HUMDRUMR: A NEW TAKE ON AN OLD APPROACH TO COMPUTATIONAL MUSICOLOGY

Nathaniel Condit-Schultz
Georgia Institute of Technology
natcs@gatech.edu

Claire Arthur
Georgia Institute of Technology
claire.arthur@gatech.edu

ABSTRACT

Musicology research is a fundamentally humanistic endeavor. However, despite the productive work of a small niche of humanities-trained computational musicologists, most cutting-edge digital music research is pursued by scholars whose primary training is scientific or computational, not humanistic. This unfortunate situation is prolonged, at least in part, by the daunting barrier that computer coding presents to humanities scholars with no technical training. In this paper, we present `humdrumR` ("hum-drummer"), a software package designed to afford computational musicology research for both advanced and novice computer coders. Humdrum is a powerful and influential existing computational musicology framework including the *humdrum syntax*—a flexible text data format with tens of thousands of extant scores available—and the Bash-based *humdrum toolkit*. `HumdrumR` is a modern replacement for the humdrum toolkit, based in the data-analysis/statistical programming language R. By combining the flexibility and transparency of the humdrum syntax with the powerful data analysis tools and concise syntax of R, `humdrumR` offers an appealing new approach to would-be computational musicologists. `HumdrumR` leverages R's powerful metaprogramming capabilities to create an extremely expressive and composable syntax, allowing novices to achieve usable analyses quickly while avoiding many coding concepts that are commonly challenging for beginners.

1. INTRODUCTION

Though digital musicology has been a productive area of research for several decades (e.g., [2, 8, 12, 16, 18–20, 22–24, 26, 28, 29]), it remains a niche field within the musical-side of academia. In fact, most cutting-edge, scientific music research has been pursued by researchers with primary training in computer science and psychology. Fortunately, recent years have seen a flourishing of "digital humanities" research in general, with increasing numbers of traditional humanities scholars adopting computational approaches. Humanist music scholars' deep, nuanced knowledge of musical culture, structure, and practice could be an invaluable asset to the computational/empirical music research community. Unfortunately, the training necessary for fruitful computational scholarship is absent from most music curricula, which cover neither the fundamental methodological principals of empirical research nor the necessary coding skills. The need to assist, and convince, music scholars to learn programming has been an area of active discussion for some time [7], in particular there is a need for software tools crafted to support their learning and research goals [32].

To make computational research truly appealing and accessible to traditional humanists and seasoned computational researchers alike, we must juggle three conflicting factors: flexibility, power, and usability. User-friendly interfaces like the Josquin Research Project's Analysis Tools [1], Theme Finder [2], or rapscience.net's Visualizer [3] can be utilized by scholars with no special training. However, such tools allow only variations of hard-coded analyses applied to limited, fixed databases. On the opposite extreme, any skilled programmer can code their own symbolic music data formats and analysis/parsing software "from scratch"—as many computational projects [4, 11] have done—, allowing for the unlimited power of the programming language of their choice, but requiring substantially more effort and experience. The most prominent modern computational musicology toolkit—`music21` [10]—, in our estimation, falls too close to the later extreme, proving quite daunting to novice coders.

This paper describes `humdrumR` ("hum-drummer"), a software toolkit for symbolic musicological data analysis intended to be appealing and accessible to traditional musicologists while also being useful to more experienced computational researchers. Learning lessons from the successes and failures of existing tools, `humdrumR` strikes a powerful new balance between flexibility, power, and usability:

1. Using the humdrum data syntax, `humdrumR` is extremely flexible and general in scope, allowing users to study any type of performance-art data that can be represented symbolically—musical scores, dance steps, trumpet fingerings, etc.

[1] http://josquin.stanford.edu/search/
[2] http://www.themefinder.org/
[3] http://rapscience.net/Analysis/gui.html

2. Based in the R programming language, humdrumR is extremely powerful, capable of complex data manipulation and interfacing with R's statistics, visualization, and machine learning libraries.

3. humdrumR is designed to present a relatively low barrier of entry for non-technical researchers, offering a concise, expressive syntax for applying music analyses while avoiding difficult coding paradigms.

In 2005, musicologist and psychologist Nicholas Cook observed that to truly engage in computational musicology "proper," scholars must achieve "sufficient understanding of the symbolical processing and data representation on which it's based" [7]. We believe that the combination of humdrum and R represents an ideal avenue for computational novices to develop this "sufficient" understanding: enough to pursue quality computational research but without having to engage with general coding paradigms that are irrelevant to their research.

In this paper, we first review the relevant philosophical and technical features of humdrum and R, noting how humdrumR incorporates these features (sections 2 and 3). We next contrast humdrum(R) coding philosophy and style with that of music21 (section 4). Finally, we describe the principle features of the humdrumR package and humdrumR syntax (section 5), including numerous code examples.

2. HUMDRUM

Humdrum[4] is a system for computational musicology research, created by David Huron [17] (circa 1995) and maintained by Craig Sapp at Stanford's Center for Computer Assisted Research in the Humanities. Though no one digital musicology framework has ever truly dominated the field, humdrum is certainly among the most widely used and influential systems, being cited as the direct inspiration for some of its most successful competitors—music21 [10] and MusicXML [13]—and with tens of thousands of scores available in humdrum format. Humdrum actually has two distinct components: the humdrum syntax and the humdrum toolkit.

The *humdrum toolkit* is a collection of Unix command-line tools for parsing and analyzing humdrum data, largely written in Bash and AWK but with more recent "extra" commands written in C++.[5] The humdrum toolkit is fundamentally entwined with the Bash shell, in particular the grep and sed commands, which rely heavily on *regular expressions* to parse and filter tokens. The humdrum toolkit also includes a number of analysis tools, notably the sophisticated windowing and n-gram tool context and the pattern finder patt. Using the toolkit, basic parsing and analysis can be achieved quickly and easily in Bash command pipes, but true Bash scripting is required for even mildly complicated tasks. In practice, after initial parsing, humdrum users must transition into a higher-level programming environment (Python, R, MATLAB, Julia, etc.) to achieve more complex data manipulation, statistical testing, or data visualization. The burden of parsing and manipulating humdrum data in the higher-level language falls on the user, substantially increasing the need for coding skills and prolonging the workflow. To make matters worse, installation of the humdrum toolkit can be fairly complicated, especially for Windows users, who must install a Unix emulator to use the toolkit at all.

HumdrumR is a successor to the humdrum toolkit, replacing all the original toolkit's functionality while adding significantly more. However, humdrumR uses the original humdrum syntax specification, and is thus compatible with all existing humdrum data. In fact, both the technical design and methodological philosophy of the humdrum syntax are fundamental to humdrumR.

2.1 The Humdrum Syntax

The humdrum syntax is an extremely general scheme for representing musicological data in plain-text. The syntax is basically tabular (tab-delineated columns) but with a few additional complexities:

- Columns of data are interpreted as *spines*, which can dynamically start, end, split, or merge, creating *spine paths*. Spine paths allow the syntax to represent many complex cases from music notation: For example, if the upper staff of a piano score splits temporarily into two voices (one with beams up, the other with beams down), this can be encoded by splitting the spine representing that staff into two columns (with a *^ token) before merging them again after the split passage (*v tokens).
- Humdrum records are tagged as *interpretation records* (representing local metadata) or *data records* (notes, chords, etc.). By simply interspersing interpretation and data records, metadata can be concisely associated with specific data points, passages, or sections. For example, key information can be quickly read, edited, or added to scores by simply inserting lines containing tokens like *f#: (f# minor). This approach contrasts with the more verbose, hierarchical attributes and tags used in XMLs.
- Finally, each "cell" (i.e., each line-column coordinate) in a humdrum file can contain multiple data tokens—a feature commonly used to represent chords but with myriad other potential use cases.

The humdrum syntax provides a data *structure* but says nothing about how information is actually encoded in data tokens. Rather, specific *interpretation* schemes must be defined. A few well-specified interpretations for music are widely used, most notably the **kern representation. However, users can freely define new interpretations, with the appropriate degree of rigour/precision/complexity for the task at hand. Thus, humdrum is not limited to traditional Western notation, but rather affords the study of non-Western, vernacular, or avant-garde musics, as any type of

[4] http://www.humdrum.org

[5] Sapp also maintains a C++ development library for humdrum tools called humlib (https://humlib.humdrum.org/).

musical feature, data, or metadata to be easily encoded in the same place. This presents a stark contrast with nearly all other music encoding schemes, including *MuseData* [15], abc [30], TinyNotation [9, ch. 16], LilyPond [25], and MusicXML, all of which are limited to representing music as, or in terms of, Western *music notation*—for instance, representing diatonic note names. Though `**kern` represents music in similar terms, the humdrum syntax in general has no such limitation. [6] Though the actively developing MEI [14] encoding standard *can* be extended to incorporate arbitrary music information, the emphasis of the MEI community is nonetheless representing music notation(s) for purposes of library science and score preservation and dissemination, not analysis.

Data encoding schemes inevitably balance read/writeability with power and flexibility [9, ch. 16]). Notation systems such as abc notation and TinyNotation achieve nearly effortless human read/writeability, but with severely limited representational possibilities. [7] On the opposite extreme, MusicXML and MEI can encode extremely complex, sophisticated score information, but are difficult to read/write directly. Humdrum achieves a balance between these extremes: though not quite as extendable as MEI, the humdrum syntax nonetheless affords a huge variety of approaches to encoding data. On the other hand, though `humdrum`'s top-down, tab-delineated format certainly makes editing slightly more cumbersome than editing abc or TinyNotation, most humdrum interpretations are comparably readable.

Like other easy notation schemes (abc, tinyNotation, LilyPond), humdrum interpretations often embed multiple pieces of information in each token. For instance, the `**kern` token `(4.aL/` encodes information about slurs (`(`), rhythm (`4.`), pitch (`a`), beaming (`L`), and stem direction (`/`). This "dense" approach allows a large window of musical time (for instance, several measures of multipart music) can be seen on a single screen. This contrasts with, for instance, MEI which spreads specific pieces of information across nested tags, making it impossible to see more than a few notes in a single screen. Such complicated tokens can also be written and edited rapidly, once you are familiar with the encoding. However, `kern`'s "dense" approach is but one option given humdrum's flexible syntax: information can be spread across multiple spines/columns (like *MuseData*) in whatever manner is most appropriate for data analysis. For example, the MCFlow corpus of rap transcriptions [6] encodes eight pieces of information across eight separate spines.

Human read/writeability is not just important to the process of data creation and curation, but is in fact essential to humdrum's entire methodological philosophy: humdrum emphasizes epistemological transparency by forcing users to engage directly with their symbolic data representations, even as they are filtered and transformed. Indeed, in traditional humdrum work flows, we apply repeated transformation to humdrum data tokens while maintaining, and visualizing the simple, readable humdrum syntax with each transformation. This dramatically improves the process of debugging and makes the series of steps from input to output clearer for novice users. Humdrum, thus, truly supports Cook's [7] call for a direct understanding of symbolic representations and processes. Consistent with this philosophy, `humdrumR` commands also reconstruct and display readable humdrum data even as the data is manipulated, maintaining a clarity and transparency which is easily lost when coding with complex data structures.

Though alternative data encodings have proven highly useful in contexts of research (*MuseData*, abc, MEI), file interchange (MusicXML), composition and engraving (TinyNotation, LilyPond, MEI), and performance (MIDI), the humdrum syntax offers an optimal combination of flexibility, read/writeability, and epistemological transparency, and is thus an ideal target for a new computational musicology toolkit. Fortunately, our focus on a single encoding scheme is not a limiting factor, as software to translate between `**kern` and most important representations—including MIDI, MusicXML, and MEI—is already available. In fact, fruitful cross-fertilization between musical data ecosystems is common: for instance, MEI's extraordinary Verovio score viewer has already been adapted as the Verovio Humdrum Viewer [8], making it easier than ever to visualize and edit humdrum data.

3. R

R [27] is a free, cross-platform, open-source "environment for statistical computing and graphics"—a domain specific programming language designed specifically for data analysis. R has a large ecosystem of data-analysis and statistics packages, most available through the Comprehensive R Archive Network (CRAN); As of now, the only programming environment with a comparable ecosystem for data analysis is Python. However, being less popular than Python for *general* programming, R's ecosystem is comparatively focused and uncluttered, with a higher quality floor. R's standard library ("base" R) itself contains analysis and visualization features which will satisfy the needs of many musicological research projects. Even when using external libraries, R users rarely encounter dependency issues—in fact, the process of installing R, `humdrumR`, and its dependencies is trivial on any operating system, even for beginner programmers, with no need for external package managers, virtual environments, "sandboxes," etc.

R excels at exploratory, ad hoc data analysis in short scripts or in the read-eval-print loop (REPL), making it easy to quickly manipulate, filter, and visualize data "on the fly." Indeed, a focus on simple scripting and "constrained" projects, without concern for more general software development issues, is core to R (and `humdrumR`) philosophy, making R an excellent avenue for learning programming purely for data analysis. This coding paradigm

[6] Conversely, humdrum/`**kern` is not as optimized for representing the details of music engraving as LilyPond, MEI, or MusicXML.

[7] However, `music21` includes an API for extending TinyNotation, useful to those with the requisite coding skills.

[8] http://verovio.humdrum.org/

is well suited to the constrained, stand-alone projects typical of theory-driven musicological research projects. The R ecosystem also includes a number of useful, free, software tools for enhancing the productivity, reproducibility, and presentation/sharing of research conducted in R. Notably, RStudio is a free, high quality integrated development environment for R. R is also one target of the Jupyter project [21], with RStudio too incorporating a suite of mark-up, notebook, and presentation tools. Finally, RStudio's `shiny` package [5] provides an easy means of creating interactive R data visualizations in the browser.

Though R is generally best suited to a combination of procedural and functional styles of programming, it nonetheless includes Object-Oriented Programming features suitable for defining simple classes. R's `S4` object system is oriented around multiple dispatch—generic functions can be defined which call specific methods based on the types of any or all of the function's arguments, not just their first argument. In languages featuring multiple dispatch (Julia, Common Lisp, Smalltalk, etc.), methods are not bound within classes. As a result, the object system operates in the background: novice users benefit from the features afforded by the object system—for instance, common functions like `summarize` can be applied to nearly any type of R object, getting useful results—without ever having to consciously engage with the system.

One of the core philosophies of R is "vectorization": treating data collections (especially vectors and arrays) as conceptually singular objects. `HumdrumR` leans into this philosophy, allowing users to think and operate on `humdrumR` data collections holistically. One concrete realization of this approach is that `humdrumR` users can completely avoid explicit iteration (i.e., loops): Iteration is abstracted by the `humdrumR` API, allowing users to decide solely *what* processes to apply to data tokens, not *how* to apply them to each token. `HumdrumR` defines a number of useful data classes, yet the R approach makes these classes an implementation detail that novice programmers need not understand.

3.1 Metaprogramming

The final key to the R ecosystem's concise data analysis syntax is its subtle use of metaprogramming [31, ch. 17–21]. In particular, numerous R packages use a shared domain specific language for specifying statistical models using R "formula" [31] objects.[9] Created using the ~ operator, R formulae capture ("quote") surrounding R expressions as well as their local namespace. R's metaprogramming features allow the programmatic manipulation of these formulae, for instance, using the `update` routine. Again, though metaprogramming is essential to R code, only advanced developers will ever need to explicitly engage with metaprogrammming concepts.

[9] For example, `Y ~ X*Z + (1|G)` describes a linear model predicting the variable `Y` using predictors `X` and `Z`, and the interaction between `X` and `Z`, with random-effect intercepts specified for each level of the grouping factor `G`.

4. `music21`

`Music21` is a Python library for symbolic music generation and analysis, with an extensive set of tools extending well beyond the capabilities of the humdrum toolkit, and which easily integrates with Python's extensive ecosystem (statistics and graphics libraries, etc.). Though `music21` and `humdrumR` overlap significantly in use case, `humdrumR` offers a fundamentally different coding philosophy and style.

Python syntax is famously simple to read/learn, yet the Pythonic coding style of `music21` nonetheless presents challenges to would-be computational musicologists. Working with `music21` requires one to engage directly with a hierarchy of complex classes (with *numerous* attributes and methods) and write many explicit control and looping structures, including (in typical analyses) multiple nested `for` loops. In contrast to humdrum, which relies on plain-text strings to encode information, `music21` parses musical scores into numerous complex data objects. Notably, `music21.Note` contains a rich set of attributes and methods for describing "notes." This complexity, though highly useful to experienced coders, is a barrier to entry for novices, for whom the practical reality of carrying out a computational musicology project is all but impossible. This is in part due to the style of `music21`'s User's guide, which necessarily gets bogged down explaining detailed functionality of how to represent, extract and manipulate low-level features (e.g., parts, notes, etc.) at the expense of explaining larger-scale processes like, for instance, how to search through a corpus of music to compare n-grams. Finally, `Music21`'s object hierarchy primarily represents musical score features—representations for arbitrary extra-musical data (e.g., dance steps, fingerings) or musical metadata (e.g., formal labels, manual annotations) is not supported.

`Music21`'s *Stream*-based object model is (purposely) extremely flexible [1]. As discussed above, the humdrum syntax includes some scope for flexible variation (using spine paths, for instance), yet `humdrumR`'s data backend (section 5.1) nonetheless always has the same structure; thus, one can always assume the same data structure and thus that certain commands/routines will always work. In contrast, `music21` users must always first determine the Stream-hierarchy of their data: are notes nested within measures, within parts, or within chords, etc? Similarly, `music21`'s highly sophisticated data classes (like `music21.Note`) are fairly inflexible, whereas humdrum's plain-text tokens are completely flexible. Again, this later approach is consistent with `humdrumR` philosophy, forcing users to think about and grapple with the transparent details of how their musical information is encoded and *not* about details of data structure.

As dynamically typed, interpreted languages, explicit loops in either R or Python are notoriously slow. However, whereas `music21` makes much explicit use of `for` loops, `humdrumR` enables users to exclusively use "vectorized" R and an optimized split-apply-combine backend, to achieve fast execution of most commands. As a result, humdrumR is generally much faster than music21.

5. HUMDRUMR

The humdrumR library defines a number of data object classes and a suite of functions for manipulating these classes. Most significant is the humdrumR class itself, which encapsulates a corpus of parsed humdrum files, and serves as the primary interface through which users interact with humdrum data. HumdrumR includes a complete humdrum syntax parser, which reads any valid humdrum data—including all valid spine paths—into a humdrumR data object. Invalid humdrum files are automatically flagged and skipped, with line-by-line syntax error reports generated on request. Consistent with the humdrumR philosophy, the syntax for reading files is concise and powerful: users simply specify one or more regular expressions matching files on their local disc. For instance, the command

```
readHumdrum("Bach/Chorales/chor.*krn")->chor
```

validates, reads, and parses all files matching the regular expression "chor.*krn" in the directory "Bach/Chorales" (370 files on our machine), wraps them in a humdrumR corpus object, and assigns this object to the variable chor. A set of summary functions are included to quickly describe the size, content, and structure of a loaded humdrumR data objects. In addition, an extensive suite of functions and classes for representing and manipulating pitch and rhythm data is also included—these tools reproduce much of the functionality of the original humdrum toolkit—as well as music21's core class hierarchy—, with some significant improvements and additions.

A great strength of the original humdrum toolkit is its use of the Bash | ("pipe") operator to concisely chain a series of operations—a syntax that is highly accessible to novice programmers. In recent years, the piping approach has become popular in R [3], especially magrittr's %>% pipe operator. HumdrumR too incorporates a pipe operator—%hum>%—which appears in all our subsequent code examples.

5.1 Data Model

R's primary native data structure is the tabular data.frame. HumdrumR utilizes a popular extension of the base R data.frame, the data.table[10]: an optimized data.frame which achieves database manipulation performance comparable to Python's Pandas module, including an extremely fast split-apply-combine routine. The HumdrumR class stores humdrum data in a list of data.tables, with each individual data token assigned to a single row. Data and metadata for each token are encoded in named columns of the data.table, called *fields*. The original string is encoded in the Token field, with global and local metadata associated with that token spread across other fields. Additional fields encode the "location" (which file, spine, path, record, etc.) of each token, encoding the structure of the original humdrum data so that it can be reconstructed for visual inspection after each modification. Users can freely create new fields; for instance, **kern tokens can be parsed into various pieces of information, each saved into a separate field: For example, the commands

```
chor %hum>% as.recip -> chor$Duration
chor %hum>% as.midi  -> chor$MIDI
```

create two new fields—Duration and MIDI—by applying the functions as.recip[11] and as.midi to the default Token field. These new fields can then be referenced like any other field in subsequent calls.

5.2 API

Much of R's expressive power arises from a subtle usage of metaprogramming to manipulate data.frames: Several base R functions—including subset, with, and within—allow users to input arbitrary R expressions which are then *evaluated within the data set* using the table's named fields as a local namespace. HumdrumR extends this paradigm to humdrum data, allowing users to apply arbitrary expressions to humdrum data stored in the underlying data.table back-end. Users capture expressions as R formulae and humdrumR API applies them to the data. These expressions can refer to any field in the data, including fields created by the user. The command

```
chor %hum>% ~table(MIDI[Duration == "4"])
```

(using the MIDI and Duration fields defined in the previous code block) extracts all MIDI values where the corresponding rhythm is a quarter note—using the standard R indexing ([]) operator—and then applies the base R table function, to tabulate these MIDI values. In a sense, this approach is an abstraction of function definition: One creates an expression—equivalent to the *body* of a function—but specifies no function arguments, as all data fields from the humdrum data are automatically passed into the expression if referenced.

The true power of the humdrumR arises through special *keyword* formulae which modify the API's behavior. Most notably, the by keyword can be used to split-apply-combine humdrum data. For instance, the command

```
chor %hum>% c(~table(MIDI[Duration == "4"]),
              by ~ File)
```

applies the exact same processing as the previous code block except the expression is applied separately to each file in the corpus, creating 370 separate tables. Since the humdrumR data fields include all data and metadata in the dataset, *any* field can be used to group data.

Other keyword formulae afford complex windowing, including n-grams, overlapping fixed-length windows, and various dynamic windowing possibilities. Finally, another set of keywords can be used to directly manipulate R's built-in visualization settings. Since formulae (including keyword formulae) are first-class objects in R, all of these expressions can be easily saved, composed, manipulated,

[10] https://cran.r-project.org/web/packages/data.table/index.html

[11] "Recip" is short for "reciprocal"—the humdrum term for standard Western duration categories (eighth-notes, sixteenth-notes, etc.).

and combined. For instance, we could save the tabling expression used above and use it in combination with various keyword formulae:

```
tabQuarters <- ~table(MIDI[Duration == "4"])
chor %hum>% c(tabQuarters, by ~ File)
chor %hum>% c(tabQuarters, by ~ Spine)
chor %hum>% c(tabQuarters, by ~ Clef)
```

The humdrumR API also includes routines for filtering and indexing a humdrum corpus. The standard R indexing operators ([] and [[]]) can be used to select pieces, records, or spines, either numerically or by matching regular expressions. More sophisticated filtering can be achieved through the use of humdrumR formulae: For instance, the command

```
filterHumdrum(chor,
              ~Token %~% '[EeAa]-',
              by ~ File)
```

selects all the files in the data set which contain the notes E♭ or A♭.

To bring it all together, a simple, yet complete humdrumR analysis script might look like this:

```
readHumdrum("Bach/Chorale/chor.*krn")->chor
chor %hum>% (~ as.midi(Token))
     %hum>% (~ Pipe - 60)
     %hum>% c(doplot ~hist(Pipe,
                           title = Clef,
                           xlim = c(30, 80)),
              by ~ Clef,
              mfcol ~ c(2,2))
```

These commands load the Bach chorale dataset, convert the original data tokens to MIDI numbers, subtract these numbers by 60 (to center them on middle C), then create a separate histogram for each clef in the data (in a 2x2 grid[12]). Note the use of the base R hist (histogram) function, including the use of the title and xlim (x-axis limits) arguments to give each plot a meaningful title and place all plots on the same scale.

5.3 Developer Tools

HumdrumR is designed to be highly extensible. Even novice users can quickly begin to create and save their own routines as R functions, or simpler yet, as combinations of humdrumR formulae. However, humdrumR also includes several features to support development by more sophisticated coders. All humdrumR data classes are intended to serve as extensible bases for further development—for instance, developers might choose to implement counterpoint analysis algorithms using humdrumR's basic tonalInterval class and its wealth of useful methods (transposition, inversion, etc.). However, the most significant tool for developers is humdrumR's Regular-Expression Method Dispatch System (REMDS). Interacting with humdrum data requires extensive string manipulation, typically using regular expressions, as one works to extract the information one is interested in from humdrum's "dense" character tokens. Using the REMDS, developers need only define normal R functions to manipulate the information they are concerned with and regular expressions to match that information. The REMDS can then be used to create generic functions which read an input string and dispatch the appropriate method based on matching regular expressions—what's more, these methods can (optionally) be applied "in place," only affecting the substring which matches the desired regular expression. For example, the humdrumR pitch module defines a number of functions which translate specific pitch encodings (note names, solfege, intervals, etc.) to/from humdrumR's common tonalInterval pitch representation. The REMDS is then used to generate generic pitch translation functions which call the desired method when a specific regular expression is matched. For instance, the function as.midi can be applied to strings containing a variety of pitch representations, even when embedded with other information (i.e., rhythm, beaming):

```
as.midi("4.cc#J") # "cc#" => **kern  =>  73
as.midi("4.C#5J") # "C#5" => **pitch =>  73
as.midi("4.-M9J") # "M9"  => **mint  => -14
as.midi("4.soJ")  # "so"  => **solfa =>   7
```

This approach allows users to use humdrumR functions without having to explicitly manipulate strings or use regular expressions, one of the major barriers to learning in the original humdrum toolkit. Many of humdrumR's own functions (like as.midi) are written using the REMDS, and developers can utilize it to significantly reduce coding effort when defining new functions.

6. FUTURE

The package source of HumdrumR 0.3.0 is currently available on github (natsguitar/humdrumR); when development solidifies, version 1.0.0 will be made available on CRAN under the terms of the GNU General Public License. However, releasing code is not enough to support humanist scholars interested in coding—it is imperative to provide high quality documentation and learning materials in a style that is digestible for users who may be new to computer programming. The most important contribution of the original humdrum project was neither the syntax nor the toolkit, but Huron's extensive user guide [17].[13] The Humdrum User Guide offers a gentle introduction to empirical/computational research from a humanistic perspective, walking readers through the practical and philosophical details and challenges of digital humanities work and the conceptual transformations necessary to convert humanistic thought into concrete code, using examples from real musicology projects. HumdrumR too will ultimately be accompanied by a *humdrumR User Guide*, including interactive online content. Our goal is to not just teach the mechanics of operating software in a friendly, hands-on format, but also the conceptual framework needed to think about music as data, introducing key scientific principles/methods (data sampling, statistics, hypotheses, etc.) while maintaining a holistic, humanistic perspective.

[12] The keyword mfcol is a base R graphics parameter which controls the layout of plots in a grid.

[13] http://www.humdrum.org/guide/

7. REFERENCES

[1] Christopher Ariza and Michael Scott Cuthbert. The music21 Stream: A New Object Model for Representing, Filtering, and Transforming Symbolic Musical Structures. In *Proceedings of the International Computer Music Conference*. Citeseer, 2011.

[2] Claire Arthur. Taking harmony into account. *Music Perception: An Interdisciplinary Journal*, 34(4):405–423, 2017.

[3] Stefan Milton Bache and Hadley Wickham. *magrittr: A Forward-Pipe Operator for R*, 2014. R package version 1.5.

[4] John Ashley Burgoyne, John Wild, and Ichiro Fujinaga. An Expert Ground Truth Set for Audio Chord Recognition and Music Analysis. In A Klapuri and C. Leider, editors, *Proceedings of the 12th International Society for Music Information Retrieval (ISMIR) Conference*, pages 633–638, Miami, FL, 2011.

[5] Winston Chang, Joe Cheng, JJ Allaire, Yihui Xie, and Jonathan McPherson. *shiny: Web Application Framework for R*, 2018. R package version 1.2.0.

[6] Nathaniel Condit-Schultz. The Musical Corpus of Flow: A digital corpus of rap transcriptions. *Empirical Musicology Review*, 11(2):124–147, 2016.

[7] Nicholas Cook. Towards the Complete Musicologist? Invited talk at ISMIR, 2005.

[8] Tim Crawford and David Lewis. Early modern cover song detection: finding concordances in mixed-notation corpora of early music. In *Music Encoding Conference, Tours France*, 2017.

[9] Michael Scott Cuthbert. music21: User's Guide, 2011.

[10] Michael Scott Cuthbert and Christopher Ariza. music21: A toolkit for computer-aided musicology and symbolic music data. In *Proceedings of the International Society of Music Information Retrieval*. Proceedings of the International Society for Music Information Retrieval, 2010.

[11] Trevor de Clercq and David Temperley. A Corpus Analysis of Rock Harmony. *Popular Music*, 30(01):47–70, January 2011.

[12] Johanna Devaney, Claire Arthur, Nathaniel Condit-Schultz, and Kirsten Nisula. Theme and Variation Encodings with Roman Numerals: A New Data Set for Symbolic Music Analysis. In Meinard Müller and Frans Wiering, editors, *Proceedings of the 16th International Society for Music Information Retrieval (ISMIR) Conference*, pages 728–734, Malaga, Spain, 2015.

[13] Michael Good. MusicXML: An internet-friendly format for sheet music. In *XML Conference and Expo*, pages 03–04, 2001.

[14] Andrew Hankinson, Perry Roland, and Ichiro Fujinaga. The music encoding initiative as a document-encoding framework. In *Proceedings of the International Society for Music Information Retrieval*, pages 293–298, 2011.

[15] Walter B. Hewlett. *MuseData: Multipurpose Representation*, chapter 27, pages 404–407. MIT Press, 1997.

[16] David Huron. Note-Onset Asynchrony in J. S. Bach's Two-Part Inventions. *Music Perception*, 10(4):435–443, July 1993.

[17] David Huron. *Music Research Using Humdrum: A User's Guide*. Stanford, California: Center for Computer Assisted Research in the Humanities, 1999.

[18] David Huron. Tone and Voice: A Derivation of the Rules of Voice-Leading from Perceptual Principles. *Music Perception*, 19(1):1–64, September 2001.

[19] David Huron and Deborah A. Fantini. The Avoidance of Inner-Voice Entries: Perceptual Evidence and Musical Practice. *Music Perception*, 7(1):43–47, October 1989.

[20] Nori Jacoby, Naftali Tishby, and Dmitri Tymoczko. An Information Theoretic Approach to Chord Categorization and Functional Harmony. *Journal of New Music Research*, 44(3):219–244, 2015.

[21] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. Jupyter notebooks-a publishing format for reproducible computational workflows. In *ELPUB*, pages 87–90, 2016.

[22] David Lewis, Tim Crawford, and Daniel Müllensiefen. Instrumental idiom in the 16th century: Embellishment patterns in arrangements of vocal music. In *Proceedings of the International Society for Music Information Retrieval*, 2016.

[23] Kristen Masada and Razvan Bunescu. Chord Recognition in Symbolic Music Using Semi-Markov Conditional Random Fields. In *Proceedings of the 18th International Society for Music Information Retrieval Conference*, pages 272–278, 2017.

[24] Eric Nichols, Dan Morris, Sumit Basu, and Christopher Raphael. Relationships Between Lyrics and Melody in Popular Music. In *Proceedings of the 10th International Society for Music Information Retrieval (ISMIR) Conference*, pages 471–476. ISMIR, 2009.

[25] Han-Wen Nienhuys and Jan Nieuwenhuizen. LilyPond, a system for automated music engraving. In *Proceedings of the XIV Colloquium on Musical Informatics (XIV CIM 2003)*, volume 1, pages 167–171, 2003.

[26] A. D. Patel and J. R. Daniele. Stress-Timed vs. Syllable-Timed Music? A Comment on Huron and Ollen (2003). *Music Perception: An Interdisciplinary Journal*, 21(2):273–276, 2003.

[27] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2018.

[28] David Temperley and T. de Clercq. Statistical Analysis of Harmony and Melody in Rock Music. *Journal of New Music Research*, 42(3):187–204, 2013.

[29] Paul Von Hippel and David Huron. Why Do Skips Precede Reversals? The Effect of Tessitura on Melodic Structure. *Music Perception*, 18(1):59–85, October 2000.

[30] Chris Walshaw. *ABC2MTEX: An easy way of transcribing folk and traditional music, Version 1.0*. University of Greenwich, London, 1993.

[31] Hadley Wickham. *Advanced R*. Chapman and Hall, 2 edition, 2019.

[32] Frans Wiering and Emmanouil Benetos. Digital Musicology and MIR: Papers, Projects and Challenges. In *Proceedings of the International Society for Music Information Retrieval*, 2013.