and Patches The module **Out** exists in practically every user patch, since it is the only way for them to produce a sound (for testing purposes you can produce sound with any Unit). But besides producing sound, Out works with two other modules, **In** and **Macro**, to enable user macros. It works like this.

The user adds at least one Out and one In to his patch. Out has several InputUnits (one of which is normally used to output sound) and InputModulations. In has several OutputUnits and OutputModulations. These InputOutputs are special: the user can edit their names. The user can attach to these InputOutputs

to provide input and output to the whole patch when it is used as a **Macro**. Then the user saves the patch (which just serializes out all the modules in the Rack). The user can then load the patch as a Macro, at which point the all of the modules are put in a list managed by the Macro rather than by a Rack. The original patch's Out module's InputUnits and InputModulations are attached directly to the Macro's OutputUnits and OutputModulations; and likewise, the In module's OutputUnits and OutputModulations are attached to the Macro's InputUnits and InputModulations. Thus the whole patch becomes a module just like any other.

Normally modules are registered with Sounds; but modules inside a Macro are not. Instead the Macro is registered as a module with a Sound as usual, but when various event methods are called on it, it calls the equivalent on all of its subsidiary modules. This process is recursive:³⁵ Macros can contain Macros which contain Macros.

4.1 Building a Modulation

A **Modulation** is a module which emits modulation values. It may also accept modulation values (ours will). Let's make a trivial square wave generator. Every N samples (at 44100 Hz) it will change from outputting a 0 to outputting a 1. We'll add some more bells and whistles to it as we go along. We start with a basic class:³⁶

```
package flow.modules;
import flow.*;
public class MyModulation extends Modulation
{
    public MyModulation(Sound sound)
    {
        super(sound);
    }

    public static String getName() { return "My Modulation"; }
}
```

The Sound class defines the voice which owns the Modulation. The call to `super.sound()` is pretty critical, as it allows the Modulation to register itself with the voice in the first place. The `getName()` method is optional and tells Flow what name to put in your module's title bar. If you don't provide it (and many don't), the class name will be used. Notice that the method is *static*. It has to be. A non-static version of this method will be ignored.

Output Something Next let's output *something*, say 0.3:

```
package flow.modules;
import flow.*;
public class MyModulation extends Modulation
{
    public MyModulation(Sound sound)
    {
        super(sound);
    }

    public void go()
    {
        super.go();
        setModulationOutput(0, 0.3);
    }

    public static String getName() { return "My Modulation"; }
}
```

³⁵It's all down to your computational power.

³⁶Yes, the codebase is indented in Whitesmith's. Resist the urge to burn it with fire.

The `go()` method is pretty central. It will be called whenever your module is being asked to update itself, by grabbing modulation and other information from elsewhere (if it needs to) and updating the modulation value it will output when asked to. We are calling `setModulationOutput(...)`, which tells the module to return 0.3 whenever someone connects to our output modulation port #0 and calls `modulate(port)` on it. Be sure to call `super.go()`.

Handle Time Next let's change our output values over time.

```
package flow.modules;
import flow.*;
public class MyModulation extends Modulation
{
    int firstTick = -1;

    public MyModulation(Sound sound)
    {
        super(sound);
    }

    public void reset()
    {
        super.reset();
        firstTick = getSyncTick(true);
    }

    public void go()
    {
        super.go();
        if (firstTick < 0) return;
        int tick = getSyncTick(true);
        if (((tick / 44100) - firstTick) % 2 == 0)
            setModulationOutput(0, 0.0);
        else
            setModulationOutput(0, 1.0);
    }

    public static String getName() { return "My Modulation"; }
}
```

The method `getSyncTick(true)` gives us the current *tick*. This will be an integer value representing the number of samples that have been generated so far.³⁷ Here we're using it to switch from 0.0 to 1.0 every second. Note that we pass *true* into `getSyncTick(...)`: this tells Flow that if the MIDI clock is running, it should use the MIDI clock as the source of ticks rather than the wall clock. Thus if the MIDI clock runs faster or slower, our ticks will come faster or slower as well.³⁸

The method `reset()` is called when our module is entirely reset. We should override it to set ourselves to a pristine state. Here this means grabbing the current tick and using that as our base for future timing. Be sure to call `super.reset()`. As you can see, if `firstTick` hasn't been set yet (it's -1), we don't update.

Handle Note-On Right now we have a **free running** LFO of sorts: it changes from 0 to 1 regardless of what we're playing. Instead let's it whenever we play a note. Also, let's add in *another* modulation output, perhaps one which outputs the opposite of modulation #0:

³⁷`getTick()` is currently an integer for threading efficiency reasons. But this means it overflows after about 13 hours. We may change it to a long later.

³⁸If the MIDI clock is running at 120BPM, then the ticks from `getSyncTick(...)` will be exactly the same as if we were using wall clock time.

```

package flow.modules;
import flow.*;
public class MyModulation extends Modulation
{
    int firstTick = -1;

    public MyModulation(Sound sound)
    {
        super(sound);
        defineModulationOutputs(new String[] { "Out", "Other" });
    }

    public void reset()
    {
        super.reset();
        firstTick = -1;
    }

    public void gate()
    {
        super.gate();
        firstTick = getSyncTick(true);
    }

    public void release()
    {
        super.release();
        if (!free) firstTick = -1;
    }

    public void go()
    {
        super.go();
        if (firstTick < 0) return;
        int tick = getSyncTick(true);
        if (((tick / 44100) - firstTick) % 2 == 0)
            setModulationOutput(0, 0.0);
        else
            setModulationOutput(0, 1.0);
        setModulationOutput(1, 1.0 - getModulationOutput(0));
    }

    public static String getName() { return "My Modulation"; }
}

```

What are we doing here? Well first, we've added the method `gate()`. This method is called whenever a note is pressed (there's an equivalent method called `release()` called when a note is released). Here we're doing the same thing we used to do in `reset()`. As usual, be sure to call `super.gate()`. And now in `reset()` we're just resetting the first tick to 0.

Additionally, in `go()` we're setting output modulation port #1 to be the opposite of what we just set in output modulation port #0. By default Modulation subclasses sport a single modulation port called "Out". Because we now have two ports, we have to define them. So we're doing that in our constructor.

Get Modulation from Elsewhere and Add Custom Options Next let's receive modulation from another source to define our modulation rate. Additionally, we'll add two options, *free* (which determines whether our LFO is free) and *type* which adds unipolar options.

```

package flow.modules;
import flow.*;
public class MyModulation extends Modulation
{
    int firstTick;
    boolean update;

    boolean free = false;
    public boolean getFree() { return free; }
    public void setFree(boolean val) { update = val; free = val; }

    public static final int TYPE_FULL = 0;
    public static int TYPE_UNIPOLAR_POSITIVE = 1;
    public static int TYPE_UNIPOLAR_NEGATIVE = 2;
    int type = TYPE_FULL;
    public int getType() { return type; }
    public void setType(int val) { type = val; }

    public MyModulation(Sound sound)
    {
        super(sound);
        defineModulationOutputs(new String[] { "Out", "Other" });
        defineModulations(new Constant[] { Constant.ONE }, new String { "Rate" });
    }

    public void reset()
    {
        super.reset();
        firstTick = getSyncTick(true);
        update = free;
    }

    public void gate()
    {
        super.gate();
        if (!free) update = true;
    }

    public void release()
    {
        super.release();
        if (!free) update = false;
    }

    public void go()
    {
        super.go();
        if (!update) return;
        double mod = modulate(0);
        int tick = getSyncTick(true);
        double val = 1.0;
        if (((int)((tick / (mod * 44099))) - firstTick + 1) % 2 == 0)    val = 0.0;
        if (type == TYPE_UNIPOLAR_POSITIVE)
            val = (val / 2.0) + 0.5;
        else if (type == TYPE_UNIPOLAR_NEGATIVE)
            val = (val / 2.0);
        setModulationOutput(0, val);
        setModulationOutput(1, 1.0 - val);
    }

    public static String getName() { return "My Modulation"; }
}

```

You'll notice that we've modified `go()` to allow for building up the variable *val*, but you should be able to convince yourself that it's basically the same. The first thing we do now is call `modulate(0)`, which gives us the current modulation value of our modulation *input* port. We then use this as a rate (notice that avoid dividing by zero: modulations go from 0.0 to 1.0 inclusive). Finally, depending on whether we're currently unipolar, we may modify our value more. We then set the modulation outputs to the value.

Notice also that we're now either resetting the tick in `reset()` or in `gate()` depending on the value of *free*; and we've `release()` now to turn off a boolean called `update` which we'll now use to determine whether to update the wave or not.

Since we're assuming a modulation input which defines our rate, we have to define it (modulations by default have no modulation inputs). We do that in the constructor. Notice that we provide not only a name for the input, but also an initial value. Initial values are in the form of `Constant` objects. A `Constant` is a very simple `Modulation` which only outputs a single value between 0.0 and 1.0 and never changes unless you manually set it. You can make your own `Constant`, or you could choose from one of several predefined ones in the `Constant` class itself. Here we're choosing `Constant.ONE`, which not surprisingly outputs a 1.0 (which will correspond to us using a rate of 44100).

Enable Custom Options in the GUI Currently our new *free* and *type* variables cannot be set by the GUI, but rather must be set programmatically. We need to change that. Flow could have handled this with some elaborate manipulation of Java Reflection, but we have opted to go the simple route: by adding two general methods, `getOptionValue(...)` and `setOptionValue(...)`, which just call the appropriate methods internally for our **options**. We add the following code to our class:³⁹

```
public static final int OPTION_FREE = 0;
public static final int OPTION_TYPE = 1;

public int getOptionValue(int option)
{
    switch(option)
    {
        case OPTION_FREE: return getFree() ? 1 : 0;
        case OPTION_TYPE: return getType();
        default: throw new RuntimeException("No such option " + option);
    }
}

public void setOptionValue(int option, int value)
{
    switch(option)
    {
        case OPTION_FREE: setFree(value != 0); return;
        case OPTION_TYPE: setType(value); return;
        default: throw new RuntimeException("No such option " + option);
    }
}
```

Notice that since we have two options, we have two option values, 0 and 1. All options accept and return integer values. A boolean option (like *free*) should be mapped to the values 0 and 1, as we have done here.

In the GUI, Flow will display boolean options as checkboxes, and non-boolean options as comboboxes. But where does the GUI get the names for the options to display them? Answer: we need to add a `defineOptions(...)` method to the constructor. We modify the constructor like this:

³⁹Let's not replicate the class code over and over again any more: it's getting longer than a single page!


```

public MyModulation(Sound sound)
{
    super(sound);
    defineModulationOutputs(new String[] { "Out", "Other" });
    defineModulations(new Constant[] { Constant.ONE }, new String { "Rate" });
    defineOptions(new String[] { "Free", "Type" },
        new String[][] { { "Free" }, { "Full", "Unipolar", "Negative Unipolar" } });
}

```

The method `defineOptions(...)` takes two arguments: an array of names for the options, and an array of arrays of names for possible values the options can take on. But if an option is boolean, and is being displayed as a checkbox, then what are its “option values”? `defineOptions(...)` will treat any method with *zero* or *one* option values, or a null option values array, as a boolean option. The checkbox label will be the option name, not the option value. We could have put an empty String array there, or null, but modules normally stick a single String there, the same as the option name. So here, by adding the single option name “Free”, with a single option value String, we have defined a checkbox which will simply say “Free”.

The other option, “Type”, will be displayed as a combobox with three options, “Full”, “Unipolar”, and “Negative Unipolar”, representing values 0, 1, and 2 respectively. The title of the box will be “Type”. Note that “Negative Unipolar” is quite long, resulting in a long combobox and a thick module panel — and wasted space. You might want to think about how to shorten names as much as possible.

Add Triggers Let’s add a **trigger** to our Modulation. Every time we transition from low to high, we want to also set a trigger. Attached modules can detect this trigger and use it to pulse themselves. We modify the `go()` method as follows:

```

boolean transitioned = false;
public void go()
{
    super.go();
    if (!update) return;
    double mod = modulate(0);
    int tick = getSyncTick(true);
    double val = 1.0;
    if (((int)((tick / (mod * 44099))) - firstTick + 1) % 2 == 0)
    {
        val = 0.0;
        transitioned = false;
    }
    else
    {
        {
            val = 1.0;
            if (!transitioned)
            {
                setTrigger(0);
                transitioned = true;
            }
        }
    }
    if (type == TYPE_UNIPOLAR_POSITIVE)
        val = (val / 2.0) + 0.5;
    else if (type == TYPE_UNIPOLAR_NEGATIVE)
        val = (val / 2.0);
    setModulationOutput(0, val);
    setModulationOutput(1, 1.0 - val);
}

```

We want our trigger to happen only when we transition high. So notice that when we transition to 1.0, we set the trigger only if it’s not been set since the last transition to 0.0 (where we reset things).

When we call `setTrigger(0)`, this sets the trigger in outgoing modulation port 0. Why don’t we clear it? Because a trigger only occurs for one tick. Inside `super.go()` it’ll automatically clear all triggers next time.

Add Presets Let's finish out by adding some presets. Modulations (and Units) which permit Presets implement the Presetable interface.⁴⁰ So we'll change our class declaration as:

```
public class MyModulation extends Modulation implements Presetable
```

To implement this method we need to create two new methods. The first gives our list of presets, and the second responds when the user has chosen one.

```
public String[] getPreset()
{
    return new String[] { "Free and Full", "Not Free and Positive" };
}

public void setPreset(int preset)
{
    if (preset == 0) // free and full!
    {
        setFree(true);
        setType(TYPE_FULL);
    }
    else if (preset == 1) // not free and positive!
    {
        setFree(false);
        setType(TYPE_UNIPOLAR_POSITIVE);
    }
}
```

That should do the job. Of course, you might want to make a table of preset values and names instead of doing it this way. But there you go.

Test the Code Now let's try it out. Run Flow as:

```
java -Dmodule=flow.modules.MyModulation flow.Flow
```

... and the MyModulation module should appear as a modulation in the menu. Later you can make this permanent by adding the `flow.modules.MyModulation.class` class to the list of modules called `static final Class[] modules` in `additive/gui/AppMenu.java`.

Now create an empty patch, add an **Out** and a **MyModulation** to it, and hook up our *out* and *other* ports to Out's 1 and 2 dials. Check "free", and you should start seeing stuff on the oscilloscopes. Change the rate and see what happens.

There's one oddity: the title bar is pink, and it appeared in the "Modulation Shapers" menu, not the "Modulation Sources" menu. This is because Flow thinks that MyModulation is meant to revise or filter an incoming Modulation. But of course it's not; MyModulation really is a modulation source. Why does Flow think this? Because MyModulation has a modulation input port. We can clue Flow in by modifying MyModulation's class declaration as:

```
public class MyModulation extends Modulation implements Presetable, ModSource
```

ModSource simply tells Flow that MyModulation is a modulation source, not some kind of modulation shaping thing. Now it's in the right menu and the right title bar color.

Saving and Loading Patches Flow saves and loads patches to very simple files: they're just GZipped JSON text files. Typically your module will load and save without any issue. However you should be aware of some items.

⁴⁰I don't know if this should be Presetable or Presettable. The latter is shorter though.

First: there is a special optional JSON object available for you if you need to save custom data. It's accessed by overriding the methods `getData()` and `setData(...)`. See `Wavetable` and `Draw` for examples.

Second: the key for each modulation input/output, each unit input/output, and each option in your module is a string which is normally just the title of the item. For example, the key for our `ModulationOutput` entitled *Other* is simply `"Other"`. This can be a problem if you have elements of the same kind (`ModulationOutputs` say) which have the same name, or a name which changes. For example, the `Envelope` module has a bunch of modulation inputs all called *Level*. To deal with this you can override the method `getKeyForModulationOutput(...)` (or `getKeyForOption(...)`, etc.) to return a designated unique key.

Third: you might also change your module, which causes problems when trying to be backward-compatible during loading. If you update your module, you might override your modulation version in `getVersion(...)`. If you've changed your options, modulation inputs/outputs, and unit inputs/outputs, perhaps adding a new one or deleting some, you might need to check for this immediately before and/or immediately after they are loaded and set automatically, so as to revise them. The methods `preprocessLoad(...)` and `postprocessLoad(...)` may be helpful. You could completely override everything, if you wanted, by overriding the methods `loadModulations(...)` or `loadOptions(...)` etc.. Or you could just override `load(...)`.

The Current Patch Format The format is in GZipped JSON and is as follows:

- `{ "flow":flow-version, "name":patch-name, "by":patch-author, "on":patch-date, "info":patch-info, "v":patch-version, "modules":modules, "sub":optional-subpatches }`
- *flow-version*: an integer ≥ 0 indicating the version of Flow which created this file. Presently 0.
- *patch-name*: The name of the patch, as stipulated by the patch creator. A string.
- *patch-author*: The patch creator. A string.
- *patch-date*: The patch creation date. A string.
- *patch-info*: Useful information about the patch, as stipulated by the patch creator. A string.
- *patch-version*: The version of the patch, as stipulated by the patch creator. A string.
- *modules*: *module*,*
- *module*: `{ "v":module-version, "class":module-classname, "id":module-id, "opt":options, "unit":unit-inputs, "mod":modulation-inputs, optional-constraints, optional-data }`
- *module-version*: the version of this kind of module. An integer ≥ 0 . Modules are presently largely at 0.
- *module-classname*: the class name of this module. A string.
- *id*: A unique identifier of this module, which is referenced by other modules when they state that they are connected to it. A string.
- *optional-constraints*: A JSON object in the form: `"constrain":{ "id"=id, "at"=modulation-unit-name, "type"=constraints-type, "not"=constraints-inverse }` This object does not need to appear in the patch at all. Additionally, all of the elements inside the patch are optional: they may appear or they may not, except that "id" and "at" must appear together.
- *constraints-type*: The value of the constraints type (an integer).
- *constraints-inverse*: Whether or not the options is inverted (a boolean).

- *optional-data*: A JSON object in the form "data":{...} which the module may use as it likes to store additional data. This object does not need to appear in the patch at all. See Wavetable and Draw for examples of usage of optional data. Additionally, both the In and Out modules use optional-data to store the user-defined names for their modulations and units (as an array of Strings called "mod" and another array of Strings called "unit").
- *options*: { *option-name:option-value,** }
- *option-name*: An option name (a string). Normally this is the same as returned by `getOptionName(...)`.
- *option-value*: The value of the option (an integer).
- *modulation-inputs*: { *modulation-input-name:modulation-input,** }
- *modulation-input-name*: A modulation input name (a string). Normally this is the same as returned by `getModulationName(...)`.
- *modulation-input-value*: Either a double between 0.0 or 1.0 representing the modulation dial value, or a *modulation-output*.
- *modulation-output*: { "at":*modulation-output-name*, "id":*id* } The id is for the attached module.
- *modulation-output-name*: A modulation output name (a string). Normally this is the same as returned by `getModulationOutputName(...)`.
- *unit-inputs*: { *unit-input-name:unit-input,** }
- *unit-input-name*: A unit input name (a string). Normally this is the same as returned by `getInputName(...)`.
- *unit-input-value*: { "at":*unit-output-name*, "id":*id* } The id is for the attached module.
- *unit-output-name*: A unit output name (a string). Normally this is the same as returned by `getOutputName(...)`.
- *optional-subpatches*: *subpatch,**
- *subpatch*: { "voices":*subpatch-voices*, "gain":*subpatch-gain*, "pan":*subpatch-pan*,
"midi":*optional-subpatch-midi-channel*, "note":*optional-subpatch-note*,
"by":*patch-author*, "on":*patch-date*, "info":*patch-info*, "v":*patch-version*,
"flow":*flow-version*, "name":*patch-name*, "modules":*modules* }
- *subpatch-voices*: The requested number of voices (Sounds) for this subpatch (an integer).
- *subpatch-gain*: The requested gain for this subpatch (a double ≥ 0.0).
- *subpatch-pan*: The requested pan for this subpatch (a double 0.0–1.0).
- *optional-subpatch-midi-channel*: The subpatch's midi channel, 0–16. 0, or missing, means "No channel".
- *optional-subpatch-midi-note*: The MIDI note for the subpatch, 0–128. 0, or missing, means "Any note". Otherwise the note is the value, minus 1.

Cloning One obscure feature of Flow is that Modulations and Units can not just be *moved* in the Rack, but can in fact be *copied*. This is done by holding the Control key (on the Mac, Option also works) while dragging the Module’s title bar. This in turn calls the Module’s `clone()` method. You need to override this method to insure that it performs a deep clone. In many situations you’d need not implement this method at all. But if your Modulation or Unit is holding an array or perhaps some other object which must be copied, you’ll need to implement it like this:

```
int[] stuff;    // let's say this is the array you need to copy into the cloned object
public Object clone()
{
    MyClass obj = (MyClass)(super.clone());
    obj.stuff = (int[])(stuff.clone());
    return obj;
}
```

4.2 Building a Unit

Read Section 4.1 (Building a Modulation) first. It covers a lot of important stuff that we’ll build on here.

A **Unit** is a module which emits partials. If you like, it can also accept partials, and can both accept and emit modulations. Let’s make a trivial low pass filter: we’ll zero out the amplitude of a partial if the partial is less than a certain value, which we’ll define as a modulation.

We begin by making a Unit subclass. Note that Unit is a subclass of Modulation, so we’re also making a Modulation. However, while Modulations by default have one modulation output defined (“Out”), Units have zero defined by default. They do, however, have one partials output defined (also called “Out”).

```
package flow.modules;
import flow.*;
public class MyUnit extends Unit
{
    public MyUnit(Sound sound)
    {
        super(sound);
        defineModulations(new Constant[] { Constant.HALF }, new String { "Cutoff" });
        defineInputs(new Unit[] { Unit.NIL }, new String { "In" });
    }

    public void go()
    {
        super.go();
        double mod = modulate(0);

        pushFrequencies(0);
        copyAmplitudes(0);

        double[] amplitudes = getAmplitudes(0);
        double[] frequencies = getFrequencies(0);

        // we'll do more here...
    }
}

public static String getName() { return "My Unit"; }
```

We define one modulation input called “Cutoff”, and set up `go`. Since we’re receiving partials, we define one incoming partials port, called *In*. By default the source module for this port is `Unit.NIL`, which represents the “empty Unit”, with standardized frequencies and all-zero amplitudes. You can’t use `null` here: every partials input must be a real Unit. When the user wires up the port to some other module,

Unit.NIL will be replaced with that module's output.

It's important to understand what the `pushFrequencies(0)` and `copyAmplitudes(0)` lines are doing. `pushFrequencies(input, output)` instructs the module to set the frequencies of all partials in output port *output* to those in input port *input*. Because in the large majority of cases we don't have more than one partials output, `pushFrequencies(input)` does the same thing as `pushFrequencies(input, 0)`. If you have pushed your frequencies (or your amplitudes), you may read them but you **may not modify them**. You'd push frequencies or amplitudes if you're not modifying them, because pushing is more efficient than copying them.

Because we're modifying amplitudes, we call `copyAmplitudes(0)` instead of `pushAmplitudes(0)`. The method `copyAmplitudes(input, output)` makes a copy of the amplitudes from *input* and puts the copy in *output*. And similarly, `copyAmplitudes(input)` is just `copyAmplitudes(input, 0)`.

So what are the amplitudes and frequencies of our partials? They are each an array, normally of length 128, of doubles. Both amplitudes and frequencies must be ≥ 0 . We extract the current amplitudes and frequencies from output port 0 with `getAmplitudes(0)` and `getFrequencies(0)` respectively. Remember that the extracted frequencies are **not to be modified** because we pushed them.

Notice that we've said nothing about the Orders. This is because changing Orders is so rare that they're pushed automatically for you in `super.go()`.⁴¹

Do the Filter Next let's determine our cutoff frequency and do some cutting-off!

```
package flow.modules;
import flow.*;
public class MyUnit extends Unit
{
    public MyUnit(Sound sound)
    {
        super(sound);
        defineModulations(new Constant[] { Constant.HALF }, new String { "Cutoff" });
        defineInputs(new Unit[] { Unit.NIL }, new String { "In" });
    }

    public void go()
    {
        super.go();
        double mod = modulate(0);
        double cutoff = modToFrequency(makeVeryInsensitive(mod));
        double pitch = getSound().getPitch();

        pushFrequencies(0);
        copyAmplitudes(0);

        double[] amplitudes = getAmplitudes(0);
        double[] frequencies = getFrequencies(0);

        for(int i = 0; i < amplitudes.length; i++)
        {
            if (frequencies[i] * pitch > cutoff)
                amplitudes[i] = 0;
        }
    }

    public static String getName() { return "My Unit"; }
}
```

Now we're cooking! We just do a for-loop over the amplitudes and zero them if the corresponding

⁴¹If for some reason you do *not* want this behavior, then in your constructor you should call `setPushOrders(false)`; and be sure to copy or push the orders yourself manually in your `go()` method.

frequency is too high. But notice the usage of *pitch*. Frequencies in partials aren't absolute frequencies: they're multiples of the pitch frequency of the current note (which typically, but not always, corresponds to the frequency of the fundamental). So if we want to get the true frequency of a partial, we have to multiply it by the pitch. The pitch is extracted from the module's **Sound**, that is, the voice it's assigned to. Sounds have a number of items they can provide: pitch is one of them.

Now, modulation values go from 0...1. How do we convert that into a frequency? We could just multiply by half the sampling rate (remember that anything over half the sampling rate is beyond the Nyquist limit), that is, something like `cutoff = mod * Output.SAMPLING_RATE`. But frequency is exponential, so we'd like something exponential-ish. That's what the Modulation utility method `modToFrequency(...)` does.

The problem with `modToFrequency` is that by itself it takes too long to reach a reasonable value: so `modToFrequency(0.5)` is only 11 Hz. This is a waste of dial space. So we make the dial much less sensitive to the exponential-ish effect by including `makeVeryInsensitive(...)`. Now the two together yield 6567 Hz at 0.5.

Add Trigger Response Now let's throw in a monkey wrench. Imagine that we'd like to swap between low-pass and high-pass depending on the state of a trigger sent to us, oh, let's say, the Modulation we just developed in Section 4.1. We might write:


```

package flow.modules;
import flow.*;
public class MyUnit extends Unit
{
    boolean lowpass;

    public MyUnit(Sound sound)
    {
        super(sound);
        defineModulations(new Constant[] { Constant.HALF, Constant.ZERO }, new String { "Cutoff", "Trigger" });
        defineInputs(new Unit[] { Unit.NIL }, new String { "In" });
    }

    public void reset()
    {
        super.reset();
        lowpass = true;
    }

    public void gate()
    {
        super.gate();
        lowpass = true;
    }

    public void go()
    {
        super.go();
        double mod = modulate(0);
        double cutoff = modToFrequency(makeVeryInsensitive(mod));
        double pitch = getSound().getPitch();

        pushFrequencies(0);
        copyAmplitudes(0);

        double[] amplitudes = getAmplitudes(0);
        double[] frequencies = getFrequencies(0);

        if (isTriggered(1)) lowpass = !lowpass;
        for(int i = 0; i < amplitudes.length; i++)
        {
            if ((lowpass && frequencies[i] * pitch > cutoff) || (!lowpass && frequencies[i] * pitch <= cutoff))
                amplitudes[i] = 0;
        }
    }

    public static String getName() { return "My Unit"; }
}

```

We created a boolean variable, `lowpass`, to toggle between a low-pass and high-pass filter. In `reset()` and `gate()` we set `lowpass` to true. Next, we added an additional modulation for the trigger. To get this trigger information, we call `isTriggered(input)`. There also exists a method called `getTrigger(input)` which returns the *count* of how many times the trigger has been fired. Finally, we use the trigger to toggle between low-pass and high-pass. Recall that the trigger is reset the next iteration.

Add Constraints One item many filter-style Units have is a *constraints facility*. This is a facility that the user can use to restrict which partials we're allowed to modify. In fact we'll modify all of them anyway, but then the constraints facility will restore some of them. The method that does this is called `constrain()`. We modify `go()` thusly:

```

public void go()
{
    super.go();
    double mod = modulate(0);
    double cutoff = modToFrequency(makeVeryInsensitive(mod));
    double pitch = getSound().getPitch();

    pushFrequencies(0);
    copyAmplitudes(0);

    double[] amplitudes = getAmplitudes(0);
    double[] frequencies = getFrequencies(0);

    if (isTriggered(1)) lowpass = !lowpass;
    for(int i = 0; i < amplitudes.length; i++)
    {
        if ((lowpass && frequencies[i] * pitch > cutoff) || (!lowpass && frequencies[i] * pitch <= cutoff))
            amplitudes[i] = 0;
    }

    constrain();
}

```

That's it! We're constrained. Because constraints are so common for filter-style Units, the GUI for them is added automatically. If you don't want constraints for some reason, you turn them off by overriding the method `isConstrainable()` to return false.

Sometimes you want a smarter constraints than just setting all the partials and then resetting a few, because constraining some partials might affect your algorithm in some way. Instead, you can call the method `public int[] getConstrainedPartials()`. This will tell you which of your partials must be constrained: the array returned holds the indexes into your frequencies and amplitudes array reflecting to-be-constrained partials.

A Note about Sorted Order and Constraints Units are required to keep their partials **sorted by frequency**. If we had fooled around with frequencies of the partials, they might be out of order. We'd want to restore sorted order by calling `simpleSort()`; at the end of our `go()` method.

It can happen that we modified frequencies, but carefully kept things in sorted order (perhaps increasing all frequencies by 10), but then when we called `constrain()`, some frequencies were restored and we're now out of sorted order. The `constrain()` method returns a boolean value which tells us whether it might have yanked things out of sorted order. We could use that to say something like `if (constrain()) simpleSort();`

Finally, in addition to frequency and amplitude, partials have a secret third array: *orders*. In this array, each partial has a unique integer, and even as partials are moved about, sorted, constrained, and so on, we can use the orders array to determine which partial has gone where. You will probably never use this array, and normally it's updated automatically, but there are modules which rely on it to do their dirty work. If you for some reason physically move partials from one slot to another, you might want to move the orders value too. Just make sure that you don't duplicate orders values or lose any: every partial should have an orders, and there should be exactly 128 of them, 0...127.

Test the Code Now let's try it out. Add the `flow.modules.MyUnit.class` class to the list of modules called `static final Class[] modules` in `additive/gui/AppMenu.java`. Now create an empty patch, add an **Out** and a **MyUnit** to it, and hook up our *out* port to Out's *A* dial. Add a **Sawtooth** module and feed it into our *in* port. Add our **MyModulation** module and feed it into the *trigger* port (you could also use an LFO). Now play with the cutoff and see what happens!

Because we created a partials input, Flow thinks that we are a "Partials Shaper" and will color our titlebar and put us in that menu appropriately. This is good, because we are! But if we were instead a Partials Source,

we could always say:

```
public class MyUnit extends Unit implements UnitSource
```

Customize the Cutoff Frequency Display Right now our cutoff frequency dial just gives values from 0...1. That's not very helpful. Instead it'd be nice to have it provide the actual frequency. We do this by overriding the method `getModulationValueDescription(modulation, value, isConstant)`, which returns a `String` to display for the provided input modulation (dial). In our case, the cutoff dial is input modulation 0. So we might say:

```
public String getModulationValueDescription(int modulation, double value, boolean isConstant)
{
    if (isConstant) // we don't have anything plugged in
    {
        if (modulation == 0) // cutoff -- there's only one
        {
            return String.format("%.4f", modToFrequency(value));
        }
        else return super.getModulationValueDescription(modulation);
    }
    else return "";
}
```

We might wish to change the popup menu as well. This can be done in a variety of ways, but here's an easy one: we add two methods, `getPopupOptions(...)` and `getPopupConversions(...)`. The first returns a list of popup text strings; the second returns a list of double values (0...1) corresponding to them. So we could do this:

```
public String[] getPopupOptions(int modulation)
{
    if (modulation == 0) // cutoff -- there's only one
    {
        return new String[] { "10 Hz", "100 Hz", "1000 Hz", "10000 Hz" };
    }
    else return super.getPopupOptions(modulation);
}

public double[] getPopupConversions(int modulation)
{
    if (modulation == 0) // cutoff -- there's only one
    {
        return new double[] { 0.155, 0.2475, 0.359, 0.55183 };
    }
    else return super.getPopupConversions(modulation);
}
```

Customize the Module Panel It's possible to customize how the GUI for our Unit looks. To do this, we're responsible for adding in the appropriate number of modulation inputs and outputs, unit inputs and outputs, options choosers, and up to one constraints chooser. If you're interested in seeing how this is done, take a look at `flow.modules.Morph.getPanel()`. For our purposes here, let's add a `PushButton` to the `ModulePanel` which toggles the lowpass/highpass filter. To do this, we'll override `getPanel()` like this:

```

import flow.gui.;
import javax.swing.*;
import java.awt.*;

...

public ModulePanel getPanel()
{
    return new ModulePanel(Morph.this)
    {
        public JComponent buildPanel()
        {
            JComponent comp = super.buildPanel(); // this does the standard panel
            JPanel panel = new JPanel();
            panel.setLayout(new BorderLayout());
            panel.add(comp, BorderLayout.CENTER);

            PushButton button = new PushButton("Toggle")
            {
                public void perform()
                {
                    toggleFilter();
                }
            };

            panel.add(button, BorderLayout.NORTH);
            return panel;
        }
    };
}

```

Here we're using a utility GUI object called PushButton which makes it easy to do buttons and pop-up menus without relying on even listeners. Obviously this could have all been done better by adding an Option in the form of a checkbox, like we did when we made a Modulation in Section 4.1. But then we'd not learn anything right?

The next step is to implement the toggleFilter() method. That's the big deal here. The ModulePanel will be calling this method not from the underlying Sound threads but from the GUI thread. So we'll need to obtain a lock on the Sounds so we can muck with them safely. And it won't be calling the method once on each underlying copy of MyUnit (one per Sound thread) So we need to distribute the information ourselves. Here's how you do it:

```

void toggleFilter()
{
    int index = sound.findRegistered(this);
    Output output = getSound().getOutput();
    output.lock();
    try
    {
        int numSounds = output.getNumSounds();
        for(int i = 0; i < numSounds; i++)
        {
            MyUnit myunit = (MyUnit)(output.getSound(i).getRegistered(index));
            myunit.toggle = !myunit.toggle;
        }
    }
    finally
    {
        output.unlock();
    }
}

```

We acquire the lock via `output.lock()` (and release it with `output.unlock()`). Acquiring this lock will stop all partials generation in its tracks (though sound output will still continue). We will have complete control over all of the modulations in all of the sounds. You want to hold onto this lock for as little time as possible.

What's going on here is as follows. Recall that the `ModulePanel` is associated with a single `Modulation` or `Unit` (normally the one belonging to `Sound #0`). We can take advantage of this to determine what position we are in the `Sound`'s list of registered modules: all `Sounds` have their modules in the same order. We then acquire the lock, then go through all the `Sounds` and look up the same `Module/Unit` in each sound. It'll be the same kind of class as ourselves, so we just change its toggle.

You want to do all this as fast as you can: holding the sound lock for very long can cause audio disruptions.

Test the Code Once again, run `Flow` as:

```
java -Dmodule=flow.modules.MyUnit flow.Flow
```

... and the `MyUnit` module should appear as a `Partials` in the menu. Later you can make this permanent by adding the `flow.modules.MyModulation.class` class to the list of modules called `static final Class[] modules` in `additive/gui/AppMenu.java`.

Try it out! And that concludes our little tutorial.