Teachable Moments in Functional Audio Processing

Martin Henz
Shang-Hui Koh
Samyukta Sounderraman
National University of Singapore
Singapore

Abstract

The atomic entity of digital audio processing systems is a digital audio signal, i.e. a sequence of sound samples that represent the amplitude of a sound wave at discrete time intervals. Such signals are transformed additively, by combining them into more complex signals, and subtractively, by subjecting them to digital filters. In order to cover digital audio processing in a classroom from first principles, we need to form collections of samples in streams or arrays, and define operations on these collections in accordance with the constraints of digitization. In this work, we pursue an alternative approach, where the atomic entity is a continuous wave function. We present additive synthesis operations, including wave envelopes and musical abstractions in a purely functional setting. The final continuous wave function is digitized in order to make the sound audible. We report our experiences with what we call *functional audio processing* as an example domain for teaching functional programming to first-year students, where simplicity and conceptual elegance outweighs the inherent limitation to additive synthesis. We describe a sequence of teachable moments that highlight the potential of functional audio processing at an early stage in the learning process, before streams or arrays are introduced.

CCS Concepts: • **Software and its engineering → General programming languages**; *Integrated and visual development environments.*

Keywords: teaching programming, JavaScript, audio processing, functional programming

ACM Reference Format:
Martin Henz, Shang-Hui Koh, and Samyukta Sounderraman. 2021. Teachable Moments in Functional Audio Processing. In *Proceedings of the 2021 ACM SIGPLAN International SPLASH-E Symposium (SPLASH-E '21), October 20, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3484272.3484967

SPLASH-E '21, October 20, 2021, Chicago, IL, USA

ACM ISBN 978-1-4503-9089-7/21/10.
https://doi.org/10.1145/3484272.3484967

1 Motivation

The functional-first approach to CS1 introduces students to programming with purely functional programs whose execution is understood as a step-by-step process of simplification and function unfolding, akin to familiar mental models in mathematical calculation and equation solving. Programming languages used in this approach include Racket, Scheme, OCaml, and Haskell and prominent textbooks are *How to Design Programs* [4] and *Structure and Interpretation of Computer Programs* [1] (SICP). While intuitions from mathematics are helpful, educators recognize the value of experiential learning in CS1. A prominent example of teaching functional programming through engaging application domains is Paul Hudak's *Haskell School of Expression* [8], which introduces programming through multimedia applications, including graphics, animation, and computer music.

For audio processing, the dominant conceptual model is *digital signal processing* (DSP), where a network of signal processing nodes is constructed, and digital signals—streams of discrete samples—are transferred between these nodes along the network links. The signal processors transform the signals additively, by combining them into more complex signals, or subtractively, by subjecting them to digital filters [17]. Examples of languages for DSP-based audio processing are Nyquist [3] and SuperCollider [13] .

Audio processing provides a rich set of abstractions and concepts and thus potentially offers teachable moments in computing and computational thinking. Robert Havighurst describes a "teachable moment" as follows.

> A developmental task is a task which is learned at a specific point and which makes achievement of succeeding tasks possible. When the timing is right, the ability to learn a particular task will be possible. This is referred to as a 'teachable moment.' It is important to keep in mind that unless the time is right, learning will not occur. Hence, it is important to repeat important points whenever possible so that when a student's teachable moment occurs, s/he can benefit from the knowledge." [6]

In this paper, we argue that audio processing offers a coherent set of teachable moments even before the students are exposed to the programming techniques necessary for stream processing. In Section 3 we introduce a *functional*

sound as a pair consisting of a function with a *continuous* domain, representing the sound wave, and a number, representing the duration of the sound. With functional sounds, a CS1 course that follows a functional-first approach can pursue audio processing from first principles as soon as pairs are discussed.

In order to explain the behaviour of even the simplest audio processing nodes such as the simultaneous combination of two sounds, a DSP-based approach needs to employ the processing of digital signals. In Section 4, we show simple additive sound composition operators operating on functional sounds, and the teachable moments that arise from them. Section 5 shows generic abstractions, including transformations and transformation generators, as higher-order functions on functional sounds. Section 6 demonstrates the design of synthetic musical instruments as applications of such transformation generators. Sections 7 and 8 explore abstractions for manipulating musical structures. Section 9 handles the final step of functional audio processing, which is the digitization of the functional sound for making it audible, and Section 10 discusses implementation and performance issues of digitization. Section 11 reports on our experiences with this approach in our CS1 course and points to future enhancements as well as its inherent limitations. Before we embark on our journey in functional audio processing, we review related work.

2 Related Work

Audio processing is inherently experiential, because the results of student programs are immediately audible. As stated by Seymour Papert in 1980 (emphasis added),

> Children create programs that produce pleasing graphics, funny pictures, *sound effects, music,* and computer jokes. They start interacting mathematically because the product of their mathematical work belongs to them and belongs to real life. [14]

Since then, the pedagogical potential of sound processing has been realized by many educators and has been widely published. Here, we highlight three studies spanning 30 years. In 1991, Guzdial [5] employed guided (musical) discovery for engaging young learners in computational thinking. In 2006, Hechmer et al. [7] present musical composition as an open-ended, creative endeavor using LogoRhythms, a library for Papert's language Logo. In 2020, Köppe [12] emphasizes the potential of audio processing in engaging students of all genders and prior computing background.

Functional programming excels in the modeling of conceptually rich domains, and thus plays a prominent role in the use of audio processing in education. Hechmer et al. [7] use higher-order functions to create harmonies and melodies in Logo. Karczmarczuk [10] represents audio signals as corecursive lazy streams in the functional language Clean. The most extensive use of audio processing in the realm of functional programming is possibly Hudak's *Haskell School of Music* [9], which recreates a rich repertoire of audio processing techniques in the functional language Haskell.

To our knowledge, all existing work in the use of audio processing for computing education is based on digital signal processing, using streams or arrays of discrete sound samples as fundamental building blocks. The language FAUST [19] might come closest to our style of audio processing. FAUST coerces primitive operators such as + such that they are applied to streams in the C++ programs to which FAUST programs compile. FAUST programmers might not always be aware of the underlying stream processing, but are inherently limited to abstractions that fit into the digital audio processing paradigm.

3 Sound Waves as Functions

An analog signal is a function $a : \mathbb{R} \to \mathbb{R}$ that maps continuous time to amplitude. We limit ourselves to sounds, i.e. analog signals of a particular duration d that start after a time 0, and restrict the range of the amplitude to $[-1, 1]$[1]; a sound is a function $s : [0, d] \to [-1, 1]$.

We use JavaScript, a dynamically-typed functional curly-bracket programming language. All examples translate one-to-one to Python[2], Scheme, Lisp or Racket, and with appropriate type declarations to OCaml, Haskell or Java[3]. We use `pair`, `head`, and `tail` to construct pairs and select their components, following the JavaScript edition of SICP [2]. (The corresponding functions in Scheme, Lisp and Racket are `cons`, `car`, and `cdr`.) A pure sine wave with a frequency of 440Hz can be represented in JavaScript by a function

```
const concert_pitch_A_wave =
   t => Math.sin(2 * Math.PI * 440 * t);
```

A *sound* is a pair whose head is a wave and whose tail is its duration in seconds. The concert pitch sound A of duration 1.5s is constructed by

```
const concert_pitch_A =
        pair(concert_pitch_A_wave, 1.5);
```

We insist on using the following selectors on sounds:

```
const wave = s => head(s);
const duration = s => tail(s);
```

We insist that the wave function of a sound returns 0 for any time larger than its duration. The constructor `make_sound` enforces this invariant:

```
const make_sound = (w, d) =>
        pair(t => t >= d ? 0 : w(t), d);
```

[1] It is recommended not to enforce this range strictly. See section 10 for more details.
[2] In the case of Python, the digitization function of Section 9 cannot rely on tail recursion; an explicit loop is required.
[3] ditto for Java

We call the invariant *sound discipline*: Any sound `s` constructed by `make_sound` fulfils

```
wave(s)(t) = 0, if t > duration(s)
```

> **Teachable Moment 1.**
> The importance of data abstraction: `make_sound` guarantees sound discipline, `pair` does not.

We define a sound generator for pure sine-wave sounds of a frequency `f` and duration `d`.

```
function sine_sound(f, d) {
    return make_sound(
            t => Math.sin(2 * Math.PI *
                          f * t),
            d);
}
play(sine_sound(500, 1));
```

We make a sound audible using the digitization function `play`, which we will examine in Section 9.

> **Teachable Moment 2.**
> Practical use of higher-order programming: `sine_sound` returns a sound which contains a wave function. The function `play` then calls the wave function to digitize the sound.

A function that takes a frequency and a duration as arguments and produces a sound of the given frequency and duration is called a *wave form*. The function `sine_sound` above is a wave form, and students program other common wave forms such as `square_sound` and `sawtooth_sound` with similar ease.

4 Programming Sound Composition Operators

Naturally, the two primary composition methods are to append and to overlap two sounds. The following function shows the latter; the former is intriguingly similar and left to the reader.

```
function simultaneously(sound1, sound2){
    const w1 = wave(sound1);
    const d1 = duration(sound1);
    const w2 = wave(sound2);
    const d2 = duration(sound2);
    return make_sound(
               t => (w1(t) + w2(t)) / 2,
               d1 < d2 ? d2 : d1);
}
play(simultaneously(sine_sound(500,1),
                    sine_sound(505,1)));
```

With functional sounds, composition operators are defined by simply constructing the resultant wave function. These compositions themselves require constant time and space, similar to stream-based digital signal processing where composition operators just set up a node in the signal processing network, and the stream operations take place when the signal processing is activated. Section 9 discusses the impact of composition on the time and space complexity of the digitization function `play`.

> **Teachable Moment 3.**
> The importance of composition in programming: Application domains typically come with domain-specific composition operators.

> **Teachable Moment 4.**
> `simultaneously` makes use of the sound discipline, invariant in action!

More teachable moments are provided by the need to generalize composition operators to take more than two sounds as arguments.

5 Functional Abstraction through Audio Processing

Students often have difficulties recognizing the power of functions to form abstractions. Kolb's experiential learning cycle [11] emphasizes the interplay of concrete experiences with abstract conceptualization. Audio processing is inherently experiential, and every successful abstraction becomes a memory-forming experience. Before we handle the abstraction of a sound transformation—a function that takes a sound and returns a sound—we encourage the students to record their own voices and sounds in their environment. A typical transformation programmed by the students reverses a sound in time, which is at the same time conceptually enlightening and highly entertaining.

```
function reverse(sound) {
    const w = wave(sound);
    const d = duration(sound);
    return make_sound(t => w(d - t), d);
}
```

> **Teachable Moment 5.**
> The significance of transformation operators (unary operators whose return type is the same as the input type) lies in their compositionality: Function composition forms audio processing tool chains.

6 Synthesizing Instruments

Envelopes are contours that alter the behaviour of sounds over time. The first envelope generator was developed by

Robert Moog in the analog era of electronic synthesizers, where a changing voltage output was triggered when a keyboard key was depressed. Piping the output to voltage-controlled oscillators and filters enabled the generation of dynamic sound effects. Envelope generators are now standard features of digital synthesizers, the most common form being Attack, Decay, Sustain and Release (ADSR) [15].

Employing our sound discipline, we implement a modified ADSR envelope.

> **Teachable Moment 6.**
> Students program an ADSR envelope by manipulating the wave function according to the SDSR specification, as a function `adsr` that takes the relative ADSR parameters as arguments and returns a sound transformation on the amplitude that respects the proportions of the ADSR phases.

Synthetic instruments can be developed additively by applying different envelopes to different wave forms at a given base frequency and its multiples, with much potential for further teachable moments.

7 Musical Scales

Conventional music provides a rich domain for abstraction. The human ear perceives rational frequency relations and an exponential progression of twelve pitches for doubling the frequency underlies many musical traditions worldwide. Not to exacerbate student anxiety stemming from a lack of prior knowledge or exposure to music theory, we do not emphasize the scientific pitch notation ($A\flat_2$, etc) and instead use the MIDI scale, where a frequency of 440 Hz corresponds to the MIDI note 69 and every increase by 12 (an "octave") on the MIDI scale doubles the frequency. The function `midi_note_to_frequency` converts a MIDI note to a frequency.

```
function midi_note_to_frequency(note) {
  // MIDI note 69 has 440 Hz
  return 440 *
         Math.pow(2, ((note - 69)
                              / 12));
}
```

The notion of an integer interval (distance between two MIDI notes) is easy to grasp for non-musically trained students, and we present a musical scale as a an increasing sequence of MIDI notes characterized by a sequence of intervals. Without dwelling too much on its harmonic properties, we present a conventional major scale by the interval sequence 2, 2, 1, 2, 2, 2, 1, starting with a chosen note.

> **Teachable Moment 7.**
> The linking of musical standards to programming provides a teachable moment about abstraction: students need not gain an in-depth understanding of Western musical theory when they can simply trust the above transformers to convert the notes for them.

Students with musical training find opportunities to rediscover their musical knowledge through their newly acquired programming skills. We borrow the musical abstractions from established functional programming libraries and systems for digital audio processing such as the *Haskell School of Music* [9].

8 Rhythmic Structures

Rhythmic structures provide an excellent opportunity to teach functional programming by linking them to the Konnakol system and nested lists. Konnakol is the art of creating rhythms through vocal syllables such as "Tha", "Ka", "Dhi", and "Mi", each representing a distinct sound, possibly from a percussive instrument. An example of a rhythm would be "Tha Ka Dhi Mi Ta Ri Ki Tha Nom". We define a *simple rhythm* as a list of non-negative integers, each representing a distinct Konnakol syllable. We also recognise that within a rhythm, sections can be repeated multiple times. Therefore, we say that all rhythms can be represented recursively as:

- a simple rhythm, or
- a list of rhythms, or
- a pair whose head is a rhythm and whose tail is an integer that represents the number of times the rhythm must be repeated.

An example of a rhythm in this *rhythm language* is

```
pair(list(pair(list(0, 1, 2, 3), 3),
          pair(list(1, 5, 2), 2)), 2)
```

> **Teachable Moment 8.**
> The importance of language design: We routinely invent new formal languages as required by our application domains.

We instruct students to build a function that converts any rhythm to its simple rhythm equivalent, which can then be further processed into a functional sound.

> **Teachable Moment 9.**
> The ubiquity of language processing: A rhythm converter can be seen as an interpreter for the rhythm language.

9 Digitization

Functions in programs are not directly audible. We need to convert them into air pressure waves that mirror the wave functions programmed by the students. In this "last mile" of functional audio processing, we evaluate the wave function at fixed time intervals to produce a list of sample values. We explore the following tail-recursive function.

```
function samples(sound, sample_rate) {
  const w = wave(sound);
  const d = duration(sound);
  const sample_dur = 1 / sample_rate;
  function accumulate_samples(t, acc) {
    return t < 0 ? acc
           : accumulate_samples(
                t - sample_dur,
                pair(w(t), acc));
    }
    return accumulate_samples(d, null);
}
```

> **Teachable Moment 10.**
> The importance of tail recursion: Any non-tail-recursive solution will quickly fail for sounds of longer than around 300 milliseconds as call stack limits are generally within the range of 10000 to 15000. Students are also encouraged to experiment with this function and experience differences in processing speed, depending on the complexity of the given sound.

With `samples`, the function `play` can be explained as follows, where a natural choice for `global_sample_rate` is 44100.

```
const global_sample_rate = 44100;
function play(sound) {
  play_using_computer_audio_system(
    samples(sound, global_sample_rate));
}
```

10 Implementation

Students will perceive *aliasing* when listening to their sounds using this `play` function and are encouraged to compare the phenomenon to strobe effects in everyday life. The underlying *sampling theorem* given by Shannon[16] states:

> If a function $f(t)$ contains no frequencies higher than W, it is completely determined by giving its ordinates at a series of points spaced $1/2\ W$ seconds apart.

Fourier analysis teaches the students that their sounds can be seen as being composed of sine waves of various frequencies. Unfortunately, the function `square_sound` in Section 3 and similarly defined sawtooth and triangle waves are not band-limited; there is no W that permits a suitable choice of `global_sample_rate` in Section 9. When played with a high frequency fundamental, these wave forms are therefore perceived as inharmonic. We guide the students to pursue two directions to mitigate the resulting aliasing by (1) increasing the sample rate, and (2) developing alternative implementations for the basic wave forms. We currently use the second approach in our library and provide a primitive `play` function that works like the `play` function in Section 9 but produces an array instead of a list. In our web-based environment for JavaScript programming, we direct the output to an audio buffer in WebAudioAPI which plays the sound in the web browser.

The sampling time for simple sounds is several orders of magnitude shorter than their duration and thus sampling can be easily performed on the fly, which eliminates any perception of digitization. The situation changes as the students creatively explore the musical possibilities discussed in Sections 7 and 8. The digitization time for the most complex student creations approaches and occasionally exceeds their duration and thus might lead to perceptible digitization run times. As part of our CS1 course, we conduct an annual functional audio processing contest, where students compete in achieving musical effects. Typical contest entries [18] are covers of popular songs.

11 Results and Future Work

Our CS1 course has included sound processing since 2014. From 2014 to 2018, the course used a hybrid approach where functional sounds where converted to digital sounds, which were then combined using conventional digital audio processing techniques. We radically simplified our CS1 audio component in 2019, with very positive student anonymous feedback, especially in 2020: "What I liked about the module [course]" (emphasis added)

- . . . a lot of interesting problems such as runes, *sounds*, curves and manipulating these with pre–declared functions. This made the module more fun and interesting.
- . . . The many different components integrated into source (*sound*, robots etc) widens the scope of learning.
- . . . The quests and missions allowed us to revise our concepts and also allowed us to apply our skills on real–life concepts, like *audio processing* and video imaging processing.
- . . . Interesting assignments that use *sound*, graphics. . .

The annual sound contest is established as one of the highlights of our first-semester programme.

The extreme simplicity of functional audio processing makes it possible to integrate it—along with other experiential domains such as graphics and robotics—into a functional-first CS1 course for computer science first-year students. The inherent limitation to additive sound processing does not cause problems in this context, and the potential runtime limitations imposed by complex musical creations have not negatively affected students. As devices continue to increase in speed, we expect functional audio processing to become more feasible for audio processing and software courses other than first-year CS1. We are currently exploring the parallelization of the digitization process and expect significant speedups due to the inherent embarrassing parallelism of the wave function generated with the abstractions presented in this paper.

References

[1] Harold Abelson and Gerald Jay Sussman. 1996. *Structure and Interpretation of Computer Programs* (2^{nd} ed.). MIT Press, Cambridge, Massachusetts.

[2] Harold Abelson and Gerald Jay Sussman. 2022. *Structure and Interpretation of Computer Programs, JavaScript edition.* MIT Press, Cambridge, MA. Adapted to JavaScript by Martin Henz and Tobias Wrigstad.

[3] Roger B. Dannenberg. 1997. Machine Tongues XIX: Nyquist, a Language for Composition and Sound Synthesis. *Computer Music Journal* 21, 3 (1997), 50–60. https://doi.org/10.2307/3681013

[4] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to Design Programs: An Introduction to Programming and Computing* (2^{nd} ed.). MIT Press, Cambridge, Massachusetts.

[5] Mark Guzdial. 1991. *Teaching Programming with Music: An Approach to Teaching Young Students About Logo.* Technical Report. Logo Foundation. https://el.media.mit.edu/logo-foundation/resources/papers/pdf/teaching_progr.pdf.

[6] Robert James Havighurst. 1953. *Educational Psychology.* Longmans, Green, New York.

[7] Aaron Hechmer, Adam Tindale, and George Tzanetakis. 2006. LogoRhythms: Introductory Audio Programming for Computer Musicians in a Functional Language Paradigm. In *36th Annual ASEE/IEEE Conference: Frontiers in Education.* IEEE, San Diego, California, 11–16. https://doi.org/10.1109/FIE.2006.322438

[8] Paul Hudak. 2000. *The Haskell School of Expression: Learning Functional Programming through Multimedia.* Cambridge University Press, New York.

[9] Paul Hudak and Donya Quick. 2018. *The Haskell School of Music: From Signals to Symphonies.* Cambridge University Press, New York. https://books.google.com.sg/books?id=Y1vCtAEACAAJ

[10] Jerzy Karczmarczuk. 2005. Functional Framework for Sound Synthesis. In *PADL 2005, Practical Aspects of Declarative Languages, 7^{th} Internat'l Symposium (LNCS)*, Manuel Hermenegildo and Daniel Cabeza (Eds.), Vol. 3350. Springer, Berlin, Heidelberg, New York, 7–21. https://doi.org/10.1007/978-3-540-30557-6_3

[11] David A. Kolb. 1984. *Experiential learning: Experience as the source of learning and development (Vol. 1).* Prentice-Hall, Englewood Cliffs, NJ.

[12] Christian Köppe. 2020. Program a Hit – Using Music as Motivator for Introducing Programming Concepts. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, Andrew Luxton-Reilly and Monica Divitini (Eds.). ACM SIGCSE, New York, 266–272. https://doi.org/10.1145/3341525.3387377

[13] James McCartney. 2002. Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal* 26, 4 (2002), 61–68. https://doi.org/10.1162/014892602320991383

[14] S. Papert. 1980. *Mindstorms: Children, computers, and powerful ideas.* Basic Books, Inc., New York.

[15] Trevor Pinch and Frank Trocco. 2004. *Analog Days: The Invention and Impact of the Moog Synthesizer.* Harvard University Press, Cambridge, MA.

[16] C. E. Shannon. 1949. Communication in the Presence of Noise. *Proceedings of the IRE* 37, 1 (1949), 10–21. https://doi.org/10.1109/JRPROC.1949.232969

[17] Julius O. Smith. 2010. *Physical audio signal processing: For virtual musical instruments and audio effects.* Center for Computer Research in Music and Acoustics, Stanford University. https://ccrma.stanford.edu/~jos/pasp/

[18] Students of our CS1 course, CS1101S, at National University of Singapore. 2021. Sound contest entries. https://github.com/source-academy/source-programs/tree/master/src/module-demos/sound

[19] Stéphane Letz Yann Orlarey, Dominique Fober. 2009. FAUST: an Efficient Functional Approach to DSP Programming. In *Computational Paradigms for Computer Music*, Gérard Assayag and Andrew Gerzso (Eds.). DELATOUR FRANCE; Musique/Sciences edition, Sampzon, France, 65–96.