Cite as: Eamonn Bell. “Forms as algorithms: The case of the ‘normal form’”, Sound and Sonorities: Form and F

Forms as algorithms: The case of the “normal form”

Introduction

In this short paper, I discuss how the musical set-theory concept of the “normal form” of a pitch-class set is presented *as* and, sometimes, conflated *with* an algorithm. Algorithm designers in all fields, including music studies, seek dependable processes that allow us to cope with the messiness of the world, refracted through a model that expresses salient features of that field’s reality as data. The algorithmic framing of the normal form is tool that makes certain dissimilar contigencies—the various forms of musical “surfaces”—appear similar, normalizing them in a way expressly designed to reveal their essential characteristics. In the case of pitch-class sets, this approximates the interval content of the set.

On another level, as an apparently popular way of explaining a desired learning outcome in the discipline of music theory—that is, being able to determine the normal form of any (two) pitch-class set(s)—it attempts to smooth over differences between individual learners by offering them a repeatable, deterministic, and general strategy for coping with messy data: the manifestly diverse “presentations” of “pitch-class sets”, alias “post-tonal music”. This regulative function in the specific context of twentieth-century music theory can be understood as one of many consequences of the influence of what I call a computational attitude in musical thought, especially in music theory, which I assert was felt strongly during the 1960s, the period of the widespread adoption of commercially produced digital computers.

To achieve this, I attempt a first approximation at a working definition for what an algorithm actually is, and explore how algorithms relate to more widely-applicable category of computationalism in twentieth-century scientific thought. To do this, I refer to a number of very recent writings from the emerging discipline of what is sometimes called “critical algorithm studies.” Next, discuss formal resemblances between descriptions of the notion of “normal form” as they appear in modern post-tonal theory textbooks and computer code. Granting this analogy, I then suggest that the computational aspect that these descriptions have taken on is not merely a result of the familiar whiff of the logical-positivist program that typifies twentieth-century music theory, but is the result of a deliberate choice to describe the concept using syntactic and lexical conventions that approximate the quasi-linguistic constraints that are typical of actual computer code. I make use of the term "pseudocode" to refer to descriptions of algorithmic processes that use a controlled vocabulary and stripped down-syntax to present their basic structure so that they may be readily converted to computer code, without being tied to any one specific programming language or dialect. By making use of the rhetoric of pseudocode, these authors "algorithmize" the notion of normal form. Only by learning to recognize such algorithmic expressions of musical concepts (be they formal processes, techniques of composition, techniques of analysis, or more) can we begin to decide whether we can live with or without them as musicians and teachers of music.

Algorithms everywhere?

It is now a trope of the genre to motivate discussions of algorithms with assertions of their ubiquity:

> [D]ozens of key sets of algorithms are shaping everyday practices and tasks, including those that perform search, secure encrypted exchange, recommendation, pattern recognition, data compression, auto-correction, routing, predicting, profiling, simulation and optimisation.[1]
>
> Algorithms are everywhere. They already dominate the stock market, compose music, drive cars, write news articles, and author long mathematical proofs—and their powers of creative authorship are just beginning to take shape.[2]

If these select quotations can be said to reflect some discursive agreement that algorithms are everywhere, there is less consensus as to what exactly they might be. As Tarleton Gillespie puts it in the useful collected volume *Digital Cultures*, "we find ourselves more ready to proclaim the impact of algorithms than to say what they are."[3]

1. Kitchin (2017), p. 15. Kitchin's laundry list is inspired by the nine algorithms described in MacCormick (2013). MacCormick's account is a more popular survey.
2. Finn (2017). p. 15.
3. Gillespie (2016), p. 189.

For instance, in a 2013 report for the Tow Center for Digital Journalism at Columbia University, Nicholas Diakopoulos defines an algorithm "as a series of steps undertaken in order to solve a particular problem or accomplish a defined outcome."[4] Perhaps such a definition is surprising. For one, it does not refer to any of the specific socio-technical systems—specifically the computational infrastructure required to make use of algorithms at the scale of demanded by a global financial and cultural economy—whose existence are implied by the contemporary abundance of examples of algorithms in the wild. Notably, its passive voice is indeterminate as to who (or what) is taking the steps; it is mute as to whether algorithms are the domain of human, animal, or machinic activity.

Italicized text on the slide indicates that Diakopulos's definition is in fact a paraphrase of a more detailed definition appearing in the Unabridged Merriam-Webster:

> a procedure for solving a mathematical problem (as of finding the greatest common divisor) in a finite number of *steps* that frequently involves repetition of an operation; broadly: a step-by-step procedure for *solving a problem or accomplishing some end* especially by a computer.[5]

Diakopulous suppresses the more technical aspects of this dictionary definition, and with good reason: from his perspective, the problems that algorithms solve today are economic, social, and cultural, as much as they are mathematical or computational. However, a fuller definition, thanks to Merriam-Webster, moves beyond a definition of "algorithm" almost exclusively teleological (that is, something used "to solve a particular problem or accomplish a defined objective") to a definition which suggests that algorithms have something to do with mathematics and computers, and, relatedly, qualifies algorithmic procedures as both finite and frequently repetitious.

In his 2017 book, *What Algorithms Want*, Ed Finn distinguishes between two kinds of definitions of algorithm: that of the pragmatist, and that of the computationalist. As a particularly acute example of a pragmatic definition, Finn cites the sparer-still definition of the computer scientist Robert Sedgewick, Finn's own computer science teacher and author of an extremely influential introduction

4. Diakopoulos (2013), p. 3
5. Webster's Third New International Dictionary, Unabridged, s.v. "algorithm," accessed April 21, 2018, http://unabridged.merriam-webster.com.

to algorithms for undergraduates. To Sedgewick, an algorithm is simply a “method for solving a problem”. Diakopolous, Sedgewick, and others typify pragmatist definitions of “algorithm” since, as definitions, they describe their referent merely in terms of its “utility”: algorithms are thing that succeed in doing other things; they illuminate “pathways between problems and solutions”, but how they do so is not of any import or interest to the user.[6] All that matters is that they solve the problem their designer sets out to solve.

On this view, good algorithms are those that “just work”, while bad algorithms are those that fail. On the other hand, computationalist definitions task “algorithm” with ontological work, asserting that algorithms hold the promise of explanatory value not by providing solutions to the problems they are designed to solve, but also with respect to *how* those problems are (or must be) solved in reality, outside the domain of the algorithm. From this perspectives, “algorithms do not merely describe cultural processes with more or less accuracy: those proceses are themselves [held to be] computational machines that can be mathematically duplicated.”[7] The computationalist understanding fuels optimism that the decades-old promissory note of general artificial intelligence will be redeemed when a sufficiently faithful algorithmic model of cognition can be described and implemented precisely because the “cultural process” of cognition is not just reducible to (per the pragmatist view), but *is in fact* “a series of steps undertaken in order to solve a particular problem”.

Neither of these candidate definitions of “algorithm” are without their problems. The breadth of the utilitarian definition of the pragmatist means that almost any procedural solution to a given problem could be understood, packaged, and sold as “algorithmic”, questionably lending it a set of virtuous associations. As Gillespie points out, algorithms are thought—incorrectly or otherwise—to be “mathematical, logical, impartial, [and] consistent.”[8] On Contrastatively, the ontological overreach of the computationalist definition, despite its popularity, risks confusion on the part of practicioners of algorithms between the useful fictions of algorithmic reasoning and the empirical reality of cognitive processes. There is no *a priori* reason to be believe that human behavior in the main is well-explained

6. Finn (2017), p. 18.
7. Finn (2017), p. 22.
8. Gillespie (2016), p. 23

by mechanistic analogies with a twentieth-century notion of computation. Thus, both pragmatic and computationalist definitions have their potential pitfalls. Understanding which, if any, of these definitions are operable in writing about musical algorithms, is the first step toward avoiding these slips.

I am not putting the various senses supported by the word "algorithm" in evidence so that we can resolve some blurry notion into a coherent and exhaustive set of mutually interdependent meanings. Rather, I want to remind you that despite its apparent stability as a term of art for computer scientists, originating in the everyday mathematics of the *bāzār*, the stakes for understanding in what sense something is claimed to be algorithmic are not just limited to the concerns of technologists, but extend to those of us interested in cognition, and in particular, the cognition of cultural products: music, naturally, included.

As musicologists and music theorists, we ought to take particular heed of the shades of meaning of "algorithm" in our discussions about the future of music in the (second? third?) age of artificial/algorithmic intelligence. If this brief discussion hasn't succeeded in raising algorithmic consciousness, it should at least precipitate a respect for "algorithmic conscientiousness": a careful attention to the specific conceptions of "algorithm" as they manifest themselves in writings outside of the domain of technology proper, and, as we will see, in writings about music theory.

Algorithms closer to home

The question to be answered, then is this: if algorithms—whatever they are—are really everywhere, are they perhaps to be found closer to home: that is, in the discipline of music theory? If they are, it seems like a perfect opportunity to exercise our algorithmic conscientiousness and tease out the meaning of these algorithms, how they came to our attention in the first place, and what we ought to do about them,

To that end, a quick review of the concept of the "normal form". A core objective of most introductory musical set-theory courses is to ensure that the student knows how to compute the normal form (and, relatedly, the prime form) of a pitch-class set. Loosely, the normal form of a such a set is the ordering of that set (with duplicates removed) that is most densely packed. (The prime form of a pitch-class set is the the most normal (in the sense just defined) of: (a) its normal form transposed such that it begins with pc0; (b) its normal form, inverted, and transposed such that it begins with pc0). In a passage that has remained relatively unchanged since the appearance of the first edition almost thirty years ago, Joseph N. Straus motivates the introduction of the concept in his *Introduction to Post-Tonal Theory*, in the following way:

> A pitch-class set can be presented musically in a variety of ways. Conversely, many different musical figures can represent the same pitch-class set. If we want to be able to recognize a pitch-class set no matter how it is presented in the music, it will be helpful to put it into a simple, compact, easily grasped form called the *normal form*. The normal form—the most compressed way of writing a pitch-class set—makes it easy to see the essential attributes of a set and to compare it to other sets.[9]

According to Straus, there are two reasons to be interested in the normal form of a pitch-class set, then. First, to understand the "essential attributes" of a set; second, to compare it to other sets. John Rahn echoes the second of these reasons in his earlier *Basic Atonal Theory* (1980), writing: "in order to be able to compare sets easily, it has been found necessary to choose one particular standard order to list them in."[10] It is typical to describe the process of arriving at the normal form—this "particular standard order"—as a sequence of imperative directions to the student. Straus again:

> 1. Excluding doublings, **write** the pitch classes as though they were a scale[.] [...]
> 2. **Choose** the ordering that has the smallest interval from first to last[.] [...]
> 3. **If** there is a tie [...] **choose** the ordering that is most clustered away from the top. [...] **If** there is still a tie, **compare** the intervals between the first and second-to-last notes[,] [...] and so on.
> 4. **If** [this process] still results in a tie, then **choose** the ordering beginning with the pitch class represented by the smallest integer.[11]

In this description, an ordered sequence of imperatives directs the student to manipulate a musical representation of the pitch-class set, notation, without direct reference to an overall goal. Each

9. Straus (2005), p. 35
10. Rahn (1980), p. 31.
11. Straus (2005), p. 36

instruction stands alone as a direction to fiddle with the pitch-class set on the level of its consituent parts: **write** its pitch class representatives as a scale, **compare** this interval with that, **choose** an ordering based on the tie-breaking pitch-class magnitude. The desired global property—dense packing or "normality"—is the result of the faithful application of local manipulations. Additionally, this description asks the student to intermittently stop the process to evaluate whether certain facts obtain: "**if** there is a tie", "**if** there is still a tie". If the conditional is true, then the student performs further steps; if not, they do not. Finally, we note that this procedure is implied by the author to guarantee the correct outcome when applied to any pitch-class set. That is, it is insensitive to the particular pitch-class set under consideration: it is asserted that all pitch-class sets we might encounter will yield up their normal form if these steps are followed; there are no edge cases.

I suggest that each of these three features make this description of the process of putting something in normal form loosely resemble a computer program. First that the normal form procedure is here made to consist in an ordered sequence of imperative statements. In computer code, computational directives are specified to the computer as a sequence of statements, according to some conventional ordering. The computer then applies this conventional ordering in the execution of the code, proceeding from statement to statement in accordance with that convention. As with instructions for humans (such as recipes for cooking or this normal form procedure), the most common ordering of statements in a computer program is the literal order in which it will subsequently be executed, but this need not be the case.

Second, Straus' description of the process makes use of "if…then" constructions, which interrupt the regular flow of the procedure when certain conditions hold. If there's one thing that digital computers are good at, it's acting on distinctions between between true and false, since the overwhelming majority of digital computers make use of the binary system to internally represent the state of a computer program as a sequence of zeros and ones. The use of "if…then" constructions in programming languages are commonplace ways of providing optional and alternative paths through computer code, allowing the conventional order of execution (alluded to above) to be temporarily suspended so that special cases or conditions for the intended termination of a program can be handled predictably. This feature of computer code (and, by analogy, the algorithmic description of the “normal form” concept) is called “conditional branching”.

Finally, because of the guarantee—implied but not proven in Straus’ description—that the procedure will generate an accurate normal form for all pitch-class sets, the procedure can be written down once in a sufficiently general way and reused, no matter what specific pitch-class set is to hand. In actual computer code, this procedure would be modeled as a subroutine. That is, a modular and reusable chunk of code that can be specified once and reused *ad lib* throughout a program, or over the useful lifespan of a software package. To write a subroutine so that it can be reused, statements inside the subroutine are constructed so that they do not refer to internal representations of a specific object, but to a placeholder, which, like a Cloze test or MadLibs, can be populated with arbitrary (but syntactically valid) objects. In the case of the normal form procedure, Straus’s description doesn’t make reference to a specific pitch-class set, but to a some variable pitch-class set, considered in the abstract. This ensures that the procedure is understood by the reader to be useful with all pitch-class sets, not just, some finite subset of all possible pitch-class sets: it is a general procedure.

Robert Morris’s description of the procedure for the normal form (published in 1991 in his *Class notes for atonal music theory*) also yields to a similar reading, even more proudly displaying its computational colors in pared-down language evidencing yet other tropes characteristic of actual computer code: variable assignments (“Let D = #D -1”), cryptic variable names (“call this E”), pruning and deletion, all along with along with control flow statements, such as “if…then” conditional branching, and “go to” statements that reference specific individual steps of the procedure. In this instance, Morris is apparently under no compunction to represent this process as anything other than an algorithm, naming it so in his text.

Rahn/Morris Algorithm: Normal-Form Representative

Definition: *Span* (sub-K) of an ordered set X:

$$S_K(X) = X_K - X_0 \text{ where } K \leq \#X - 1$$

1. D is a member of some SC Z. Write pcset D as an ordered set in ascending numerical order. Call this E.

```
2. Let $K = \#D - 1$.
3. Construct the set $E_r$ consisting of all rotations of $E$ and $RT_0IE$. If $E_r$ contains duplicate members, prune all duplicates.
4. Find $S_K$ for each member of $E_r$.
5. Find the smallest value of $S_K$ from members of $E_r$. Call it $m$.
6. Delete all members of $E_r$ with $S_k$ greater than $m$.
7. If $E_r$ has only one member, go to step 11.
8. $K = K - 1$.
9. If $K = 0$, find the member of $E_r$ with the lowest first pc, delete all other members of $E_r$, and go to step 11.
10. Go to step 4.
11. Transpose the remaining member of $E_r$ to begin pc $0$. The result is the normal-form representative of SC $Z$ and can be found on both the Rahn and Morris set-class tables.[12]
```

Why might these two authors structure their descriptions in accordance with these strained linguistic conventions? What is this distinctive rhetorical practice, which sits somewhere between recipe book, ordered checklist, and actual computer code?

I recognize it as pseudocode, "a notation resembling a simplified programming language, used in program design". The use of pseudocode is a commonplace of contemporary computer science literature, since it provides a mechanism for expressing computational procedures in a way that is agnostic of the reader's fluency in any one programming language, making it a useful template for comprehension and, importantly, implementation by as many readers as possible. Implementation is the process, usually undertaken by a computer programmer, that bridges the gap between algorithm and computer by taking a rigid specification of the algorithm, such as that (hopefully) provided by a clear and accurate pseudocode description of the algorithm, and transmediating it, line by line, in to the statements of a specific computer programming language, for a specific system, in a specific context of production. Implementation is the process by which the algorthmic idea is reified as an actually running computer program.

Pseudocode texts, then, are instructions for humans. I make the suggestion here that in these explanations of the "normal form", as they appear in music theory textbooks, the rhetoric of pseudocode functions on two registers at once: it serves both as a template for the student interested

12. Morris (1991), 1991

in computational implementation of the concepts in described therein, but also as a portable and intersubjectively robust set of procedures that can be used by hand, even without expertise in a specific programming language. In so doing, pseudocode descriptions of the normal form procedure “algorithmize” the concept of the normal form—a concept that could be, and **was**, defined otherwise—in a way that views students as both potential programmers and as interlocutors to be programmed.

If you believe that Straus’s and Morris’s description are somehow like computer code, and if we recall the pedagogical functions of their texts, then it is both provocative and productive to think—even fleetingly—about the idea that the student themselves is the thing that is being programmed here, being programmed with a set of routines instrumental to the educational objectives of these textbooks. Whether school students should be equipped with certain basic algorithms—let’s say learning the general principles of computing long division by hand—became a flashpoint in debates among educationalists in the US, as the virtue of rote learning in mathematics education fell under suspicion during the late 1980s.

Similarly, arguments for and against the inclusion of imperative, algorithmic specifications of musical processes in the music theory curriculum balance the need for drilling the basics of the mathematics that appear to be useful for the description of certain kinds of music with a desire for other ways to encourage investigation and play with musical materials. Such ways are, perhaps, less structured: their guarantees of intersubjectivity are less robust; their claims to generality more tenuous; their repeatability as tools for thinking about music more suspect. In closing, I tentatively suggest that this is not always a bad thing. I fully intend to be equivocal when I say that, in short, becoming sensitizied to the notion that there are “algorithms” of music theory is the first step toward design musical pedagogies and musical practices that are–for better or worse–less algorithmic.

Works cited

Diakopoulos, Nicholas. 2013. “Algorithmic Accountability Reporting: On the Investigation of Black Boxes.” A Tow/Knight Brief. Tow Center for Digital Journalism.

Finn, Ed. 2017. *What Algorithms Want: Imagination in the Age of Computing*. Cambridge, MA: MIT Press.

Gillespie, Tarleton. 2016. “Algorithm.” In *Digital Keywords: A Vocabulary of Information Society and Culture*, edited by Benjamin Peters, Princeton University Press, 18–30. Princeton, N.J.; Oxford, UK.

Kitchin, Rob. 2017. “Thinking Critically About and Researching Algorithms.” *Information, Communication & Society* 20 (1):14–29. https://doi.org/10.1080/1369118X.2016.1154087.

MacCormick, John. 2013. *Nine Algorithms That Changed the Future: The Ingenious Ideas That Drive Today’s Computers*. Princeton, N.J.: Princeton University Press.

Morris, Robert. 1991. *Class Notes for Atonal Music Theory*. Hanover, NH: Frog Peak Music.

Rahn, John. 1980. *Basic Atonal Theory*. New York: Longman.

Straus, Joseph Nathan. 2005. *Introduction to Post-Tonal Theory*. Upper Saddle River, N.J.: Prentice Hall.