JITModular: Teaching Synthesis by Using JITLib as a Modular Synthesizer

H. James Harkins

Xinghai Conservatory of Music

jamshark70@zoho.com · jamshark70@gmail.com

ABSTRACT

The Just-In-Time Library (JITLib) in SuperCollider supports free experimentation with signal processing by separating the design of the connections from signals' content. My JITModular quark serves as a conceptual bridge between SuperCollider signal processing and modular synthesizer design, taking full advantage of JITLib's hot-swapping and hot-patching capabilities for a free and flexible work flow. This approach facilitates both teaching audio synthesis and SynthDef development; this paper introduces the framework primarily from a pedagogical perspective, and describes the problems which JITModular addresses.

1. INTRODUCTION

SuperCollider is infamous for its steep learning curve, a reputation earned in part by a certain fragility in its "vanilla" methodologies. Who among us has never reassigned a new synth or pattern player to a variable, so that the old one becomes unreachable, and unstoppable, while still playing?

The Just-In-Time Library (JITLib) [1] alleviates this difficulty and others by managing synth nodes under a convenient umbrella called a `NodeProxy`, which consists of a signal source and a bus hosting the signal. Reassigning an audio source replaces the old signal without orphaning it, reducing the need for beginners to track nodes carefully. Meanwhile, the proxy's bus remains a stable reference point to access the current signal, enabling free interconnections between signal processors.

For synthesis pedagogy, I favor modular designs over VST-style instruments because the modules call attention to individual components. Since the goal is to streamline teaching and the development of synthesis design, monophony is sufficient for most cases. JITLib NodeProxies are already practically modules, with inputs, signal processing, and an output (with the advantage over modular-synthesizer emulators such as VCV Rack, that they can be redefined without breaking existing connections). In practice, however, I encountered a few troublesome aspects of JITLib's interfaces for this usage style. My JITModular quark [2] extends JITLib with a few new features and workflow improvements (importantly, the ability to save and restore a patch's state), and establishes best practices to streamline the use of a `ProxySpace` as a monophonic modular synthesizer.

To introduce JITModular's approach, I will first walk through my typical first lesson with new students, building a simple subtractive synthesizer incrementally. Then I will discuss problems in native JITLib and JITModular's solutions to them, outline best practices and more advanced features, and conclude a brief assessment of the quark's effectiveness.

2. Modular Design Advantages: A Student's First Exposure

In JITModular, the basic unit of signal processing is a module, rather than a SynthDef. Modules may be as simple as a single line of code. Also, if each module is visually separated from other modules, then students can focus full attention on one module at a time and understand its elements: DSP operations, its several inputs and single output, and implementation or syntactic details. Building one component at a time also structures the discussion of signal processing around incremental additions; we can pause to understand an oscillator fully before moving on to a filter or envelope generator.

Emulating the design of modular synthesizers includes two elements: defining modules, and connecting them.

Modules exist within an object, `JITModPatch`, defined in the JITModular quark: `p = JITModPatch.new;`. For convenience, I often assign it to `p`, but the current patch can always be reached from `JITModPatch.current`. Creating a patch opens a new IDE code document as well as a graphical window including buffer and MIDI controller display, a textual representation of the patch's connections, and a JITLib `ProxyMixer` for interactive control over parameters. Each patch contains a JITLib `ProxySpace`: an Environment where entities assigned to environment variables are converted into `NodeProxy` signals. When a JITModPatch's code window is focused, the patch's `ProxySpace` becomes active; users can open multiple patches at the same time and switch between them using document tabs.[1]

Modules are defined in environment variables. In the first lesson, I begin with a single-line sinewave

1 Windows users should be aware of two bug reports: "scide: Document.current nulled on Class Library Recompile" [3], and "Document's envir feature sometimes doesn't sync up in Windows" [4].

oscillator: `~osc = #{ SinOsc.ar(440) };`. At this point, SuperCollider syntax is entirely unfamiliar to the students, so I take plenty of time to explain each syntax element in turn: the environment variable as an identifier, the function braces, class names, methods, and method-call arguments. For students without prior programming experience, this is already a lot to absorb!

As in standard JITLib usage, merely creating a module does not automatically connect it to the audio hardware. Any module may be "play"-ed at any time (including use of the `ProxyMixer` "play" button), but I found it easier and more consistent to route all audio through an `~out` module with a stereo input. The tail of the DSP chain should be connected to this `~out` module, using the left-to-right chaining operator: `~osc <>> ~out`. (Multiple modules may be chained at one time, such as `~osc <>> ~filter <>> ~out`.) The target of a chaining operation is a `JMInput.ar` UGen, a wrapper for NamedControl that initializes to two channels of audio-rate input.

Having obtained a sound, we then add a frequency control. This addition introduces the use of function argument lists to expose module parameters for external control. The new control input is automatically available in GUI, by clicking the "ed" button next to the `~osc` module in the `ProxyMixer` panel. Adding a `freq` control raises student engagement because the effect of the slider is easy to hear, and also, the slider "just works," because its range is already defined: immediate positive feedback. Next comes the idea of explicit parameter ranges. If we change the oscillator to `VarSaw`, we can introduce a width parameter, whose range 0.0–1.0 must be explicitly defined using the `addSpec` method from the JITLibExtensions quark [5] (Figure 1).

```
~osc.addSpec(\width, [0, 1]);
~osc = #{ |freq = 440, width = 0.5|
   VarSaw.ar(freq, width)
};
```

Figure 1. An oscillator module with an explicit parameter range.

At this point, the main syntax elements are on the table, so it becomes straightforward to introduce different oscillator types and expand to filters and envelope generators. By the end of the first lesson, then, we have a working monophonic subtractive synthesizer (Figure 2), in four clearly delineated modules (including `~out`) and about 35 lines of code.

```
~out = #{ |amp = 0.2| amp * JMInput.ar };
~out.play;

~osc = #{ |freq = 440| SinOsc.ar(freq) };

~osc <>> ~out;

// try another waveform
~osc.addSpec(\pwidth, [0.01, 0.99]);
~osc = #{ |freq = 220, pwidth = 0.5|
   VarSaw.ar(freq, pwidth)
};

~filter.addSpec(\ffreq, \freq, \rq, [1, 0.05,
\exp]);
~filter = #{ |ffreq = 2000, rq = 1|
   RLPF.ar(JMInput.ar, ffreq, rq)
};

~osc <>> ~filter <>> ~out;

// try a third waveform
~osc = #{ |freq = 220, pwidth = 0.5|
   Pulse.ar(freq, pwidth)
};

~eg.addSpec(
   \atk, [0.01, 2, \exp],
   \dcy, [0.01, 2, \exp],
   \sus, [0, 1],
   \rel, [0.01, 2, \exp]
);
~eg = #{ |gt = 1, atk = 0.01, dcy = 0.15, sus
= 0.6, rel = 0.08|
   JMInput.ar
   *
   EnvGen.kr(Env.adsr(atk, dcy, sus, rel),
gt)
};

// need to write only eg's source and target
~filter <>> ~eg <>> ~out;
```

Figure 2: A functioning subtractive synthesizer.

This, of course, is not the shortest possible representation of this synthesizer; the corresponding `SynthDef` would comprise 9 lines. But brevity is not clarity. A `SynthDef` factors all of the signal processing into a single block. For beginning students, a unified synthesis function blurs into a wall of text from which specific features may be difficult to disentangle. In a `JITModPatch`, modules may be easily located by a descriptive variable name, and each module's boundaries are clearly marked by function braces. Students can more easily focus on one module when needed and ignore the rest temporarily. Further, the patch code reflects not only the final design, but also includes a written record of modules and connections that were tried and abandoned. It also defines parameter ranges per module, supporting the parameter faders in the graphical interface. The automatic GUI is extremely useful for beginning students! Lastly, the ability to replace and repatch modules on-the-fly makes the design process more interactive, and facilitates the audition of partial results in progress.

3. JITLib Problems and Solutions

At first, it might seem that JITLib is already natively equipped for this usage style. There are a few sticking points, however.

The first problem arises, in fact, with the first "simplest oscillator" example in section 2: the `SinOsc` is monophonic, while the `~out` module

expects a stereo input, so the result will be heard only in the left channel. To simplify students' first exposure, JITModular expands audio NodeProxies to stereo by default.[2] (Control-rate modules remain mono.) In Figure 2, `~osc` specifies a single unit generator (no `.dup`, no `!2`, no panning), but its output is automatically duplicated onto the right channel as well. A stereo module should connect to a stereo input. Native SuperCollider syntax is `NamedControl.ar(\in, 0!2)`. I felt that the `0!2` is awkward to explain to students. So JITModular provides a `JMInput` class as a factory for a `NamedControl`, where the default at audio rate is stereo. End-to-end stereo may consume more CPU cycles, but the cost is negligible compared to the benefit of avoiding confusion over channel counts.

The second problem stems from the fact that a SuperCollider code document does not unambiguously represent the final state of the patch. The final state is the aggregate result of the history of statements that were issued. SuperCollider's interactive REPL means that statements can be executed in an order different from that in which they appear in the document; thus the document is not sufficient to recover a given state. Imagine that a patch passes a signal through a filter and a distortion module. The sound will be very different depending on whether the filter or the distortion module comes first. If both orderings were tried, the code document might retain instructions for both; the active ordering could then not be inferred from the code document. JITModular provides two features to alleviate the issue. First, a panel in the GUI window lists current connections. In the above example, this panel would show either `~source <>> ~filter <>> ~dist <>> ~out` or `~source <>> ~dist <>> ~filter <>> ~out`, depending on the active state. Second, JITModular saves patches into its own file format, restoring modules' definitions, parameter values, connections, and even buffers and MIDI controller mappings, as well as the user's original code document (at whatever level of disorganization). There are some minor restrictions, which are easily avoided by following a few best practices, discussed below. A comprehensive save/load mechanism is unusual in the SuperCollider ecosystem, and is essential for classroom usage: to grade students' work, it is necessary to restore it reliably.

The third problem is buffer management. I chose to base `JITModPatch` on `ProxySpace` to reduce the typing burden for module definition and chaining. But a `ProxySpace` can contain only NodeProxies; buffers cannot be stored in its environment variables. Also, if buffers are to be restored when loading a patch from disk, then they cannot be stored in free-standing variables. JITModular's solution is to implement `addBuf` and `freeBuf` methods on the `JITModPatch` object. Buffers are stored in the patch under symbolic keys; saving the patch creates a folder alongside the patch file where buffers are saved as WAV files, and automatically reloaded. Audio modules access buffers by way of control-rate "buffer-index" modules created by `addBuf`. Figure 3 illustrates. The `addBuf` call associates a `JMBuf` (which extends `Buffer` for waveform display in the GUI window) with the key `\a11`, whereupon the expression `~a11.kr(1)` in a module accesses the buffer number. Using control-rate proxies for buffer numbers allows users to hot-swap buffers: if the user calls `addBuf` for a buffer name that already exists, the old buffer is released and the buffer proxy's buffer number is updated, so that the change transparently cascades through the whole patch.

```
JITModPatch.current.addBuf(\a11,
JMBuf.read(s, Platform.resourceDir +/+
"sounds/a11wlk01.wav"));

~loop = #{
   var buf = ~a11.kr(1);
   PlayBuf.ar(1, buf, BufRateScale.kr(buf),
loop: 1)
};

~loop <>> ~out;
```

Figure 3. Buffer usage in a JITModular patch.

4. Usage details and best practices

Space does not permit complete documentation of all JITModular features; this section is necessarily abbreviated.

One best practice has already been mentioned: to reserve one module, `~out`, to represent hardware output. This provides a single, consistent locus to mute or adjust the volume of the entire patch. I also usually reserve `~eg` for the main volume envelope, as shown in Figure 2. Note also that the keyword `gate` is reserved for `NodeProxy`'s internal use, so a gate for the patch should be written `gt`.

I recommend using closed functions for module definitions. Module definitions can be saved into the patch file only if they are closed, i.e., if they do not refer to any variables declared outside the function's scope. In SuperCollider, a function may be explicitly closed by preceding the opening brace with a hash mark: `#{ ... }`. If the hash mark is omitted and the function meets the criteria to be closed, then it will be closed internally; strictly speaking, then, the hash mark is optional. But, if you write the hash mark at the head of a function that is accidentally open, the resulting compilation error will protect you from creating a patch that cannot be saved to disk.

Splitting a single note's processing across multiple modules raises a question about argument names. Taking a `freq` argument as an example, should the

2 A module can override the two-channel default by initializing to a specific number of channels before defining the module. For example, when building a vocoder, the multiband analysis stage may be initialized by running `~bandVolume.ar(40)` for 40 bands; when the DSP function is provided after this, the 40 channels will not collapse down to stereo. However, there is no need for beginning students to confront this type of fine print at their first exposure.

patch have one single global frequency, or should each module's `freq` parameter be independent of the others? In JITModular, NodeProxies are components working together rather than independent layers, all, so same-name arguments are synchronized across all of the patch's modules. This is different from standard JITLib usage, where parameters are independent between NodeProxies. Parameters that should be independent need to have different names: note in Figure 2 that the oscillator has its `freq`, while the filter has `ffreq`.

Sequencers may be added into the patch using a new "proxy role," `\psSet` (short for "proxyspace set"). A pattern assigned to this role will set parameters within the patch. (That parameters are synchronized simplifies sequencing: the user is relieved of the burden of specifying which modules to update.) Figure 4 shows a simple sequencer that may be applied to the patch in Figure 2.

```
~player = \psSet -> Pbind(
   \midinote, Pseq([33, 45, 40, 50, 43],
inf),
   \dur, 0.25,
   \legato, Pwrand([0.6, 1.01], [0.8, 0.2],
inf)
);
```

Figure 4. A simple psSet sequencer.

Patch properties that do not otherwise fit into the `JITModPatch` structure, such as clock tempo for sequencing, can be set up in a `customInit` function assigned to the patch object. A companion hook, `cleanup`, can release any persistent resources that `customInit` created. When assigning `customInit`, the function is evaluated automatically. Figure 5 shows the recommended way to set sequencer tempo.

```
JITModPatch.current.customInit = {
   TempoClock.tempo = 128/60;
};
```

Figure 5. Custom initializer for tempo.

Modulators should generally be implemented as single-channel control-rate modules, which may be patched into audio modules' control inputs using `set()`. A helper class, `JMModulation`, makes it simpler to implement modulation in the standard way found in many modular components and VST instruments, in terms of deviation away from a set value.

External controls are based on MIDI continuous controllers. Calling `initMidi` on the patch prepares it to receive external controller data. Controls may be mapped by number (`addCtl`), or "learned" by touching a MIDI control on the hardware (`learnCtl`). Mappings are saved in the patch, and are visible in the GUI window's top-left panel. For Open Sound Control mapping, a `midiOSCBridge` can associate incoming OSC command paths with MIDI controller numbers. If the OSC bridge is initialized in the `customInit` function, it will be restored when loading the patch.

A weakness of JITModular is that it does not natively handle polyphony. A patch would need to be translated into a SynthDef and performed polyphonically using a standard Pbind. For courses in synthesis theory, monophonic synthesizers may be sufficient; polyphony could be introduced separately to advanced students. To assist, JITModPatch has an experimental decompiler: clicking the "SynthDef" button will attempt to render the patch into a SynthDef. The output is not optimal for human reading, and may in some cases require some minor hand-editing. But, as a quick-and-dirty way to get a reusable SynthDef from a patch, it is certainly faster than hand coding.

In the future, I would like to improve server node ordering in JITModPatch. In JITLib, node ordering is arbitrary; audio-rate controls may introduce control-block delays. The patch decompiler already knows the proper node order; I have not yet applied this to the patch nodes, however. Another enhancement would be a graphical display of modules and their connections; this is low priority at this time.

5. Conclusion

I have used JITModular with master's students in my digital audio and interactive media seminar. These students' first language is not English, so they struggle with spelling and capitalization more than English-speaking students would. In a way, that is an ideal test case; these students would be less likely to engage with large blocks of code, and they get more benefit from the isolation of signal processors into separate, short code blocks.

Additionally, `JITModPatch` streamlines the development process for synthesis designs. The automatic GUI in particular and the ability to save patch files are especially useful. I expect this will be part of my workflow for a long time to come.

6. REFERENCES

[1] J. Rohrhuber, A. de Campo, "Just In Time Programming." In: D. Cottle, N. Collins, S. Wilson, eds., *The SuperCollider Book*. MIT Press, Cambridge (MA), 2011.

[2] H. J. Harkins, "JITModular." Github repository. https://github.com/jamshark70/JITModular. Accessed February 2, 2025.

[3] Github user jrsurge, SuperCollider bug report #4413, https://github.com/supercollider/supercollider/issues/4413, accessed February 2, 2025.

[4] Github user jamshark70, SuperCollider bug report #6082, https://github.com/supercollider/supercollider/issues/6082, accessed February 2, 2025.

[5] A. de Campo. "JITLibExtensions." Github repository. https://github.com/supercollider-quarks/JITLibExtensions. Accessed February 2, 2025.