Tim Bedford

Development of DAVIS

The Daft Architecture Vocal Improvisation System (DAVIS) is an interactive music system, designed to perform improvised music alongside myself. Unlike some interactive music systems, DAVIS is not intended to be used with unpredictable collaborative sources. It is designed specifically to interact with a single monophonic audio source based on a human voice—my own.

I originally intended to develop DAVIS as a single, monolithic program. Its only input and output would be raw audio. However, after encountering the "PfQ" model developed by Young and Blackwell, I decided that breaking up my system into the three suggest components would make the system easier to work on[1]. *P* could be described as the system's "ear," *f* the system's "brain," and *Q* the system's "mouth." *P* receives incoming audio data from a microphone as its input and outputs symbolic note data. *f* receives *P*'s output as its input and outputs new symbolic data in the same format. *Q* receives *f*'s output as its input and outputs synthesized sound through a loudspeaker. I decided early on that I would use the Open Sound Control (OSC) protocol for all inter-component communication, even if I wasn't originally sure what kind of information I would be communicating.

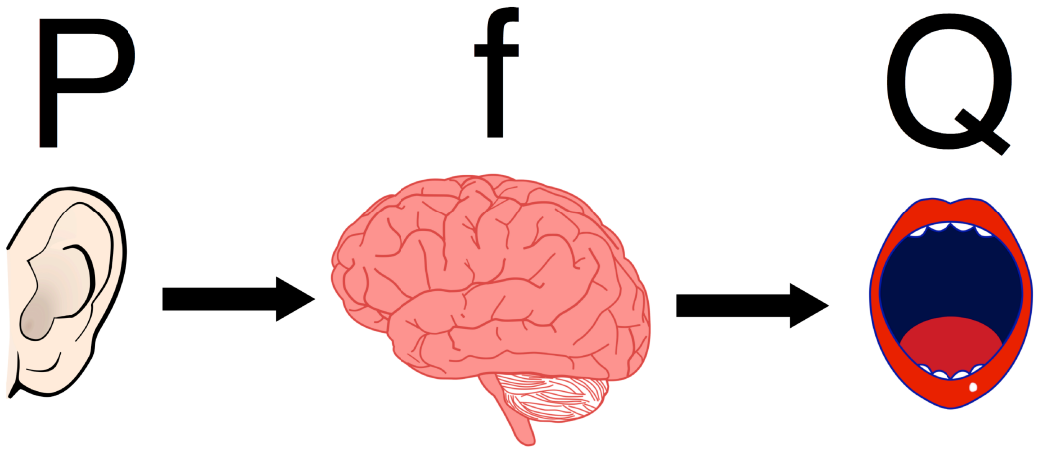


Illustration 1: Bodily metaphors for the three components that make up DAVIS

1 Blackwell, Tim; Bown, Oliver; Young, Michael. "Live Algorithms: Towards Autonomous Computer Improvisers. Computers and Creativity."

The development process involved creating basic versions of each component, one at a time. Then, with a working system done, I moved freely between the components as I refined the system as a whole. Beginning my work with *P*, I spent an exorbitant amount of time developing different versions of the component. Months went by as I tested out new pieces of software to experiment with. I began with SuperCollider[2], a relatively popular and versatile audio programming language. I had written some simple programs with SuperCollider a few years ago—at the time, I had never programmed before—but when returning to the language, I found the syntax's clunky mixture of object-oriented and functional programming styles to be obtuse and distracting. Using a tool I'm more comfortable in seemed like a better idea, so I began looking for audio analysis libraries for language I have used the most: Python.[3]

I tried out two Python libraries, Aubio[4] and Librosa.[5] Aubio is written in C and includes Python bindings, while Librosa is written entirely in Python. Both libraries are in active development, but both are relatively early in their development cycle and have minimal documentation. This situation made development difficult, especially because I was trying to use the libraries for a task neither was designed for: working with real-time audio. Venturing outside of the tasks outlined in the few tutorials required digging into the source code, considerably slowing down my work. While exploring the inner workings of each library was certainly interesting, I desperately needed a reliable platform to make some progress.

I ended up developing the final version of *P* with the software Max[6] (sometimes referred to as Max/MSP or Max/MSP/Jitter), one of the most common software packages used in experimental audio and video art. In Max, programs are represented visually as "objects" that connect to each other through "patch cords." The visual design has made it a more approachable software than its text-based

2 https://supercollider.github.io/
3 https://www.python.org/
4 https://aubio.org/
5 https://librosa.github.io/librosa/
6 https://cycling74.com/products/max/

rivals. I first encountered Max several years ago and have used it occasionally since then. The biggest benefits of Max are that it is stable, well-documented, has an active user community, and is designed for real-time projects. This means that I was able to quickly find user-made objects for pitch and amplitude estimation and get a barebones version of *P* running within days. I ran into trouble when I couldn't figure out a way to effectively detect silence. For example, a 500 millisecond note followed by a 500 millisecond period of silence was being registered as a single 1 second note. When I couldn't come up with a solution, I decided to make the bug a feature and eliminate rests from my singing approach.

I was initially reluctant to use Max for one reason: it is expensive, proprietary software and I don't want anyone who wishes to look at my system's code to have to pay to do so. However, I learned that a free version of Max is available, the only restriction being that the user cannot save any edits they make to a program. I hope to later write an entirely new *P* component with Aubio or Librosa, but Max will suffice for now. With a version of *P* completed, I moved onto the next component.

I developed *f* in Python simply because it is the language I am most comfortable in. I first worked on retrieving notes from Max over OSC and sending these same notes back over OSC to Max. I wrote a basic placeholder for *Q* in Max that played incoming notes from *f* with one of my computer's built-in MIDI instruments. The python-osc[7] library made this process quick and easy. Now, with the simplest possible version of *f* working, I could begin playing with the data. I decided early on that basing *f*, and consequently the first version of the system, around the discovery and manipulation of musical motifs would be a solid conceptual starting place. In this context, a motif can be defined as any short section of music that appears and reappears at various points in a piece of music. Repetition of motifs can help to imply form and intentionality.

My source for motif manipulation was the permutation of sets that atonal, twelve-tone composers, such as Arnold Schoenberg, used to generate new material. I focused on four types of

7 https://pypi.python.org/pypi/python-osc

permutation: retrograde (reversing the order of notes), transposition (uniformly raising or lowering the pitch of each note), stretching/shrinking (uniformly changing the duration of each note), and transformation (replacing one note with an entirely unrelated note). I also added a function to randomly generate a new motif with no connection to anything previously played.

Motifs are detected by searching for patterns in the most recent notes I have sung. The longest pattern that hasn't previously been detected is catalogued for later use. With each of the permutations implemented as its own function, I could randomly permutate generated and detected motifs and spit out the results. It wasn't the most sophisticated approach, but it was enough that I felt ready to move on to the next goal.

Various functions in *f*, such as randomly generating a new motif or checking whether or not to send the next note to *Q*, are triggered through various regularly occurring messages sent over OSC from *P*. This is the one instance where DAVIS's components communicate outside of the main "/note" API. As it is currently designed, *f* cannot properly work without my specific implementation of *P*. Ideally, none of the components should require any other component to work. Though I regret using such a design, my previous solution—calling these functions inside other commonly called functions—was extremely messy and difficult to maintain. I hope to find a better solution in the future.

I didn't begin work on the final component, *Q*, until late in the system's development. I held off because I initially had no idea what kind of sounds I wanted to create. I imagined creating some sort of abstract synthesis, but had no idea how to begin building something so vague. Once I decided to imitate the human voice, I was able to research the various methods for vocal synthesis and narrow down my options. With a small list of possible synthesis techniques, I began looking for audio software to use them in. I initially expected to use Max, but found the pre-made objects lacking. If I was going to implement the techniques by hand, I would prefer to use it in a text-based language, since I find large-scale Max projects difficult to keep organized. I took another look at SuperCollider, but decided that the awkward syntax wasn't worth the effort either.

Finally, I tried Csound.[8] With its first version arriving in 1986, Csound is the oldest audio synthesis language still in use, and its age shows. Its name is a reference to the fact that its synthesis engine is written in C; while virtually all low-level audio software is now written in C or C++, doing so was apparently notable three decades ago. Csound's syntax is an unconventional mix of C, assembly, and markup languages (such as HTML) that I can't imagine every being designed after the 1980's.

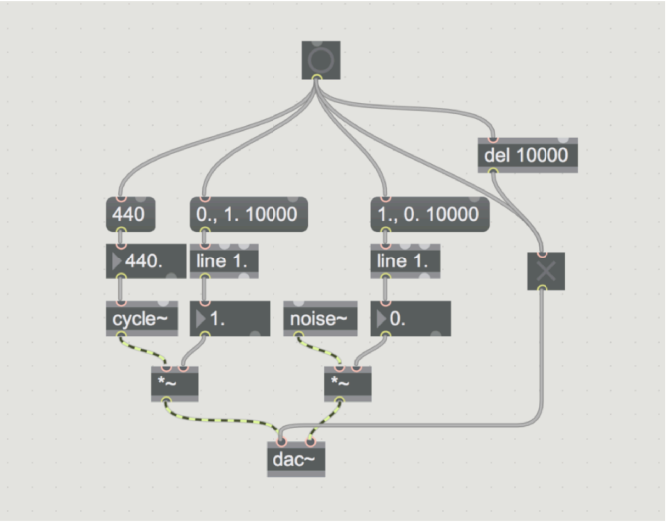
Despite all of this baggage, I ended up sticking with Csound. Why use such an esoteric language? There are a number of reasons. For one, Csound includes a built-in opcode—closer in concept to an object in Max than a true assembly opcode—for FOF synthesis. FOF—"Fonction d'onde formantique," French for "formant wave function"—is a well-established technique for building an imitative vocal sound out of formants, or "sinusoid[s] at the formant center frequency with an amplitude that rises rapidly upon excitation and decays exponentially."[9] Though far from the most advanced vocal synthesis technique, FOF offers a recognizably vocal sound through a relatively simple concept. Another reason for using Csound is that the developers make a point not to remove any opcodes from the language, meaning that although the standard library may appear unwieldy, a user can be sure that their program will still run on the latest version of Csound decades after being written. I don't expect my Max patch, for instance, to have the same longevity, so knowing that at least one of my components will require little to no maintenance far into the future is reassuring. Finally, learning Csound gave me an opportunity to learn a new programming language I would have otherwise never dreamed of using. Though the language is strange, I haven't found it difficult to learn. I prefer working with opinionated programming languages that expect their users to follow established styles, and Csound certainly fits the bill.

Among the three components, the initial work on *Q* took the least amount of time. The most significant reason is that synthesis was not the focus of my project. I knew that, due to time restraints

8 http://www.csounds.com/
9 Singing Voice Synthesis, 40

and my limited knowledge of synthesis techniques, creating a realistic vocal instrument was out of the question. The human voice is quite possibly the most difficult instrument to convincingly recreate and I would have likely had to devote my entire project to that goal. Instead, I decided that a simple form of synthesis would suffice. My first version of *Q* was a simple adaption of Csound's FOF opcode example, which layers the minimum number of FOFs (five) needed for something resembling human speech. The only change I made was the method of input; instead of manually entering note data into the program, the notes were received in real-time over OSC.

Illustration 2: The three components of DAVIS (each represented by their implementation software's logo) and the OSC messages they use to communicate




```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 128
nchnls = 2
0dbfs = 1.0

instr 1
  ksineamp line 0.0, 10, 1.0
  asinetone oscil ksineamp, 440

  knoiseamp line 1.0, 10, 0.0
  anoise noise knoiseamp, 0

  out asinetone, anoise
endin

</CsInstruments>
<CsScore>
i 1 0 10
</CsScore>
</CsoundSynthesizer>
```

Illustration 3: The same sound is implemented in Max (left) and Csound (right): over the course of 10 seconds, a 440 Hz sine tone in the left channel goes from silence to maximum amplitude and white noise in the right channel goes from maximum amplitude to silence, both linearly.

With the three components usable and communicating with each other, I had a working version of DAVIS. However, the results weren't exactly spectacular. Without many adjustments to Csound's FOF example, DAVIS sounded incredibly stiff. Though I wasn't aiming for realism, I did want to create a sound that was at least approaching humanity. I began tweaking the various parameters of each FOF function in order to create variation. Randomly altering each parameter by a small amount helps give the sound some character. I used a few of the techniques outlined in what is possibly the oldest

FOF tutorial on the internet.[10] For example, fading in the same pitch a few octaves below towards the end of a note resembles the slight croaking sound sometimes called "vocal fry." Applying some vibrato to one of the formants—but not all of them—helps to make each note a little more unique. At one point, I tried writing explicit motif-like patterns in Csound instead of randomizing each parameter. For example, the vocal fry might be exaggerated for every other note. However, I felt that doing so would imply the sort of intention that I only want to be present in *f*. I removed this feature, though perhaps these patterns could return if and when I decide to relax my use of the "PfQ" model.

The problem with *f* was how unmusical its output was. Even though I had decided that DAVIS should be an object without anything resembling human-like intelligence, I thought I should at least attempt to improve the aesthetic experience within the theoretical framework I had developed. Small changes, such as ensuring that every motif repeats, drastically improved DAVIS's musicality. Motifs now functioned as genuine motifs, instead of random phrases.

During my first round of debugging, I printed all relevant information—motifs generated, motifs detected, notes being received from *P*, and notes being sent to *Q*—directly to my terminal. Trying to manage all of this information as it cascaded down my screen was near impossible, so I wrote a simple user interface with a Python wrapper for the curses library.[11] Curses is used for developing text-based instances that mimic more common graphical interfaces. Each of the categories I mentioned is displayed independently in its own sub-window in the terminal. Whenever new data is sent to a sub-window, it can be added to whatever is already there, in the case of motifs, or it can replace everything in the sub-window, in the case of received and sent notes. Because the program crashes if any sub-window becomes full—and because I wanted to have a better handle on the data at all times—I implemented regularly occurring motif deletion. Now with only a few motifs stored at any one time, I can better conceptualize the data as it appears. I've found the process of deciding which data to

10 Clarke, Michael. "A FOF Synthesis Tutorial."

11 https://docs.python.org/3/library/curses.html

represent and how to do so to be very interesting, so while this interface was originally designed only to streamline the debugging process, I plan to further develop it within the context of human-computer interaction (HCI) theory.

When I began working on DAVIS, I thought almost all of my effort would be put towards coding the *f* component and pondering issues of computational creativity and artificial intelligence. While I certainly put more work into *f* than the other components, and those themes were at the front of my mind during development, my research has drawn me to new interests as well, such as HCI and vocal synthesis. My long term goal for DAVIS is to develop many new versions of each of the three components, all radically different in design from the ones I have already made. Through this process, I plan to test the limits of my current API and develop a more robust and versatile formal specification for vocal-oriented interactive music systems. I hope DAVIS can then serve as a framework for broader studies in improvisation, audio analysis, synthesis, and more.

Bibliography

Blackwell, Tim; Bown, Oliver; Young, Michael. “Live Algorithms: Towards Autonomous Computer Improvisers.” Computers and Creativity. Edited by McCormack, Jon and d'Inverno, Mark. 2016. Springer, Berlin.

Clarke, Michael. “A FOF Synthesis Tutorial.” https://cara.gsu.edu/courses/Csound_Users_Seminar/csound/3.46/CsTutorials.html

Cook, Perry. “Singing Voice Synthesis: History, Current Work, and Future Directions.” *Computer Music Journal* 20, no. 3 (1996): 38-46.