(AUTOMATED) LISTENING IS MORE MEANINGFUL WITH THE SOURCE CODE

2

ABSTRACT

Machine Listening encompasses many active areas of research. An especially interesting area is nonhuman audition of music, including music information retrieval and music recommendation. Currently, most approaches analyze music audio data without information about its origins, i.e. without information about how the audio was generated. For music audio, this specifically means knowledge of music-making (performance, composition, meta-composition, etc.). For example, music recommendation via content-based models might focus on acoustic features without information about how those features arise within a generative model of music. In this paper, we focus on music that is generated partially or wholly by code (i.e. source-code), and how both simple and more theoretical knowledge about source-code can (a) inform listener expectations from an analytic standpoint, as well as (b) reveal the dependencies of outputs on inputs for metacomposers of music-making systems from a generative standpoint and (c) resolve problems beyond the reach of SOTA machcine listening problems while introducing intriguing new problems. After introducing the spectrum of "codedness" and other algorithmic properties of "coded" music, we present three case studies, which progress from a hermeneutic non-automated analysis of the source code of a very algorithmic piece of music to progressively more automated analyses of source code. Ultimately, we discuss future directions for analysis of source code-audio pairs, as well as domains which can benefit from a similar integration of data and data-generator.

1. WHAT IS MUSIC SOURCE CODE, AND WHY STUDY IT?

One can define music source code relative to a model of score-interpreter-output. A interpreter P is an interpreter which takes in a source of some form x, and outputs an audio signal $P(x) = y$. In traditional instrumental performance, for instance, the source is a visual score, the interpreter is an orchestra, and the audio signal is the recording. However, in many musics produced today, the score is itself a program, and the interpreter is an automated compiler that takes in such a program and produces an audio file. Thus, when one talks about music source code, it refers to any artifact which can be *directly* processed by an interpreter to produce an audio output. The word "directly" implies a useful criterion for evaluating what constitutes source code material - for instance, a process which caused a composer to hard code a particular device chain for each track of their audio project is not itself source code, since the process isn't itself evaluated by the interpreter, but the device chain itself is part of the source code.

1.1 Limitations of Machine Listening

1.1.1 Literature Review on Machine Listening

Machine learning techniques have gained significant momentum in the field of audio analysis in recent years, particularly in areas such as audio classification, audio event detection, and speech recognition. A pioneer in the use of machine learning for audio analysis is [9], who utilized traditional techniques for signal processing for audio signal classification into speech and non-speech categories. After that, several studies have employed various machine learning algorithms such as decision trees [7], support vector machines [13], and other early machine learning algorithms.

In recent years, deep learning models, such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), have been widely adopted for audio analysis tasks and shown to outperform traditional machine learning methods [10]; [19]). For instance, [4] proposed a deep audio feature learning framework that leverages both CNNs and RNNs to improve the accuracy of music emotion recognition. [14] proposed an attention-based deep learning model for audio event recognition and showed that the model outperforms traditional CNN-based methods. Another study by [20] introduced an attention-augmented convolutional neural network for music genre classification and demonstrated that the attention mechanism helps to improve the performance of the model.

1.1.2 Literature Review on Musical Machine Listening

In the domain of Machine Listening to music, deep learning models have shown significant advances in recent years. One of the key challenges in this area is music classification, which involves categorizing music into various genres, moods, or styles. A number of studies have employed deep learning models such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) to tackle this problem. For example, [21] proposed a music genre classification model that leverages a combination of CNNs and RNNs to capture both local and global features of music. The study showed that the proposed model outperforms traditional machine learning algorithms.

Another recent study by [6] introduced a deep transfer learning framework for music classification that utilizes pre-trained CNNs to extract high-level features from music. The study showed that the proposed method outperforms traditional machine learning algorithms and can effectively handle the challenge of small-sized datasets in music classification.

In addition to music classification, deep learning models have also been applied to other tasks in Machine Listening to music such as music auto-tagging [11] and music similarity measurement [17]. These studies demonstrate the potential of deep learning models for various tasks in Machine Listening to music and highlight the need for further research in this area.

1.1.3 Limitations of Current Machine Listening

Machine Listening, and machine listening to music in particular, is in its infancy. Even determining what instruments are playing [18], or how to annotate in MIDI any instrumental parts [5], is extremely non-trivial and usually imperfect.

One possible explanation for difficulty making progress in machine listening is that integrated analysis is necessary for understanding interdimensionality of highly complex or multidimensional systems, such as nonlinear effects in nonlinear systems. Blind source separation is often provably undedetermined. According to most paradigms, the key to solving underdeterminism is locating and then imposing external structural constraints [12]. Integrated listening implicitly proposes structural constraints by narrowing the space of possible phenomena that can be occurring so that they fit all modalities.

1.2 Source-code analysis reveals new research directions in (automated) listening

Quoting from Scheirer's thesis on music listening [15], one important auditory scene analysis task is determining whether two pieces are similar. On the surface, this may seem simple with the correct tools (some sort of metric of audio similarity between small bits of audio combined with a way to process ordered sequences). However, when we consider the program, a key question arises: Do we define a piece with similar meta-orchestration (use of basic materials and sound processing techniques) as similar, even if we can't necessarily hear the similarity? To realize that this might be something we'd want to consider, note the extended analysis performed on Charles Ives' String Quartet in relation to the roles assigned to various instruments. These instruments can't be source-annotated by audio alone (possibly even by a human), because they are voiced by Violins 1 and 2. Similarly, suppose two electronic artists come up with what sounds like very similar files, but one artist used 100 different sound sources in their piece, while the second used a single sound source processed in 100 ways. Is it the same piece?

1.3 Music as perceptual/conceptual/neurosymbolic

Cognitively, we deal with highly structured data (e.g., programs) and gestalts (e.g., raw audio) differently. "Music" is both! Part of the music creation process is about hearing things that are perceptually appealing (e.g., I really like this sound more with more reverb added), and another part can be about constructing a narrative or logical process that is logically appealing or elegant (e.g., Occam's razor). Research in both machine learning and cognitive psychology groups have focused on "Neurosymbolic" learning, or learning about the relationship between the perceptual and the logical/conceptual [8].

1.4 Even non-machine analysis can benefit from a focus on codedness and source code

In this paper we focus on integrated listening where the integration is between source code x, an interpreter P, and the audio y that results by applying $P(x)$. In terms of causality, the implications of this paradigm is that when we read source code in a language such as Max/MSP or Ableton, we are dealing with a very special case where we know how our causal model should operate - we know the semantics of Max/MSP, so given a program and a audio file, we know how we should be able to interpret the latter as being caused by the former. This does not always mean that we will be able to predict the audio before listening to it from the source code - the code may involve randomness or complexity beyond our cognitive capabilities to process in advance. But we can be sure that what we are listening to is nevertheless caused by that code being executed by the interpreter. Such a situation occurs more often than one would thing - one can interpret a Mozart string quartet visual score as x, and the string quartet players as P, leading to a recording y. Such generalization is useful, for example, when measuring "codedness" (a metric defined below): There's a difference between how we analyze a Beethoven score and a graphic Morton Feldman score, and this can be analagous to the difference between analyzing a DAW project that only uses MIDI + instruments without modification, and analyzing the output of a Max/MSP program; as such, measuring "codedness" in the context of digital performance also could be applied to modern graphic and process music. However, there are also useful distinctions to be made between performers and programs which interpret source code: The relationship between source code and audio output is often deterministic, or only non-deterministic in very prespecified ways. Even if the source code is non-deterministic, the "code interpreter" has a predefined semantics, unlike in human music where the composer won't know the "semantics" of the first violin player.

1.5 (Automated) listening as a composer can be enhanced by focus on the source code

One way of learning how to compose music of a certain kind is imitation. It is much easier to learn to imitate how a particular reverb is used by learning how it's parameters change in response to time or other changes in the piece then trying to audiate that - we simply don't have the educational system for that kind of audiation. Furthermore, in learning to compose algorithmic music, it's crucial to understand the algorithms that produced them if you want to produce something that's equally algorithmic. Mechanistic/rule based methods in composition have always been around, but with computers have proliferated; it is important to study compound, complex rules, algorithms or programs, such as their complexity, logical control flow, etc in order to understand how to make "realistic" music in many genres, not exclusively to "algorithmic music" (note the many tutorials on Youtube dedicated to explaining how to chain effects in Ableton, an interpreter used for creating popular hip-hop).

1.6 Defining codedness

While the above examples show that codedness is clearly a spectrum, and there are several intuitive ways of thinking about the level of codedness, we only provide a tentative definition:

Relative to a listener L (who is abstracted as a function from an audio file y to facts that are definitely true to the listener about the audio file y) and a universe of predicates that could be true about any audio file y', how many additional facts can be obtained by having the source code? To see this is a viable definition in an integrative framework, consider the difference between a sonata by Mozart and the output of a Markov Model on melodies. A composer with perfect ability to transcribe playback of a Mozart piano sonata by MuseScore or the same piano soundfont playing a Markov model-generated piece would learn nothing new about the Mozart sonata by being given the piece as a visual score (as they would be able to ascertain every such fact by listening), but would learn a considerable amount about the Markov model piece, such as its likelihood among all the possible pieces that could have been generated by the "composer", or that the most unexpected moment in the piece should have been when the motif "A C B" was followed by "A B C". Note that this definition assumes that "facts about the piece" include its likelihood of having been generated, which some consider outside of the scope of analysis of the given file. Thus, we can only define codedness (as noted above) relative to a universe in which a class of predicates exist. In a universe where *probabilistic information* is relevant, the Markov piece is more coded than the Mozart sonata; in a different universe it might not be. Similarly, if the universe includes facts about both instrument identity and instrument sound, such as a *hermeneutically rich* universe, and if the listener has perfect transcription abilities, a piece which manipulates a pair of instruments into radically different timbres using an additive synthesis algorithm is more coded than a traditional score for a Woodwind Quintet with equivalent sound, because we learn "which instruments did what" in the first case, while in the second we learn nothing new in this respect. Finally, in a universe where the composer's mental process is relevant (an *intentional universe*), we learn a great deal more by examining a deterministic algorithm involving morphing Euclidean rhythms which generates MIDI than by examining a MIDI piano-roll alone (and thus the algorithm is more coded), because the algorithm tells us that the composer intended for it to display the nature of evolution of nearly-circular rhythms.

Other problematic definitions of codedness One might argue that there are better definitions of codedness that don't require parameterizing on a listener model and a relevance universe. Here we present several alternative candidates and why they are problematic.

1. Kolmogorov complpexity may seem like a natural fit. However, note that we are measuring not how well a piece of music could be coded, but how coded the implementation actually is. Two pieces with identical output but different implementations are coded differently, even if they'd have the same Kolmogorov complexity.
2. Ratio of file size to output length (for comparing between programs in a similar language) - such an approach may work when we are comparing a python program which literally hard-codes a MIDI sequence than a Markov model which generates MIDI, for instance. However, in general file size is not a good measure of codedness, as, for instance, in an Ableton file the use of a single device with a long definition will cause file size to grow massively, even if one is only using that device as a MIDI interpreter without changing any of its parameters.
3. User-facing complexity (measuring complexity only of the aspects that the user interfaces with) or Human-Resource-Weighted File Length - the idea here would be to either measure the expressivity of the parts of the code that the user is relying along non-trivially (parameterized by some definition on non-triviality of usage), or weight the file size by how labor-intensive doing each action manually would be. This is probably the most successful alternative definition. However, it is even more arbitrarily parametric than our proposed definition, since it requires knowing what is a "trivial" usage of a device as opposed to a "coded" use of it (just passing the problem farther along), and weighting lines of code by resource consumption requires making arbitrary guesses about how much effort goes into each piece of the code, and, by extension, how each piece of the code is produced. Note that we are only given the source code, not the steps the composer took to achieve the source code!

1.7 Properties associated with but not equivalent to codedness

1.7.1 Diversity of Behavior

Many pieces of music that are "coded" to some extent are not only non-deterministic, but can produce very different sounding output in each performance, for example *open form*. It is definitely true that more algorithmic pieces tend to produce a wider variety of behavior if they do exhibit nondeterminism, since this non-determinism may be more structural. However, it is possible to create pieces which are completely algorithmic and not deterministic at all. Young's work on sonifying algorithms [22] provide several examples - according to the author, there was a strict bijection between the algorithms being sonified and the output, so that the output was strictly determined by the source material, and the same source material wasn't able to produce two different outputs.

1.7.2 Audiational agency

"Audiational agency" is a quantification of the level of uncertainty a composer has about how their music will sound at any point before, during, or after composition (intention-consequence gap, which arises because of uncertainty about

the line of actualization from premusic to music). Unlike codedness, audiational agency in its truest sense is not completely determined by the code itself - some composers write the whole algorithm before listening, some will listen back after every change. It also depends on how complex of interactions we imagine the composer can precalculate in their head.
We will choose to define audiational agency of individual changes to the code and relative to an “audiational task” and a resource (short term/long term memory, computing abilities, pitch sensitivity) model R. Fix an audiational “task” that a listener might have to accomplish (e.g., what pitches are playing, what the dynamics are, which instruments are more salient, etc.) How much does the one change to the code influence the composer’s ability to predict the results of performing the task on the code’s output, assuming they’ve heard the output right before changing the code? Again, more codedness tends to coincide with less audiational agency, but a listener can’t assume this relationship is strict. **From hermeneutic to automated analysis of source code-audio pairs: three case studies**

2. A CLOSE READING OF THE SOURCE AND OUTPUT OF “MIXING” BY MATTHEW CHUNG

How might a listener be informed by the source-code of an algorithmic composition?

A recent example in algorithmic composition is Chung’s *Mixing* system [3], a program that generates an indefinite number (a class) of renditions, each of which is presentable as a complete musical work itself. Each rendition has a unique, autonomously (without human intervention) generated musical form (macrostructure), phrasing (mesostructure), and synthesis texture (microstructure). The source-code is in Python, SuperCollider, and Open Sound Control. A listener can consider (i) a single piece, (ii) a set of pieces, or (iii) the entire set of all pieces generated by the *Mixing* system, and can also consider from the perspectives of (iv) analyzing the system itself, and (v) the methodology by which the system was created (metacreation).

Perhaps the most straightforward value is that knowing the source-code helps one listen with informed *expectations* or confirm aspects of a listening. For example, one can learn simple information from the source-code of *Mixing* that could be equivalently inferred from multiple listenings to multiple renditions, e.g. that there is only one stream of grains using FM synthesis, that Lorenz attractors are used, how random variables and conditionals are used, and that some parameters are hard-coded while others are left configurable by the user. In general, studying how the program produces music is a kind of ‘ultimate answer guide’ to many questions that can be uncertainly answered by inference and multiple listenings, especially those questions that have to do with expectations (e.g. tension arising from uncertainty about the fulfillment or subversion of expectations). Source-code informed or confirmed expectations in turn can help one direct resources to essential features of the work and avoid wasting energy deciding whether a feature is relevant or not. For example, one can study the source-code of algorithmic compositions to infer typical features of distributions by either statistical or algorithmic methods. Thus, *source-code can help inform listener expectations: what to expect about one particular rendition, about a statistically typical rendition, about all renditions, and also what not to expect, such as what is not possible in any renditions*. This closely resembles listening to a more traditional piece of music with the score, where one can visually confirm aural information.

A value for composers is that one can study the particular way an algorithmic composition is implemented, which can be thought of as a kind of *reverse engineering*. This is much like studying the brushwork of a painter or orchestration approaches of a composer. How did they produce that visual or aural effect? For algorithmic composers, how did they invent particular algorithms to solve general compositional problems, such as producing music that is interesting for its entire duration, that exhibits diverse behavior between renditions, that exhibits musical spontaneity, and so forth? In the case of *Mixing*, a particular important aspect is that the source-code is autonomous, i.e. produces a complete musical work without any intermediate human assistance. This required inventing not only a parameter space that was interesting in many regions but also trajectories that consistently passed through them in interesting ways. This involved compositional intuition about what sounds or parameter configurations might be interesting, which was formalized and captured by the source-code. Thus, *one can study source-code of algorithmic music to extract formalizations of compositional intuitions*. This resembles listening to those aspects of music that ‘work’ and studying the score to learn why (note that one must listen to know what ‘works’).

Generalizing this discussion, it is valuable to recognize that *the same ends can be reached by entirely independent means*. For example, a feature that seems perceptually important might be produced as an accidental artifact, an instance of randomized behavior, or specifically hard-coded. Furthermore, this sometimes communicates composer intent; the feature might have been produced intentionally by the composer through meticulous craft, or ‘discovered’ as-is and left carefully undisturbed (any visual artist knows it is almost as important to know when to stop adding and adjusting!), or unimportant to (or simply unnoticed by) the composer. Thus, *knowing the means taken to achieve a particular end might help one understand the end better, sometimes but not always including understanding composer intention*. In terms of *Mixing*, there is a rendition called *Sensory Maps, Version C* which features a few unusually long, low-frequency grains about 9 minutes in, that sound like held tones and might stand out perceptually to some listeners simply due to the nature of human hearing and traditional listener conditioning. A listener might wonder if this ostensible anomaly is hard-coded, accidental, or simply atypical but still entirely algorithmic. Source-code would verify that it is atypical but purely algorithmic (i.e. a statistical outlier). Thus, *source-code can sometimes help understand composer intentions, but ultimately only serves as a record of particular means taken to arrive at the music.*

3. A SEMI-AUTOMATED ANALYSIS OF "EUCLIDEAN SYMPHONY" USING HOARE LOGIC

3.1 Ableton, Max4Live, and user-facing expressivity

Ableton is one of the industry standard Digital Audio Workstations (DAW's). As such, it is used by many producers in a variety of genres, as well as occasionally playing a role in the nice area of algorithmic composition. The full language's expressivity is greatly enhanced by the inclusion of several devices which were built in a more expressive language (Max, similar to PD described above), and with the inclusion of just 3 of these basic devices Ableton can be shown to be Turing complete. There is a library dedicated to enabling anyone to create a Max patch, and develop a user interface for it in Ableton such that the user doesn't need to know about the inner workings of the code.

There is no "typical user" of Ableton; however, a cursory glance at youtube's hip-hop tutorial pages shows that a huge niche is "device consumers" - those who don't have the programming skills to write their own devices, but will pay for devices which look useful. We can model these users as "shallow" programmers - programmers who can only write code with a finite, predetermined set of functions. Thus, we need a language for analyzing the usage of a finite set of devices with fixed but black-box semantics.

3.2 Non-determinism in Ableton and the need for provable properties

Many max4live devices are designed to rely heavily on randomness, as are certain built-in devices. As such, the output each time one renders a given file may be vastly different. Therefore, one cannot understand the space of ideas the composer was willing to entertain without finding out facts for which logic can prove that they hold in all cases, since even if, e.g., you listen to a piece 90 times and never hear a violin, there is no guarantee that the composer didn't include a violin sound somewhere in the piece.

3.3 Hoare Logic and its applications to Ableton pieces

We suggest that Hoare logic is a useful tool in this case. Hoare logic, as pioneered by Tony Hoare, is a type of logical reasoning which is not itself automated, but for which validity can be automatically verified.[1] In Hoare logic, we are given rules about the semantics of a function in terms of $PF(f)Q$. P, the precondition, states what we're assuming we know is true about the state of the program before we invoke function f. $F(f)$ describes a way of using f. Q states what we then know is true of the state of the program after this invokation. There can be multiple "Hoare triples" per function, reflecting the multiple propositions that can be true of a function, and the multiple conclusions we will arrive at based on starting initial assumptions.

Usually, Hoare triples are used in sequential imperative programs, such as the following:

```
Triples given:
x % 2 == 0 {x = f(x)} x == 0
y % 3 == 0 {y = g(y)} y == 1
z % 4 == 1 {z = h(z)} z % 2 == 1
```

```
Program:
def y():
    x = 4
    x = f(x) + g(x)
    x = z(x)
    return x

Program Annotated with triples;
{tautology} x = 0 {x % 2 == 0 /\ x % 3 == 0};
{x % 2 == 0 /\ x % 3 == 0} x = f(x) + g(x); {x == 1}
{x % 4 == 1} x = z(x) {x % 2 == 1}

 Conclusion:
 y() % 2 definitiely equals 1
```

In our analysis below, we will assume that we have been given a fixed set of Hoare triples for each device we care about. Our task is then to use those Hoare triples to determine provable facts about a piece which uses these devices. Note that it is common in partially-automated theorem proving to be given a description of the behavior of a black-box object, and to make conclusions based on these descriptions.

3.3.1 Defining Hoare logic in Ableton

In general, Hoare logic in Ableton will be based on Ableton's sequential device model, in which an arbitrary list of models which transform MIDI are followed by an instrument which turns MIDI into audio, followed by an arbitrary number of effects which turn audio into audio. Preconditions then, depending on the type of device (MIDI device, instrument, or audio device), have as preconditions facts about MIDI or audio, and have as postconditions how facts about MIDI or audio. For an example, consider the following devices: We have a MIDI effect "ComplexSequencer" which ignores its input and produces arbitrarily complicated MIDI about which we can say nothing, a "Scale" MIDI effect which states that regardless of input, the output will be in C Major, a "Note length" MIDI instrument which affects note length but preserves pitches, an instrument which preserves harmonic spectra, and a "normalizer" audio effect that sets the average volume within a single FFT frame to 1db as well as a "compressor" audio effect which preserves harmonic spectra but affects volume by dividing it in two.

```
Hoare Triples:
{True} {input = Scale(input)} {input in C Major}
{pitches(input) = x} {input = NoteLength(input)} ->
                             {pitches(input) = x}
{pitchesInScale(input, x)} {input = Instrument(input)
                             {spectraInScale(x)}
{spectra(input) = y} {input = Normalizer(input)} ->
                             {spectra(input) = y}
{spectra(input) = y} {input = Compressor(input)} ->
                             {spectra(input) = y}
```

Note the difference in strength of these different Hoare triples. In the first Hoare triple, we have a strong precondition and a tautology post-condition, telling us nothing about the output given any information about the input. In contrast, the second Hoare triple tells us that regardless of the MIDI given, we will always know that the output will be in C major - a much more informative piece of information.

To use these to prove that a piece has a harmonic spectrum consisting of notes in C major, we will simply chain those Hoare triples together as shown in Appendix A:

3.3.2 Analysis of "Euclidean Symphony" by Halley Young

"Euclidean Symphony" [23] is a piece which relies on orchestral-sounding samples, but has a distinctly electronic sound do to the heavy use of audio effects. In the version of it included on soundcloud, it includes a driving pulse which morphs into a more chaotic rhythm before "resolving" to the same pulsing beat. What we can learn about Euclidean Symphony by observing its source code and applying some Hoare logic:

1. This piece which relies on several devices allowing little audiational agency and some non-determinism (remove a random x% of notes, use complicated melody-generating device relying on randomness)

2. This piece provably moves across 3 scales throughout the piece *regardless of how many times it is re-generated*

3. This piece relies heavily on nonlinear processes by using a signal envelope to modulate note length on one track, which effects volume (and future input to signal envelope)

4. This piece relies heavily on continuous "gestures" (creating long curves of automation of devices). See Smith on the importance of continuous gestures in many types of electronic music.

4. HOW TO ANALYZE PIECES BUILT IN RENOISE - AN INTRODUCTION TO AUTOMATED STATIC ANALYSIS OF CODE + SOUNDS

4.0.1 How Renoise Works

Renoise is a somewhat unusual Digital Audio Workstation. It is organized into a pattern editor, sample editor, sample modulation sets, and track effect chains, which interact with each other. The pattern editor, while providing one command for non-determinacy, is immutable and states how samples used in the piece are supposed to be initially processed, as well as how each track is to be post-processed after the initial production of sound. The sample editor specifies default settings for processing of samples, including which modulation set should be applied to each sample. A modulation set specifies ways of modulating the pitch, panning, volume, resonance, and filter on a sample, and is composed of chains of devices multiplied or added together. Similarly, an effect chain consists of a sequence of common effects (compressors, reverbs, etc.) as well as more interesting meta-devices (such as a hydra which can take input from any signal and send it to any parameter in any other device). Each file is a compression of an XML file containing all data except the samples, and a folder containing the samples used.

4.0.2 Renoise and the field of automated static analysis

Automated static analysis is the term used to describe programs which are given a class of programs in a given language, and produce analyses with a given goal fact or set of facts to be determined. For instance, in Doop, a tool used to analyze Java, one of the goals is to determine if you ever have provably infinite loops, or whether a certain function is defined but provably never used. It is referred to as "static" analysis because it is accomplished through a careful analysis of the source code, and not by running the code and seeing what happens. Static analysis usually involves two steps: parsing the code for "surface-level" facts, and performing inductive analysis in a language such as Datalog [16] to learn more complex facts. For instance, parsing a java program may easily provide clauses such as

```
Java program contains:
public int func1()
    var1 = 3
    return var
end

Parser outputs:
definedInFunction(func1, var1, 3)
```

Then, the inductive rules find more general facts.

```
Datalog program inputs:

definedInFunction(func1, var1, 3)

constantValueInFunction(func, var) :-
definedInFunction(func, var, val),
not isPointer(val),
not (assignedInFunction(func, var2, val2),
val != val2)$.

Output:
constantValueInFunction(func1, var1).
```

4.1 Applying static analysis to Renoise

As in our example with Java, there are certain facts that can be parsed easily from a Renoise XML file - for instance

```
modulationAppliedToSample(sample1, chain3).
```

In addition, we can perform all the typical analyses on the audio samples that the renoise file uses to produce additional clauses: e.g.

```
hasLowSignalToNoiseRatio(sample1).
hasEstimatedPitch(sample2, 20).
isClassifiedByCNN(sample3, "string").
```

Finally, we can apply inductive logic to learn more interesting facts about the piece:

```
Datalog Program Fragment:

modulationAppliedToSample(sample2, chain3).
chainIncludesDevice(chain3, "randomStepper").
hasLowSignalToNoiseRatio(sample2).
sampleUsedInPattern(sample2, 3).
```

```
pieceRandomizesDrumSoundInPattern(pat) :-
modulationAppliedToSample(sample, chain),
chainIncludesDevice(chain, "randomStepper")
hasLowSignalToNoiseRatio(sample),
sampleUsedInPattern(sample, pat).

Output conclusion:
pieceRandomizesDrumSoundInPattern(3).
```

4.1.1 A case study - "LFO" by Danoise

There are a lot of fascinating conclusions that can be drawn by analyzing a piece titled "LFO" as a program. "LFO" is a piece which would probably be considered "EDM" in the quality of its sounds and in some of its rhythmic grooves, but has moments of irregular pure ambience. It pays homage to the Renoise software with occasional punctuations of the acronym "RNS" presented in a robotic voice. Here are just a few things we learned from looking at it as a set of materials (samples) that a program is applied to:

1. The variation in timbre in the strings is entirely random, not modulated (something you could never know for sure without seeing the project file)

2. In general, he relies on developing more complex instruments and exposing that complexity as opposed to making more complex track audio effects or using many different instruments

3. He tends to hard-code repetition of phrases, and organize sample material into 1-second chunks which are concatenated into these phrases

4. He tends towards using simple materials (sine waves, triangle waves, etc. - samples which can be described using ¡200 individual points of sample material) except when using speech

5. "DISTANT LISTENING" AND CORPUS ANALYSIS OF A DATASET OF RENOISE FILES

We can also do basic learning on a corpus of Renoise files. A total of 2796 renoise files were collected for this purpose. We considered asking a few questions as an entry into exploratory data analysis: For instance, we can look at the likelihood of different sub-programs to see which types of effects are typically evokes contiguously inside a track device-subprogram 5 5.

5.1 Can we develop methods to quantify and visualize the spectrum of non-determinism?

Methods of non-determinism in Renoise There are several ways of introducing non-determinism into renoise. For instancee, one can have non-determinism in the hard-coded commands attached to each individual note or line - you can always include a Y command to either make a line not be executed within a track or to choose exclusively between different notes or phrases. "1 Instrument - 32 Lines" by Bellows [2] includes Y commands nested within instrument phrases, leading to an incredibly complex piece from 32 lines (2 measures) and a single sample.

There also is the possibility of non-determinism in the instruments. The "Stepper" device has a setting which

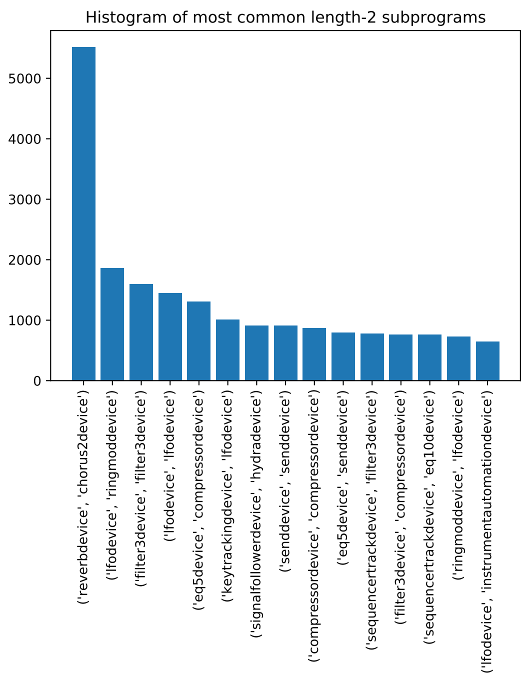


Figure 1. A histogram of the top 15 bigrams in renoise subprograms.

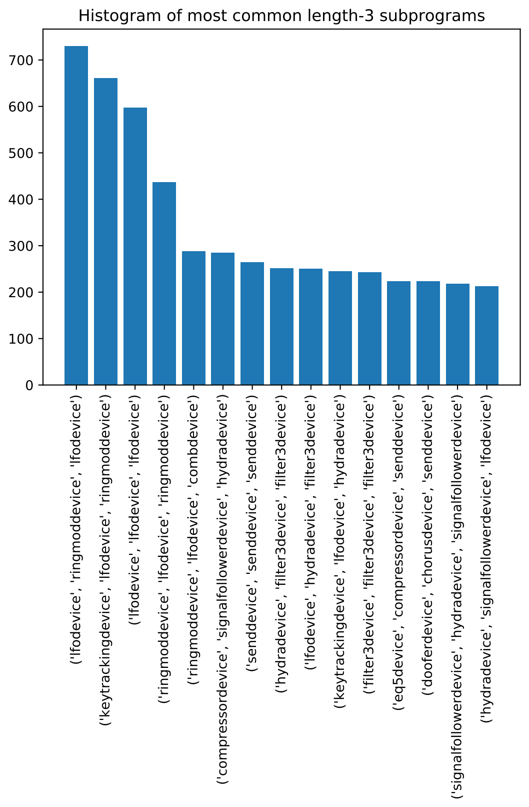


Figure 2. A histogram of the top 15 trigrams in renoise subprograms.

allows randomizing a choice of discrete steps, while the "LFO" effect actually can produce a continuous random signal and multiply/add it to any parameter's value. The Youtube composer "Zensphere" has a number of videos detailing how he uses these forms of non-determinism in instruments to produce "psy-trance" pieces.

Finally and most commonly, there can be non-determinism in the track effects. The LFO device has a random starting point, unless specifically reset. While this might not seem like enough to create very different renditions each time, if the LFO maps to a sub-device-chain (known as a "doofer" which modulates which phrase of an instrument is played), or is used in a device-chain with heavy feedback mechanisms, it can lead to complex differences between instances of the "same" piece. We can to some degree quantify and compare levels of nondeterminism as in 3 4. **C**an we develop methods to quantify and visualize the spectrum of economy of materials?

1. Define Aural Dimension Count of a single signal as the number of variables related to the processing of the signal which have at least two values over the course of the piece, the Aural Dimension Count of the entire piece as the sum over all samples, and Economy of Materials as $\frac{ADC(x)}{|S(x)+c*I(x)^2|}$, where ADC(x) is the total Aural Dimension Count in file x, S(x) is the number of samples in file x, I(x) is the number of unique instruments in x, and c is some coefficient balancing the two measures of economy of material. Both elements in the denominator are necessary because while more samples necessarily implies using more materials, in many cases there are hundreds of samples which are sampled from a single real-world instrument to provide maximum realism in evoking that instrument.

2. Aural Dimension Count can be calculated by looking at effect automation and instrument design

3. Algorithm to find Aural Dimension Count:

```
1) Loop through every modulation
    set chain.
If an item in the chain is either
    a) assigned to a Macro or
    b) preceded by an LFO or stepper,
    increment the dimension
    count of the sample by 1.
2) Loop through each track device.
    Check for meta-devices.
    For each input-output pair
    in the meta-device,
    increment the dimension
    count of the sample by 1.
3) Look at the automation
    of each track device
    in every track where the
    instrument appears.
    If it varies at any point
    in the piece,
    increment the dimension
    count of the sample by 1.
```

6. ANALOGOUS PROBLEMS IN OTHER FIELDS

There are generative and analytic problems in other domains which would benefit from a focus on the interaction of the product and the source code. For instance, videogame analysis has been performed by trying to build a model of how pixels continuously transform in response to user input. Similarly, some deep learning researchers have even proposed building videogames on the basis of deep models which learn to generate the right pixels in response to a user's input. However, generating video-games as a state-action-state pair of pixels-keystrokes-pixels is fundamentally different than generating valid programs in Unity (a game design engine), and we argue that an approach based on program analysis is a lot more informative and tractable. In 3d shape generation, Blender (a 3d design program) can either be given a huge number of points to specify a surface, or simply give a program which enumerates the parts as platonic-like objects (spheres, splines, etc.) and their parameters, as well as actions performed on those parts (extrusion, intersection, etc). We argue that it is probably more tractable to build a model which learns the space of highly structured programs in Blender than to learn the space of all 3d objects in terms of point-sets. However, it is worth noting that in these areas, one could claim that there is less of an aesthetic imperative to "understand" the output as there is in an inherently aesthetic field such as music. Nevertheless, even in terms of practical tractability, it is probably worthwhile to focus on improving program analysis in Blender or in Unity.

7. CONCLUSION

In this paper, we introduce the notion of "source-code informed listening", or the usage of the source code of a given piece of 'coded" music to better understand and analyze the audio file that is the aural artifact of that music. We start with a hermeneutic (close and very deliberately manual) reading of a specific coded piece, discussing what added value the source code brings. We then, drawing from the literature on automated program analysis, define two more automated forms of analysis, of which the second enables corpus study and "distant listening" (an analogy to the practice in digital literature studies of "distant reading.") We conclude by examining further areas for exploration, including interdisciplinary future work.

8. REFERENCES

References

[1] Krzysztof R Apt. "Ten years of Hoare's logic: A survey—Part I". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 3.4 (1981), pp. 431–483.

[2] Bellows. *1 Instrument, 32 lines.* `https://forum.renoise.com/t/3-0-xrns-1-instrument-32-lines/41089`. 2014.

[3] Matthew F Chung. *Mixing: Composition Theory and Chaos in an Autonomous Music-Making System.* University of California, San Diego, 2021.

[4] Zijing Gao et al. "A novel music emotion recognition model for scratch-generated music". In: *2020 International Wireless Communications and Mobile Computing (IWCMC)*. IEEE. 2020, pp. 1794–1799.

[5] Josh Gardner et al. "Mt3: Multi-task multitrack music transcription". In: *arXiv preprint arXiv:2111.03017* (2021).

[6] Deepanway Ghosal and Maheshkumar H Kolekar. "Music Genre Recognition Using Deep Neural Networks and Transfer Learning." In: *Interspeech*. 2018, pp. 2087–2091.

[7] Perfecto Herrera, Alexandre Yeterian, and Fabien Gouyon. "Automatic classification of drum sounds: a comparison of feature selection methods and classification techniques". In: *Music and Artificial Intelligence: Second International Conference, ICMAI 2002 Edinburgh, Scotland, UK, September 12–14, 2002 Proceedings*. Springer. 2002, pp. 69–80.

[8] Pascal Hitzler et al. "Neuro-symbolic approaches in artificial intelligence". In: *National Science Review* 9.6 (2022), nwac035.

[9] Mark Kahrs and Karlheinz Brandenburg. *Applications of digital signal processing to audio and acoustics*. Springer Science & Business Media, 1998.

[10] Jongpil Lee and Juhan Nam. "Multi-level and multi-scale feature aggregation using pretrained convolutional neural networks for music auto-tagging". In: *IEEE signal processing letters* 24.8 (2017), pp. 1208–1212.

[11] Jongpil Lee et al. "Sample-level deep convolutional neural networks for music auto-tagging using raw waveforms". In: *arXiv preprint arXiv:1703.01789* (2017).

[12] Yuanqing Li et al. "Underdetermined blind source separation based on sparse representation". In: *IEEE Transactions on signal processing* 54.2 (2006), pp. 423–437.

[13] Michael I Mandel and Daniel PW Ellis. "Song-level features and support vector machines for music classification". In: (2005).

[14] Koichi Miyazaki et al. "Weakly-supervised sound event detection with self-attention". In: *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2020, pp. 66–70.

[15] Eric David Scheirer. "Music-listening systems". PhD thesis. Massachusetts Institute of Technology, 2000.

[16] Bernhard Scholz et al. "On fast large-scale program analysis in datalog". In: *Proceedings of the 25th International Conference on Compiler Construction*. 2016, pp. 196–206.

[17] Mohamadreza Sheikh Fathollahi and Farbod Razzazi. "Music similarity measurement and recommendation system using convolutional neural networks". In: *International Journal of Multimedia Information Retrieval* 10 (2021), pp. 43–53.

[18] Arun Solanki and Sachin Pandey. "Music instrument recognition using deep convolutional neural networks". In: *International Journal of Information Technology* 14.3 (2022), pp. 1659–1668.

[19] Bo Sun et al. "Audio-video based multimodal emotion recognition using SVMs and deep learning". In: *Pattern Recognition: 7th Chinese Conference, CCPR 2016, Chengdu, China, November 5-7, 2016, Proceedings, Part II 7*. Springer. 2016, pp. 621–631.

[20] Yu Wu, Hua Mao, and Zhang Yi. "Audio classification using attention-augmented convolutional neural network". In: *Knowledge-Based Systems* 161 (2018), pp. 90–100.

[21] Rui Yang et al. "Parallel Recurrent Convolutional Neural Networks Based Music Genre Classification Method for Mobile Devices". In: *IEEE Access* PP (Jan. 2020), pp. 1–1. DOI: `10.1109/ACCESS.2020.2968170`.

[22] Halley Young. "Algorithm as Determinant of Form in Music." In: *ICCC*. 2019, pp. 350–351.

[23] Halley Young. *Euclidean Symphony*. `https://soundcloud.com/user-332259139/euclidean-symphony`. 2023.

A. HOARE LOGIC PROGRAM FOR ANALYZING "EUCLIDEAN SYMPHONY"

```
Remember, there is a hoare triple
{True} {input = Scale(input)} {input in C Majo
for all input, and no other functions
affect tonality, so the following is valid:

{input = silence}
    {input = complex_sequencer(input)}
                {True}
{True}
    {input = Scale(input)} ->
                {input in C Major}
{pitches(input) in C major}
        {input = NoteLength(input)} ->
                {pitches(input) in C major}
{pitchesInScale(input, C major)}
        {input = Instrument(input)} ->
                {spectraInScale(C major)}
{spectraInScale(C major)}
        {input = Normalizer(input)} ->
                {spectraInScale(C major)}
{spectraInScale(C major)}
        {input = Compressor(input)} ->
                {spectraInScale(C major)}
```

B. FIGURES QUANTIFYING NON-DETERMINISMM IN A RENOISE PROGRAM

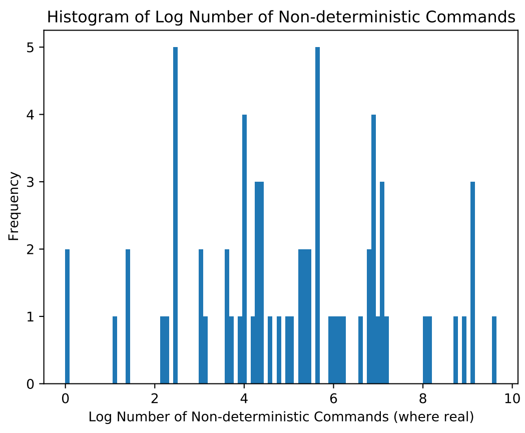


Figure 3. Histogram of number of non-deterministic effect commands across corpus

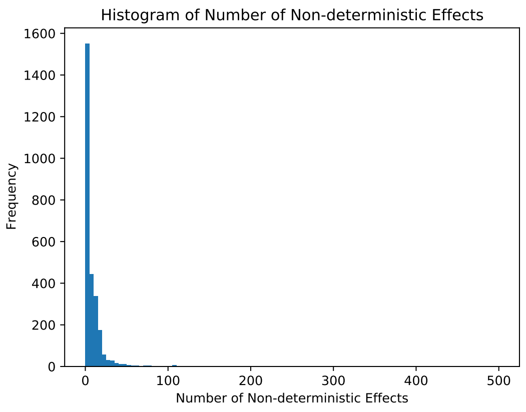
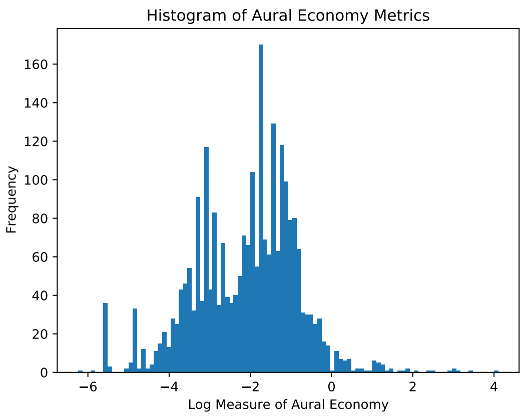


Figure 4. Histogram of number of non-deterministic effect chain across corpus

C. AURAL ECONOMY METRICS


Histogram of Aural Economy Metrics
Frequency
Log Measure of Aural Economy