Proceedings of the 19th Linux Audio Conference (LAC-25), Villeurbanne, France, June 26–28, 2025

MFP: A GRAPHICAL PATCHER EXTENSIBLE IN PYTHON AND FAUST

Bill Gribble

grib@billgribble.com

ABSTRACT

MFP is a graphical patching system distinguished by the ready availability of Python data types and operations in patches. Recent work has added support for live coding of message-processing patch elements in Python, and of signal-processing patch elements in Faust. Its extensibility and rich set of message data types give MFP some nice capabilities for interactive patching.

1. INTRODUCTION

My first exposure to live coding for audio was Pure Data (Pd) [1]. As a programmer, I have always loved the read-eval-print loop (REPL) as a way of exploring solutions and incrementally building and running code. Pd's interface immediately connected to the part of my brain that likes REPL development. It's such a great way of experimenting, testing, and, once you get something going, improvising and performing on a patch that just keeps running even when you make mistakes.

The original development of MFP was inspired by the realization that a dataflow patcher's "REPL" really is just code, expressed in an unconventional form. I know this isn't exactly a stroke of genius – one of the mottos of Pd is "the diagram is the program", so it's basically printed on the tin. What was interesting to me was looking at what it means for a graphical patcher to be a programming language. How do the concepts of a conventional programming language map onto the expressive vocabulary of a patch? Are there gaps in the correspondence that can lead to insights towards better patching systems? Can the expressive power of the implementation language give a boost to the patcher? Can the need for real-time performance be reconciled with a rich and dynamic data model?

When regarded as programming languages, dataflow patchers have many aspects that connect them to the programming language "family tree". They are somewhat like functional languages, in that values are returned from functions and marshaled into parameter lists of other functions without resorting to variable names or side effects. Message passing, the main mechanism patchers use for data interchange and code invocation, goes back to Smalltalk and beyond. Patchers support modular programming via the ability to start with simple units and compose them to make more complex behavior. They have a variety of data types, input/output mechanisms, and modes of interaction with the wider system. So it's reasonable to expect patching languages to support a pretty wide variety of usual programming paradigms and patterns.

However, when I started trying to make more complex patches in Pd, especially ones that leaned into the manipulation of non -audio data, I quickly ran into frustrations with the data model that made it quite difficult to do things that should (in my opinion) have been simple. I wondered if breaking out of the space -delimited textual syntax and limited type-vocabulary of Pd objects, and using the data types from a dynamic language such as Python instead, could give a graphical patcher more of the power of a full programming language.

The result of my explorations is MFP. The patch editor and message processing engine are implemented in Python and directly expose the Python data model, library, and evaluator to the user. The real-time audio engine is implemented in C and is loosely coupled to the UI and message-processing system via an RPC mechanism. The overall experience is similar to Pd or Max/MSP, but there has been no attempt at compatibility.

When I presented it at the 2013 edition of LAC [2], MFP was functional enough and interesting enough that I thought it was worth continuing to develop. Now, some 12 years later, it has evolved in various foreseen and unforeseen directions and is, I think, worthy of a re-introduction to the Linux audio community.

2. WHAT IS IT?

My vision for MFP is as a multitool for REPL-style incremental development that can fit in to music composition, performance, and recording workflows wherever some bespoke functionality is needed.

In operation, MFP will look pretty familiar to users of Pd or Max/MSP. See Figure 1 for a sample session. Its interface allows the user to interactively build *patches*, which are sets of directed graphs of *processors*. A processor has a set of inputs, which handle either *messages* (discrete pieces of data of any type) or *signals* (streams of floating-point audio data). It produces outputs which are likewise either messages or signals. The behavior of a processor is determined from a type-name and arguments provided at creation. The type-name can resolve to either a builtin MFP processor type, of which there are over 150, a user-created patch, or an arbitrary Python callable.

In the patch editing window, processors can appear as plain boxes, or can have a variety of visual representations that allow for patch control or feedback: message boxes, number boxes, buttons, sliders, dials, X/Y plots, text with Markdown formatting. Patches used as building blocks by other patches can export a UI of their choice to appear on their placeholder in the containing patch. Direct connections between processor nodes are represented by lines.

Named message and signal buses, comparable to Pd's `[s]` and `[r]`, appear as dots terminating connection lines, called *vias*. Names and scoping are an important part of MFP, and vias represent one of the main user-visible aspects of naming: each via is paired with its companion(s) by a name, visible in the UI, that must be resolved to find the intended connection-partner.

The patch as a whole is modeled as a stack of *layers*, which are something like Pd *subpatches*: they are primarily a visual organizational tool to reduce clutter in the patching interface. One layer of a patch is visible at a time. Multiple patches can be displayed simultaneously, using a multi-page tiled workspace manager.

The nomenclature of *layers* and *vias* comes from electronics design, where a printed circuit board is made up of a multilayer sandwich of conductive and nonconductive material. To me, a dataflow patch looks a bit like a circuit board populated with chips and other components. A *via* is a plated conductive hole, effectively a wire, that connects from one layer through to another. The visual representation of a via as a dot/circle is intended to evoke that.

3. CAPABILITIES

An exhaustive listing of MFP's features would be quite long and probably boring. Instead, I would like to highlight a curated set of features that might be of interest when trying to decide if you want to download MFP and give it a try.

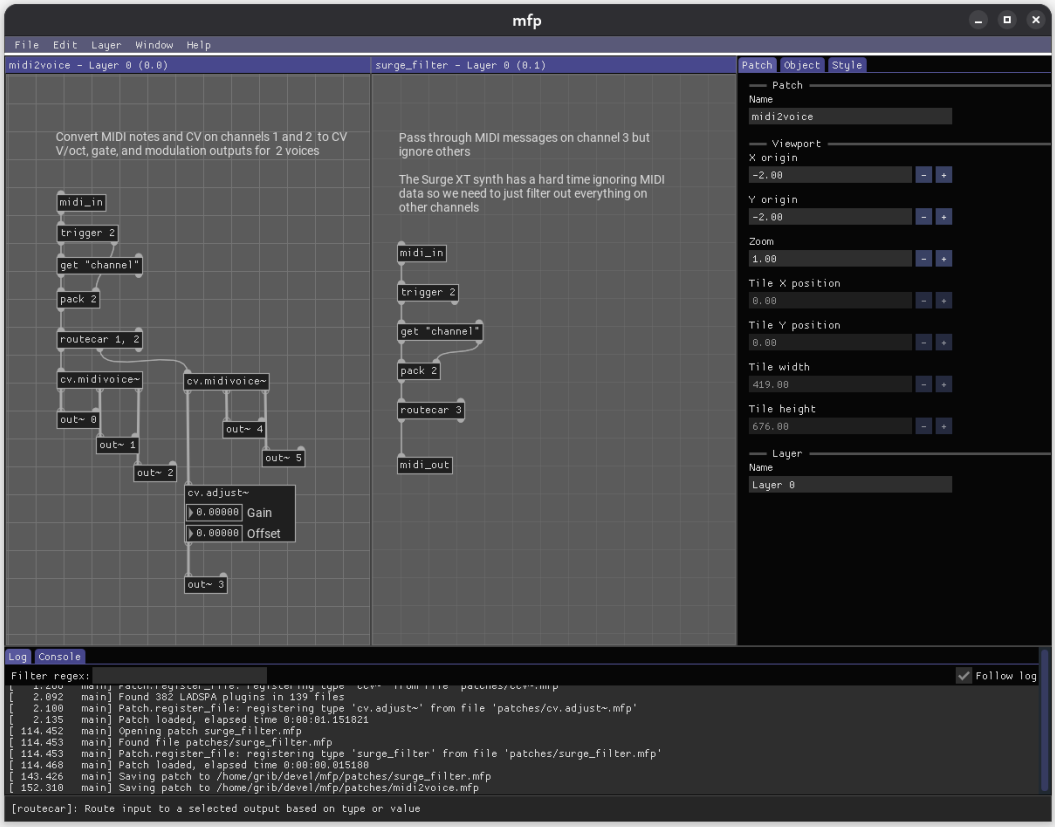

Figure 1: *View of MFP app window, featuring tiled workspace, object info panel on the right, and log window with status/tooltip line below.*

3.1. Data model

Message data is plain Python objects of any type. Numbers, strings, arrays, dictionaries, sets, custom classes, anything. Literal data typed into the UI is evaluated as it would be in a Python REPL. This means that message boxes and processor argument lists can contain Python expressions to be evaluated at object load time. See Figure 2 for some examples.

Partial application syntax. A special syntax allows for literals that represent partially-applied functions. When sent as a message, `@funcname([args], [kwargs])` will be consumed by its first recipient and applied to the recipient as a method call.

Deferred evaluation syntax. Another special syntax extension allows for literals that are not evaluated until they enter the processing network. A initial comma preceding an expression will wrap it in a Python `lambda` form that is not called until the message is sent. This allows for time-varying results from a "literal" message box.

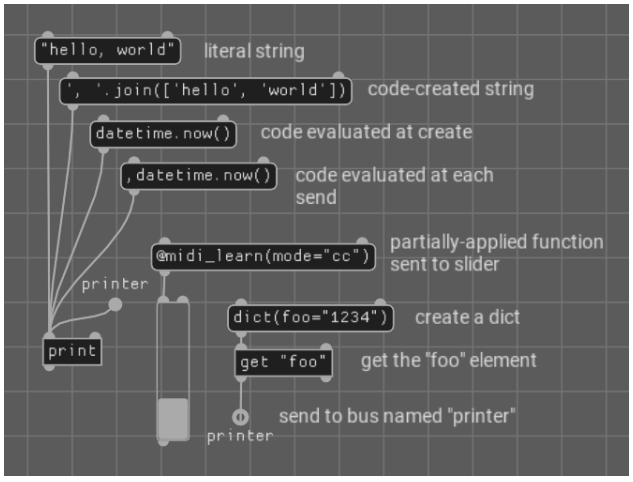

Figure 2: *Patch fragments demonstrating features of the MFP data model, including vias, partial application, and deferred evaluation*

3.2. Python live-coding

The message-passing system at the heart of MFP is implemented in Python, and there are several pathways enabling the user to access the standard Python library and/or write Python code directly into the patching interface.

The `func` processor. `[func]` is a very thin wrapper around Python's `lambda`. It allows you to write a single Python expression directly in a processor box. The arity of the lambda is extracted via Python's `inspect`, and inlets are created for each argument.

Library access. Any callable Python object can be automatically wrapped as a MFP processor element. If you create a processor element called `re.split`, which is not an MFP builtin, it will look for a callable bound to that name. As with `func`, `inspect` is used to find its argument signature and create an object with the appropriate inputs.

Custom processor types. This enables live coding of new message processors in Python. To create a custom processor type called `myfunc`, we can attach the Python code to define a

function called `myfunc` directly to the processor via its "Custom code" parameter, and allow the autowrapping process to discover it.

Offline coding. A user-provided RC file, loaded at startup, can define new data types and builtin processor types using MFP's internal framework.

See Figure 3 for some examples of these mechanisms in use.

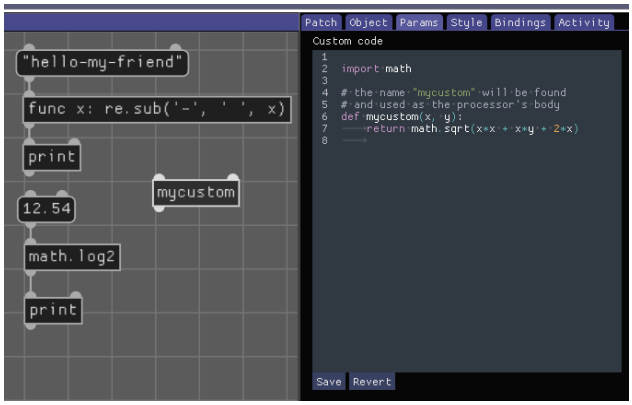

Figure 3: *Several mechanisms for live-coding Python in MFP: [func], custom code, and automatically wrapped Python library functions*

3.3. Faust live-coding

The `[faust~]` object supports live coding of DSP algorithms in Faust. Code can be typed directly into the UI or loaded from files, and can be parameterized or customized via Python at compile time.

This is possible via the "Custom code" parameter mentioned above. The `[faust~]` processor looks for a special Python variable called `faust_code` in the custom code block. This variable can be defined as a constant, literal string (the Python triple-quote syntax works well to support a formatted Faust program in a Python string), it can be the result of Python file operations to load from a named file, or it can be assembled by Python string formatting and manipulation operations.

Once compiled to LLVM within the `mfpdsp` process, the Faust DSP object is inspected to extract parameters and I/O channels so that the processor element's input/output profile matches the signature of the DSP code.

See Figure 4 for how this looks in practice.

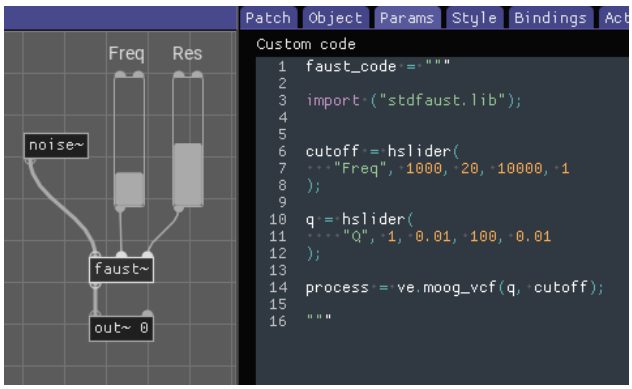

Figure 4: *Live coding in Faust using the* `[faust~]` *object*

3.4. Debugging

Since MFP is still very much a work in progress, a lot of the debugging functionality is built as much to debug MFP as it is to debug user patches, but it's useful for both.

Python console. There is an always-available Python console that can directly access and manipulate the app's data. For example, the default patch that is created when you open MFP with no arguments is `app.patches['default']`

Breakpoints and step execution. MFP supports step debugging of patches with the `[bp]` breakpoint setter. A message encountering a `[bp]` causes the Python console to enter a simple debug mode, inspired by `pdb`, which allows for direct inspection of internal processor state and tracing of the marshalling and dispatch process.

Snoop mode. Enabling "snoop mode" on a connection briefly displays any messages passing through the connection.

UI state inspector. A UI state inspector, modeled after the Redux devtools inspector, is provided by the Flopsy state management library [3]. It allows for detailed analysis of the time sequence of changes to the UI.

3.5. MIDI and OSC

Every element in a patch can be controlled with MIDI or OSC messages. MIDI and OSC events can be easily handled, created, and sent within a patch.

MIDI message types. A set of Python types representing the various flavors of MIDI messages are available to patches: `NoteOn`, `NoteOff`, `MidiCC`, `MidiPitchbend`, etc. These types can be used to filter/route events by type with the `[route]` family processors. Fields are accessible using the `[get]` and `[set!]` processors.

Automatic controls. Every element has an OSC address and can accept OSC messages. The "Bindings" tab of the processor info display shows the currently active OSC routes and data types. OSC messages to the displayed address will be sent to the leftmost inlet of the targetted processor.

Learning controls. To connect an external MIDI controller to a processor, send the message `@midi_learn` to the processor or choose "Learn MIDI controller" from the context menu. The next MIDI events received will be analyzed and a filter and value accessor will be determined from the properties of the event.

Explicit message handling. The builtin processor types `[midi_in]`, `[midi_out]`, `[osc_in]`, and `[osc_out]` enable sending and receiving of MIDI and OSC messages outside the automatic and learned controllers on specific objects. Figure 1 shows some of these processors in action.

3.6. Patches as plugins

An MFP patch can be saved as an LV2 plugin. There are limits on the live editing that can be done (for example, the number of inputs and outputs of the plugin are fixed at save time) but for the most part the patch works as normal.

LV2 plugins exist as "bundles" which are (to simplify) a shared library implementing the plugin, plus metadata, in the form of the `manifest.ttl` file (describing tha plugin) and any additional plugin-specific metadata.

For an MFP patch, the shared library is the `libmfpdsp.so` library that implements the MFP DSP engine, and the plugin-specific metadata includes the MFP patch save file. Since the Python runtime is an essential part of patch loading and operation, we need to make sure that it is loaded on plugin initialization.

Normally, when MFP is launched, the Python "main" process is launched first, and it forks the DSP engine and communicates with it through a socket. The LV2 entry points in `libmfpdsp.so` reverse this process, forking the Python main process (if it is not already running) in a mode that enables it to know what is happening.

Since the Python runtime also manages the MFP UI, the management of UI for the plugin is not currently left for the

LV2 host to control via the normal mechanism of the LV2 UI extension. Rather, an "Edit" control added to the plugin causes the MFP graphical editor to open the patch. This may change in the future.

3.7. Dynamic object and scope creation

When creating a patch as a versatile building block, it may be important to have flexibility to support dynamic numbers of inlets and outlets or other patch resources. For example, there's no right number of inputs for an "audio mixer" patch; it depends on the application how many channels you want to mix.

This could be accomplished by making separate patches for each useful number of channels, like `[mixer4]`, `[mixer8]`, etc. But a nicer way would be to make a single patch called `[mixer]` and create it with the number of channels as an argument, i.e. `[mixer 4]` for a 4-channel mixer. If done properly, the interface could scale up as well, showing a slider or even a full channel strip for each channel.

MFP enables this via a message called `@clonescope` that can be sent to the patch at load time. `@clonescope` makes a specified number of copies of all the objects in a specified *lexical scope*, creating a unique new scope for each set of copies.

Lexical scopes define how the names used to label processors and vias are resolved to actual objects. As in any language, the process of turning a name into a value starts looking in the local scope and becomes more global until the name can be resolved.

Objects in a patch default to their names being in the scope `__patch__`, which is the "global" scope in a patch, but each layer can set a different default scope for objects in itself, and each object can be directly assigned to any other scope via the parameter editor.

With multiple scopes existing in a patch, the same name (of a processor or send/receive via) can refer to different objects depending on the scope in which it is resolved. `@clonescope` uses this to make copies of entire scopes at once. The objects, their interconnections, and send/receive vias can all be duplicated cleanly by just changing the scope name for each object when copying.

A patch wanting to take advantage of this can inspect its own creation arguments, and then send itself a message to duplicate selected parts of its implementation as needed.

See Figure 5 for how this looks in a patch. `togglegrid` is a user-defined patch that creates an NxM grid of toggle buttons, each of which has an inlet and an outlet. The implementation is more than a single screenshot can capture, but the file `doc/togglegrid.mfp` is part of the MFP source distribution and can be opened to inspect how lexical scopes, layers, and `@clonescope` work together to make this happen.

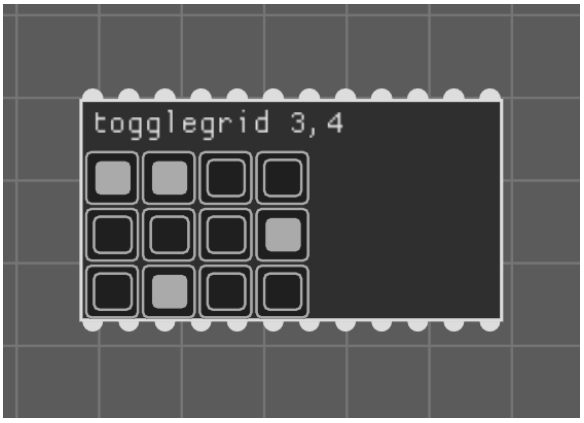

Figure 5: *Dynamic object creation using the* `@clonescope` *message.* `togglegrid` *is a user-defined patch that creates an NxM grid of toggle buttons.*

4. IMPLEMENTATION

4.1. Runtime components

When running, MFP consists of several components working in concert.

mfpmain: A Python program that is the primary execution environment of MFP. It handles the processing of discrete message data, delegating all real-time signal processing to the mfpdsp process.

mfpgui: A Python program that implements the UI. It currently supports 2 backends: the original Clutter/Gtk UI and a newer one built with Dear ImGui.

mfpdsp: A C program that implements the DSP engine and JACK client interface.

These three are loosely coupled, primarily communicating via an asynchronous RPC mechanism implemented in the Carp [4] package. To enable the special case of displaying signals and other rapidly-changing values in the UI, shared memory buffers are used by the `[buffer~]` object to shortcut directly from the DSP engine to the UI.

Each runtime component implements its own representation of the patch's graph of processing nodes, so there are "parallel" representations of each patch built in the UI process, in the main process, and in the DSP engine, using representations specific to that process.

The DSP and UI processes are launched by default but may be suppressed with command line options. A no-DSP, no-UI "batch mode" is available to run a patch with input from `stdin` and write resulting objects to `stdout`.

Much of the functionality of mfpdsp is built in to a shared library called libmfpdsp. This library implements the LV2 plugin API, which allows MFP to be used as a plugin in any LV2-enabled host.

4.2. User interface

As of version 0.8, the default user interface for MFP is built on the Dear ImGUI library [5], via the ImGUI Bundle bindings [6]. The move to ImGUI from the previous implementation in Gtk/-Clutter was somewhat painful, but worth it. ImGUI's flexibility and the rich ecosystem of extensions and addons made things like enabling Markdown in comments and making more sophisticated X/Y plots relatively easy.

4.3. Help system

MFP's help system is modeled after Pd's, with builtin processor types providing "help patches" that describe and demonstrate their functionality. Help patches are not yet available for most MFP builtins, but I am working on it.

In addition to the per-processor help (available in the context menu), there are some global help patches available under the app's Help menu. A Tutorial patch walks through the basics of creating a patch, and a Reference patch attempts to organize information about the builtins and how to use them.

5. STATUS

The first public release of MFP, 0.1, was right about the time that the submission window for LAC-2013 closed. The most recent release, 0.8, is happening right around the close of the submission window for LAC-2025. Funny how that works!

A significant amount of the recent work on MFP has been in getting the Dear ImGUI UI up to parity with the Clutter UI. A lot of new features have been added along the way, but they were not the primary focus. Going forward, I would like to spend my time on (1) stabilizing existing features and (2) implementing items from the priorities list below.

I use MFP for my own music making and I think it's in good shape for others to use as well. As far as testing its capabilities to do more general-purpose programming, I have tackled the Advent of Code [7] challenge for the last 2 years, creating MFP patches for my solutions [8]. While I have yet to make it all the way through a month of puzzles, I blame myself and my holiday travel plans more than MFP's capability as a programming environment.

I have been fortunate to have the time and energy to work on MFP on a regular basis recently, and I hope to continue to do so. There's not really a significant user community at this point, but only you can change that!

5.1. Development priorities

To reach what I would consider to be a "1.0" level of functionality for MFP, I would like to implement most or all of the following:

"mfp4ardour". Ardour has a deep OSC API, and building good wrappers (perhaps modeled after the way max4live does it) could make Ardour + MFP a great combo.

Expanded UI functionality. The table widget, improved graphing options, simpler handling of oscilloscope displays, better connection routing and manual routing support, themeability, loading images and textures...

JACK MIDI. For sample-accurate MIDI event processing and more capable LV2 implementation, it's necessary to build a library of builtin functions in the DSP domain to deal with JACK MIDI data.

LV2 plugin hosting. LV2 is the current Linux standard for plugins. The current LADSPA-only `[plugin~]` is inadequate.

Spectral processing with libfft. Embedded Faust gets a lot of great functionality, but does not enable working in the frequency domain. New DSP builtins will be needed for this.

Improved file format. I would like to have a save format that includes resources such as sample data. I have done some preliminary work on using SQLite as a storage backend.

Multirate/variable-blocksize DSP operations. Antialiasing, timestretch, and many other DSP algorithms require oversampling, decimation, and other primitives that are best implemented using variable sized blocks of data, at least internally. Currently MFP deals only in blocks that are the size of the JACK buffer.

Multichannel connections. Recent versions of Max/MSP support multiple channels on single "wires" for easier multivoice patching. I think this should be a feasible extension to MFP.

5.2. Getting MFP

Source code and issue tracking for MFP are on GitHub:

`https://www.github.com/bgribble/mfp`

The project is licensed under the GNU General Public License (GPL) version 2. Your interest and participation is invited and welcomed.

6. REFERENCES

[1] M. Puckette, "Pure Data," `http://www.puredata.info`, 1997.

[2] B. Gribble, "Music for programmers (mfp): A dataflow patching language," *Proceedings of Linux Audio Conference 2013*, 2013.

[3] B. Gribble, "Flopsy," `http://github.com/bgribble/flopsy`, 2023.

[4] B. Gribble, "Carp," `http://github.com/bgribble/carp`, 2022.

[5] O. Cornut, "Dear ImGUI," `http://www.github.com/ocornut/imgui`, 2014.

[6] P. Thomet, "ImGUI Bundle," `http://www.github.com/pthom/imgui_bundle`, 2022.

[7] E. Wastl, "Advent of code," `http://www.adventofcode.com`, 2015.

[8] B. Gribble, "Advent of Code solutions," `http://github.com/bgribble/advent_of_code`, 2023.

[9] M. Puckette, "Max/MSP, currently distributed by Cycling '74," `http://www.cycling74.com/max`, 1989.