

MaxPy: An open-source Python package for programmatic construction and manipulation of MaxMSP patches

Ranger Liu
Parsons School of Design
66 5th Avenue
New York, NY 10011
rangerliu@newschool.edu

Satchel Peterson
Columbia University
70 Morningside Drive
New York, NY 10027
sp3914@columbia.edu

Richard Lee
Columbia University
2960 Broadway
New York, NY 10027
rtl2118@columbia.edu

Mark Santolucito
Barnard College
3009 Broadway
New York, NY 10027
msantolu@barnard.edu

ABSTRACT

MaxMSP is a visual programming language for creating interactive audiovisual media that has found great success as a flexible and accessible option for computer music. However, the visual interface requires manual object placement and connection, which can be inefficient. Automated patch editing is possible either by visual programming with the [thispatcher] object or text-based programming with the [js] object. However, these objects cannot automatically create and save new patches, and they operate at *run-time* only, requiring live input to trigger patch construction. There is no solution for automated creation of multiple patches at *compile-time*, such that the constructed patches do not contain their own constructors. To this end, we present MaxPy, an open-source Python package for programmatic construction and manipulation of MaxMSP patches. MaxPy replaces the manual actions of placing objects, connecting patchcords, and saving patch files with text-based Python functions, thus enabling dynamic, procedural, high-volume patch generation at *compile-time*. MaxPy also includes the ability to import existing patches, allowing users to move freely between text-based Python programming and visual programming with the Max GUI. MaxPy enables composers, programmers, and creators to explore expanded possibilities for complex, dynamic, and algorithmic patch construction through text-based Python programming of MaxMSP.

Author Keywords

MaxMSP, visual languages, automation, generation, metaprogramming

CCS Concepts

•Applied computing → Sound and music computing;

1. INTRODUCTION

Programming languages have long held foundational importance in sound synthesis and audio processing, driving the frontier of computer music. Both text-based languages like CSound [2] and SuperCollider [7] as well as visual programming languages like PureData [8] and MaxMSP [1] have given computer musicians many options for writing code. In particular, visual languages have allowed non-programmers to create complex computational artifacts with relative ease due to their accessible and intuitive interfaces.

However, the main drawback of visual languages is the amount of manual interaction required to create a program. In MaxMSP, objects and patchcords must be manually placed and connected, which becomes inefficient for large or complex patches. These manual actions can be automated with the [thispatcher] object, which takes message inputs to create, destroy, connect, or disconnect objects within the current patch. The [js] object also provides text-based programming control over [thispatcher] functionality.

These automations can be considered a limited form of *metaprogramming*, which refers to languages that enable the programmatic manipulation of other programs. In this case, Javascript is used to manipulate MaxMSP code, or MaxMSP code itself is used to manipulate MaxMSP code. However, neither [js] nor [thispatcher] can be used to create or save new patches, and neither allows for *compile-time* behavior that would enable patch construction to occur outside of MaxMSP.

In this paper, we introduce MaxPy, an open-source tool for true metaprogramming of MaxMSP using Python, enabling compile-time creation, editing, and saving of Max patches. Metaprogramming has found enormous success in traditional programming contexts, including C++11's `constexpr` [5], Template Haskell [9], and Scala macros [3]. Following the success of metaprogramming in other domains [6], we believe that MaxPy will empower new modes of interaction with visual programming languages for human-centered audio programming and computer-aided composition.

2. MOTIVATING EXAMPLE

We first consider a motivating example of generating patches from chess data. Given the Forsyth-Edwards Notation (FEN) for a particular board configuration, we want to 1) place an abstraction for each piece in play, 2) input their color and position coordinates as messages, and 3) connect their outputs to an [~ezdac] (Fig. 1).



Licensed under a Creative Commons Attribution 4.0 International License (CC BY 4.0). Copyright remains with the author(s).

NIME'23, 31 May–2 June, 2023, Mexico City, Mexico.

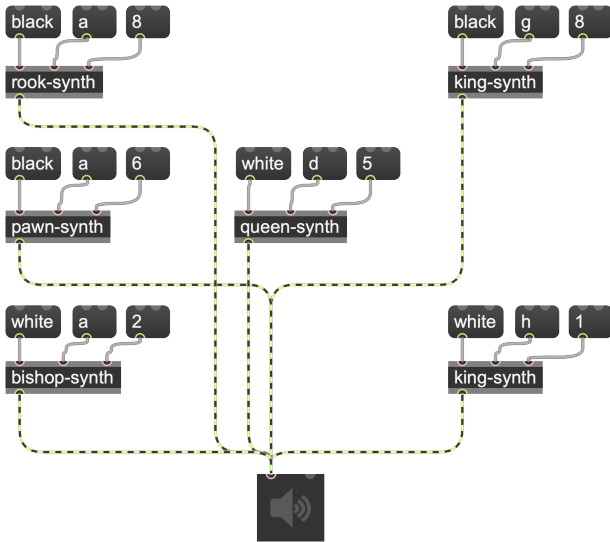
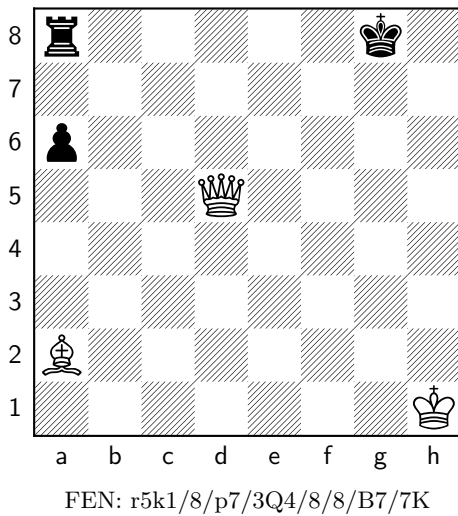


Figure 1: Example of chess-to-maxpatch generation.

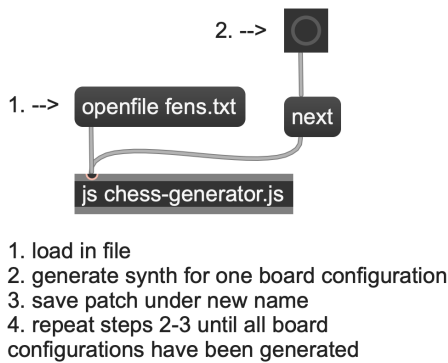


Figure 2: Single-patch generation from chess data using [js]. This patch must be manually saved after each iteration.

Manually creating the patch shown in Fig. 1 is not difficult. However, generating new patches based on different configurations becomes challenging, as MaxMSP lacks automated control over creating and saving patch files. Parsing FEN strings, placing objects, and connecting patchcords can all be automated with [thispatcher] or [js], but saving each generated patch as a separate file requires manual input (Fig. 2). This tedious process scales poorly at high volumes: imagine generating a different patch for every state of every game in the Games of the Day Archive [4] from the past year. Without metaprogramming, such a task would be insurmountable. MaxPy thus offers an efficient solution for automated patch generation, especially at high volumes¹.

3. MAXPY USAGE

This section gives a basic, non-comprehensive overview of MaxPy usage. Most recent code and documentation can be found at <https://github.com/Barnard-PL-Labs/MaxPy>.

3.1 Creating An Empty Patch

To use MaxPy, we first import the library and instantiate a MaxPatch.

```
1 import maxpy as mp
2
3 patch = mp.MaxPatch()
```

When given no arguments, the default MaxPatch constructor creates an empty patch.

3.2 Creating and Placing Objects

We can use MaxPatch.place() to create and place multiple objects at once:

```
1 counter, button = patch.place(["counter",
2                               "button"])
```

Alternatively, we can first create a "floating" MaxObject and then place it in the MaxPatch:

```
1 counter_obj = mp.MaxObject("counter")
2 button_obj = mp.MaxObject("button")
3 patch.place([counter_obj, button_obj])
```

MaxPy currently supports all max/msp/jit objects from a vanilla MaxMSP installation. External MaxMSP package support is discussed in Sec. 5.

3.3 Object Arguments and Attributes

Arguments and attributes are specified in the object creation string:

```
1 patch.place(["counter 4 @carryflag 1"])
2
3 #or...
4 counter_obj = mp.MaxObject("counter 4
5                               @carryflag 1")
6 patch.place([counter_obj])
```

Common box attributes are specified during object creation:

```
1 counter_obj = mp.MaxObject("counter",
2                               ignoreclick=1)
```

¹Code for the [js] and MaxPy implementations can be found at <https://github.com/Barnard-PL-Labs/MaxPy/tree/main/examples/chess-paper-example>.

3.4 Connecting Patchcords

Each MaxObject contains `MaxObject.ins`, a list of Inlets, and `MaxObject.outs`, a list of Outlets, both numbered left-to-right starting from 0. We connect patchcords by specifying (Outlet, Inlet) pairs:

```
1 outlet = button_obj.outs[0]
2 inlet = counter_obj.ins[0]
3 patch.connect( (outlet, inlet) )
```

3.5 Saving Patches

Patches can be saved to the `.maxpat` format by specifying a file name:

```
1 patch.save("my.maxpat")
```

3.6 Loading Existing Files

Instead of instantiating an empty patch, MaxPy can also load in an existing `.maxpat` file:

```
1 patch = mp.MaxPatch(load_file="my.maxpat")
2
```

This loaded patch can be manipulated like any MaxPy-generated patch, allowing users to easily switch between the Max GUI and MaxPy.

3.7 Abstractions and .js Files

MaxPy will locate and link abstraction files and `.js` files as long as they are saved in the current directory.

```
1 abstraction = mp.MaxObject("my-abs")
2 linked_js = mp.MaxObject("js script.js")
```

4. CASE STUDIES

4.1 Quantum Audiovisualizer

This data sonification project shows MaxPy's ability to dynamically generate patches based on data processed with external Python libraries².

Given a number of quantum circuit components, MaxPy creates one oscillator per component (Fig. 3). These oscillators are symmetrically detuned and panned, with exact amounts depending on the total number of components (Fig. 4). MaxPy saves these custom oscillator blocks as individual patch files and places them inside the main synth patch as abstractions.

The quantum circuit is then simulated and the results are used to generate a sequence of notes for five synth voices. MaxPy is used to create and place a sequencer for each synth voice (Fig. 5). Python then uses OSC messages to trigger each sequencer.

In creating the synth and sequencers, MaxPy leverages Python's data-processing capabilities to calculate detune and pan amounts, convert quantum data into a sequence of notes, and parse the sequence into separate voice tracks. The entire project hinges on MaxPy's ability to easily integrate with Python's `quiskit` package, which enables quantum data to be used directly. As exemplified in this project, MaxPy's access to Pythonic data processing creates enormous potential for data-driven sonification and visualization.

²Code for this project can be found at <https://github.com/ryurongliu/quantum-audiovisualizer/blob/main/quantfinal.py>.

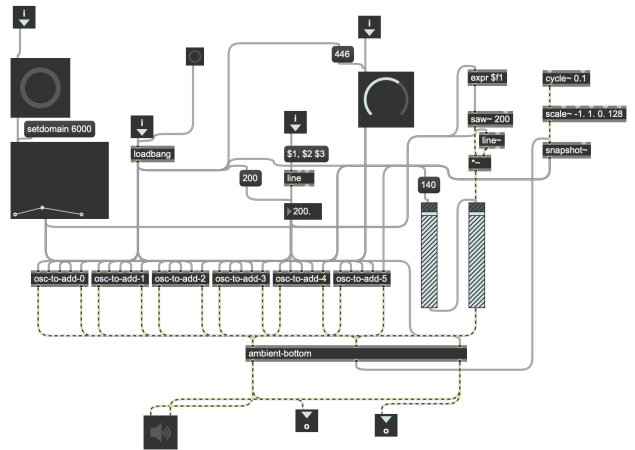


Figure 3: MaxPy-generated additive synth with a variable number of added oscillators.

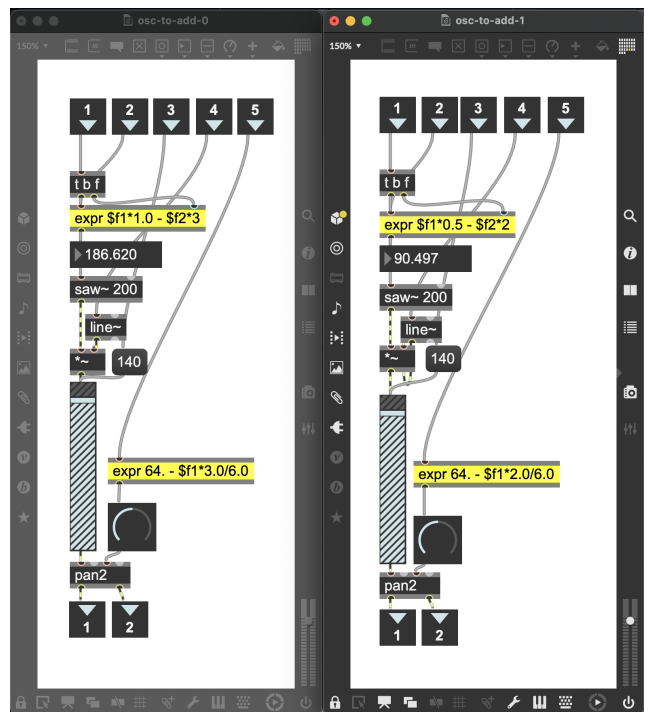


Figure 4: A comparison of two MaxPy-generated oscillators for the additive synth. The highlighted [expr] objects are uniquely calculated for each oscillator.

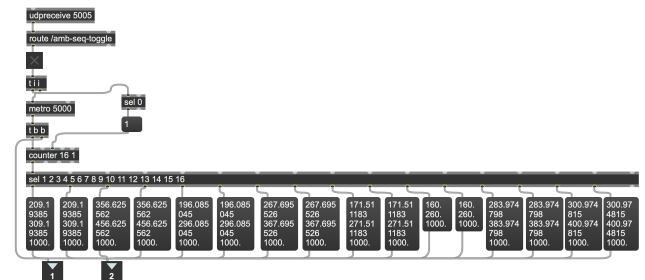


Figure 5: A MaxPy-generated sequencer with hardcoded frequency and duration amounts.

4.2 you are god

This experiential music game uses MaxPy as a bridge between Processing and MaxMSP to automatically edit an open patch, based on user input to Processing.

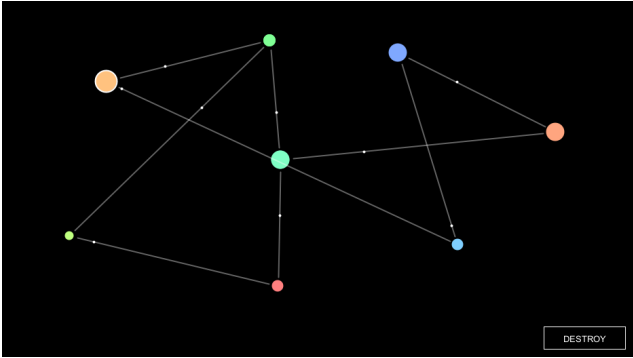


Figure 6: Screenshot from you are god.

In the game, the user places a series of circles and connects them with lines. Small dots then bounce back and forth along these lines, triggering each circle’s unique sound on impact (Fig. 6).

The patch generation for this game is fairly simple: create one synth voice for each dot, and connect them to trigger inputs and a signal output³.

Python is used as a bridge between Processing and MaxMSP (Fig. 7). Processing first uses OSC to communicate the number of dots to Python. Python then uses MaxPy to create that number of synth voices in a MaxPatch (Fig. 8). Finally, Processing uses OSC to trigger these newly loaded voices in MaxMSP.

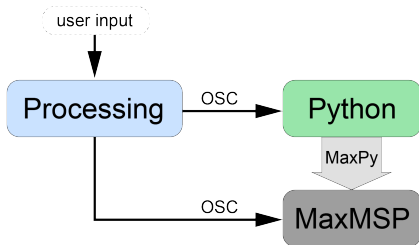


Figure 7: Diagram of dataflow within you are god.

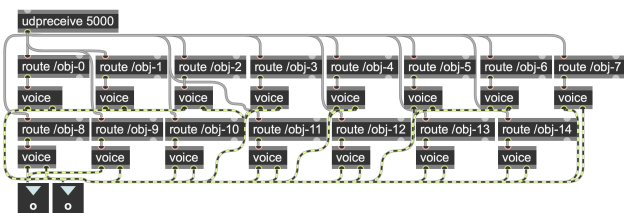


Figure 8: The MaxPy-generated abstraction [yag-voices], containing a variable number of voices connected to OSC triggers.

This game is a simple example of MaxPy’s possibilities as a bridge between MaxMSP and other input sources. MaxPy allows data processing and patch generation to occur outside of both MaxMSP and the input method, freeing up

³The MaxPy code can be found at <https://github.com/ryurongliu/you-are-god/blob/main/yag.ipynb>.

computational load on both ends. This is particularly useful for interfacing with embedded systems that have limited on-board memory but require significant data processing.

5. CONCLUSION

In this work we have presented MaxPy, an open-source Python package for metaprogramming in MaxMSP. In addition to the promising results so far, our initial experiments with MaxPy have yielded a number of areas for future work.

One of the attractions of MaxMSP is the ability for external developers to create their own Max objects. MaxPy can support these external objects, but the current importation process is inefficient. We are developing a streamlined way to import objects that will allow MaxPy contributors to add MaxPy support for external packages.

Another open issue is typechecking in MaxPy, which would check for illegal patchcords attempting to bridge incompatible datatypes. This would enable several potential applications of MaxPy, such as the generation of random patches or the creation of an NLP patch generator.

Finally, proper typechecking in MaxPy will likely stem from (or lead to) a definition of the formal semantics of MaxMSP. Unfortunately, the wide variety of MaxMSP objects makes it difficult to pin down such a definition, particularly when considering externally developed object packages that use custom datatypes. Defining a formal semantics of MaxMSP remains an ongoing effort.

6. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CCF-2105208.

7. REFERENCES

- [1] Cycling. ’74. MaxMSP product page. <https://cycling74.com/products/max>, 2023. accessed 1-27-2023.
- [2] R. Boulanger. *The Csound book: perspectives in software synthesis, sound design, signal processing, and programming*. MIT press, 2000.
- [3] E. Burmako and M. Odersky. Scala macros, a technical report. In *Third International Valentin Turchin Workshop on Metacomputation*, number CONF, 2012.
- [4] C. Games. Games of the day archive. <https://www.chessgames.com/perl/gamesoftheday>, 2023. accessed 1-27-2023.
- [5] N. M. Josuttis. *The c++ standard library: a tutorial and reference*. 2012.
- [6] Y. Lilis and A. Savidis. A survey of metaprogramming languages. *ACM Computing Surveys (CSUR)*, 52(6):1–39, 2019.
- [7] J. McCartney. Rethinking the computer music language: Super collider. *Computer Music Journal*, 26(4):61–68, 2002.
- [8] M. Puckette et al. Pure data: another integrated computer music environment. *Proceedings of the second intercollege computer music concerts*, pages 37–41, 1996.
- [9] T. Sheard and S. P. Jones. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16, 2002.