A PATTERN ORIENTED DATA STRUCTURE FOR INTERACTIVE
COMPUTER MUSIC

Adam Lockhart, B.A.

Thesis Prepared for the Degree of
MASTER OF ARTS

UNIVERSITY OF NORTH TEXAS
May 2011

APPROVED:
Andrew May, Major Professor
Kathleen M. Swigger, Minor Professor
Jon C. Nelson, Committee Member
Joseph Klein, Chair of the Division of
Composition Studies
Lynn Eustis, Director of Graduate Studies
James C. Scott, Dean of the College of Music
James D. Meernik, Acting Dean of the
Toulouse Graduate School

This essay describes a pattern oriented data structure, or PODS, as a system for storing computer music data. It organizes input by sequences or patterns that recur, while extensively interlinking the data. The interlinking process emulates cognitive models, while the pattern processing draws specifically from music cognition. The project aims at creating open source external objects for the Max/MSP software environment. The computer code for this project is in the C and Objective-C computer programming languages.
Copyright 2011

by

Adam Lockhart
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. INTRODUCTION AND BACKGROUND</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Music Cognition</td>
<td>1</td>
</tr>
<tr>
<td>1.3 Interactive Computer Music</td>
<td>1</td>
</tr>
<tr>
<td>1.4 Machine Listening</td>
<td>1</td>
</tr>
<tr>
<td>1.5 Agents</td>
<td>1</td>
</tr>
<tr>
<td>1.6 Heuristics</td>
<td>1</td>
</tr>
<tr>
<td>1.7 Markov Models</td>
<td>1</td>
</tr>
<tr>
<td>1.8 Timbre</td>
<td>1</td>
</tr>
<tr>
<td>1.9 Examples of the Genetic Algorithm</td>
<td>1</td>
</tr>
<tr>
<td>2. SOFTWARE BACKGROUNDS</td>
<td>11</td>
</tr>
<tr>
<td>2.1 Max/MSP</td>
<td>11</td>
</tr>
<tr>
<td>2.2 Objective-C</td>
<td>11</td>
</tr>
<tr>
<td>2.3 Searching NSMutableArray</td>
<td>11</td>
</tr>
<tr>
<td>3. INTRODUCTION TO PODS</td>
<td>16</td>
</tr>
<tr>
<td>3.1 PODS: A Pattern Oriented Data Structure</td>
<td>16</td>
</tr>
<tr>
<td>3.2 Suggestions for Implementation</td>
<td>16</td>
</tr>
<tr>
<td>3.3 A Simple Implementation Example</td>
<td>16</td>
</tr>
<tr>
<td>3.4 Walking Through the Example</td>
<td>16</td>
</tr>
<tr>
<td>3.5 Another Implementation Approach</td>
<td>16</td>
</tr>
<tr>
<td>3.6 Time Buffer Example Walkthrough</td>
<td>16</td>
</tr>
<tr>
<td>4. PODS INTERLINKED DATA</td>
<td>32</td>
</tr>
<tr>
<td>4.1 Storing Related Data</td>
<td>32</td>
</tr>
<tr>
<td>4.2 Detailed Overview of Data Storage</td>
<td>32</td>
</tr>
<tr>
<td>4.3 Starting Pitch Storage Example</td>
<td>32</td>
</tr>
<tr>
<td>4.4 Conclusion</td>
<td>32</td>
</tr>
<tr>
<td>APPENDIX: ALGORITHM</td>
<td>39</td>
</tr>
</tbody>
</table>
REFERENCE LIST
CHAPTER 1

INTRODUCTION AND BACKGROUND

Introduction

Music is an experience of time as articulated by the perception and cognition of sound. The many dimensions of the musical experience can be represented as data for analysis and composition. This essay describes software tools for interactive computer music systems utilizing a pattern oriented data structure. This data structure, PODS, is designed to function similarly to cognitive models, particularly concerning music cognition. PODS is an extension to the Max/MSP software, the most widely used development environment for real-time computer music. PODS addresses the need for a more robust means of storing and parsing real-time data in systems for music performance, composition, improvisation, and analysis. The project is currently under development; at a later stage, it will provide open source opportunities for other researchers to use and to develop customized implementations that suit individualized needs and goals.¹

The impetus for this project came from an artistic goal for a particular piece of music. The idea was for a human and computer duet on Disklavier (a Yamaha piano fitted with sensors for capturing performances as MIDI data, and robotics for reproducing MIDI data on the piano), where the computer would "listen" to somewhat improvisatory

input and generates improvisatory responses evolving from the input. The first attempts were built within Max. This includes a linked list implementation for coll (collection object), and then an adaptation for making trees. There are major limitations to this. Foremost, the Max list type is limited to containing 256 items. Also it is a convoluted process, using Max objects to handle a task that should be addressed on a lower level of coding language. Using Max for this adds unnecessary storage and processing costs. The answer to this problem was to create a Max object that would handle data processing independently.

Music Cognition

From a cognitive standpoint, one plausible definition of music involves recognition of patterns in sound over time and the ability to imagine those patterns modified in various ways. Whether listening, analyzing or creating music, the intellectual activity described happens as part of a larger experience, bundled with everything perceived from one’s environment, and related to a network of meaning created by one’s entire perceptual history. Eduardo Reck Miranda goes as far as to relate emotional response to music with this idea of intellectual activity. For the purpose of modeling and manipulating music on the computer, however, the first step is to represent these patterns as data in a way that is plausibly connected to human cognition of music. While the ideas of pattern identification, mutation, prediction, etc. are certainly

---


4 Miranda, “Regarding Music, Machines, Intelligence, and the Brain,” 1.
intellectual activities involved in music, it is more likely that emotion in music is derived from and communicated by means of preconceived meaning.

Our cognitive (and even emotional) response to music relies on our expectations. David Huron describes the psychological mechanisms involved in musical expectation – imagination, tension, prediction, reaction, and appraisal – all of which affect the listener’s experience.\(^5\) Reward systems are critical to both motivation and learning. Whether or not our expectations are met gives either negative or positive reinforcement and encourages the accurate formation of future expectations.\(^6\) Huron proposes that these activities set up a reward system in our brains that can account for much of our enjoyment of music.

Interactive Computer Music

Interactive computer music can be closely related to chamber ensemble music. There are seldom more than a few performers, and if there is more than one human performer, they often rely on listening and watching each other to keep up. Robert Rowe proposes, “Interactive computer music systems are those whose behavior changes in response to musical input. Such responsiveness allows these systems to participate in live performances, of both notated and improvised music.”\(^7\) Because of its interactive role, the computer may be thought of as a member of the ensemble rather than an instrument. In a similar way that the performer affects the computer’s behavior, the

\(^6\) Ibid., 7-15.
reaction of the computer influences the performer in various ways; the performer(s) and
computer are therefore joined in an interaction.

In an interactive computer music performance, an audience member can see and
hear a performer play a note and hear the computer respond (or not respond) to it. This
adds a further dimension to Huron’s expectation model on both a musical and an
extramusical level. The listener doesn’t know exactly what to expect, but is attentive in
anticipation, waiting to make connections about how the interaction works.

Machine Listening

The centuries-old discipline of music theory has proposed many models of
organization of music. More recently the field of music cognition has proposed a variety
of alternate models. Music cognition’s focus on process is intrinsically different from
music theory’s focus on the work of music as an artifact. Music cognition provides a
particularly strong foundation for defining core elements required for interactive
computer music. Machine listening involves a number of ways in which the computer
may parse input data to determine salient features of the music. In live, interactive music
this is done in real-time and is therefore modeled after human listening. Models of
music theory may also be incorporated, particularly the notion of musical analysis.

Robert Rowe points out the interdependencies of these two ideas:

The contrast between listening and music analysis should be drawn here: analysis
is related to listening, as are all musical skills, but differs . . . musical analysis has
random access to the material; the analyst . . . can consult [the score] in any order
regardless of the temporal presentation of the piece in performance. . . . The
listener, by contrast, is constrained to hear the piece from left to right . . . the
music must be processed as it arrives. The groupings and relations the listener
forms can only arise from cognitively available perceptual events with short- and
long-term memory. Seen in this light, the problems of music listening are simply a part of the larger problem of human cognition.  

In Rowe’s real-time music analysis implementations, the process does not claim to “reverse engineer” the way humans listen but aims at capturing musicianship and identifying enough musical elements to interact with human performers on their terms.  

Agents  

In interactive computer music, the computer has a role as a participant in an interaction. In AI terms, the computer's role is as an agent. "An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors." By this definition, the human performers meet the criteria of being agents as well. A rational agent is one which can be said to do the right thing. In any interactive performance, one presumes that the human performers are rational. If there are computer agents involved, it is a design goal for the computer to carry out actions that will cause it to be the most successful. In order to determine success, there must be an appropriate set of performance measures in place. Efficiency in processing time is one very important performance measure in live performance. To a human performing alongside a computer performer, the main performance measure involves making it through the end of a piece without any major malfunctions. To an audience, the ultimate performance measure may be related to aesthetic quality, which is

---

8 Ibid., 96.  
10 Ibid., 31.  
11 Ibid., 32.
a matter of personal perspective and perhaps not the first priority in determining whether an agent for interactive computer music is functioning successfully.

If an agent bases its actions entirely on a prescribed set of rules, it is said to lack autonomy. Incorporating an ability to learn would likely always be a design advantage. Rule-based systems may provide convincing opponents for video games, however they are by definition unable to improvise. For an improvisatory musical interaction to take place, a computer must be able to generate new musical material that is appropriate for a given context. The PODS project intends to aid this type of process.

Heuristics

This word, coming from the Greek *heuriskein*, meaning “to find” or “to discover,” has its own unique history in the field of artificial intelligence: “Currently, heuristic . . . [refers] to any technique that improves the average case performance on a problem solving task, but does not necessarily improve the worst case performance. In the specific area of search algorithms, it refers to a function that provides an estimate of cost.” David Huron also uses this term to describe “imperfect rules of thumb” used in forming expectations. However similar in meaning, it does not describe the same thing.

A best-first search involves a heuristic given from some means of evaluation that is intended to improve efficiency in processing. For example, when searching through nodes of a tree, the nodes are expanded in an order of best to worst options, based on a

---

12 Ibid., 35.
value given by an evaluation function. There are a variety of best-first algorithms with
differing means of evaluation.\(^{15}\)

Heuristics, outside the realm of search algorithms, typically pertains to some type
of machine learning. Among the examples presented here, some are considered to be
genetic algorithms, also know as evolutionary algorithms. They operate similarly to the
biological principle of survival of the fittest. Better solutions evolve from previous
solutions, which are then measured by a fitness function. Two such implementations of a
genetic algorithm involve timbre.

Markov Models

One extensively explored approach for machine composition involves Markov
models. PODS stores everything needed to implement higher order Markov models for
generating musical output. For some dimension of music, i.e. the pitch interval domain, a
first order model would show, for any given previous choice, possibilities for the next
choice and their weights (likelihoods derived from score values). An nth order model
gives possibilities and probabilities based on n previous choices. David Cope’s work in
this area involves filling a database with musical examples of some particular composer
or style and analyzing it into nth order Markov models.\(^{16}\) Models of musical construction
in works of Bach, Beethoven, Chopin, Joplin, and others have been used by David Cope
to generate output corresponding to the composers’ style. For a PODS nth order Markov
model, n is limited by the tree depth (defined by length of lists sent to PODS), but it can
be used to implement Markov models for machine composition.

Cope’s approach addresses composition in terms of musical pitch and rhythm structures. Newer versions of his programs consider dynamics and timbre as well.\(^\text{17}\) While somewhat multidimensional, only a specific set of dimensions and interrelations can be used, i.e. it can not be used for music without pitches. PODS functions without the need to specify what dimension it is or which dimension organizes another. While these distinctions can be made with the PODS data, the representation is not limited by them. In other words, the same data are available, but the hierarchy can be changed around, i.e. rhythm could be moved to the top level and be organized by dynamics, and instead of pitches it could organize possibilities of non-pitched sound shapes.

**Timbre**

A useful definition for timbre in terms of electroacoustic music where sounds may be abstract – not tangible in the real world – comes from Denis Smalley. The model he calls *spectromorphology* describes any sound by its vertical and horizontal aspects: “The two parts of the term refer to the interaction between sound spectra (spectro-) and the ways they change over time (-morphology). The spectro- cannot exist without the – morphology and vice versa: something has to be shaped, and a shape must have sonic content.”\(^\text{18}\) This serves as a basis for a definition of timbre from a cognitive perspective, that some shape of sound spectrum over time is the subject of timbral identity, which is afforded only by a process of cognition and recognition by an observer.\(^\text{19}\) A given sound may have a layer of identities. For example, you have the sound, but it could also be that


sound in some room, its relative location in the room, or its spectral balance might be altered by the proximity effect; however, the identity of the sound does not become altered or obfuscated by these other layers of timbral identity.20

Examples of Genetic Algorithms

In order for a computer to process timbre, many aspects of the sound must be broken down, parsing the data into single attributes.21 This tangle of data can be addressed by use of a genetic algorithm. Each feature described becomes a “chromosome” or in other words, a trait that may be passed to offspring in the search for the most fit offspring. One proposed model includes forty dimensions. Some of these are in the time domain, attack time, RMS energy, zero crossing rate, and crest factor. The frequency domain includes spectral centroid, spectral spread, spectral flatness, flux, presence, and roll off, and there are twenty-nine others in the categories of partial domain (harmonics), trajectory attributes, periodic attributes, and statistical attributes.22

A useful application of this type of work is being implemented at IRCAM by Tardieu, Carpentier and Rodet,23 who have put together a computer aid for orchestration. Their system attempts to determine the closest combinations of instruments to imitate a

---

20 Ibid.
22 Ibid., 180-81.
given sound.\textsuperscript{24} While this has no obvious direct use in interactive performance, they have reported it to be efficient and to yield interesting results with large orchestras.

One real-time implementation of a genetic algorithm is designed to control an improvisatory robotic xylophone player.\textsuperscript{25} The robot analyzes input, which can be either MIDI or audio from human players.\textsuperscript{26} The output is informed by the analyzed input; however, the base population for generating the evolved responses includes approximately forty melodic excerpts of different lengths and styles.\textsuperscript{27} The fitness function compares similarity to the input. There have been pieces written and successfully performed with the robot.

\begin{footnotesize}
\begin{enumerate}
\item \textsuperscript{24} Ibid., 188-91.
\item \textsuperscript{26} Ibid., 353.
\item \textsuperscript{27} Ibid., 353.
\end{enumerate}
\end{footnotesize}
CHAPTER 2
SOFTWARE BACKGROUNDS

Max/MSP

Max/MSP is a graphic object oriented computer programming environment optimized for real-time computer music performance. It is the most commonly used of such environments, largely because of its intuitive interface and ease of use. The Max environment provides a graphical user interface with objects representing specific functions to process events or digital audio signals, and "patch cord" style connections between these objects. This provides an efficient way to create objects that send messages to each other by linking outlets to inlets of other objects. The number of inlets and outlets for an object may depend upon arguments given at the object’s creation, allowing for flexible parsing of multiple data and audio streams; however, the basic data types are extremely limited. Events may be either integers, 32-bit floating point numbers, strings, or lists (which may contain any combination of the other types). One-dimensional arrays and text-based collections of lists are available through objects built into the software. However, models of music tend to require a great deal of variety, dimensionality, and linkage between data. Max can overcome such shortcomings because it allows for external objects to be coded by users with the Max Software Developers Kit.28 Max externals are normally coded in the C language. Many Max

externals are written in C++, which is a strict superset of C. C++, Objective-C, or any other strict superset of C can be used to write Max externals, since their compilers can include the libraries necessary to create a Max object. Objective-C is the best fit for the needs of the PODS project, as explained below.

Max was chosen for this project because it is widely used for interactive computer music, and it provides a useful set of tools for parsing data from live input to be sent to PODS. However, the Max libraries only define a way for the Objective-C classes that define PODS to communicate within the Max interface. Therefore, PODS is not limited to working with Max, and it can be adapted to work with other software environments in the future.

Objective-C

The primary task of the PODS object is to compare incoming data with stored data in order to determine how the incoming data should be stored and linked to data previously received. Objective-C provides particularly robust and well-defined frameworks for this sort of data comparison. The existing Objective-C classes discussed in this paper are identified with the prefix “NS”. This abbreviation comes from NeXTStep, which is an operating system developed by NeXT Computer. NeXT licensed Objective-C from Stepstone, whose efforts focused on creating an object oriented programming language like Smalltalk, by simply adding onto the C

---

programming language. Therefore Objective-C is a strict superset of the C language.\textsuperscript{30} Like C++, an Objective-C compiler can compile any C code, though it may not work for C++ code. No matter what software languages are used, a new Max external object can be created, provided a compiler will compile it.

Every Objective-C object inherits from the root class, NSObject. The important method for comparison of all Objective-C objects is defined in the NSObject class as isEqual. Certain PODS objects override the isEqual method by redefining it within their class. In order for two objects to be equal, they must return the same hash value. Therefore, hash, the method that returns a hash value, is redefined for those same classes.

NSNumber is a wrapper class that may contain numbers of various types, and it allows for comparison of values of different types, i.e. a 32-bit integer and a 64-bit double precision floating-point number can be compared based on the value they describe. The NSNumber method isEqualToNumber handles these comparisons, whereas a simpler isEqual (inherited from NSObject) would see values of different types as unequal. Similarly, NSString can compare string values by more advanced means, however they are not necessary for PODS. The PODS object assumes that a string, “1.0”, a floating-point number, 1.0, and an integer, 1, are distinctly different inputs, as they are in the Max environment. So, the isEqual and hash methods are suitable for all comparisons. This saves some extra steps and reduces processing time, because all values can be compared without the need to determine data types, and it maintains the distinction between data types.

NSDictionary stores data associated with strings used as key values. The method objectForKey retrieves any object stored for a given key value. NSArray can be used to store many different data structures, because it stores objects. It can easily be implemented as a tree or linked list structure. NSArray has a method, indexOfObject, which searches the array and returns an index value. NSMutableArray is simply a resizable version of NSArray.

Searching NSMutableArray

With the exception of the tree structures, a search through an NSMutableArray object is facilitated by a method inherited from NSArray, indexOfObject, which returns the lowest index for an array value equal to an object. An object is considered equal if isEqual returns YES. In order for indexOfObject to work properly for PODS, the isEqual method is overridden in PODS node classes and Synapse class. The PODS Synapse class is for interlinking data. It stores an index for the linked data and a score indicating the number of times an association has been made. When searching through Synapse objects, the index is the only relevant datum. Indices into NSArray are always returned as type NSUInteger (an unsigned integer value), therefore PODS uses indices of the same type (recall that NSNumber is an object wrapper for various types of numbers including NSUInteger). When a Synapse object is compared to an NSNumber object containing an NSUInteger, the Synapse object uses the hash value of the stored index (instead of the hash value for that object), which is also an NSUInteger. Since they are the same type, if the values are equal, then the hash values will be equal as well. The nodes work the same way. At the node level, an object for comparison will either be NSString or NSNumber.
Either way the hash value for the incoming data is compared to the hash value for the main datum contained in the nodes. So without the override, it would compare the hash value for the node object instead of the main datum.
CHAPTER 3
INTRODUCTION TO PODS
PODS: A Pattern Oriented Data Structure

A goal for this project is to create a music-optimized data structure that is suitable for efficient retrieval of the most significant patterns given to it, and other interlinked data. The organization assumes that humans find musical significance in the patterns that occur most often within the context of a given work, and that these patterns may rely on multiple dimensions of data. The overall goal is not only to store meaningful relationships among analysis data, but also to provide a well-structured database for real-time machine composition in an interactive performance.

A pattern oriented data structure, or PODS, object is an instance of a self-organizing storage class. Everything that it responds to is scheduled by the Max/MSP environment. The user defines a means of real-time analysis involving at least one salient feature of music, according to the needs and context of a given project (e.g. a composition or an improvisation environment). The PODS object may receive multiple input data of varying types, which are stored in separate interlinked data structures. The main advantage of the PODS object is to track the recurrence of patterns input as ordered lists of numbers, and possibly store their association with other patterns or values describing other salient features of music or other additional relevant information.

Another main design goal for PODS is to be flexible, adaptable, and portable. The classes that define PODS are not strictly for Max. Only the outer level of PODS interfaces with Max, so it can be adapted to other environments. PODS stores objects rather than specific data types, so it can store any assortment of C data. In some cases the user may need to define an NSObject wrapper class for the data, but this is very simple to do. It is not unreasonable to assume that PODS could be adapted for use in unrelated fields of interest, like analyzing business data.

PODS offers a number of possibilities for machine composition. The Max environment allows communication between objects through messages, and the PODS data could be traversed this way. In most cases, traversing the PODS data structure by using messages comes with the added cost of converting the data in some way, however it can be useful for distributing processes over a network using Open Sound Control.\textsuperscript{32} The best solution for using the data is to create custom PODS traversal objects that are designed with specific compositional goals in mind. A PODS traversal object can send a \textit{location} message a PODS object, which will then output the memory location for the outer level NSDictionary for all of the PODS data. No other data is needed to traverse a PODS data structure.

Suggestions for Implementation

It is up to the user to decide what salient features of music should be stored in the PODS object. It may only be necessary to track pitch, and pitch intervals. Alternatively one might wish to create an implementation where rhythm and dynamics data are parsed

\footnotesize{\textsuperscript{32} “Open Sound Control” accessed on December 9, 2010, http://www.opensoundcontrol.org.}
and organized within the PODS object. It could equally be used for generalized data describing gesture or shape.

Timbre analysis, as described earlier, can include a multitude of related data sets. The PODS object includes "Synapses," or integrated links between multiple data sets. The more generalized the relationships are, the more meaningful associations can be made, because patterns and their recurrence will be more evident. Generalization can mean rounding floating-point numbers to integers, but it can be done in a way that more closely models the way that humans process timbre. Consider each dimension of timbre as an identity and a given sound as having a layer of identities. The human process involves recognition of layered identities bundled in a single percept. These identities can be considered by the perceiver as a unique identity or as individual identities, in detail or in general, and can be related on any such level to any other percept in his or her entire perceptual history. In contrast, the computer can only recognize general attributes of a given sound based on analysis of very specific data. Therefore, a group of generalizations can be stored, interlinked to the specific data, and in turn interlinking it to other data associated with those generalizations.

An example of using PODS for an autonomous agent for machine composition involves best-first traversal through data that varies according to a weighted randomness. It is weighted towards the highest scoring choice (a choice with the most occurrences is the best choice), but may randomly choose other near-best options. A given PODS

structure contains reliable data about starting pitches, interval sequences, rhythmic sequences, and sequences of note dynamics and the interrelations therein. Each serves as a gene for a genetic algorithm. So, offspring are created from various combinations of related, high-scoring values from each dimension. The weighted traversal might or might not branch out to related data according to chance, as shown in the following routine description.

Generating offspring from the PODS data, benefits from a good starting place. In this example, it is simple: the pitch of a note presently or most recently played in a performance. The routine for generating offspring operates is as follows:

1. Find that pitch in an array of starting pitches.
2. From the starting pitch node, traverse best and near-best pitch interval sequences.
3. Either use those sequences or find other best and near-best starting pitches for those sequences, and traverse other interval sequences from those pitches, sometimes transposing back to the original starting pitch.
4. For every sequence, use best or near-best rhythmic patterns linked to those sequences.
5. Use sequences of note dynamics linked to either the rhythmic sequence or the interval sequence.

This yields many options of musical material, offspring, that are then compared using a fitness function. The fitness function looks for compatibility with recent music input, and chooses the best offspring.
A good fitness function is important. Depending on the artistic goals of a given musical context, this might pertain to mode mapping, timbre, rhythmic structures, melodic contour, gesture, rhythmic or tonal density, etc. In a real-time implementation for live music, it is beneficial to specify a set of rules that are not entirely based on input but are also defined according to specifications or tendencies of a particular section of music.

A Simple Implementation Example

The following example shows the PODS object as it parses and stores pitch interval sequences. A circular buffer of six data is maintained, updated at each new note event. Each datum contained in the buffer describes the pitch of a note as a MIDI pitch value, an byte integer (0 - 127) where middle C is 60. Between the six pitches in the buffer exist five pitch intervals, or differences between the values of consecutive pitch (specifically a pitch value subtracted from the value of the previous pitch). The musical phrase shown in figure 1 will be the basis of this example.

![Figure 1. Example musical phrase.](image)

When the buffer has been filled and each subsequent time the buffer is updated, the intervals between pitches are calculated and the sequence of intervals is sent to the PODS object. Figure 2 shows the buffer as the seventh pitch comes; the first pitch is
discarded, and the sequence of pitch intervals is calculated as shown. Positions in the list
of intervals are labeled here as A, B, C, D, and E.

![Pitch buffer for interval analysis implementation](image)

**Figure 2. Example pitch buffer state.**

The output list of intervals \{1, 3, -8, 1, -12\}, may be partitioned into subsets of
consecutive intervals, shown in figure 3. The PODS object represents these subsets in a
minimally redundant data structure. This is needed both to allow easy detection of
patterns and recurrence, and to eliminate the redundancy of data that comes with
successive passes through the circular buffer. In figure 3, the subsets are grouped by
their initial datum, and those groups are labeled according to the position of that datum in
the list, as depicted in figure 2.
Because of the nature of the circular buffer, subsequent sets of intervals will contain some subsets that are identical to the current subsets. Figure 4 shows all of the possible subsets for the subsequent list of intervals. The blacked out subsets are already present in figure 3. In group A of figure 4 to the subsets in group B of figure 3. The subsets in group B of figure 3; they are identical to groups B, C, D, and E. To avoid redundancy, only group A subsets (i.e. those containing the oldest interval) will be parsed by the PODS object.
In order to track statistical data about the frequency of occurrence of each subset, it is not necessary or helpful to store identical portions of each subset separately. The subsets can be stored as linked nodes, each of which is given a score, an integer value representing the number of times that particular subset has been processed. So the first subset \{1\} is represented at a node (1) linked to a child node (3) which represents the subset \{1, 3\}, and so on. Linked nodes (1)-(3)-(8)-(1)-(12) then represent all of the group A subsets, and each node is associated with its own score.

As more sets are evaluated and stored, smaller subsets are likely to occur most often and are likely to be a common element of many differing larger subsets. Thus the data are well represented in a tree structure. The scores given to each node facilitate an approach similar to a best-first heuristic for searching through the tree. Patterns are stored with minimal redundancy and with a built-in count of their frequency of occurrence.

Walking Through the Example

Figure 5 shows the first five intervals taken from the melody in figure 1 as stored by the PODS object after the circular buffer is first filled. The PODS object adds the set \{3, 1, 3, -8, 1\} to its tree, giving each subset a score of one. In figure 5, the node (3)\(^1\) represents subset \{3\} with a score of one. Node (1)\(^1\) – being the child of the node above – represents subset \{3, 1\} with a score of one, etc.
Figure 5. First five intervals added to tree structure.

When the seventh note of the melody is received (as in figure 2), the PODS object must store the data set \{1, 3, -8, 1, -12\}. Starting from the root of the tree in figure 5, there is no child that matches the smallest subset \{1\}. The PODS object adds the child to the root, with an initial score of one. Likewise, the remaining subsets are added to the tree with a score of one, as shown in figure 6.

Figure 6. Next tree state.
The next set given to the PODS object is \{3, -8, 1, -12, 7\}. Illustrated in figure 7, the first subset \{3\} is already present as a child of the root. The PODS object adds one to that node’s score. The subsequent elements are not present, so they form a new branch, with each node initialized with a score of one.

![Figure 7. Branch added at node (3).](image)

The next data set \{-8, 1, -12, 7, 4\} causes a new child along with its children to be added to the root with initial score values of one. \{1, -12, 7, 4, 1\} brings the score of the existing subset \{1\} from one to two, but there is no child subset representing \{1, -12\}, so it is added along with its children, bringing the tree to the state represented in figure 8.
As the PODS object continues to receive data, the next interval set \{-12, 7, 4, 1, 3\} branches from the root. \{7, 4, 1, 3, -8\} followed by \{4, 1, 3, -8, 4\} also branch from the root. Bringing the tree to the state represented in of figure 9, \{1, 3, -8, 4, 1\} adds one to root child \{1\} – making a score of three – then adds to the subsequent children \{1, 3\} and \{1, 3, -8\} – giving them both a score of two, and the remaining children are added to the tree with initial score values.
Another Implementation Approach

The example above ignores the dimension of time, a crucial determinant of musical content. Fast notes are treated identically to slow, sustained ones. Segmentation according to long rests is not available. Calculation stops when no new note events occur to trigger output. If a musical passage begins with sparse and long-sustained notes, the buffer takes too long to fill up, particularly if it is much larger than the previous six note example. A buffer for event data that occur within a moving window of time produces different and potentially more useful results.

In the following example, a moving window of time is used to evaluate the musical material shown in figure 1. Pitches are added to a buffer of arbitrary size as they are input, but are removed from the buffer according to elapsed time. A window duration of N milliseconds (set by the user) represents the span of time in which pitch intervals can be considered as a pattern to be sent to the PODS object. A clock counts time
continuously (from the time at which the program is launched). When a pitch is stored, the current clock time is stored with it.

The clock is also used to query the buffer at a regular time interval of \( M \) milliseconds. \( M \) should be less than the smallest expected note duration, but not so small as to overburden the processor with redundant calculations. At each query, the stored time stamps are compared with the current time. If the difference between the current time and the time stamp of a given pitch exceeds \( N - M \) (meaning that it is at the front of the buffer, but should not be in the buffer the next time it is queried), output is triggered, and that pitch datum is removed from the buffer.

Whenever output is triggered, the intervals between all pitches contained in the buffer are evaluated. If no intervals are present (i.e. just one pitch is present in the window), no output is sent. Otherwise, interval sequences are sent to the PODS object. The pitch that triggered this process is removed from the buffer.

If the onset of a note occurs before the end of the buffer, which in real-time analysis approximates the present time, then the interval between it and the previous note is included in an output regardless of whether the note is still being played. In other words, the note need not be completed, because the interval information occurs between the cessation of one note and the onset of another.\(^{34}\) As described, when there is only one note in the buffer, there is no interval, because there is no other note with which to compare it. In this case, no output is triggered, and it is simply removed from the buffer. It would be problematic to

\(^{34}\) This assumes a monophonic instrument, on which only one note can sound at any time. Polyphonic instruments (like piano or guitar) present more complicated problems.
remove the only note in the buffer when it reaches the beginning of the buffer, if that note were still being played. In such a case the cheapest solution is to reinsert the note into the buffer, however this is not a necessary process in the context of the following example.

Time Buffer Example Walkthrough

Figure 10 shows the first few groupings of data triggered. For this example, the duration of one and a half measures of musical material as performed is exactly equal to the size of the time buffer; while it doesn’t model real world input, it makes the example clearer. The first grouping includes measure one and the first half of measure two, with the beginning of the buffer being exactly between beats two and three of measure two. Similarly, the next group stops in the middle of the following measure. For the third group, the half note at the end of measure three is still being played when output is triggered by the third note of the musical passage arriving at the front of the time buffer. The fourth group begins halfway through measure two and ends directly between measures three and four, and so on.

Figure 10. Data parsed by periods of time.
The entire phrase is played and all of the notes in the buffer expire. Every note (except for the last note) has caused the output of sets of intervals, as follows:

\[
\begin{align*}
\{3, 1\}, \{1, 3, -8, 1, -12, 7, 4\}, \{3, -8, 1, -12, 7, 4, 1\}, \\
\{1, -12, 7, 4, 1, 3\}, \{-12, 7, 4, 1, 3\}, \{7, 4, 1, 3, -8\}, \{4, 1, 3, -8\}, \{1, 3, -8, 4\}, \\
\{3, -8, 4\}, \{-8, 4, 1\}, \{4, 1\}, \{1\}\}.
\end{align*}
\]

It is immediately apparent that the sets vary considerably in length. Figure 11 shows these sets as represented in a PODS object.

Notice that the some of the nodes representing the first interval of each ordered list – the children of the root – have higher scores than the previous example. This is because the intervals at the end of the musical phrase were processed at the beginning of their own list of consecutive intervals list (specifically, the last four sets processed in the present example).
In both examples (figure 9 and figure 11) a similar result occurs: of all the nodes at depth three (i.e. four-note patterns), the only one with a score greater than 1 is \( \{1, 3, -8\} \). Given the assumption that recurrence is an important feature of music, this pattern is of evident musical importance: not only does \( \{1, 3, -8\} \) occur twice, but it does so within a relatively small span of time. Upon examining the entire interval series given in these two examples,

\[\{3, 1, 3, -8, 1, -12, 7, 4, 1, 3, -8, 4, 1\}\]

it is evident, however, some important information is missing. It is not clear whether the repeated pattern involved the same notes or whether it was transposed. To remedy this, another dimension of data needs to be added to the system.
CHAPTER 4
PODS INTERLINKED DATA

Storing Related Data

As mentioned, the pattern oriented data structure, or PODS, object can store other relevant data with each input. To improve on the implementation given in the previous example, the starting pitch of each list of intervals can be stored and related to each subset along with a score describing the recurrence of a given starting pitch for each subset. Suppose the previous example is modified to include the starting pitch. As the musical example is processed, the subset \{1, 3, -8\} is linked with two starting pitches, 63 and 59, each with a score of one. Now it is clear that this pattern was repeated in a transposition.

The PODS object accepts creation arguments including an identifier for the PODS object, an integer indicating the number of different ordered lists in a combined input, followed by identifiers associated with each one, followed by an integer indicating the number of singular (non-list) data that it will receive, along with identifiers associated with those data. All lists are treated as ordered lists, even if the order of the data contained in the list is irrelevant, to maximize efficiency.

Data are interlinked by associations that tell where other data are stored. In the case of storing the integer value for a pitch, it is somewhat convoluted to store it somewhere and separately store another integer somewhere else to describe its location. It slightly increases the amount of storage and processing required. However, it is
necessary so that all data associations can be stored and retrieved using the same simple data class, the Synapse class, regardless of type. The Synapse class has two variables: an index and a score. The score is always a score for its index at the node in which the Synapse object is stored, not the score for the node that the index points to.

More Objective-C Background

Data sent to the PODS object need only be distinguished as a list or non-list. The type of data within a list is not a necessary distinction, nor is the type of non-list data. Max does distinguish these data types, which helps to define which Objective-C class is appropriate for each datum, i.e a string is stored in an NSString object, an integer is stored in an NSNumber object as type NSInteger, etc. But regardless of type, all comparisons within the PODS object are performed using the same methods.

Detailed Overview of Data Storage

When a PODS object is created, a combined input is defined by identifier strings that associate with individual inlets, which are initialized to accept either lists of data or individual data. The identifier strings are also used as a key values that associate data at every level of the data structure. The entire data structure exists in an NSDictionary object, which stores an object for each key. From this outer level, the object associated with each key is an NSMutableArray.

Input data of type list are stored in tree structures. The tree structures are stored with a root node as the first element in the NSMmutableArray object. The root node, defined in PODS by the RootNode class, is simpler than other tree nodes, because it only needs to store the indices of its children ordered by score. Tree nodes, defined in PODS
by the TreeNode class, store a main datum, a score, a parent index, indices of children ordered by score, and a structure for interlinked data.

All individual, non-list data are also stored in instances of NSMutableArray. The nodes, defined in PODS by the by the SingleNode class, store data similar to the tree nodes; the exception being that there are no parent and children indices. Also, there is no root node at the beginning of the array. SingleNode objects have a data structure interlinking them with other data, exactly like TreeNode object nodes. This structure, for both types of node, consists of an outer-level NSDictionary object, containing an NSMutableArray object for each key value, other than the key value for the array in which the node is contained. The NSMutableArray objects at the node level, will only contain Synapse objects.

Starting Pitch Storage Example

This example is an extension of the previous example, modified to store starting pitch. The PODS object is given creation arguments initializing it with a tree structure for input of ordered lists identified by the key, intervals, and a structure for singular data identified by the key, start_pitch. The first output is triggered, and a set of intervals, \{3, 1\}, and the starting pitch, 60, are sent to the PODS object. These data are processed as illustrated in figure 12. The set \{3, 1\} is added to the tree structure. The first value, three, is searched for among the root’s children, but is not found. New node (3) is initialized with a score of one, and the index for the root node, zero, is stored as its parent and the index for node (3) is stored as a child of the root. The second value, one, is searched for among the children for node (3) but is not found. Node (1) is initialized with
a score of one, and the index for the node (3) is stored as its parent, and the index for node (1) is stored as a child of node (3). Note that indices of new nodes are added onto the end of arrays in which they are stored, but when a child node’s score is being updated, the array of child indices is reordered with the highest scoring values first.

The starting pitch array, like the interval tree, is stored in the outer level NSDictionary object. Within NS Mutable Array associated with the key start_pitch, the new pitch datum is searched for among all previous pitch data. The array is empty, so it is not found. A new node is created for the starting pitch with a score of one. The new node is stored at the next available position in the array, index zero.

Notice that figure 12 illustrates the differences between these two types of nodes. The start_pitch node doesn’t need to store a parent or children, and the array containing those nodes does not begin with a root node. Aside form these distinctions, the two structures store their main data and interlinking data the same way. Synapse objects are created after every node for this combined input of intervals and start_pitch, are created, because the starting pitch node will need to store only the index of the deepest node of the interval list. Using the deepest index, the parents can be traced back up to the root.
Resuming the example, the starting pitch node has been added to the \textit{start\_pitch} array, and it’s key and index are cued to be linked to other data. All of the interval nodes have been initialized and stored, and their key and node indices are cued for linking. For the key, \textit{start\_pitch}, the deepest node of the \textit{intervals} set’s array index, two, is evaluated for storage; the node is checked for an existing Synapse object with the index value of two. It is not found so a Synapse object is created and stored in the \textit{start\_pitch} node. For the key, \textit{interval}, array indices one and two are cued to be linked. For the \textit{interval} node (3), a Synapse object with an index of zero for \textit{start\_pitch} is searched for but not found, so a new Synapse object is created and stored in the node. The same happens for node...
(1). Note that, like the child indices, new Synapse objects are added onto the end of arrays in which they are stored, but when a Synapse object’s score is being updated, the corresponding array is reordered with the highest scoring values first.

An informal algorithm in figure 13 describes the PODS storage routines. Recall that the user defines some number of list inputs and some number of non-list inputs, and that each of them is identified by a key. PODS keeps track of whether an input is a list or non-list for each key. The test for this only involves looking up a stored Boolean value for a given key. There are two main outer loops: one for creating or updating nodes followed by interlinking.

Conclusion

With high-resolution data, particular in dimensions that are difficult to quantize (such as time and amplitude), it can be difficult to identify patterns. Recurring patterns that can be detected intuitively can appear as very different data sets. Thus, issues of resolving and quantizing data must be addressed before the PODS object can become useful. This is an inherent limitation of the PODS model. The biggest limitation in interactive performance is computation time. Whatever composition algorithm is used, it must operate within strict time deadlines. While PODS allows for a multidimensional data structure, there may be a point where processing data in many dimensions would cause the performance to suffer. With a genetic algorithm, for example, the more dimensions there are to process, the fewer offspring can be produced in a given period of time.
At present, this project is in its beginning stages. Progress can be tracked online, and anyone is invited to join. The Synapse class is online, and the SingleNode class is almost completed. The TreeNode class is similar and is to be completed next, accompanied by the RootNode class. There is also a testing interface in development for these classes, and each will be posted online only after being tested; then the PODS class for Max can be written. The goal is to have all of these classes fully tested and available for others to develop their own customized PODS traversal objects for machine composition.

Music cognition involves expectations created in real-time by comparing perceived music to a vast network of meaning that comes from an extended perceptual history. In contrast, for musical material to be evaluated by a computer, it must be parsed into individual data with little intrinsic meaning. In order for the computer to function as a rational agent in a musical environment, it is necessary for these disparate aspects to somehow be reassembled into a plausible model of music. A network of interrelations between multiple data sets must be generated, and a criterion for significance of those interrelations must be built into the model. PODS is designed to fulfill these criteria and to be versatile and useful in a variety of applications.
//create or update nodes
for each key value {
    if (input is a list) {
        for each element in the list {
            for each child node { /* starting at root for key */
                if (list element value equals child value) {
                    add 1 to child score
                    make child current node
                }
                else {
                    create new node
                    value = list element value
                    score = 1
                    make new child current node
                }
                if (last list element) {
                    store current node index with current key value
                }
            }
        }
    }
    else {
        if (input value equals a value in array for key) {
            add 1 to node score with that value
        }
        else {
            create new node
            value = input value
            score = 1
        }
        store node index with current key value
    }
}

//link data
for each key value /* current key */ {
    for each key value /* stored index key */ {
        if (stored index key is not equal to current key) {
            if (data for current key is list type) {
                do { /* starting at index for current key */
                    if (link exists for stored index) {
                        update link score
                    }
                    else {
                        create new link
                        score = 1
                        index = stored index
                    }
                } while (parent is not root)
            }
        }
        else { /* starting at index for current key */
            if (link exists for stored index) {
                update link score
            }
        }
        else {
            create new link
            score = 1
            index = stored index
        }
    }
}}}}
REFERENCE LIST


