JAMOMA: A MODULAR STANDARD FOR STRUCTURING PATCHES IN MAX

Tim Place* and Trond Lossius†

*Cycling '74, Electrotap LLC, University of Missouri – Kansas City
tim@electrotap.com
†Bergen National Academy of the Arts, Department of Fine Arts
trond.lossius@khib.no

Abstract

Jamoma is a system for developing high-level modules in the Max/MSP/Jitter environment. Jamoma consists of two parts: A recommendation and an implementation of that recommendation. Jamoma offers a compelling set of benefits to users. These benefits include fast and flexible interchange of modules, patch-building, and module construction, as well as possibilities of advanced control of the modules in performance. Jamoma modules may encapsulate any type of functionality that can be performed by Max, MSP, Jitter, its components (such as Java or JavaScript), or any third-party objects.

1 Introduction

Max/MSP is a flexible environment that has become something of a *lingua franca* for practitioners of computer-based live performance. A wide range of libraries of low-level externals and abstractions are available for this environment. Comparatively few high-level patches are shared within the community, and they generally seem to be more difficult to share. A possible explanation might be that Max imposes no particular way of working on its users. Indeed, the metaphor for what it provides users is a blank page (Zicarelli, 2002). Patching-style can be highly idiosyncratic, making it difficult to merge patches by different authors into a cohesive system. The general lack of consistency in how patches are automated or controlled also means that users of shared high-level patches must thoroughly learn many systems to use patches created by others. In the words of David Zicarelli, "Max/MSP does not enforce readability, consistency, or efficiency on its users. There are no real standards for interoperability at the level of the patcher..."

The first goal of the Jamoma project is to address concerns of sharing and exchanging Max patches in a modular system. This means creating a structured framework that does enforce consistency, readability, and standards for interoperability while not placing daunting restrictions on users. Zicarelli states, "A system in which components talk to each other in a standard way allows for the sharing of expertise among people working on a variety of interests and with varying levels of expertise."

The second goal of Jamoma is to leverage this structured environment for effective, efficient, and powerful means of automating and controlling Max patches. The Jamoma framework has created a wide variety of possible interactions for automated control without enforcing any particular paradigm to which a user must conform.

A number of similar initiatives have been introduced in the past. One of these is the Pluggo architecture, which provides a way of encapsulating Max/MSP patches as plug-ins for general audio software. Other modular environments typically serve a specific purpose or setting. These include Lloopp[1], Framework[2], UBC Max/MSP/Jitter Toolbox[3], and the Jade[4] application – extractions distilled from Jade served as the origin of the Jamoma project. These environments generally do not share all of the properties of Jamoma, namely that it is open-source, free, fully modular, and generic use (not limited to only audio, only video, etc.).

2 Jamoma Features

2.1 JIG: Jamoma Interface Guide

The Jamoma Interface Guide is a recommendation for common issues in Max patches regarding construction, state handling and interfacing. The recommendation ensures inter-operability of modules so that they might be reused, exchanged and documented in a generic way. At the same time the guideline is designed to be open-ended and extendable so that it does not introduce restrictions concerning what tasks modules might possibly perform as compared to the underlying structure of MaxMSP/Jitter itself. Jamoma strives to maintain the flexibility characteristic of new media as defined by Manovich (2001).

Jamoma is modular. All parts of the Jamoma system are implemented as modules, self-contained components of a

[1] http://lloopp.klingt.org/start.html
[2] http://www.leafcutter.33-rpm.net/Downloadframework.htm
[3] http://www.opusonemusic.net/muset/toolbox.html
[4] http://www.electrotap.com/jade/

system, with a well-defined interface to the other components.

The leftmost inlet and outlet of any module is reserved for communication of control information to and from the module using the Open Sound Control protocol (Wright and Freed, 1997). Modules are expected to maintain a record of their current state, and this state can be queried and stored. Modules may define parameters and messages. Parameters alter the state of the module. Arguments passed can be of any type permitted by the Max environment. If a parameter expects a floating point or integer argument, a range can be defined where upper and/or lower limits enforced. In addition, the parameter may be set to ramp to new values over a period of time given as a second optional argument. At this time only linear ramping is supported; more ramping modes will be implemented in the future. Messages behave the same as parameters, except that they are stateless.

Optional additional inlets and outlets are used for audio signals or passing video as Jitter matrixes. This can easily be expanded to include other types of data if required. The Jamoma specification poses no restrictions on the number of inlets or outlets used for signals, and neither is it required that the number of inlets and outlets are identical. A module might mix various types of signals, though no such modules are implemented in the current distribution.

It is recommended that all modules implement certain standard messages and parameters to ensure that common tasks are dealt with consistently. Additional standard parameters and messages are recommended for audio and video modules respectively. These will be further described in the following sections.

Jamoma specifies the user interface to have fixed dimensions based on the paradigm of conventional rack mount hardware. The standard size of a module is 510x60 pixels. Modules might be several rack units tall, and/or a half rack unit wide. The width of 510 pixels was chosen to maximize the number of modules that can fit on a monitor with a resolution 1024 pixels wide, encouraging efficient and consistent use of screen real estate. While this may seem like an unnecessary restriction on first glance, the enforcement of this lattice vastly simplifies organization of modules on the screen for efficient use by both automated and human processes.

2.2 Jamoma Modules

A number of externals and abstractions (Max patches made to behave like externals) have been developed to facilitate the development of modules. In Jamoma terminology these are called components.

Figure 1 shows jmod.hub, the central brain of the module. jmod.hub maintains all control communication to and from the module, as well as communication to and from other components in the module. It also keeps track of the state of the module, working in tandem with pattrstorage, part of the new regime offered as part of Max/MSP 4.5 for improved state handling. Arguments to jmod.hub are required for setting up local wireless communication of messages within the module, describing the name and size of the module, the number of signal inlets and outlets, what kind of data the module is dealing with, and a description of the module. jmod.hub also has the ability to auto-generate HTML documentation for the module.

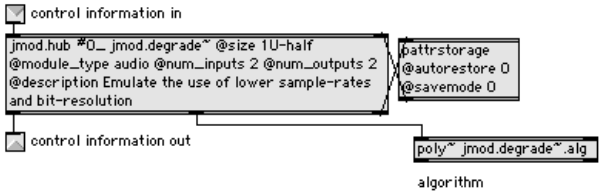

Figure 1. jmod.hub communicates with the outside world, handles internal state, and controls the logical algorithm.

Currently three basic types of modules are implemented, dealing with control data, audio and video respectively. Other types of modules might be implemented in the future.

Each module is made up of three functional elements.

Graphical User Interface (GUI). The graphical user interface provides visual feedback and interaction with the module. The interface aims to provide access to as many parameters of the module as possible, while remaining efficient in terms of screen usage.

The jmod.gui component, loaded as a bpatcher, forms the backdrop of the GUI section. Arguments to jmod.hub set the size of jmod.gui, as well as the look, depending on the kind of data the module is processing. Common tasks of modules are implemented as part of jmod.gui. The GUI uses a skinnable system to allow for customization of the look and feel of a module.

For audio modules an optional widget offers monitoring of output levels as well as the ability to mix dry and wet signals, mute or bypass the module and change internal sampling rate, as illustrated in Figure 2.

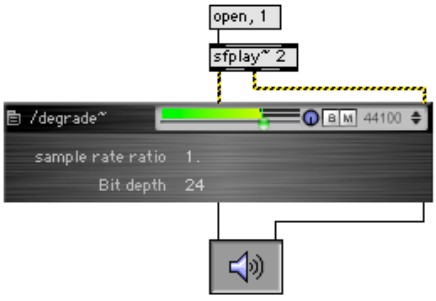

Figure 2: The jmod.degrade~.mod emulates use of lower sample- and bitrate.

Figure 3 illustrates the pop-up menu shared by all modules. The content of the menu varies depending on the type of module. For all modules, screen updates of the user

interface elements can be disabled to conserve CPU cycles. HTML documentation of the module can be accessed and the algorithm (the internal logic of the module) may be viewed in a separate window. Current state can be saved as an XML preset file, previously saved presets may be loaded, or the default preset recalled. For video modules additional menu items offer possibilities of bypassing, muting or freezing the video processing.

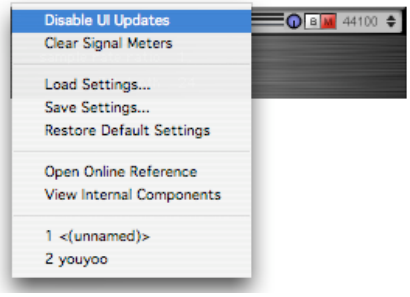

Figure 3: Several common messages and parameters can be accessed from the pop-up menu.

The main section of the GUI is reserved for graphical user interface objects for interacting with various parameters and messages specific to the module. The design of this part of the GUI is left to the whim of the module creator.

Parameter Handling. The two components jmod.parameter and jmod.message are used to define what parameters and messages the module accepts. They also deal with any boundaries on range and type of argument, as well as the ramping mechanism. The components might be connected to graphical user interface objects for user interaction and feedback. They connect wirelessly to jmod.hub for communication with the rest of the patch, and the outside world.

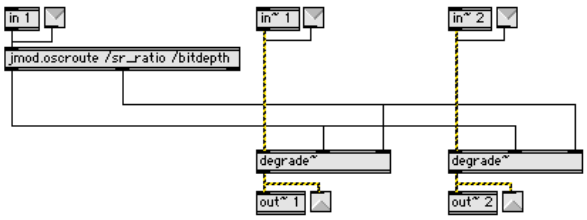

Figure 4. A stereo algorithm for audio degrading.

Algorithm. The logical task of the module is generally implemented as a subpatch and saved separately. Figure 4 shows one such subpatch, or algorithm, for a simple audio processing module. The algorithm shares several conventions with the module. The leftmost inlet and outlet is used for communicating control data to and from the algorithm using Open Sound Control messages. Additional inlets and outlets are used for passing audio and video signals to and from the algorithm. Audio algorithms can be loaded in the module using the poly~ external, incorporating the possibility of several parallel instances as well as the capability for muting of the algorithm, or downsampling it, to save CPU cycles. In this case the leftmost inlet will be a shared control-data and signal inlet. The algorithm differs as compared to a module by being stateless and not offering any graphical user interface.

Documentation. For all modules distributed as part of Jamoma, HTML documentation is provided as well as help patches illustrating the use of the module. There are help patches for custom externals and components. Tutorials are provided on how to build modules, and templates for modules are provided to simplify the process.

3. Examples of Usage

3.1 Controlling Jamoma Modules

Modules might be controlled in a number of different ways. A system for remote communication enables a set of control modules for controlling other modules. Several such modules are implemented. jmod.cuelist.mod loads a text-based script of event cues, and is able to control all modules provided that they have been given unique names via Max's script-name inspector. The cues can be executed in arbitrary order. A WAIT syntax can set the execution of a cue on hold for a specified amount of time, opening up the possibility for scripting of complex events evolving over time. The current state of all modules can be queried, and used to create new cues. Figure 5 provides a simple example.

```
##############################
CUE sweep
##############################

# Module filter~
     /filter~/cf 3000 3000
     WAIT 6000
     /filter~/cf 200 3000
```

Figure 5. A cuescript. The center frequency of the filter~ module ramps to 3000 Hz over 3 seconds, holds for 3 seconds, and ramps down to 200 Hz over 3 seconds.

Other control modules exist for mapping of parameters between different modules, and for enabling network communication using Open Sound Control. Such modules are able to control themselves as well, offering possibilities for complex recursive generative systems.

3.2 Various approaches to using Jamoma

The Jamoma distribution includes a number of modules, and more modules will be added in the future to form a library targeting a number of specific and common tasks for audio and video processing.

For experienced users of Max/MSP/Jitter the structure itself and the possibilities for control offered may be more interesting than the included modules themselves. In performance Jamoma offers efficient use of screen estate, a

standardized way of dealing with presets and parameter handling, and simplifies complex handling of parameters. The authors have found that modules ported to Jamoma tend to maintain more of the possible parameters available for interaction than when working with unstructured Max patchers, resulting in richer expressive possibilities. In large projects involving several artists or programmers Jamoma offers a standardized framework to simplify collaborative development. As Jamoma uses Open Sound Control it is simple to extend the system to large projects running on a network of connected computers.

For newcomers to Max/MSP/Jitter Jamoma can simplify the process of structuring patches so that they become useful and reliable. It is the authors' hope that the Jamoma structure will encourage reuse and exchange of modules within the Max community.

In the case that one does not want to adapt to the modular structure of Jamoma, the underlying algorithms might still prove useful as lower-level building blocks. The guidelines might be useful on their own, and could be extended/adjusted to work with other programs for audio processing. This could eventually be extended into a partly standardized OSC namespace.

3.3 Jamoma modules as plugins

To demonstrate the flexibility of Jamoma, the official distributions contain two examples of how to use Jamoma modules in Pluggo-based[5] plug-ins. Pluggo provides a system whereby Max/MSP patches can be transformed into plug-ins usable in any VST, AudioUnit, or Pro-Tools host. By using Pluggo's objects together with Jamoma it is possible to quickly create complex and powerful plug-ins for music production environments.

3.4 Obtaining Jamoma

As of this writing Jamoma version 0.3.1 is publicly available for Windows XP and Mac OSX Universal Binary from the Jamoma web site[6]. In spite of the low version number, extensive development has been carried out. The development of the Jamoma specification and implementation has, to a large degree, been informed by concerts, performances and installations by the authors and others. Jamoma has proven a stable tool expanding artistic flexibility and possibilities.

Jamoma is licensed under the terms of the GNU Lesser General Public License[7].

4. Further Development

Further development on the kernel will focus primarily on performance improvements, possibly by porting many patcher components to C-based externals. The support of additional parameter ramping modes will allow both more flexibility and better performance. In addition support is planned for non-real-time rendering of audio and video.

Remaining improvements will be focused on the module-level. For control modules this includes additional automation facilities and paradigms, and different approaches to handling time-based works. Current audio module development includes work on spatialisation, effect processing, and synthesis facilities. Modules for spatialisation will offer access to several different techniques, including vector based amplitude panning (Pulkki, 2000) and ambisonics (Schacher and Kocher, 2006). Video module development is currently focused on analysis of gestures, with The Musical Gestures Toolbox (Jensenius, Godøy and Wanderley, 2005) being ported to Jamoma.

There is an ongoing dialog with the Integra[8] project led by UCE Birmingham Conservatoire to ensure compatibility between the two projects.

5 Acknowledgments

The Jamoma project wishes to thank Electrotap for providing the initial resources for the project and for open sourcing software required by Jamoma. Development has been sponsored in part by Bergen National Academy of the Arts as part of a research fellowship in the arts. The jamoma.org website is hosted by Bergen Center for Electronic Arts. Additional support has been provided by Alexander Refsum Jensenius, Jeremy Bernstein at Cycling '74, and the members of the Jamoma SourceForge project.

References

Jensenius, A. R., R. I. Godøy and M. M. Wanderley. 2005. Developing tools for studying musical gestures within the Max/Msp/Jitter environment. In *Proceedings of the International Computer Music Conference 2005,* pp. 282-285. Barcelone, Spain: International Computer Music Association.

Manovich, L.,2001. *The language of new media*. Cambridge, Massachusetts: MIT Press.

Pulkki, V, 2000: Generic panning tools for MAX/MSP. In *Proceedings of International Computer Music Conference 2000.* pp. 304-307, Berlin, Germany: International Computer Music Association.

Schacher, J. C. and P. Kocher, 2006: Ambisonics Spatialization Tools for Max/MSP. In *Proceedings of the International Computer Music Conference 2006*, New Orleans, US: International Computer Music Association.

Wright M. and A. Freed. 1997. Open sound control: A new protocol for communicating with sound synthesizers. In *Proceedings of the International Computer Music Conference 1997*, pages 101–104, Thessaloniki, Greece: International Computer Music Association.

Zicarelli, D. 2002. How I Learned to Love a Program That Does Nothing. *Computer Music Journal* 26(4), 44-51.

[5] http://www.cycling74.com/products/pluggo
[6] http://www.jamoma.org
[7] http://www.gnu.org/copyleft/lesser.html
[8] http://www.integralive.org