ChoraleAnalyzer

Development of a Simple Implementation into an Effective Pedagogical and Musicological Tool

Charles Peyser
Advisor: Dr. Robert Dondero

Table of Contents

Charles Peyser

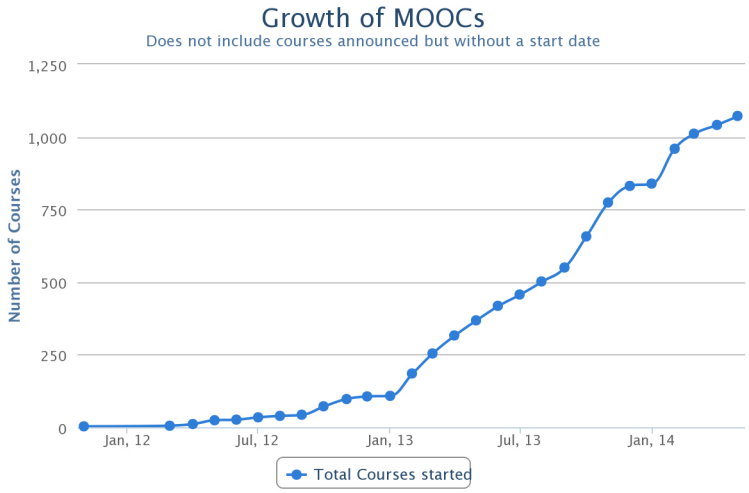
Spring 2014 Independent Work
Advisor: Dr. Robert Dondero
Submitted: May 6th, 2014

Note to the Reader

This project is largely one in online musical education, and was entirely made possible by a prior semester of work given to solving the problem of automatic chorale verification and grading. As such, it is difficult to discuss the project's goals and results without heavy reference to the original ChoraleAnalyzer tool, its design, and its capabilities. Except for the reworking of the tool to be faster and more adaptable and the addition of a substantial new analysis feature, however, development of the original ChoraleAnalyzer was the work of last semester. For reference as to the specifics of the tool, and for a primer on the relevant topics in music theory, I invite the reader's attention to my first Junior Paper: "ChoraleAnalyzer: An Automatic Tool for the Verification of Bach-Style Chorales". This paper will seek to restrict itself to the progress made in the spring of 2014.

Introduction

At this point in the progression of education technology it is difficult to dispute the claim that large, automated online courses will play a substantial role in the way that we will educate in the 21st century. Since the first MOOC, or Massive Open Online Course, was launched in 2008 at the University of Manitoba, Canada (*Marques*), the model has grown to encompass over 200 universities, 1300 instructors, and 10 million students. In fact, since the beginning of 2013 the number of courses offered online has exploded, from about 100 at the start of the year to more than 1200 today (*Shah*).



MOOCs are growing at an increasing rate (*Shah*)

Online education has become popular largely because it works from a business perspective. While an online course might be expensive to create, sometimes costing over $100,000 to pay for the same kind of production value that might be expected from a television series (*Goldstein*), the incremental cost of adding another user is practically nothing, making MOOCs difficult to compete with in an environment where anyone can log in and learn. MOOCs also open up markets that do not exist for the traditional pedagogical model;

working professionals, parents, and educators can participate in an online course more easily than they can attend an actual lecture (*Swope*).

The first MOOCs were largely technical (*Marques*), and in fact, programming and other engineering topics dominated online course offerings in their earliest years. Recently, however, subjects in the humanities have overtaken other fields to become the largest category of online course offered by major providers (*Shah*). The challenge with the dominance of this sort of class is grading – as essays and projects take over for computation and multiple-choice exams, evaluation becomes more subjective and requires more human attention (*Duhring*). Many online educators address this problem by employing peer grading, a process by which students read and assign grades to each other's work. This system, however, suffers from the assumption that students are qualified to grade an assignment, and that they will read and understand grading directions given by professors with little incentive (*Rees*).

One field for which the grading barrier has played a substantial role in slowing the progress of online education is music. Evaluation in music history and music theory often is difficult in a multiple-choice or essay format, making it hard to grade students in an online course (*Covach*). The beginner music theory curriculum stands at a peculiar place between the technical and aesthetic, teaching composition through both analytical and artistic procedure. As such, grading is hard. A composition must follow the rules of the particular style in question, and it falls outside the ability of both the average student and a simple program to grade a general submission.

This paper, and the tool that it describes, deals with one style in particular that is covered in introductory and intermediate music theory classes – the Bach chorale style. The style encompasses both the rules of melody and of harmony that are typically put forward in such a class, and thus, a composition assignment in the chorale style often acts as a capstone project. The Bach chorale style is characterized by a large number of melodic and harmonic "rules" that govern how notes and chords ought to, but do not necessarily have to, relate to each other. Following these rules generally produces a stylistic and artful composition. Bach himself, however, often violated the rules of the style in compliance with some aesthetic motivation; thus, evaluating a composition in the Bach style is an exercise in validating general compliance with the rules while finding justification for the points at which the rules are ignored.

Needless to say, a problem as complex as Bach-style analysis thwarts basic peer-grading schemes. A Bach chorale is different from an essay in that a non-expert cannot reliably evaluate the chorale. The mechanisms by which it operates are simply too technical to be checked by students but too non-standard to be easily handled by an automatic grader. Grading Bach-style chorales has become a notoriously tedious job delegated to graduate students and teacher's assistants, and stands in the way of a full migration of the standard music theory curriculum online.

This paper discusses an advanced tool for the evaluation of Bach-style chorales, the chord progression-based mechanism it employs to take rule violations in context of where they

occur, and the use of the tool through an online interface that provides pedagogical feedback to instructors.

Prior Work

Music21

ChoraleAnalyzer, in both its original form and the form presented in this paper, is built on top of a set of Python scripts called music21. Created and maintained by Dr. Michael Cuthbert and his team at MIT, music21 is self-described as "a toolkit for computer-aided musicology". It contains methods for parsing musical scores from various formats into an internally-defined set of Python objects, and conducting analysis on those objects (*music21*). While music21 featured prominently in the improvements made to the original ChoraleAnalyzer as part of this project, it is more relevant to the original construction of the tool, and is thus only referred to as necessary in this paper.

The Original ChoraleAnalyzer

The original ChoraleAnalyzer was a large group of Python scripts sitting on top of music21. The program repeatedly scans through a chorale and accompanying Roman numeral analysis for rule violations of various types. I created the tool in the fall of 2013, and at the time of completion it was able to reliably discover the following errors in a chorale:

- Parallel intervals, including parallel octaves, fifths, and unisons. Hidden parallels.

- Voice leading errors, including tritone intervals in the same part and repeated bass tones.
- Single-chord errors, including parts out of standard range, intervals between parts that are too large, and voice crossings.
- Harmonic violations, including incorrect modalities, inversions, and non-standard progressions.
- Discrepancies between the provided chorale and Roman numeral analysis.

The tool also supported a basic grading metric called the "Bachness" score, which compared the number of violations per measure of a chorale in each rule type to data collected from a sample of original Bach chorales, and gave a score based on conformity. The tool, at the time, did not adjust the score based on the violation in context within the chorale.

The original tool ran from the command line of a machine that had music21 installed. It ran extremely slowly, as the mechanisms for finding parallel intervals were taken from the music21 libraries and were ill suited for the task. A single chorale could take longer than twenty minutes to process.

A Faster, More Comprehensive Backend

Parallel Interval Validation

Development of the original ChoraleAnalyzer program into a viable pedagogical tool required a number of changes from the implementation completed in the fall. In particular,

parallel interval validation in the original ChoraleAnalyzer derived from music21's built in `theoryAnalysis.theoryAnalyzer.getParallelFifths()` method, which is extraordinarily inefficient and slowed execution on chorales of average size (20-30 measures) considerably. It became obvious that in order to make the tool public and to handle a large number of submitted chorales, some alternative, faster mechanism would have to be devised.

The solution that was devised and eventually implemented relied on a technique that a human grader might use to find parallel intervals. Each pair of parts is inspected individually in a horizontal manner, looking for rhythmic points at which the parts sound simultaneously and checking the next note in both parts for illegal parallel intervals. In order to implement this mechanism, two pieces of data are first gleaned from the chorale: a list of offsets and an offset dictionary. The list of offsets, just called `offsets` in the code, is a Python list of measure/beat elements that give all of the rhythmic points at which a note begins in the chorale. The measure/beat elements are represented as a Python float in the form `measure.beat`, the assumption being that no measure will have more than ten beats. The offset dictionary, called `offsetDict`, maps offsets in `measure.beat` form to music21 `Note` objects. Between the two simple data structures, a chorale can be traversed horizontally and intervals can be computed.

For each pair of parts (there are six unique pairs for four parts), the two offset lists are searched for matches, where a match represents a point at which the parts both sound, and a candidate for the first pair of tones in an illegal parallel interval. When a match is found,

the forward interval, that is, the interval between the tone and the one that immediately follows it, is computed for both voices and compared. If the intervals are found to be the same and of illegal size, the violation is flagged and a message is passed up the logic hierarchy to the frontend. Otherwise, the match is discarded and the process proceeds onward.

As can be seen in the profiling results below, the new offset-based implementation vastly outpaces the original music21 implementation, offering a feasible alternative to a mechanism that was slowing execution down to an intolerable level. Profiling of the music21 implementation using the Python cProfile module reveals that the speed problems in the `theoryAnalysis.theoryAnalyzer.getParallelFifths()` method lie deep in the music21 codebase. Further investigation would be required in order to determine what in particular causes the method to take so long. The music21 team has been made aware of the issue. Hopefully, music21 will adopt my implementation over that which is currently in its codebase.

Chorale Number	Music21 Implementation	New Implementation
1	9m 34.577s	4.919s
2	8m 8.663s	3.425s
3	4m 46.927s	2.573s
4	5m 5.516s	2.416s
5	27m 14.560s	6.186s
6	2m 6.132s	1.672s

It should be noted that the sets of parallel intervals identified by the two implementations differ meaningfully. In particular, the music21 implementation identifies a number of parallel intervals that the offset-based implementation does not, but these violations do not appear to follow the standard definition of parallel intervals. Further inquiry into the mechanism behind the music21 implementation is required to determine what exactly it searches for, but for the purposes of this project the new implementation appears sufficiently thorough.

Progression Tracking

One of the problems with the original ChoraleAnalyzer is that flagged errors without any notion of context: a student could know how her error profile compares to Bach quantitatively, but for any given violation could not know how likely Bach would be to ignore that rule in that place. This, of course, is the fundamental problem with automatic

Bach-style grading: the rules in question are more like guidelines than anything else in particular, and seeing violations in a black-and-white way limits perspective on the chorale.

This iteration of ChoraleAnalyzer takes a tentative step towards tackling the problem of treating violations by their context by tracking progressions of chords together with violations, and associating violations with the chord progressions in which they occur. The algorithm and implementation for gathering n-gram chord progressions from a chorale in XML format arises entirely from the work of Dr. Dmitri Tymoczko of the Princeton Department of Music, who has produced a substantial library of code for parsing, analyzing, and extracting data from compositions of various types.

Dr. Tymoczko's code was modified so as not to return Python strings representing chord progressions, but rather `progressionTracker` objects, defined in `utilities.py` and capable of storing a chord progression together with incident Bach-style violations. A `progressionTracker` can be described well by analogy: a `progressionTracker` instance is to a chord progression of variable length as an `errorTracker` instance is to an entire chorale. A module called `gatherProgs.py` was composed that adapted Dr. Tymoczko's original progression gathering code. A second module called `testBachProgs.py` was composed that contains methods for processing long lists of `progressionTracker` instances into simpler, non-redundant lists of `progressionTrackers` that together provide an in-context error profile for a chorale, and converting that list into a Python dictionary that maps progressions to a list giving the frequency that each category of violation is incident to that progression.

More specifically, the method `combineProgs()` takes the raw output of the `gatherProgs.py` process, that is, a Python list of `progressionTracker` objects corresponding to every progression of chords in a chorale, where a progression is defined as a sequence of consecutive chords of size five or smaller, as represented in the Roman numeral analysis. These `progressionTrackers` represent *specific instances* of the progression and the errors that occur within. The method finds duplicate `progressionTrackers`, which represent the same progression occurring at multiple points in the chorale, and sum their associated error sets. The next step of processing, `createProgDict()`, accepts that list of unique `progressionTrackers` and returns a dictionary that maps string representations of progressions to lists representing error profiles. Information as to the relative occurrence of rule violations in various progressions is then readily accessible for any chorale, or set of chorales.

As alluded to above, this opens the door to addressing the problem of the subjective nature of the Bach rules. By considering a violation together with its progression context, a new, better "Bachness" metric is possible. The fundamental mechanism behind the "Context Aware Bachness" grading scheme implemented in the website is the "discounting" of error values based on their occurrences in the Bach chorales themselves. The procedure involves tallying up the errors that occur in the student chorale, but scaling down the contribution to the tally if Bach made the same mistake. The scaling system is simple: an error's contribution to Context Aware Bachness is one divided by the number of occurrences in Bach (unless, of course, the error never occurs in Bach, in which case it is divided by one).

It is worth noting that the data gathered in order to support this computation, that is, the progression to error profile dictionary on Bach himself, is itself an interesting and novel dataset. For example, it seems that a large number of parallel fifths in Bach occur over a I chord. The data is saved as a Python pickle file[1], has been published as a Git repo, and will be made available to the appropriate faculty in the music department.

The data set itself is too large to be fully described here. It contains the error profiles of 20344 distinct progressions. Interesting facts can be gleaned even from cursory perusing of the data. For example, the tenor goes out of range seven times over viio6 -> I6 -> V progressions, while only going out of range fourteen times over all progressions. Consider the below:

[1] "Pickle" is a built-in Python serialization mechanism. That is, a Python class hierarchy can be encoded into pickle format, saved to a file, and retrieved by another Python process at a later time. Pickle is comparable to JSON, in that it is used to pass data structures from program to program.

Error Type	Most Violating Progression in Bach
Parallel Unison	(Does not occur in Bach)
Parallel Fifth	V -> V7 -> I
Parallel Octave	I6 -> I -> IV
Bass Out of Range	V -> V7 -> I
Tenor Out of Range	viio6 -> I6 -> V
Alto Out of Range	viio -> Imaj7 -> ii2
Soprano Out of Range	(Does not occur in Bach)
Alto/Soprano Interval	I -> IVmaj6/5 -> V6/5
Tenor/Alto Interval	ii -> viio6 -> I6
Bass/Tenor Interval	V -> V7 -> I
Alto/Soprano Voice Crossing	V -> V2 -> i6
Tenor/Alto Voice Crossing	viio6 -> I6 -> V
Bass/Tenor Voice Crossing	V -> V7 -> I
Tritone Leap	iv6 -> V -> i
Repeated Bass Tone	V -> V7 -> I

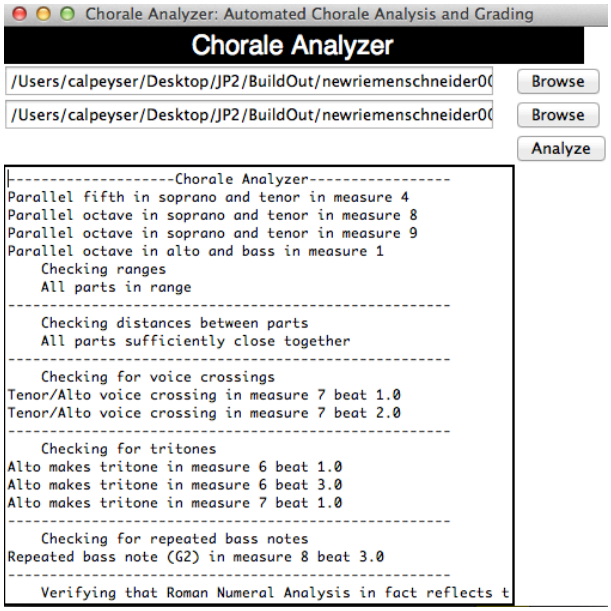
A Clearer, More Usable Frontend

One of the biggest problems with the original ChoraleAnalyzer addressed by this project is ease-of-use. The tool was powerful and adaptive but yet entirely unusable by a theorist or

instructor without background in technology. Use of the tool requires the user to install an open source library into her Python path and run the program using the Python interpreter. This was unrealistic, seeing as most of tool's audience is unlikely to know its way around a command line!

A Simple GUI

The first solution attempted to the ease-of-use problem was to build a simple graphical user interface that a user could access by double clicking an icon on her desktop. The user would then load chorales from a dropdown menu, hit an "analyze" button, and see results in an expandable text box. The user could then copy the results from the application to some other location and proceed to analyze as many chorales as desired.



An attempted GUI for ChoraleAnalyzer

An attempt was made to realize this method of interaction in a GUI built with Tkinter. The interface proved easy enough to implement – the problem arose in packaging of the application to be used on other computers. ChoraleAnalyzer relies on having access to a Python installation equipped with music21. The application crashed immediately for

machines that did not have the libraries installed. Thought was given to producing an installer that would find music21, install it, and place it in the appropriate path. Lack of knowledge of the user's operating system and specific installation of Python made such an installer seem to be a risky and unattractive option, so the GUI was abandoned.

An Online Tool - CGI

Uploading ChoraleAnalyzer to the Internet and producing a visual interface on a webpage solved the problems of the simple GUI. The specific Python libraries necessary for the program to run could simply be installed on a virtual machine provided by an online vendor, and that machine could act as the web server that would process chorales coming from the user. The operating system and Python installation of the user would be irrelevant – access would be through a browser.

The fundamental issue anticipated with building a website that runs ChoraleAnalyzer was transferring data from the user to a Python program. For this reason, the original effort to build the webpage was made in CGI, using Python to print HTML pages to the user and to process form entries that would contain the files to be analyzed. Development was begun using an XAMPP testing server running on a single machine, and a simple webpage was successfully designed that passed XML and text files to the program to be processed and returned.

It became clear during development, however, that simply to pass chorales into and out of a webpage without maintaining any state was to sell the potential of the tool short. If some

basic data about the submitted chorales could be remembered, then meaningful information as to the average error profile over a number of chorales could be generated, which could have important theoretical and pedagogical implications. In particular, if the application could inform an instructor that some particular class of errors of some particular set of progressions was causing problems for an unusually high proportion of students, that instructor might be able to save tremendous time and effort in communicating correct practice to his students. The added functionality could also be used by students themselves to learn which progressions are the most troubling.

The CGI model ultimately did not suit the integration of a database that could store meaningful user data. In particular, communication between the Python interpreter, which is a 64-bit application, and a MySQL database, which runs as a 32-bit application, stopped progress. While there certainly exist ways to use a database in a CGI context, the integration of a unique Python installation caused complications that, at first glance, were unable to be resolved by several experts. Both my advisor and I agreed that the website was entirely viable as a pure Python application, were more time to be put into solving the database issue. However, in the interest of moving forward with the project in the short time frame of a single semester, I abandoned the CGI model in favor of a more traditional website, built in HTML, PHP, and JavaScript.

A Website in PHP

While PHP was originally avoided due to its instability as a language and perceived difficulty of integration with Python programs, the traditional model was eventually successful in supporting the application in the anticipated manner. A website was built that provided a simple homepage from which a user could upload a chorale and Roman numeral analysis to be analyzed, and have the results returned in a simple textual format. The user could also access an "About" page with details on use and appropriate references, and a contact form to submit bug reports. However, if the user wants to maintain information regarding submitted chorales, she can create a user profile by submitting a username and password. If that username has yet to be taken, an account is created and the user is taken a "profile" screen.

On the "profile" page, the user again has the opportunity to upload a chorale and roman numeral analysis for error validation. However, unlike on homepage, a chorale submitted while logged in is associated with the user's profile and stored. The user can then access any of those chorales at any other time, and select any subset of them to "queue for analysis". The user can then analyze that subset, and be directed to a page providing comprehensive information on the error profile of those chorales taken together as a group. In particular, a list of average violations per chorale is given in each error type, as well as a printout of progressions that demonstrate a particular propensity for errors of a particular type.

Frontend – Design

The website is implemented on an Apache 2 server running on an Amazon EC2 instance located in northern Virginia. The server runs Ubuntu with a typical LAMP setup, including a MySQL database. The domain name "Choraleanalyzer.com" was purchased and connected via an A request to the elastic IP `54.85.217.150`.

Development Environment

As described above, development occurred on an XAMPP local server that simply served HTML and PHP files to "localhost" to be viewed in a browser. Migration occurred using a shared Git repository. Only one branch was needed, seeing as the only changes to be made to the deployment server were made on the development server. In order to allow the page to operate on both servers, a global variable named `$homepage` was maintained in each PHP file.

	Development Server	Deployment Server
Value of `$homepage`	`127.0.0.1`	`Choraleanalyzer.com`

Redirections could occur by simple reference to `$homepage`, as in:

```
header("Location: http://$homepage/ChoraleAnalyzer/somepage.php");
```

All files other than the landing page `index.php` are stored in a directory called ChoraleAnalyzer, allowing for easy migration using Git. The landing page itself was written separately for the two servers.

MySQL Database

The website relies on two MySQL tables in order to store state relevant to user profiles[2].

TABLE USERS	Field	Type
	`username`[3]	`varchar(255)`
	`password`	`varchar(255)`

As suggested by the names, this table stores usernames and passwords for users. It is populated by users when creating new profiles.

[2] The details as to column names and types are *slightly* different in the server implementation. Those details are ignored here in order to present the database in an intuitive manner.

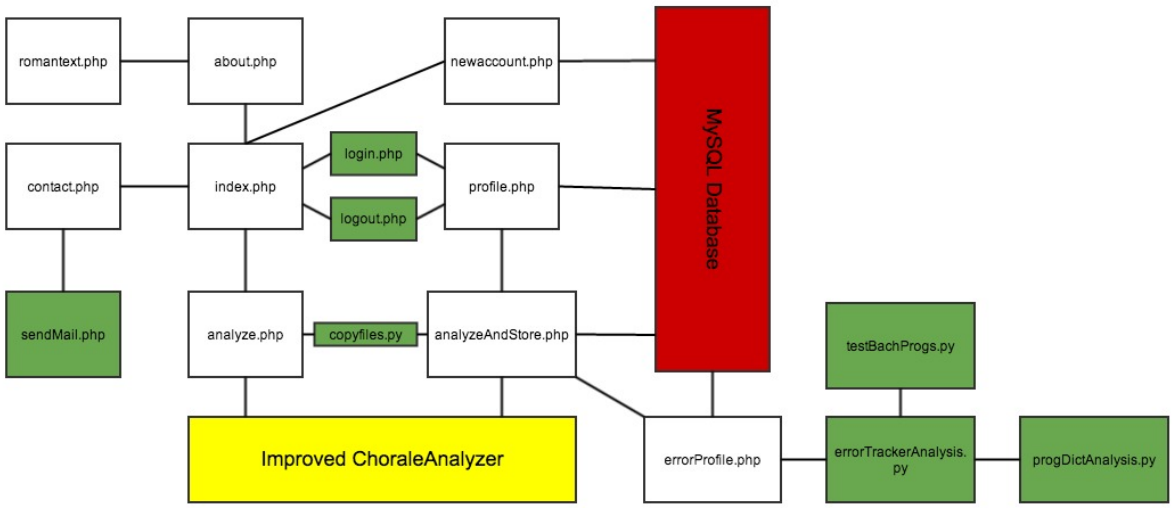
[3] The "username" entries in the two tables represent the same thing: a username associated with a profile. They are not, however, implemented as SQL foreign keys. That is, the two references contain the same information, but do not occupy the same memory.

TABLE CHORALES	Field	Type
	name	varchar(255)
	username	varchar(255)
	xml	blob
	rna	blob
	errorTracker	blob
	errTrackerNum	mediumtext

Each entry of this table corresponds to a chorale submitted by a registered user. The chorale's name is stored in order to provide some representation to give to the user when selecting subsets, and the username itself is stored so that the appropriate set of chorales can be shown. The other three entries are "blobs", or Binary Large Objects, that encompass the pre- and post-analysis states of the chorales. The first two blob entries, "xml" and "rna", are the chorale submissions in XML format and the Roman numeral analysis submission in romantext format, respectively. They allow for storage of the chorales in case data is ever required for an inquiry into student submitted chorales – presumably over time the database will grow to store a wealth of such information. The third blob entry, "errorTracker", stores the Python `errorTracker` object that contains the post-analysis error profile of the chorale. The final entry, "`errTrackerNum`", maintains a number that corresponds to the `errorTracker` on the server, to facilitate passing the object to Python without dealing with serialization.

Program Architecture

The ChoraleAnalyzer webpage consists of a number of PHP files that pass data to each other via HTTP POST requests. The following diagram illustrates the flow of data through the application, where green boxes represent scripts that contain only logic, and are thus never seen.



It will likely be most effective to address each component of the application in turn, in order to communicate the design of the program in a holistic manner.

- Homepage (index.php): `index.php` provides the landing site for the page, and fundamentally provides several points of entry into the site. A horizontal navigation bar, which persists throughout the site, points back at the homepage and to an about page, `about.php`. From the homepage, a user can also access the account creation page, a contact form, and the improved ChoraleAnalyzer itself for stateless analysis

(the chorale is analyzed and a report is returned, but the data is not stored). Finally, a user can login thorough a form at the top of the page to access her own page.

- About (`about.php, romantext.php`): `about.php` is a static HTML page that provides basic information about the tool, how it works, the nature of the input files it requires, and a the Bachness scoring scheme. It links to `romantext.php`, which is another static page that describes the romantext format for Roman numeral analysis that is requisite to use of the tool. The brief tutorial is taken from Appendix B of my paper on the original ChoraleAnalyzer itself (see above, *Note to the Reader*).
- Contact (`contact.php, sendMail.php`): `contact.php` is a simple module that displays a form allowing the user to input a message containing a comment, question, or bug report. Form submissions are handled by `sendEmail.php`, which, as the name suggests, sends an email. The code is linked to the author's personal email address.
- Account Creation (`newaccount.php`): The account creation screen is immediately accessible from the homepage and exposed the user to a form that allows him to register a new username and password. Upon validating that the form is appropriately filled, the code opens a connection to the `Users` table in the MySQL database. A list of all usernames is produced and compared against the candidature account to ensure that usernames remain unique; if a duplicate is found, the form is rendered again with a message to user asking for a different username. Otherwise, the username and password are entered into the table.

- Login (`login.php`): The login mechanism, expressed in `login.php`, makes use of the PHP `session` mechanism. When a username and password are submitted to the login form in the homepage, a connection is made to the `Users` table in the MySQL database. Once the user has been authenticated, a `session` is started. In PHP, a `session` is a global associative array that maintains information assigned to it from the time that `session_start()` is called until `session_end()` is called. The username in question is loaded into the `session`, and `profile.php` is rendered[4].
- Logout (`logout.php`): From the profile page, a user can hit a button in the upper right-hand corner of the screen to logout and return to the homepage. This action, which essentially consists of a call to `session_end()`, is handled in `logout.php`.
- Profile (`profile.php`): Once logged in, the user is presented with an interface similar to the homepage in that it offers access to the improved ChoraleAnalyzer through file uploads. When files are submitted in this page, however, Chorales are stored in the manner described above (see the *MySQL Database* section), and presented to the user in a list. The list is populated by a database call which searches all chorales for those keyed to the username in question. Selecting any subset of the stored chorales and clicking a "Queue for Analysis" button activates the `queue_chorales()` JavaScript function, which copies the selected chorale references to another list.

[4] The PHP session object works by storing cookies in the user's browser. Thus, cookies must be enabled for the site to function properly.

- Error Profile (`errorProfile.php, errorTrackerAnalysis.php, testBachProgs.php, progDictAnalysis.php`): From the final list on the profile page, the user can activate a set of scripts that gather a holistic error profile for the subset in question, and print it to the user. Control is passed to `errorProfile.php`, which validates the form and retrieves references to the `errorTracker` instances in question. The data is passed to `errorTrackerAnalysis.py`, which defines a Python object called a `groupTracker`, which is meant to be analogous to an `errorTracker` but to maintain state for a number of chorales. The `groupTracker` inherits from `errorTracker`, and thus maintains counts for each type of error and a list of `progressionTrackers`. A `groupTracker` is instantiated by passing in a list of `errorTracker` instances, which have their respective error counts summed. The `groupTracker` has only one method – a printing mechanism that computes average violations per chorale, and that calls upon the progression tracking modules to produce a summary of the worst progressions for each error type.
- Analysis without storage (`analyze.php`): If a chorale and a Roman numeral analysis are submitted from the homepage, the form contents are processed by `analyze.php`. The original design of this module simply copied these files to a temporary location, processed them with the improved ChoraleAnalyzer through the PHP `exec()` function, and printed the resulting list of violations and Bachness score. Two issues were encountered:
    - File copying without a lock: It proved difficult to copy the contents of the files submitted in the homepage to a temporary location without forking a process.

That is, a new process would be automatically spawned by PHP to handle to copying job, and control would pass to the improved ChoraleAnalyzer before the copy was complete. This problem was solved by delegating the copying job to Python instead of PHP: a simple script called `copyFiles.py` that takes filenames as arguments and copies them to a known location was composed and called with the `exec()` function. PHP waits for `copyFiles.py` to complete before moving to the improved ChoraleAnalyzer. That is, the problematic mechanism is removed by replacing PHP file handling with Python file handling.

- Concurrency: Storing the contents of the chorale and Roman numeral analysis in a file with a fixed name leads to problems if two users simultaneously call the improved ChoraleAnalyzer. In that case, one user's file can get overwritten by the other's before analysis is complete. The solution implemented involves appending a random large number to the end of the filename, generated using PHP's `rand()` function. While this solution obviously does not eliminate the possibility of a concurrency conflict, it makes it practically impossible, given the large space from which the number is taken.

- Analysis with storage (`analyzeAndStore.php`): If a chorale and a Roman numeral analysis are submitted from the profile page, the form contents are processed by `analyzeAndStore.php`. This module implements all of the above, as well as a mechanism to hold on to the submitted files and produced `errorTracker` for later

use. A connection is established with the `Chorales` table in the MySQL database, and the name of the submitted chorale is checked to prevent duplicates. The submitted files are then passed into the improved ChoraleAnalyzer, which saves the `errorTracker` instance.

Security

Upon consultation with a group of web developers working at the Princeton Department of Computer Science, a number of basic security measures were implemented in order to prevent intrusion into the database:

- **Prepared Statements**: A number of the database accesses that occur in the site involve queries composed from user input. This inherently puts the site at risk of SQL injection and jeopardizes the integrity of user data. In the site's original design, the PHP `mysql` library was used for database access. Upon realization that the `mysql` library offered no protection against SQL injection attacks, all relevant calls were replaced by calls to the similar, but more secure `mysqli` library. This library offers the ability to "prepare" an SQL query by removing all in-context meaning before passing to the database.
- **Hashing**: Before being entered into the database, passwords are passed through the PHP `crypto()` function, which implements a secure one-way hash. This way, even a malicious user that somehow gained access to the database would be unable to

impersonate another user, as the password field in the database does not divulge the actual password without knowledge of the seed used in the hash.

- **Salt**: A sequence of characters is both prepended and appended to usernames and passwords upon entry into the `newaccounts` form. By itself, salting does not add any additional security to the site. In combination with hashing, however, salting dramatically increases the difficulty of password discovery. In particular, salting helps to protect against attempts to break the hash by comparison to likely, English-based possible passwords.

User Interface

The website was transformed from raw html to a presentable interface using Bootstrap, which is a collection of CSS and JavaScript files that allow for easy selection and usage of html objects called "components". These components are pre-designed to look professional. Instead of loading all of the core Bootstrap files onto the server, the website utilizes the Bootstrap CDN, a technology that downloads the relevant files from a specified location on the internet as the page is rendered.

Future Work

In many ways, the project presented in this paper encompasses a number of "Future Work" goals set out in my paper on the original ChoraleAnalyzer. A result is that unlike for the original ChoraleAnalyzer, future work is not imperative – the tool is effectively complete to

the degree to which it solves the problem that it addresses in an easy-to-use manner, providing most readily derivable information as to the submitted chorale to the user. However, the remains substantial work that *could* be done. In particular:

- The user interface to the improved ChoraleAnalyzer could be turned into a genuine GUI, providing in-score indications of error locations. This would likely require integration with MuseScore, which is an open source musical notation package (*MuseScore*). Such a project would have a high barrier to entry; so to speak, as MuseScore itself is implemented in over 100,000 lines of C++, and an understanding of the program architecture would be required to properly integrate the improved ChoraleAnalyzer.
- There remains more advanced error-based analysis that could be done. For example, the tool could check for advanced violations like failure to resolve the third of a V7 chord up.
- The website maintains each chorale and Roman numeral analysis that is submitted in its original forms. Once the website becomes used, it is likely to amass a large set of such submissions. Who knows what this dataset might be used for in the field of musical education? Perhaps this set could provide the basis for the first musical plagiarism-detection software.
- As described above, this project has exposed a wealth of information as to the context of the rule violations in the Bach chorales themselves. This data, however, really asks more questions than it answers. It would be fascinating to explore why,

from a musicological perspective, Bach writes so many alto/tenor voice crossings over viio6 -> I6 -> V progressions. It remains for music theorists more qualified than myself to investigate the motives that caused Bach to compose in an unusual matter.

Apart from work to the program itself, a long path remains to the point at which the ChoraleAnalyzer tool with have the exposure that I expect of it. Dr. Dmitri Tymoczko in the Department of Music has asked to co-author a paper with me on the tool, which will hopefully bring it to the attention of the music theory community. I expect to commercialize the ChoraleAnalyzer as I continue to push it to be an enabling, powerful pedagogical tool. I hope that one day it will become integrated into the first comprehensive MOOC on music theory.

Acknowledgements

I wish to express deep gratitude to my advisor, Dr. Robert Dondero, for providing consistently effective direction regarding the vast array of new technologies I needed to learn this semester, for putting tremendous time and energy into advising a project on a peculiar intersection of topics, and for committing that time and energy before the project was even fully defined.

I also wish to thank Professor Dmitri Tymoczko of the Princeton Department of Music, for furnishing me with most of the data and some of the code required to implement Content Aware Bachness, and for providing guidance throughout development.

Bibliography

Covach, John. "To MOOC or Not To MOOC?" *MTO: A Journal for the Society of Music Theory*. N.p., Aug. 2013. Web. 23 Apr. 2014. <http://mtosmt.org/issues/mto.13.19.3/mto.13.19.3.covach.php?utm_source=rss&utm_medium=rss&utm_campaign=to-mooc-or-not-to-mooc-music-theory-online>.

Duhring, John. "Massive MOOC Grading Problem – Stanford HCI Group Tackles Peer Assessment." *Moocnewsandreviewscom RSS*. MOOC: MOOC News and Review, 10 May 2013. Web. 23 Apr. 2014. <http://moocnewsandreviews.com/massive-mooc-grading-problem-stanford-hci-group-tackles-peer-assessment/>.

Goldstein, Buck. "As MOOCs Move Mainstream Universities Must Pay to Play." *The Huffington Post*. TheHuffingtonPost.com, 28 Oct. 2013. Web. 23 Apr. 2014. <http://www.huffingtonpost.com/buck-goldstein/as-moocs-move-mainstream-_b_4170524.html>.

Marques, Juliana. "A Short History of MOOCs and Distance Learning." *Moocnewsandreviewscom RSS*. MOOC: MOOC News Reviews, 17 Apr. 2013. Web. 23 Apr. 2014. <http://moocnewsandreviews.com/a-short-history-of-moocs-and-distance-learning/>.

"MOOCs in 2013: Breaking Down the Numbers (EdSurge News)." *EdSurge*. N.p., 22 Dec. 2013. Web. 23 Apr. 2014. <https://www.edsurge.com/n/2013-12-22-moocs-in-2013-breaking-

down-the-numbers>.

"MuseScore." *MuseScore*. N.p., n.d. Web. 04 May 2014. <http://musescore.org/>.

"Music21: A Toolkit for Computer-Aided Musicology." *Music21: A Toolkit for Computer-Aided Musicology*. N.p., n.d. Web. 04 May 2014. <http://web.mit.edu/music21/>.

Rees, Jonathan. "Essays on the Flaws of Peer Grading in MOOCs | Inside Higher Ed." *Inside Higher Ed*. N.p., 5 Mar. 2013. Web. 23 Apr. 2014. <http://www.insidehighered.com/views/2013/03/05/essays-flaws-peer-grading-moocs#sthash.VukHeQaZ.dpbs>.

Swope, John. "How MOOCs Can Be Free and Profitable at the Same Time." *EdTech Magazine*. N.p., 16 Dec. 2013. Web. 23 Apr. 2014. <http://www.edtechmagazine.com/higher/article/2013/12/how-moocs-can-be-free-and-profitable-same-time>.