OPEN ACCESS Freely available online
I. Clester and J. Freeman,
"Distributing Generative Music With Alternator"
J. Audio Eng. Soc., vol. 71, no. 11, pp. 780–789, (2023 November).
DOI: https://doi.org/10.17743/jaes.2022.0113

Distributing Generative Music With Alternator

IAN CLESTER,[*] AND JASON FREEMAN
(ijc@gatech.edu) (jason.freeman@gatech.edu)

Center for Music Technology, Georgia Institute of Technology, Atlanta, GA

Computers are a powerful technology for music playback: as general-purpose computing machines with capabilities beyond the fixed-recording playback devices of the past, they can play generative music with multiple outcomes or computational compositions that are not fully determined until they are played. However, there is no suitable platform for distributing generative music while preserving the spaces of possible outputs. This absence hinders composers' and listeners' access to the possibilities of computational playback. In this paper, the authors address the problem of distributing generative music. They present a) a dynamic format for bundling computational compositions with static assets in self-contained packages and b) a music player for finding, fetching, and playing/executing these compositions. These tools are built for generality to support a variety of approaches to making music with code and remain language-agnostic. The authors take advantage of WebAssembly and related tools to enable the use of general-purpose languages such as C, Rust, JavaScript, and Python and audio languages such as Pure Data, RTcmix, Csound, and ChucK. They use AudioWorklets and Web Workers to enable scalable distribution via client-side playback. And they present the user with a music player interface that aims to be familiar while exposing the possibilities of generative music.

0 INTRODUCTION

From now on there are three alternatives: live music, recorded music and generative music. Generative music enjoys some of the benefits of both its ancestors. Like live music, it is always different. Like recorded music, it is free of time-and-place limitations — you can hear it when you want and where you want.
— Brian Eno, *A Year With Swollen Appendices*

In the essay "Generative Music" [1], Eno describes three kinds of music. The first is live music, in which the music comes out differently every time. Even the same musicians playing the same piece will sound a little different between performances. In musical traditions that emphasize improvisation on a form, each performance of a piece may differ significantly, but still be recognizably the same piece. Freer practices go further still, taking each performance in different directions bound together only by the improvisational spirit of the performers—or by a score that embraces randomness, with variability built-in by the composer.

The second kind is recorded music, in which music is frozen in time. The recording consists of whatever particular sounds occurred that time, e.g., samples output by an analog-to-digital converter or grooves etched into a record, and the recording does not change each time it is played (other than, perhaps, the medium degrading). Nonetheless, recorded music is wildly successful because it is convenient for both artist and listener. Once an artist records their music, their fans can listen to it across time and space, whenever they want and however they want. A recording, as information, is inherently less scarce than a performance—a performer can only play one thing in one place at a time. Digital recordings in particular are non-rivalrous: one's listening to a recording does not prevent another's listening to an identical copy of the same recording, and there is no venue to limit the size of the audience.

So on one hand, there is live performance, which is dynamic but constrained and scarce, and on the other, recordings, which are static but easy to distribute and reproduce. Then there is the third alternative: generative music. By representing music as *systems that generate audio* rather than audio itself, the best of both worlds can be had: dynamism *and* availability, possibilities *and* scalability.

Eno described that third alternative in 1996, but where is it today? Live music and recorded music are ubiquitous, but generative music remains niche. Modern music production invariably involves a computer, but playback uses almost none of the machine's potential. Languages and environments abound for making computer music, but distribution remains challenging, and the solutions that exist are typically custom-made for a particular musical work, artist, or language. From the perspective of the listener, these forms of distribution present a barrier to listening and separate generative music not only from other music, but even from other generative music.

*Corresponding author: ijc@gatech.edu

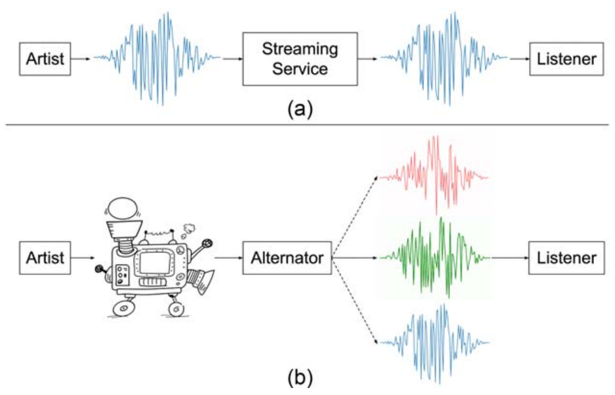


Fig. 1. Conventional streaming model (a) vs. Alternator (b).

In this paper, the authors address this state of affairs and investigate the problem of distributing generative music. They describe Alternator: a system for distributing and playing computational compositions, making the possibilities of such compositions more accessible to listeners and more useful to composers.[1] The authors originally presented Alternator at the 2022 Web Audio Conference (WAC) [2]; here, they recapitulate their previous exposition and expand on it, connecting Alternator to additional literature on generative music distribution and providing updates since their presentation at WAC.

Fig. 1 summarizes the conceptual model of Alternator. In conventional music distribution platforms [Fig. 1(a)], an artist produces an audio recording of their music and uploads it to a streaming service, which then streams it to listeners. A fixed, unchanging waveform represents the musical work, and listeners hear the same thing every time. In the Alternator model [Fig. 1(b)], the artist creates and uploads a *computational* recording of their music instead—rather than audio, a program that generates audio (represented by a contraption in the figure). At the time of playback, the program runs, and there may be many possible outcomes; listeners hear a different one each time.

1 BACKGROUND AND RELATED WORK

1.1 Musical Inspirations

This work is inspired by the long tradition of thinking about music in terms of systems and processes. Aleatoric or indeterminate music may include instructions in the score that allow for many possible outcomes at the time of performance, as in works by Cage, Brown, and Stockhausen [3].[2] Steve Reich [4] made explicit the notion of a piece of music as a process and emphasized gradual, perceptible processes as opposed to the chance of Cage (or the "seldom audible" processes of serial composers). Brian Eno [1], inspired in turn by Reich, articulated the idea of generative music as a kind of hybrid between live and recorded music and linked its future to advances in the capabilities and ubiquity of computing technology.

Alternator also builds on work in algorithmic composition; for a summary, see Essl [5]. Algorithmic thinking long predates automatic computing machines, but as Essl notes, "[by] using generative composition algorithms on computers, music can be created in real-time by an autonomous and infinite automatic process." Essl also observes that, due to the nature of generative music, "distribution on a reproductive medium such as a compact disc seems highly inappropriate." Although CDs may have since been eclipsed by streaming services, the fit is no better, because the medium (static audio recordings) remains reproductive rather than generative. As Levtov [6] puts it, "the widespread formats of today have one particular defining characteristic that is fundamentally at odds with the experiential, transient nature of live music or indeed algorithmic music; once their sonic content is defined, they are designed to sound identical at every hearing." Alternator aims to provide a more appropriate medium, suitable for the variable output and variable (potentially infinite) length of generative compositions.

Although it draws on these artistic movements and fields, Alternator does not enforce any particular aesthetic constraints. The sole requirement is that a musical work have a computable translation into sound:[3] thus, Alternator supports generative and aleatoric music as well as deterministic or conventionally recorded music—and everything in between. Despite this aesthetic agnosticism, Alternator's existence is predicated on the idea that there is value in generativity and that the idea of the musical work as a computable process—ever-changing, describing a field of possibilities—is an idea worth sharing. Alternator embraces the vision of composers of aleatoric, generative, and algorithmic music and deals with the problem *after* composition: getting music to a listener.

1.2 Technical Precursors

1.2.1 Generation

Many projects have addressed the problem of allowing for many possible outcomes from a computer-playable piece of music. SSEYO's Koan software (used by Brian Eno as described in [1]) and its successors explicitly address this under the umbrella of *generative music*, usually with an ambient emphasis. Programming languages designed for music offer a general solution, from Max Mathews's original MUSIC onward [7]. Today, a composer can use audio programming languages (Max/MSP, Pure Data, SuperCollider, ChucK, RTcmix, Csound, etc.) to describe a computational program that may depend on random values or external input and thus encompass a broad range of possibilities. A composer may also use general-purpose

[1]A live demo is available at https://ijc8.me/alternator, and the source code can be found at https://github.com/ijc8/alternator.

[2]Some aleatoric works are "indeterminate with respect to their composition," as Cage puts it (or, as the present authors might say, they vary only at "compile-time"). In this paper, the authors are more interested in those that are "indeterminate with respect to their performance" (which vary at "run-time"), because these compositions encompass multiple outcomes.

[3]Technically, Alternator requires bundles that produce up to two channels of floating point samples at consumer-audio sample rates, but these practical restrictions are not essential to the idea.

languages toward the same end, possibly in conjunction with composition/synthesis libraries such as Aleatora [8] (for Python) or JSyn [9] (for Java).

Alternator does not offer a new language for composition, nor does it enforce the use of any particular existing language. Instead, it leverages the development of WebAssembly and open-source projects such as Emscripten and Pyodide to allow the composer to choose among the large set of existing options.

1.2.2 Distribution

Generation is only part of the puzzle. After creating a generative piece using one of the many available languages or environments, the composer still needs to distribute it to listeners. Unfortunately, their options are limited; the authors lay out a few options and examples here and recommend consulting Levtov [6] for a more complete treatment.

One option is to render the piece as an audio file and distribute that instead, but then the composer must choose one fixed rendering of the piece, eliminating all other possible outputs. This approach is taken, for example, by Eno's *Music for Airports*: "although *Music For Airports* is the result of a process which can be exploited to deliver different results time after time, ad infinitum, the actual master recording of the album available today is merely a forty-eight-minute snapshot of one particular variation" [6]. To avoid this fate, a composer may opt to run a few renderings to generate and include multiple versions of the piece, but doing this for every piece (or even a single piece, if there are millions of possible outcomes) is infeasible.

Alternatively, a composer may distribute their work as software, which presents its own challenges. The composer may upload their piece somewhere as an executable and send out a link, but this requires listeners to run an untrusted binary executable. Even if listeners trust the artist completely or use a virtual machine, the playback experience is isolated: instead of their familiar music player, they see the executable's interface (which may lack basic controls such as pause, rewind, and seek), and there is no way to put it in a playlist with the rest of their music. Per Levtov [6]: "For even the most open-minded of listeners, these differences in procurement and procedure create a psychological disconnect between algorithmic music and static music, which limits the rate of its adoption. A future reality where algorithmic music is as common as static music will surely feature a listener experience where the various music formats sit together and on equal footing in the user interface."

Some artists have solved the distribution problem individually by leveraging their programming expertise; this route requires that the composer take on the additional role of programmer. For example, Jason Freeman's *Grow Old* EP consists of pieces that vary their output as the days pass.[4] New audio files are automatically generated each day and replace the old ones. Thus, the pieces are always available in the same place, can be readily played in a web browser without requiring the user to execute code, and realize new possibilities day by day.

Alex Bainter's Generative.fm [10] solves the same problem another way. On Generative.fm, listeners are presented with a web-based music player offering 50+ ambient music generators in a familiar interface. The interface is simple and consistent across pieces. Unlike the *Grow Old* EP, the pieces are generated at the time of playback. The generation occurs client-side, as the user's browser executes the code for each piece on demand.

Though these both effectively distribute generative music, they are artist-specific and technology-specific. The *Grow Old* EP is an album with all pieces written in RTcmix by Jason Freeman and rendered server-side. Generative.fm is a collection of ambient works written by Alex Bainter with the Web Audio API. As Levtov notes, "Web Audio represents yet another innovation that is founded on technologies that are unfamiliar to algorithmic music composers, this time the JavaScript programming language . . . [which] had never been used in the production of interactive audio until the introduction of the Web Audio API. . . . As such, although Web Audio represents a further widening of the potential audience for algorithmic music, it also comes with a significant narrowing of the group of potential composers capable of exploiting it until new tools which help bridge this gap are adopted" [6]. Alternator aims to bridge that gap by generalizing, serving as a platform where many composers, using many different languages, tools, and workflows, can share their music.

The nearest precursors to Alternator are automated Internet radio stations that stream real-time generative music, such as `rand()%`[5] and Streaaam [11]. Unlike the previous examples, these projects serve as independent platforms where artists (at least within a particular community) can submit their own work using a variety of languages. As radio stations, all listeners hear the same thing per the station schedule. Alternator similarly serves as a platform where artists can submit their own generative music, with the key distinction that it takes the form of a personal music player (such that listeners can independently listen to whatever they want, whenever they want) rather than a radio station. This would be difficult if Alternator rendered on the server-side like `rand()%` or Streaaam, because each simultaneous listener would require their own sandboxed composition process running on the server. Instead, Alternator executes compositions in the browser, which allows listeners to choose what they hear while avoiding the scalability issues inherent to server-side execution.

1.2.3 Platforms and Archives

Although it has a different focus, Alternator bears some relation to the mobile apps RjDj, MobMuPlat [12], and

[4]http://distributedmusic.gatech.edu/GrowOld/.

[5]`rand()%` has been down since 2007, but some information is available on the Internet Archive (https://web.archive.org/web/20070629095427/http://www.r4nd.org/rand_home.html) and at https://www.bbc.co.uk/radio3/cutandsplice/rand.shtml.

PdDroidParty,[6] as well as JSyn [9] and WebPd.[7] The three apps play Pure Data patches on mobile devices, and JSyn and WebPd enable interactive musical applications to run in the browser. Like these, Alternator runs computational compositions in the browser and on mobile, but it aims to support many different languages and provide a consistent music player interface.

Looking further back, MPEG-4 Structured Audio [13] deserves special mention. This forward-thinking standard likewise dealt with the distribution and playback of computational audio. However, it focused on the *efficiency* (in terms of compression) of describing audio computationally—an ill-fated tack given increasing bandwidth and storage—rather than the new possibilities afforded by such a general representation. Additionally, it inherited a synthesis framework from Csound (and older MUSIC-N languages) and embedded a high-level language (SAOL) in the standard itself, making it difficult to use alternative languages or approaches to computational composition (and more difficult to implement the standard). In contrast, Alternator aims to be a "common carrier," providing an executable format sufficiently low-level to accommodate all kinds of approaches to computational composition and sound synthesis.

Alternator differs from digital archival projects such as Rhizome ArtBase[8] and Miller Puckette's Pd Repertory Project [14]; it aims to enable any composer to share their work, rather than attempting to preserve works of historical significance. It also differs from platforms such as Scratch [15], EarSketch [16], and TunePad [17]; like Alternator, these are focused on computational art and music and enable sharing, but they are geared toward pedagogy rather than distribution and lack a dedicated interface for the listener.

In short, Alternator aims to fill a void. It endeavors to be a generative music player for any composer, in any language, on any device with a browser.

2 GOALS

Alternator's goal is to enable composers to easily distribute and share their compositions—including generative music, without compromises to fit it into static media. From a complementary perspective, Alternator's goal is to enable listeners to easily discover and listen to such compositions, experiencing the field of possibilities inherent in each piece without abandoning the features, interfaces, and portability they are used to finding in a music player.

This goal implies two others. The first is generality: to enable composers to easily distribute and share their compositions, Alternator must *play* their compositions. Therefore, it must avoid mandating a single chosen way to create computational compositions. If composers have to drop their existing tools, expertise, and workflows to fit into a mold, the battle is already lost. Alternator must strive to provide a general platform, suitable for the tools and languages that exist today and for those yet to be invented.

The second implied goal is stability. A platform is not useful to composers if it requires them to continually maintain and update their compositions every time the platform changes. This is unheard of in music streaming services for the simple reason that PCM data is forever—after a composer uploads the audio recording for a given track, no maintenance is required. Unfortunately, this situation is common in software, especially on the ever-changing web: an API for some service changes, or disappears entirely, and breaks anything that depends on it. This requires regular upkeep by software engineers to keep their software running as the foundations shift beneath it. As this is unacceptable for most composers, Alternator must endeavor to make its compositions more like audio files than living software with regard to maintenance requirements. The following section describes how Alternator meets this and the other goals in its design and implementation.

3 DESIGN AND IMPLEMENTATION

3.1 Architecture

The core of Alternator is a music player that executes computational compositions, generating samples on demand for playback. These computational compositions are represented as *bundles* containing all of the resources needed to run the piece: code (potentially in multiple languages), audio samples, MIDI data, models, etc.

These bundles could be executed server-side (streaming generated audio to each client) or client-side. Server-side generation, however, would require a backend capable of running a process for each concurrent listener (or else trade off some of this computation cost for storage cost or compromised variability).[9] Given that most digital music listeners today are listening on relatively powerful devices (smartphones, tablets, and laptops), client-side execution is feasible and inherently more scalable, because the available resources grow with the number of concurrent clients.

Thus, Alternator executes musical bundles client-side, in the browser. However, the browser does not understand languages like Csound or Pd. Until a few years ago, it only understood JavaScript; this limitation led to asm.js and then to development of WebAssembly (henceforth "Wasm"), a binary instruction format for a portable virtual machine. Alternator takes advantage of this development, using Wasm builds of libpd,[10] libcsound [18], ChucK,[11] RTcmix,[12] and

[6]https://droidparty.net/.
[7]https://github.com/sebpiq/WebPd.
[8]https://artbase.rhizome.org/.

[9]Note that the *Grow Old* EP, which does use server-side generation, avoids this problem by only generating each piece once per day. This optimization is only possible because it fits the artistic intent of the album; the pieces evolve slowly, day by day, so there is no need to generate them at the exact time of playback.
[10]Claude Heiland-Allen's empd: https://mathr.co.uk/empd/.
[11]https://github.com/ccrma/chuck/tree/chuck-1.5.0.6/src/host_web.
[12]The authors created a Wasm build in a fork: https://github.com/ijc8/RTcmix.

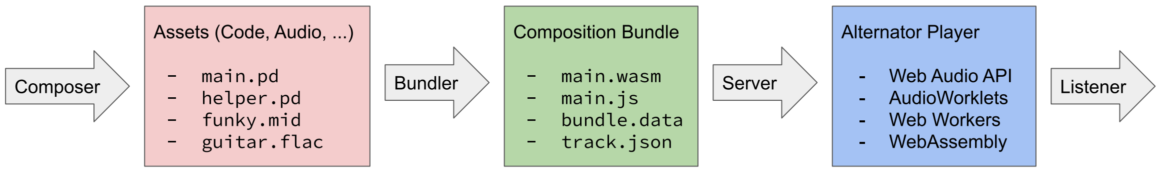
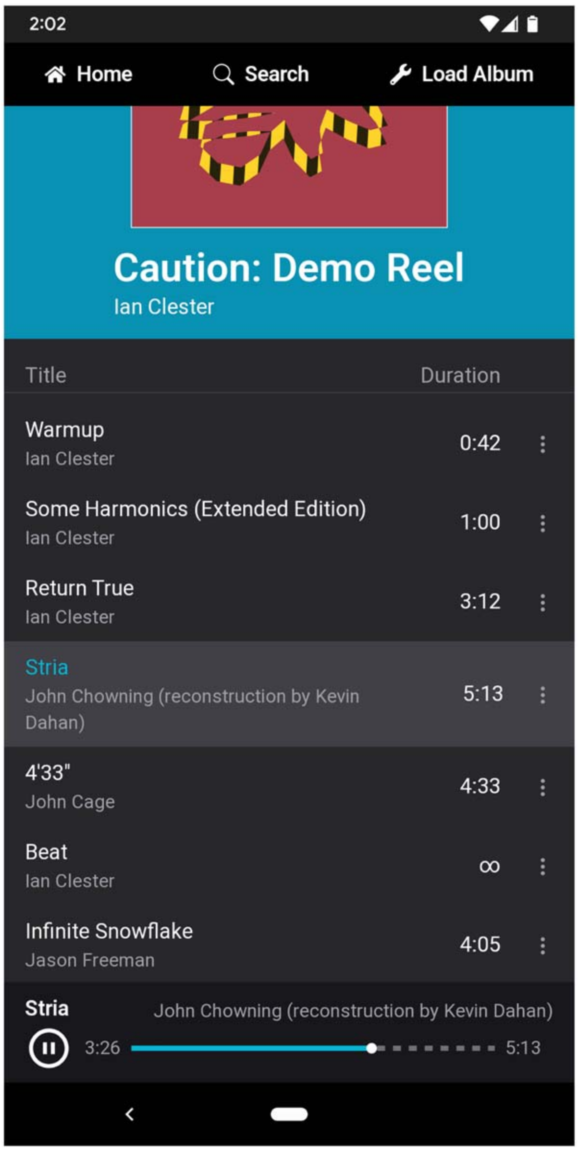


Fig. 2. How music flows from the composer to the listener in Alternator.


2:02
Home Search Load Album

Caution: Demo Reel
Ian Clester

Title	Duration
Warmup Ian Clester	0:42
Some Harmonics (Extended Edition) Ian Clester	1:00
Return True Ian Clester	3:12
Stria John Chowning (reconstruction by Kevin Dahan)	5:13
4'33" John Cage	4:33
Beat Ian Clester	∞
Infinite Snowflake Jason Freeman	4:05

Stria John Chowning (reconstruction by Kevin Dahan)
3:26 5:13

Fig. 3. It is feasible to generate and play compositions (such as this Csound reconstruction of "Stria") smoothly, in real time, in a browser or on a smartphone.

the like to enable the execution of patches and scores in the browser.

The complete flow of a piece of music in Alternator is depicted in Fig. 2. First, the composer creates a piece using their preferred tools. In this example, their piece consists of a Pure Data patch ("`main.pd`"), an abstraction ("`helper.pd`") used by the main patch, a MIDI file ("`funky.mid`") containing recorded performance data, and an audio sample ("`guitar.flac`"). The patches and MIDI/audio resources are then bundled together by the *bundler*, using the Pure Data template. The template consists of a Wasm blob and some glue JS. These are combined with the composer's assets and some metadata about the track (title, artist, etc.) to produce a bundle. Finally, this bundle is statically served to a listener's client, which executes the JS and Wasm (which may access the static assets) to generate audio in real time when the listener plays the track.

3.2 Execution

In the last subsection, Pure Data was used as an example, but the Alternator player does not have special support for any particular audio language built in. Rather, it supports a general executable format; the only requirement is that the executable can fill buffers with samples and (optionally) indicate when it is finished.

An Alternator executable is called a *bundle*. A bundle includes some JavaScript that implements two functions: "`setup()`," which takes the listener's sample rate as an argument and does any necessary preparation, and "`process()`," which takes in a buffer and fills it with samples. Unless the piece is written in JavaScript, the bundle also includes a Wasm blob. Typically this blob corresponds to a language runtime, but it may also correspond to the piece itself if it is written in, e.g., C/C++ or Rust.

Thus far, the authors have implemented templates for Pure Data, ChucK, Csound, RTcmix, Python (with the authors' composition framework Aleatora [8]), and static audio files (WAV and Ogg Vorbis). Alternator's design is intended to be future-proof: regardless of what music programming environments exist in the future, they can work in Alternator as long as they 1) can output samples and 2) fit in a Wasm blob. The first is a prerequisite for any audio programming environment (it must be audible), and the second is in a good state today thanks to efforts such as Emscripten and the Rust toolchain. It is of course impossible to predict the future, but existing support and the simplicity of the WebAssembly specification bodes well for its continued viability as a compiler target.

One important execution detail is endings: how does Alternator know when a piece is done? Pieces can end at any time by returning fewer than the requested number of samples in `process()`. This signals to the player that the piece is finished, and it will not call `process()` again. Some languages lack built-in notions of endings. In these cases, the bundle template can implement this notion in whatever way is most convenient. For example, the Pure Data template allows a patch to signal that the piece is over by sending a "bang" to a "`send`" object named "`finish`."

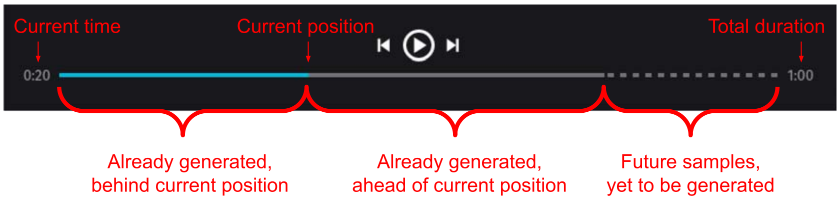

Current time
Current position
Total duration
0:20
1:00
Already generated, behind current position
Already generated, ahead of current position
Future samples, yet to be generated

Fig. 4. Alternator extends the visual language of conventional music players for computational music. Seeking to the dashed region, which represents the unknown future, will trigger faster-than-real-time rendering.

When the listener plays a piece, Alternator fetches the corresponding bundle. The JavaScript is executed in a Web Worker and typically fetches and instantiates a Wasm module. An AudioWorklet communicates with the Web Worker, transferring buffers to fill with freshly-generated samples. The AudioWorklet uses double-buffering with a moderate buffer size (1,024) to avoid hitches, connects to a `GainNode` for volume control, and finally gets audio out to the listener.

3.3 Listener Interface

In pursuit of its goal for listeners, Alternator adopts a familiar music player interface. The basic features of a music player include playing, pausing, resuming, seeking (skipping forward/backward within a track), and switching tracks. It is clear what these should do in the case of a static recording, which is pre-computed time-series data to be replayed. Alternator translates these core operations into a generative music context. From Alternator's perspective, music is something that can generate sound. To play, then, means to start generating sound; pausing pauses generation; and resuming picks up from the same point. However, some aspects of the interface require special consideration in a computational music context.

Seeking is less straightforward, but nonetheless has clear analogs. Seeking backward should replay exactly what was heard the first time, because the listener typically uses seeking backward to hear something again. So, Alternator maintains a growing history buffer for the playing piece. Seeking backward 10 s, for example, will replay the previous 10 s of generated samples, and then return to generating fresh samples from where it left off.

Seeking forward, however, requires playing samples that have not been generated yet. The only way to generate those samples is to reach that point in the piece. Thus, when the user seeks forward, beyond what has already been generated, Alternator generates the intervening samples as fast as possible (faster than real time, only limited by the performance of the bundle) and then resumes real-time audio generation and playback from the target position.

Alternator expands the visual language of the seek bar to convey these differences. Electric blue indicates the segments in the past, which have necessarily already been generated. Solid gray indicates samples in the future that have already been generated. Such samples only exist after a seek backwards; the user can seek forward to these samples instantly because they were already generated. Dashed gray indicates samples in the future that have yet to be generated; this region is still "potential sound," as yet unrealized in this playthrough and possibly indeterminate. When the current position is also the end of the generated samples (in other words, when there is no solid gray), the dashed line moves as it shrinks to convey activity: the system is in motion.

Another consideration for a computational music interface is duration. In an ordinary music player, duration is straightforward: because each recording is static, it is known in advance, and therefore, it must be finite, with a known duration. In Alternator, none of this is necessarily true. A piece may be infinite, continuing until the user intervenes. Or a piece may be finite, but with many possible durations (say, anywhere from 2:30–3:00). Due to the Halting Problem, it is impossible for Alternator to determine the duration of a piece in advance (or whether it ever ends). From a UX perspective, however, it is valuable to the listener to know what to expect. Thus, the composer declares the duration of a piece in its metadata. If the piece is finite, Alternator will display the piece's duration (or duration range) and use it in scaling the seekbar. If the piece is infinite, Alternator will display "∞" as the duration instead and will use the time of the last generated sample + 10 s as the duration in the seekbar,[13] so that the user can still skip ahead of the already-generated audio.

Finally, there is one element of Alternator's interface that has no analog in a conventional music player: the "view source" button. Pressing it opens a window showing the contents of the bundle: the code and assets that make up the piece. It also includes a link to the repository (elaborated in SEC. 3.4) containing the piece, which may have additional information about how the piece was made (such as documentation or more source code, if the piece is written in a compiled language). The implications of this feature are discussed further in SEC. 4.

3.4 Backend

The discussion so far has focused on the Alternator client and bundler. By comparison, the backend is simple: the only essential thing is static file hosting for bundles and some mechanism for composers to upload their work. Other features such as search, recommendations, and playlist management require more from the backend, but these have been solved in existing music players and do not require changes in a computational music context.

[13] In the future, this seek-ahead window may be user-configurable.

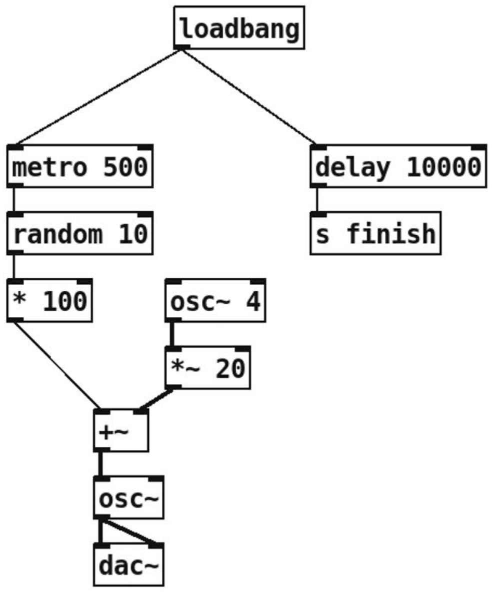


Fig. 5. Example Pure Data Patch.

Because the backend is not the focus, the existing open-source ecosystem is leveraged. Drawing inspiration from "utterances,"[14] a comment widget backed by GitHub Issues, Alternator uses GitHub as its backend for the time being. Albums in Alternator—which are both software and music—take the form of GitHub repositories. An album repo includes all the track bundles, some JSON metadata, and cover art.[15] Alternator can discover and search through these albums using the GitHub API because the repositories are marked with the "#alternator-album" tag.

3.5 Composer Interface

The preceding sections have described how Alternator works and what it looks like for the listener. This section demonstrates how it works for the composer with a concrete example.

To get started, a generative piece to bundle is needed. For this example, the authors use the simple Pure Data patch shown in Fig. 5, which plays random harmonics with vibrato for 10 s. This is an ordinary patch which is playable on its own if Pure Data is installed, but there are two details of how this connects with Alternator that are worth mentioning.

First, as in normal Pd usage, "loadbang" sends out a bang when the patch is loaded. In Alternator, this happens when the user plays (or resets) the track. Second, the patch sends a delayed bang to a special destination called finish. As mentioned in SEC. 3.2, the Pure Data template for Alternator listens for this to indicate that the track has finished playing. This is necessary because Pure Data, like other signal-processing–oriented languages, does not have a built-in notion of "endings," and it demonstrates the adaptability of Alternator's model to different languages and environments.

Next, the authors need to set up the right structure and provide some metadata. First, they create a directory for their album. Inside, they name their album art "my-album-cover.svg" and create "album.json" with the contents:

```
{
  "title": "My Cool Album",
  "artist": "Jen Rétive",
  "cover": "my-album-cover.svg",
  "tracks": ["bundles/my-first-track"]
}
```

Next, the authors save the patch as "main.pd" in a subdirectory called my-first-track. Inside, they create a file called "track.json" with the contents:

```
{
  "title": "My First Track (Some Harmonics)",
  "artist": "Jen Rétive",
  "duration": 10,
  "channels": 1
}
```

Finally, they run the bundler:

```
$ ../alternator/bundle.py pd my-first-track
Creating output directory: bundles/my-first-track
Copying main.js from pd template.
Copying main.wasm from pd template.
Bundling /main.pd
Saving finalized track.json.
Finished bundle: bundles/my-first-track
```

The bundler takes the patch and bundles it together with everything needed to run it (the Pure Data runtime compiled to WebAssembly and some glue code). Everything that Alternator needs to play the piece is stored in "bundles/my-first-track." If the patch had any additional assets it needed (e.g., a .wav file or an abstraction in another patch), these would likewise be bundled up in the track.

At this point, the authors have a playable bundle in Alternator (Fig. 6). If it is hosted somewhere on the Internet, the authors can then distribute a working link like https://ijc8.me/alternator/?u=https://example.com/my-album. If they want their album to be discoverable via Alternator's search feature, the authors can put it on GitHub as discussed in SEC. 3.4.

4 DISCUSSION AND LIMITATIONS

In this paper, the authors have presented Alternator, a platform for distributing and playing computational music compositions: music that is generated on-demand, such that each track may encompass a field of possibilities rather than a single fixed recording. In this section, they reflect on the implications of Alternator's design and its limitations.

[14]https://utteranc.es/.
[15]Beyond the bundles and metadata, album repos can contain anything else that a repository can: source code, a README, a LICENSE, documentation, etc.

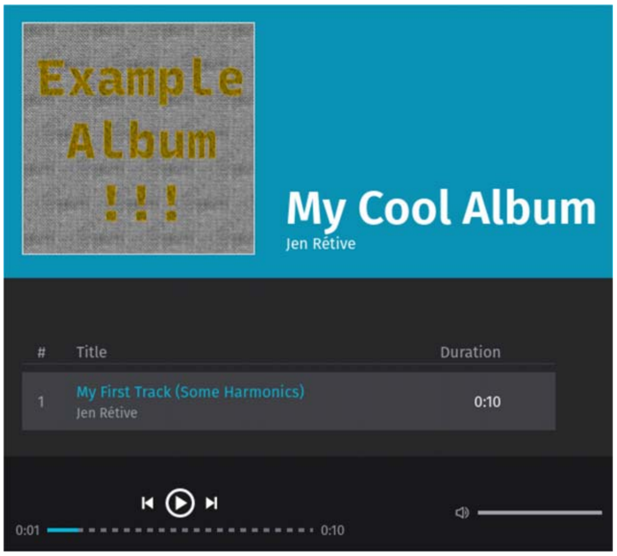


Fig. 6. The authors' example patch is now playable in Alternator!

4.1 Music as Code

Alternator's premise is distributing music as code (instructions that generate audio) rather than pre-rendered audio. This premise suggests exciting possibilities: an enthralled listener who thinks "this is amazing, how did they do it?" can look at the source and start to get a glimpse. Other musicians (typically avid listeners) can go even further, forking the repository and experimenting with it to make it their own. These ideals are shared by other contemporary projects, as in Fluid Music's [19] vision for music production and sound design shared as code.

Alternator does not guarantee any of these possibilities. Most music released today is opaque, with alternatives such as releasing stems or digital audio workstation (DAW) projects being the exception rather than the rule. A composer is free to follow this trend and distribute their music as an opaque binary blob, as with ordinary closed-source software. Indeed, this may be necessary for proprietary systems such as Max/MSP to be usable in Alternator.[16] Alternator can suggest another way through its design (the "view source" button, the repository link, hosting on GitHub) and practices (releasing the player, templates, and demo albums as open-source), but ultimately the culture will be determined by the artists.

4.2 Interactive Music

One intentional limitation is that Alternator only deals with purely generative music. It does not handle reactive or interactive music, in which the output depends on external user input. The reason for this is simple: Alternator is a music player, not an instrument. All generative music fits essentially the same interface (samples out), which translates well to the conventional music player of static recordings. In contrast, interactive music comes in many shapes and sizes, requiring different input devices that the user may or may not have or that might make sense for a listener on a smartphone but not on a laptop. It also expects a different mode of engagement from the listener. Finally, from a technical standpoint, any potential interface for interactive bundles will be more complex and likely to change than the simple `setup()` + `process()` required for generative pieces. Such a platform would thus be more prone to breaking existing compositions, making it less reliable and thus less useful for composers who expect their pieces to remain playable without maintenance.

That said, interactive music is a vibrant area that offers unique experiences impossible with purely generative works. It may be worth exploring a separate UI/API in Alternator for interactive compositions, for composers willing to accept increased uncertainty (and maintenance requirements) in exchange for interactivity. The possibility of compositions (interactive or autonomous) generating video as well as audio is likewise intriguing, especially given the large body of work and literature on generative visual art.

4.3 Updates and Future Work

So far, the authors have described the version of Alternator presented at WAC [2], in which the API is defined in JavaScript and thus each bundle contains some JavaScript "glue code" (even if the main body of code is compiled to WebAssembly). The JavaScript code runs in a Web Worker, resulting in imperfect isolation. For example, a well-meaning composer could exploit this to fetch external data (e.g., the current weather) over the network for their compositions—rendering the composition broken if the external resource should move or disappear. A less scrupulous composer could exploit this to, e.g., run a browser-based cryptominer using spare cycles.

For the goal of stability, composition code would ideally be totally self-contained WebAssembly, with as little API surface area shared with the browser as possible. Since the authors' presentation at WAC, they have experimented with moving to pure Wasm bundles, using the WebAssembly System Interface (WASI) instead of Emscripten-generated glue code to facilitate loading assets.[17] In addition to fully isolating generative bundles, the authors have found that this move has several other benefits. For one, the external state observed by the bundle [via WASI host functions such as "`random_get()`" and "`clock_time_get()`"] can fully be controlled, and thus particular outputs can be reproduced; this functionality could allow listeners to save and share interesting variations by storing random seeds. What's more, generative bundles outside of the browser can be played back, because this requires only a Wasm VM rather than a complete browser engine. In the future, this may enable generative playback through a dedicated app (lighter on resource use than the browser) or even on embedded devices. As a proof of concept, the authors have implemented a simple generative music player in Rust that can play pure Wasm bundles.

[16]Hybrids are possible: the runtime engine may be closed, whereas the composer's patch is open. Also, in the case of the Max add-on RNBO, exported code can be distributed under GPLv3.

[17]This version of Alternator is available at https://github.com/ijc8/alternator/tree/next/wasm.

One issue that remains is that Alternator uses a "statically linked" model for bundles, in which they are expected to come with everything they need to run. This means that every piece using Pure Data comes with its own Wasm-compiled version of libpd. To reduce bundle size and load times, it may be worth allowing composers to simply reference common templates (provided by Alternator or perhaps hosted on an CDN) rather than always bundling them with the composition. At a minimum, the player could cache `.wasm` files in `localStorage`.

Finally, a major limitation is lack of tooling. Currently the only way to determine if a piece will play smoothly on a given device and browser is to try it and see. Profiling tools and benchmarks on common devices could give composers more confidence in distribution. Also, the existing workflow for creating bundles is straightforward but technical. It would be simple enough to make a graphical, web-based version of the bundler (possibly embedded in the Alternator client). It would be even better to fit into artists' existing workflows, integrate with DAWs, and provide access to generative possibilities without requiring composers to code. Since presenting at WAC, the authors have begun to integrating generativity into the DAW with their work on LambDAW [20]; in the future, the authors hope to enable the export of generative bundles directly from the DAW and thus make generativity as accessible to composers as possible.

5 CONCLUSION

Behind all the technical details and design discussion, a question lurks in the background: will Alternator, or something like it, have an impact? Will generative music reach a wider audience? Is there an appetite for distributable music that changes?

The popularity of dynamically generated playlists (as on Pandora and Spotify) and endless YouTube music streams suggests an appetite for familiarity with variety: *give me more like this, but different*. Generative approaches in other media have proven popular: procedural generation is used in games like Minecraft, Dwarf Fortress, and countless roguelikes to make the game more fun and improve replayability.[18] And musicians—including songwriters, composers, and producers—are not shy about trying out and adopting new technologies on their own terms.

Ultimately, the appeal of generative music, like any music, depends on its content. Much computer music has historically been "pure" computer music, generated in-the-box with techniques, timbres, and textures that set it far apart from its contemporaries. Or else it has stuck firmly to certain genres, such as electronic and ambient music. There is opportunity to broaden the horizons of computer music, blur the boundaries, and explore hybrids: music with both conventional and computational aspects. For example, a pop musician might record several good takes in a DAW and then create a computational bundle that chooses a path through them dynamically at the time of playback. Uses like these may serve to bring generative music to a wider audience and enable more artists to take advantage of its possibilities, and the authors hope to engage with artists and conduct user studies to explore the space of generative hybrids in future work.

As Eno put it in the essay that opened this paper [1], "I too think it's possible that our grandchildren will look at us in wonder and say, 'You mean you used to listen to exactly the same thing over and over again?'" Realizing this future requires bridging the gap between the composer and would-be listener of generative music; it requires a distribution channel that opens up the possibilities of generative music to both.

[18]Indeed, some games feature generative soundtracks. In this sense, the most successful distribution platform for generative music thus far might be Valve's Steam.

6 REFERENCES

[1] B. Eno, *A Year With Swollen Appendices: Brian Eno's Diary* (Faber and Faber, London, UK, 1996).

[2] I. Clester and J. Freeman, "Alternator: A General-Purpose Generative Music Player," in *Proceedings of the International Web Audio Conference*, paper 9 (Cannes, France) (2022 Jul.). https://doi.org/10.5281/zenodo.6767436.

[3] J. Cage, "Composition as Process: Indeterminacy," in C. Cox and D. Warner (Eds.), *Audio Culture: Readings in Modern Musi*c, pp. 176–187 (Continuum, New York, NY, 2004).

[4] S. Reich, "Music as a Gradual Process," in *Writings on Music, 1965-2000*, pp. 34–36 (Oxford University Press, New York, NY, 2002).

[5] K. Essl, "Algorithmic Composition," in N. Collins and J. d'Escrivan (Eds.), *The Cambridge Companion to Electronic Music*, Cambridge Companions to Music, pp. 107–125 (Cambridge University Press, Cambridge, UK, 2007). https://doi.org/10.1017/CCOL9780521868617.008.

[6] Y. Levtov, "Algorithmic Music for Mass Consumption and Universal Production," in R. T. Dean and A. McLean (Eds.), *The Oxford Handbook of Algorithmic Music*, pp. 628–644 (Oxford University Press, New York, NY, 2018). https://doi.org/10.1093/oxfordhb/9780190226992.013.15.

[7] G. Wang, "A History of Programming and Music," in N. Collins and J. d'Escrivan (Eds.), *The Cambridge Companion to Electronic Music*, Cambridge Companions to Music, pp. 55–71 (Cambridge University Press, Cambridge, UK, 2007).

[8] I. Clester and J. Freeman, "Composing the Network With Streams," in *Proceedings of the 16th International Audio Mostly Conference*, pp. 196–199 (Trento, Italy) (2021 Sep.). https://doi.org/10.1145/3478384.3478416.

[9] P. Burk, "JSyn - A Real-Time Synthesis API for Java," in *Proceedings of the International Computer Music Conference (ICMC)*, paper 289 (Ann Arbor, MI) (1998 Oct.). http://hdl.handle.net/2027/spo.bbp2372.1998.289.

[10] A. Bainter, "Generative.fm," in *Proceedings of the International Web Audio Conference (WAC)*, p. 148 (Trondheim, Norway) (2019 Dec.).

[11] F. Hollerweger, "Streaaam: A Fully Automated Experimental Audio Streaming Server," in *Proceedings of the 16th International Audio Mostly Conference*, pp. 161–168 (Trento, Italy) (2021 Sep.). https://doi.org/10.1145/3478384.3478426.

[12] D. Iglesia, "The Mobility is the Message: The Development and Uses of MobMuPlat," in *Proceedings of the International Pure Data Convention*, pp. 56–61 (New York, NY,) (2016 Nov.).

[13] E. Scheirer, "The MPEG-4 Structured Audio Standard," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, vol. 6, pp. 3801–3804 (Seattle, WA) (1998 May). https://doi.org/10.1109/ICASSP.1998.679712.

[14] M. S. Puckette, "New Public-Domain Realizations of Standard Pieces for Instruments and Live Electronics," in *Proceedings of the International Computer Music Conference (ICMC)*, paper 59 (Havana, Cuba) (2001 Sep.). http://hdl.handle.net/2027/spo.bbp2372.2001.059.

[15] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The Scratch Programming Language and Environment," *ACM Trans. Comput. Educ.*, vol. 10, no. 4, paper 16 (2010 Nov.). https://doi.org/10.1145/1868358.1868363.

[16] B. Magerko, J. Freeman, T. Mcklin, et al., "EarSketch: A STEAM-Based Approach for Underrepresented Populations in High School Computer Science Education," *ACM Trans. Comput. Educ.*, vol. 16, no. 4, paper 14 (2016 Sep.). https://doi.org/10.1145/2886418.

[17] J. Gorson, N. Patel, E. Beheshti, B. Magerko, and M. Horn, "TunePad: Computational Thinking Through Sound Composition," in *Proceedings of the Conference on Interaction Design and Children (IDC)*, pp. 484–489 (Stanford, CA) (2017 Jun.). https://doi.org/10.1145/3078072.3084313.

[18] S. Yi, V. Lazzarini, and E. Costello, "WebAssembly AudioWorklet Csound," in *Proceedings of the International Web Audio Conference (WAC)*, paper 24 (Berlin, Germany) (2018 Sep.).

[19] C. J. Holbrow, *Fluid Music*, Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA (2021 Sep.).

[20] I. Clester and J. Freeman, "Composing With Generative Systems in the Digital Audio Workstation," in *Proceedings of the 3rd Workshop on Intelligent Music Interfaces for Listening and Creation (MILC)*, paper 15 (Sydney, Australia) (2023 Mar.).

THE AUTHORS

Ian Clester

Jason Freeman

Ian Clester composes music and programs. His research interests include the design of languages, environments, and tools for composing and performing music with computational or generative aspects. His composition and performance practice includes algorithmic composition, instrumental improvisation, and live coding, and he is a founding member of the MIT Laptop Ensemble. His work has been recognized with best paper (Web Audio Conference 2022) and best poster (Audio Mostly 2021) awards. He received his B.Sc. in Electrical Engineering and Computer Science (EECS) and Music and his M.Eng. in EECS from MIT, and he is currently pursuing a Ph.D. in Music Technology at Georgia Tech.

•

Jason Freeman is Professor of Music at Georgia Tech and Chair of the School of Music. His artistic practice and scholarly research focus on using technology to engage diverse audiences in collaborative, experimental, and accessible musical experiences. He also develops educational interventions in K-12 and university environments that broaden and increase engagement in STEM disciplines through authentic integrations of music and computing. His music has been performed at Carnegie Hall, exhibited at ACM SIGGRAPH, published by Universal Edition, broadcast on public radio's Performance Today, and commissioned through support from the National Endowment for the Arts. Freeman's wide-ranging work has attracted funding from sources such as the National Science Foundation, Google, and Turbulence. It has been disseminated through over 80 refereed book chapters, journal articles, and conference publications. Freeman received his B.A. in music from Yale University and his M.A. and D.M.A. in composition from Columbia University.