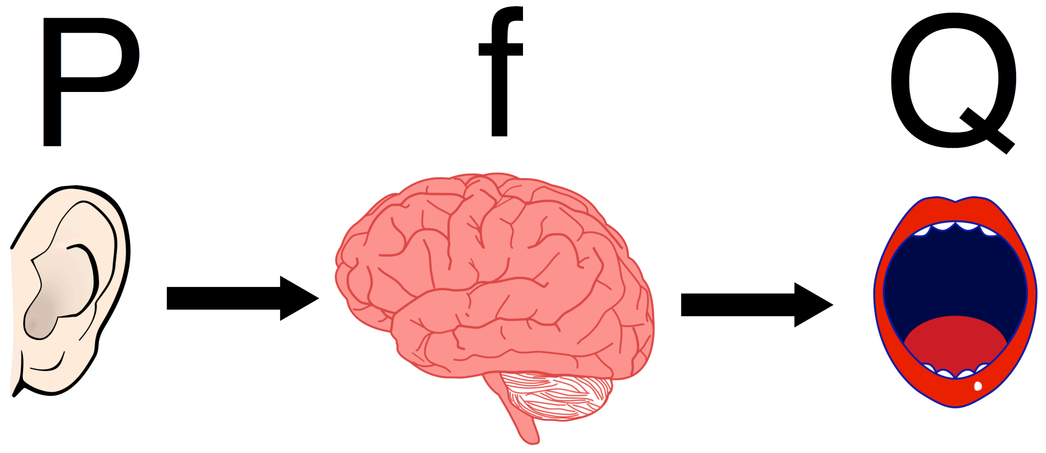
Tim Bedford

Development of DAVIS

Introduction

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. 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.



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

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. All three components were developed in different programming languages. This wasn't intentional, but it help immensely. Having to create firm divisions between the three components forced me to consciously consider DAVIS how would handle its internal data.

P

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—with no outside programming experience—but when returning to the language, I found the syntax's clunky mixture of object-oriented and functional programming styles 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 my most used language: Python.[3]

The first two libraries I tried out were 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 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.

2 https://supercollider.github.io/
3 https://www.python.org/
4 https://aubio.org/
5 https://librosa.github.io/librosa/

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. I first encountered Max about five 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. 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 can't save any edits they make to a program. With a version of *P* completed, I moved onto the next component.

f

I developed *f* with Python for a number of reasons. Because *f* only works with symbolic music data, I knew I wouldn't have to use a language that can deal directly with audio, such as Max. Any general purpose programming language would work for the task, so I picked the one I knew best.

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 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

6 https://cycling74.com/products/max/

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 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 any patterns in the most recent notes I hang 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 over OSC by *P*. This is the one instance where DAVIS's components aren't entirely encapsulated. 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.

Q

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.

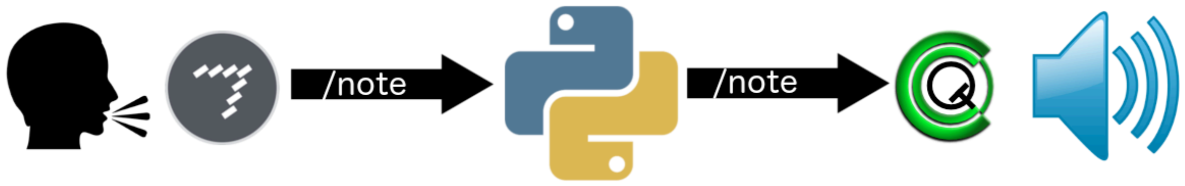
Finally, I tried Csound.[7] With its first version arriving in 1986, Csound is the oldest audio synthesis language still in use, and its age shows. Its name is a reference to the fact that its synthesis engine is written in C; while virtually all low-level audio software is now written in C or C++, doing so was apparently notable in the mid-1980's. Csound's syntax is a bizarre mix of C, assembly, and markup languages (such as HTML) that I can't imagine every being designed during or after the 1990's. Despite all of this, I ended up sticking with Csound.

Why use such a strange language? There are a number of reasons. For one, Csound includes a built-in opcode—essentially the same concept as an "object" in Max—for FOF synthesis. FOF —"Fonction d'onde formantique," French for "formant wave function"—is a simple technique for building an imitative vocal sound out of "formants." In practice, most FOF-based synthesis doesn't sound very convincing, but DAVIS was never about cutting-edge synthesis. 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. 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, it certainly isn't difficult to learn. It establishes a very particular approach to design instruments and programming in general.

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 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

7 http://www.csounds.com/

would have likely had to devote my entire project to that goal. Instead, I decided that simple 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.



All Three

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 human character. At one point, I tried writing explicit patterns in Csound instead of randomizing each parameter. However, I felt that doing so would suggest a sort of intention in *Q* that was only meant to be present in *f* and removed this feature.

The problem with *f* was how unmusical its output was. Even though I had decided that DAVIS should be an object without human-like intelligence, I knew DAVIS could stay within that framework and be a more interesting musical partner. It would need to have a more involved way of picking which motifs to permutate and when to do so.

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 this information as it flowed down my screen was near impossible, so I wrote a simple user interface with a Python wrapper for the curses library.[8] Each of the categories I mentioned is displayed independently in its own window in the terminal.

Future

My next main goal for DAVIS is to make the API more versatile. For example, there are many ways to represent timbre, and instead of explicitly requiring five formant values, a list of any length and type should be the default way to store timbral data. Once I have a mature API, I would like to create radically different versions of each component and make DAVIS a much more modular project.

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