

# An Automatic Tool for the Verification of Bach-Style Chorales

Charles Peyser

Princeton University, Department of Computer Science

January 7, 2013

## Note to the Reader

This paper is intended simultaneously as a user manual for my application and as my Junior Paper, to be submitted to the Princeton Department of Computer Science. The reader is assumed to understand the fundamentals of music theory and notation, including scales, chords, inversions, and Roman numeral analysis. Should a refresher be needed, I invite the reader's attention to Appendix A, which describes the relevant musical notions while assuming only the ability to read music in the treble and bass clefs. Deep understanding of the Bach style is not assumed, and high-level descriptions of the rules of the style are provided as needed in the body of the paper itself and in Appendix C.

## Introduction

Johann Sebastian Bach was a Baroque composer writing in the early 18<sup>th</sup> century. Arguably, Bach's largest contribution to music was the development of the 4-part chorale style. Bach arranged hundreds of prayers to be sung by choirs consisting of two male parts (bass and tenor) and two female parts (alto and soprano). While the sopranos, who have the highest part, sing the primary melody of the hymn, the other three parts sing lines called "countermelodies" that individually are melodic and together harmonize the soprano part. It is often said that in this style, one can read a score both horizontally (melodically) and vertically (harmonically). The following is an excerpt from one of these chorales, complete with Roman numeral analysis.

*Chorale #300, phrase 1*

The image shows a musical score for Chorale #300, phrase 1. It consists of two staves: a treble clef staff and a bass clef staff. The treble staff contains a single melodic line with a treble clef. The bass staff contains a single bass line with a bass clef. The music is in G major and 4/4 time. The Roman numeral analysis below the staves is: a: i 6 5/3 V<sup>4-3</sup> viio<sup>7</sup>/iv iv viio<sup>7</sup>/V V.

This style is important for two primary reasons. First, it opened the door to the development of a style of music called "counterpoint", which involved multiple melodies acting together to create harmony. Bach was a master of this style, and it is widely agreed that his two-part inventions and three to four-part fugues are among the greatest pieces of

music ever written. Second, it laid the foundation for the use of “functional harmony” in compositions of all types. Functional harmony refers to a set of rules regarding in what sequences chords may appear in a piece of music. The rules of harmony and voice leading solidified and used by Bach, a Baroque composer, were fundamental to the subsequent work of the early classical composers like Haydn and Mozart, enhanced and experimented with by late classical and early romantic composers like Beethoven, Chopin, and Brahms, and finally rebelled against by late romantic composers like Wagner and transformed into a broader, less rigid style by jazz artists. It can be said that the simple four-part chorales of the Baroque era are part of what set the history of harmony into motion (Megill).

Because of its role as the foundation for future musical development, the Bach chorales are studied in detail in beginner and intermediate music theory classes. Music students are taught to analyze the chorales and to write compositions in the style. The style, however, is characterized by complicated rules that are difficult to enforce and hard to check for, and it is both tedious for instructors to check compositions for compliance and infuriating for students to balance the multiple requirements of the style.

The motivating idea behind my project is that this is an opportunity for computerized automation to make a difficult task easier to accomplish. A “full service” chorale checker, which analyzes input for violations of the Bach style and produces a report which provides useful information as to the integrity of the chorale does not yet exist and would likely be very useful for the audience at which it is targeted, especially as education moves online.

## Prior Work

The purpose of this section is to acquaint the reader with both similar projects attempted in the field and the open-source tools used in this project.

### Harmonia

Harmonia is a member of an emerging group of automatic tools for music theory education. Created by software engineers at the University of Illinois and written in about 100,000 lines of C++, Harmonia is a general-purpose music theory tool targeted at automating assignments and grading. Harmonia is presented in a clean GUI that allows instructors to create assignments that incorporate musical scores to be analyzed in real time with student input. Harmonia is a very recent application, and will be implemented in Fall '13 for music theory courses at the University of Illinois. (Harmonia)

Many music theory classes aim to teach students in two ways – analysis and composition. Harmonia is targeted at the analysis angle. Students can be asked to provide a Roman numeral analysis of a score, and have their responses compared to an analysis submitted by the professor to produce a grade report. Compositional elements are also targeted in the software, but to a less comprehensive degree. For example, an assignment can be created in Harmonia that asks the student to fill in or move certain voices in order to fit a provided Roman numeral analysis. The automatic grading of an entire student composition in some particular style, however, is yet to be supported. It is that problem (in the Bach chorale style) which my application seeks to address.

## Music21

Music21 is a set of Python modules created by Professor Michael Cuthbert and his team at MIT, self-described as “a toolkit for computer-aided musicology”. Fundamentally, music21 is a group of Python data types that encapsulate musical data together with an elaborate `parser` class that populates those data structures from any one of a number of musical formats. In particular, a musicXML (see below) formatted file can be parsed into a `score` object, which in turn contains methods to break the score down into `voice` objects, `chord` objects, `note` objects, and the like for easy analysis.

As will be described in detail below, music21 provided most of the data structures used by my analyzer. The process described above of parsing and then breaking down `score` objects proved fundamental to the completion of this tool in one semester.

Music21 is licensed under the Lesser GNU Public License, meaning that the modules are open source and can be freely downloaded from the group’s website. Furthermore, it is legal to use to music21 in closed-source applications so long as the version of music21 used is freely editable by users. Thus, although I intend to freely distribute the source code for this project, I could presumably sell the software and owe nothing to the creators of music21 (Music21).

## MusicXML

MusicXML is a widely accepted textual format for sheet music, designed to allow sheet music to be shared by a variety of applications in a way that allows it to be dynamically

changed. That is, since musicXML represents sheet music by tracking its musical elements as opposed to maintaining a graphic of the score, software that uses it can change notes, keys, chords, and formatting with ease. Since the creation of musicXML in 2004 a large corpus of music has been collected and stored in that format. In particular, all 361 of the Bach chorales in musicXML format are freely available on the Internet (JSBChorales). As such, it is a clear choice of format for applications like Harmonia and music21.

Furthermore, since a student using any standard notation software can save his or her work as a musicXML file, it provides the bridge between the student and the software that is necessary for the present project (MusicXML).

### MuseScore, Finale, Sibelius

MuseScore, Finale and Sibelius are popular examples of musical notation packages that provide a musician with an easy interface for writing music to a format like musicXML. These programs typically display an interactive score to the user, who can drag and drop notes, rests, and other musical objects into the staves. They often include other composition tools, like audio rendering and playback from a library of synthesized instruments. Each package has its own advantages and disadvantages; while MuseScore, for example, is free and open source, paid software like Finale or Sibelius is more robust and less buggy. While MuseScore was used in the development of this application, any software that saves to musicXML format is a feasible interface between the user and the chorale analyzer (MuseScore, Finale, Sibelius).

## Romantext

While musicXML has been around for long enough to become widely accepted and integrated into most relevant platforms, no standard yet exists for computer-readable roman numeral analysis notation. This problem stems from the fact that computerized sheet music was originally born out of a need for musicians to share and edit scores, and was not motivated by the prospect of large-data analysis until recently. As such, the automatic reading of roman numeral analyses has yet to become standard in the field, and in fact the problem of automatically generating roman numeral analysis without error remains unsolved.

As such, I have chosen to use Princeton professor Dmitri Tymoczko's "romantext" format, primarily because music21's `romanText` class can read it (Music21). I have been unable to find a description of the format online and had to learn the convention by inspection; I have thus included instructions as to how romantext notation works in Appendix B.

## Functionality

The purpose of this section is to acquaint the reader with the specific capabilities of the program and to provide instructions as to how to use it. The program runs in the command line on any machine with Python 2 installed. Music21 must also be downloaded and installed in order for my program to run. The program takes two command-line arguments:



1) **The name of the file containing the chorale to be verified in musicXML format.**

A chorale can be saved as a musicXML file from almost any musical notation software.

2) **The name of the file containing the roman numeral analysis of the provided chorale in romantext format (optional).** If provided, the program will run checks relevant to harmony and to the roman numeral analysis. If not provided, that entire battery of tests will be skipped.

Suppose I were interested in checking the following score for Bach-style errors. The chorale and analysis appear below, with errors in the chorale boxed in red:

The image shows a musical score for a chorale in G major, 4/4 time. It consists of four staves: three treble clefs and one bass clef. The first staff is the melody, the second is the alto part, the third is the tenor part, and the fourth is the bass part. A red box highlights a specific error in the tenor part on the third staff, where a note is marked with a sharp sign (#) that does not correspond to the key signature of G major.

7

12

Composer: J. S. Bach  
BWV: 153.1  
Title: Ach Gott, vom Himmel sieh' darein  
Analyst: Andrew Jones  
Proofreader: Dmitri Tymoczko and Hamish Robb  
Note: please email corrections to dmitri@princeton.edu

Time Signature: 4/4

Form: chorale

Note: piece has decidedly minor-mixolydian feel

m0 b4 a: V

m1 i b2 viio6 b3 i6 b4 V4/3 b4.5 i

m2 V6 b1.5 V6/5 b2 i b3 V || b4 viio6/5

m3 i6 b2 V b2.5 V7 b3 VI b4 iio6

Note: reasonably common cadential figure in m4. If G# is an incomplete neighbor, it is i6/4, otherwise III+6 with A as a regular neighbor.

m4 i6/4 b2 V b3 i :|| b4 G: V6

m4var1 III+6 b2 V b3 i || b4 G: V6

m5 I b1.5 e: viio6 b2 i b3 V b3.5 V2 b4 i6 b4.5 viio6

Note: parallel fifths evaded by voice crossing in m. 6

m6 i b2 iv6 b3 V || b4 i

m6var1 i b2 iio6/4 b2.5 ii/o4/3 b3 V || b4 i

m7 a: VI b2 i6 b3 V b4 i

m8 i6 b2 V b3 i || b4 i

m9 V6 b2 i b3 iv6 b4 viio7/IV

m10 IV b2 viio7/V b3 V

I would execute the following Python command:

```
python choraleAnalyzer.py riemenschneider003.xml riemenschneider003.txt
```

where `riemenschneider003.xml` is a reference to the musicXML version of the

chorale and `riemenschneider003.txt` is a reference to the romantext file. I

would receive the error report on the next page printed to the terminal.

Parallel Intervals

-----Chorale Analyzer-----  
Scanning for parallel unisons  
No parallel unisons found  
-----  
Scanning for parallel octaves  
No parallel octaves found  
-----  
Scanning for parallel fifths  
No parallel fifths found  
-----  
Checking ranges  
All parts in range  
-----

Zero Order

Checking distances between parts  
Interval between alto and soprano greater than an octave in measure 7  
Interval between tenor and alto greater than an octave in measure 10  
Interval between tenor and alto greater than an octave in measure 10  
-----

Voice Leading

Checking for voice crossings  
Tenor/Alto voice crossing in measure 6  
Tenor/Alto voice crossing in measure 6  
Tenor/Alto voice crossing in measure 7  
Tenor/Alto voice crossing in measure 7  
-----

Checking for tritones  
Tenor makes tritone in measure 3  
Tenor makes tritone in measure 6  
-----

Checking for repeated bass notes  
No repeated bass notes  
-----

Harmony

Verifying that Roman Numeral Analysis in fact reflects the score  
Incorrect chord root in measure 2 beat 1.5  
No chord found for roman numeral in measure 4 beat 4  
-----

Validating chord progressions in Roman Numeral Analysis  
Measure 9: The seventh degree chord must appear in first inversion.  
Measure 10: The seventh degree chord must appear in first inversion.  
Measure 2: Invalid progression from V to viio6/5.  
Measure 3: In minor, V goes to vi6, not vi  
Measure 3: Invalid progression from iio6 to i6/4.  
-----

Bachness Score: 0.874168576635  
-----

-----Analysis Complete-----

We see that there are three broad categories of checks:

- 1) Chorale-centric checks, which use only the musicXML file. These include:
  - a. Parallel Intervals – checks for parallel unisons, fifths, and octaves<sup>1</sup>
  - b. Voice Leading – checks for other violations involving a pair of adjacent chords. Includes checks for tritones<sup>2</sup> and repeated bass tones<sup>3</sup>.
  - c. Zero Order – checks involving only one chord. Includes validation of part ranges<sup>4</sup>, distance between parts<sup>5</sup>, and voice crossings<sup>6</sup>.
- 2) Checks relevant to the roman numeral analysis, which use only the romantext file.

These include<sup>7</sup>:

- a. Validation that chords of correct modality are used, given the key.
- b. Checks for incorrect progressions, ex. V to IV.
- c. Checks that appropriate chord inversions are used.

---

<sup>1</sup> A parallel interval refers to an equal interval occurring between two parts in adjacent chords. For example, the below phrase exhibits a parallel unison in the bass and alto:



<sup>2</sup> In the Bach chorale style, a sequence of two tones in a part may not make a tritone. This is not because of the sound of the chord created, but simply because tritones are difficult to sing precisely.

<sup>3</sup> Repeated bass tones across adjacent chords are thought to deprive the harmony of its sense of movement.

<sup>4</sup> This rule arises simply from the range of the human voice. The specific note ranges by part used are given in the “Design” section below.

<sup>5</sup> Adjacent parts (excluding tenor/bass) are required to be within an octave of each other at all times.

<sup>6</sup> A voice crossing occurs when a part that is supposed to be above its neighbor is in fact below it. For example, a voice crossing would occur if the soprano voice were lower than the alto.

<sup>7</sup> For lack of room here, an explanation of the Bach harmony rules is given in Appendix C.

3) Roman Numeral Analysis validation, which compares the musicXML file to the romantext file and points out inconsistencies.

It is interesting to note that Bach in fact wrote the sample chorale given above, with all of its Bach-style violations. It is not at all unusual for Bach to violate the chorale rules in his work; in fact, almost every Bach chorale includes some kind of Bach-style “error”. For that reason, it is not entirely true to the style to make absolute judgments as to whether a student chorale is in the Bach style or not based on adherence to the rules. It is more reasonable to measure degrees of conformity with the Bach style by determining how often Bach violates each rule himself and weighting the rules accordingly.

In addition to the results of the checks, the program prints a “Bachness” score for the chorale. A chorale’s Bachness is a numerical metric that seeks to give the degree to which the chorale is consistent with the Bach rules. The Bach chorales on average have a Bachness of 1; scores greater than 1 indicate a chorale which violates the Bach rules more often than the chorales themselves and visa versa.

The computation of the Bachness score requires data on the chorales themselves, which was collected by running the program on the first 70 Bach chorales<sup>8</sup>, comprising 1006 measures of music. From this data, it was possible to construct the following chart:

---

<sup>8</sup> Of course, it would have been better to use all 361 of the chorales. Unfortunately, I was only able to obtain roman numeral analysis for the first 70 chorales.

<b>Error</b>	<b>Total Violations</b>	<b>Average Violations Per Measure</b>
Parallel Unison	0	0
Parallel Fifth	22	0.021868787
Parallel Octave	2	0.001988072
Bass Out of Range	82	0.081510934
Tenor Out of Range	6	0.005964215
Alto Out of Range	1	0.000994036
Soprano Out of Range	0	0
Soprano/Alto Closeness	22	0.021868787
Alto/Tenor Closeness	76	0.07554672
Bass/Tenor Voice Crossing	41	0.040755467
Tenor/Alto Voice Crossing	94	0.093439364
Alto/Soprano Voice Crossing	29	0.028827038
Tritone	48	0.047713718
Repeated Bass	35	0.034791252
Incorrect Chord Inversion	38	0.03777336
Incorrect Chord Modality	79	0.078528827
Invalid Chord Progression	94	0.093439364

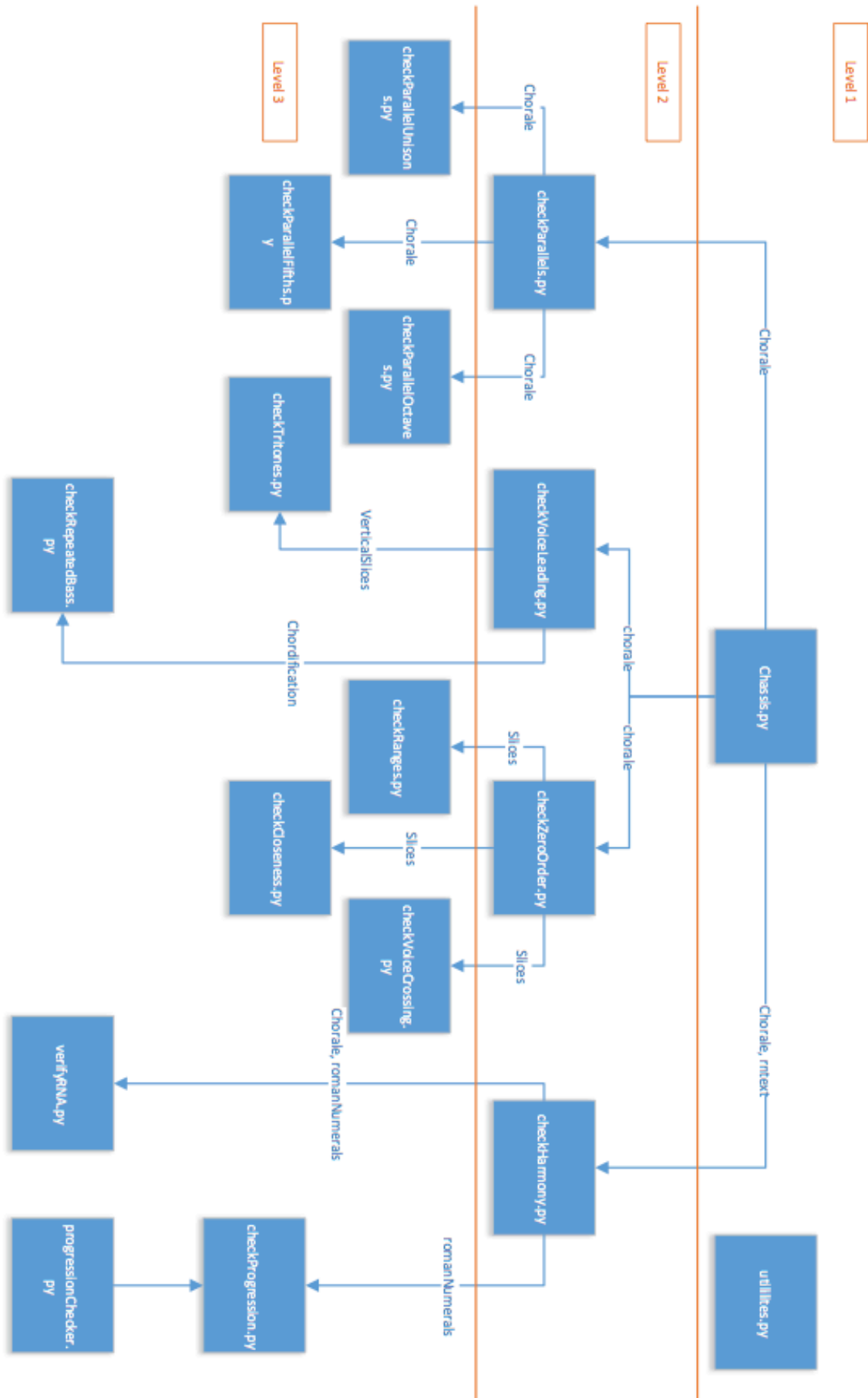
These data provide the foundation for the Bachness computation. For any chorale, the program tallies up errors and produces a similar chart, tracking average violations per measure and comparing it to the data on the Bach chorales themselves. The program then computes the ratio of violations per measure in the student chorale to violations per measure in Bach for each individual error type (errors which occurred zero times are approximated as having occurred once, to avoid division by zero). Averaging these ratios gives the submission's Bachness.

The motivation of developing the Bachness score was to bridge the gap between automated error checking and automated grading for student submissions. Unlike the judgement of a human grader, Bachness is an objective measure of a chorale's conformity with the Bach rules that accounts for the degree to which each rule is reflected in Bach's work.

## Design

The purpose of this section is to describe the techniques and data structures used to accomplish the task of chorale verification. The program is implemented in Python, primarily using music21 data types as described above. The code is organized in a three-tiered module-based system, as illustrated in the module interaction diagram on the next page. Blue boxes indicate Python modules, and blue arrows indicate the primary direction of data flow between them. The arrows are labeled with the data structure that is passed between the modules.

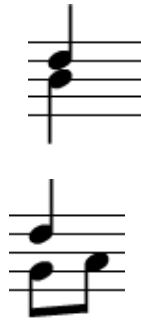




We see that the program is executed with a call to `choraleAnalyzer.py`, which parses the musicXML file into a music21 `score` object (called “chorale” in the chart and in the code), parses the romantext file into a music21 `romanText` object (called “rntext”), and passes relevant data down to the second tier of modules. These modules then do some preliminary parsing and pass data down to the third tier of modules, which contains functions that return Python lists of strings containing error messages to the second tier functions. The second tier functions then call `printErrors()` from the `utilities.py` module, which prints those error messages to the terminal. While I do not want to belabor the topic by describing code, I will describe at the general level the methods and data structures used to implement each second tier family of tests.

- 1) `checkParallels.py`: Unlike other checks, I was able to not only utilize music21 data types but also music21 methods in order to accomplish parallel interval identification. The music21 `theoryAnalysis.theoryAnalyzer` module contains a number of methods that operate on the `score` data type and return `VoiceLeadingQuartet` objects, which encapsulate pairs of parts moving from one chord to a second. In particular, the `getParallelFifths()`, `identifyParallelOctaves()`, and `identifyParallelUnisons()` methods are called on the “chorale” `score` object in order to produce appropriate error identification.
- 2) `checkZeroOrder.py`: Single-chord analysis relies on the music21 `voiceLeading.verticalSlice` object, which, as the name suggests, encapsulates data regarding a vertical “cross-section” of the chorale, including its chord type and the

four (or fewer) tones that comprise it. The verticalSlice object is useful because it comes with a getVerticalSlice() method which operates on score objects and returns a Python list of verticalSlice objects corresponding to each simultaneous quartet of tones. That is, a phrase like



will generate two verticalSlice objects, one for <B, F, B, D> and one for <C, F, B, D>.

This feature is necessary because both chords are subject to the Bach rules. Parsing into a Python list of verticalSlice objects called “slices” is done in the second tier module. The list is then passed to three third tier modules:

- a. checkRanges.py: Flags violations of the following range restrictions:
  - i. Soprano: c4 to g5
  - ii. Alto: g3 to d5
  - iii. Tenor: c3 to g4
  - iv. Bass: f2 to c4

These checks can be done easily because the music21 pitch class has a built-in comparator.

- b. checkCloseness.py: Verifies that adjacent parts are sufficiently close to each other at all times. In particular, adjacent parts must be within an octave of

each other, except for the bass part, which can differ from the tenor part by more than an octave. Validating closeness is tricky because the music21 interval class does not have a built in comparator: that is, a line like `interval.Interval(chord[1], chord[0]) > interval.Interval('P8')` will not work. The solution that was implemented after some thought involves using the interval class' cents attribute, which gives an integer metric on the size of an interval (a half step is 100 cents, a whole step is 200 cents, ect.) for purposes of comparison.

- c. `checkVoiceCrossing.py`: Returns an error message for instances of voice crossings between adjacent parts. Since pitch objects can be compared in music21, this is easy to implement.

3) `checkVoiceLeading.py`: This module covers two-chord checks which are not checks for parallel intervals. In particular, it contains:

- a. `checkTritones.py`: Flags tritone intervals in any part. The same `verticalSlice` object that was used for validating zero-order rules was used here. While interval objects cannot be compared using "<" and ">" in music21, they can be checked for equality. It became known during testing however that this check only considers intervals of the same *spelling* to be equal, that is, the interval between a C and an F# would not be considered equal to the interval between a C and a Gb, because, while they are both tritones, the first is an augmented fourth while the second as a diminished fifth. The solution to this problem that I implemented involves the creation of a custom check for an interval's equality with the tritone, implemented by maintaining a list of all

varieties of tritone intervals and trying the music21 interval equality check on each element in the list.

- b. `checkRepeatedBass.py`: Points out repeated bass tones. Identifying repeated bass tones is tricky because it is difficult to distinguish between connected tones and separate tones in a `verticalSlice` reduction. For example, the following phrase would generate two `verticalSlice` objects (because of the two tones in the alto part), and the C in the bass would look like a repeated tone.



I was unable to find a data structure in music21 which addressed this problem. The solution that I implemented involves the use of the `chordify` method, which acts on the score object “chorale” to produce a Python list of chord objects which comprise the chorale. From that list a list of bass tones was constructed, maintaining as properties the measure number and the number of beats from the beginning of the chorale (this is computed by adding the beat to the time signature times the measure number). These properties had to be explicitly tracked because music21 does not support the

extraction of a part from a list of chords in a way that preserves beat and measure<sup>9</sup>. Finding repeated bass notes given such a construction is matter of comparing each adjacent pair in that list and determining if, for the first note, the number of beats from the beginning of the chorale plus the duration of the note is less than the number of the beats from the beginning of the chorale of the second note. Under that condition, the bass note can be identified as having been repeated.

4) `checkHarmony.py`: This module consists of both the comparison of the provided Roman numeral analysis to the chorale and the validation of the chord progressions in that Roman numeral analysis. This module was difficult to create because `music21` only provides basic support of the `romantext` format, does not provide any data structures for the comparison of a roman numeral analysis to a Bach chorale, and does not provide methods for comparing an analysis to the Bach rules of harmony. The module I created does some initial parsing, so that it passes a Python list of `RomanNumeral` objects down to `verifyRNA.py` rather than a `romantext` file.

Two checks are implemented:

- a. `verifyRNA.py`: This module points out inconsistencies between the provided roman numeral analysis and chorale. I define a correct Roman numeral analysis as a set of Roman numerals, each of which line up with a chord that it correctly describes. The implemented procedure for validating a Roman numeral analysis involves the creation of a pair of Python lists of two-

---

<sup>9</sup> This likely arises from the fact that `music21` is intended to address music which generally is not chord-to-chord, by nature. That is, the separation of parts from a sequence of chords is a counterintuitive notion outside of the chorale style.

element lists. The first, called “numList”, is derived entirely from the list of RomanNumeral objects. It has as its first element a decimal value in the form  $m.b$  where  $m$  is a measure number and  $b$  is a beat number, and as its second element the music21 RomanNumeral object that occurs at that measure and beat. Note that not all beats in a measure are present in “numList”; only those at which a Roman numeral occurs are represented in the list. The second list, called “chordList”, is defined entirely from the Python list of chord objects derived from the score object “chorale” using the chordify method. It has as its first element a similar decimal value representing measure and beat and as its second element a chord object representing the chord that occurs at that measure and beat in the chorale. The two lists interface in a nested for loop that, for each RomanNumeral object in numList, searches chordList for a chord with the same beat and measure. If no chord is found, an error message is returned. If a chord is found, the chord and roman numeral are compared by a helper method which checks that that they represent the same chord in the same inversion.

- b. checkProgression.py: My implementation for the validation of the chord progression in a roman numeral analysis relies on a rather elaborate object called a progressionChecker, which I have defined in the module progressionChecker.py. The module is complex and went through several rounds of development, as it was difficult to create a data structure to encapsulate a set of rules as general as functional harmony, and which remains invariant under modulation. The notion behind the

progressionChecker object is that for our purposes a chord progression can be checked by considering only each numeral on its own and each adjacent pair of numerals. As such, a progressionChecker can be “loaded” with either one RomanNumeral object or two, and can validate the correctness of that input with a call to its isValid() method. More specifically, a progressionChecker object contains as instance variables a large group of functions (the possibility of having functions as data types is a convenient feature of Python for this application), each which check either a single numeral or a pair of numerals for some violation in particular. Initialization occurs in two parts. First the class constructor is called with only the mode of the chorale as an argument. In the constructor that subset of the checker functions that is relevant to the given modality is selected and used to populate two Python lists, “singleChecks” and “doubleChecks”, which respectively contain methods to check single numerals and methods to check pairs of numerals. Then, a call is made to the class’s setChord or setChords method in order to load up the class with either a single numeral or a pair of numerals. Which of those two methods is called determines which set of tests, singleChecks or doubleChecks, will be run when isValid() is called. The module checkProgression.py itself is essentially client code to the progressionChecker data type, loading up the object sequentially with each numeral or pair of numerals and calling isValid(). Modulations are handled by recursive calls that create new progressionChecker objects and append the generated list of error messages to those generated for the previous key.



While not ran during execution of the chorale analyzer itself, the `testBach.py` module merits brief discussion. The module is a script that runs the chorale analyzer on a given number of Bach chorales and computes the total number of violations of each rule. It makes use of the `errorTracker` object (defined in `utilities.py`), which does string parsing on the output of the program to populate local fields that represent the number of errors occurring over a large set of chorales.

## Evaluation

Ideally, testing for a program such as mine would involve running the code on a large number of student-written chorales and comparing the application's output to that of a human grader. That way, it would not only be possible to confirm that each of the checks in fact work but also to compare the ability of the program to identify errors to that of a typical graduate student teacher's assistant. It would also be possible to demonstrate the objectivity of the Bachness score as a grading metric in comparison to judgments made by human graders.

Unfortunately, because Princeton's relevant music theory class does not return grades on its chorale-style composition assignment until after the end of the semester, I was unable to obtain the body of data necessary to conduct these tests in time for this paper's deadline.

Given this limitation, I developed a battery of tests that involves the introduction of erroneous music into an arbitrary selection of actual Bach chorales. One chorale (and corresponding roman numeral analysis) was created for each family of checks, and the

output of the program on that input was compared against the list of relevant errors deliberately written into the chorale. The tests and results are as follows:

1. **Parallel Intervals:** Changes were made to Bach's 10<sup>th</sup> chorale that introduced two instances each of parallel unisons, parallel fifths, and parallel octaves over a number of pairs of parts. The program correctly identified all six errors.
2. **Ranges:** Changes were made to Bach's 6<sup>th</sup> chorale that, for each of the four parts, were above and below the ascribed range once. The program correctly identified all eight errors, although by the nature of the implementation it identified a violating part only once per measure, even if that part violated the rule on multiple consecutive notes.
3. **Distances Between Parts:** Changes were made to Bach's 22<sup>nd</sup> chorale that violated the octave limitation on distances between parts. The rule was violated three times between the soprano and alto parts and three times between the alto and tenor parts. The program correctly identified all six errors.
4. **Voice Crossings:** Changes were made to Bach's 46<sup>th</sup> chorale that caused voice crossings between each adjacent part in several locations. The program correctly identified all seven errors.
5. **Tritones:** Changes were made to Bach's 16<sup>th</sup> chorale that introduced a large number of tritones. Both ascending and descending tritones were included, of both the augmented 4<sup>th</sup> and diminished 5<sup>th</sup> variety. The program correctly identified all twelve errors.
6. **Repeated Bass:** Changes were made to repeat bass tones. Bass tones were repeated in the same measure and across measures. The program correctly identified all

seven errors.

7. Harmonic Progressions: Changes were made to the roman numeral analysis of Bach's 45<sup>th</sup> chorale. These changes introduced a number of harmonic errors, including incorrect chord modalities given key, incorrect use of borrowed chords, and incorrect inversions. A resolution IV/V to V/V was included to ensure that the program tolerates a borrowed chord preceded by its own subdominant. The program correctly identified eleven of the twelve errors. The error that the program failed to identify occurred over a modulation. Because of the initialization of a new `progressionChecker` object at each modulation, the program does not check for two-chord errors that occur as the key changes.

## Other Known Limitations

The previous section identified a limitation of the program. This section describes known bugs and limitations.

1. The user will find that the program runs substantially slower than would be ideal, taking between five and ten minutes on twenty measures of music. The culprit is the `getVLQs` method inside the `theoryAnalysis.theoryAnalyzer` in `music21` that, as the name suggests, retrieves all possible `VoiceLeadingQuartet` objects from the score and is integral to the process of searching the chorale for parallel intervals. The `music21` developers have been made aware of the speed issue with this method. In consultation with my advisor, I have included an option in execution that skips the checks for parallel intervals. If a third argument (of any

kind) is provided to the program, that condition will be activated and execution will complete in several seconds without printing parallel intervals. I hope to release an update of the software that uses a custom `getVLQs` method to retrieve only the data that are needed for parallel interval checking.

2. Testing revealed some amount of unpredictable behavior in the `verifyRNA.py` module, which, as described above, checks consistency between the provided chorale and roman numeral analysis. In particular, ornamental objects such as decorated bar lines were shown to disrupt the modules inner measure counting, leading to a number of false error identifications. Further investigation revealed that the measure renumbering was in fact a failure of the MuseScore implementation, which breaks measures that include such ornaments into two in the musicXML `<measure number>` header read by the music21 parser. In consultation with my advisor, I determined that such implementation details are outside of my control as a programmer, and I will content myself with warning the user of possible issues with this function in particular.

## Future Work

With only a few months to design and write this program, there's quite a bit that I wish that I could have done to make the program more useful. In particular:

- 1) While the program covers what are widely considered the most essential of the Bach rules, there are a few checks that I would like to have added. For example, I

think that it would be valuable to be able to check that the leading tone is always resolved down by step, that two leaps do not occur simultaneously in one part, and that the numerals in a Roman numeral analysis not only each represent a chord in a chorale, but together cover all of the chords in the chorale.

- 2) An automated chorale analyzer opens the door to rigorous statistical analysis of the Bach chorales themselves, which can inform grading standards for student compositions. The “Bachness” score is an example of this, but there is potential for much more sophisticated implementation. For example, chord progressions could be compared against gathered data so that error messages like “illegal progression from IV6 to iv” could be replaced by “Bach uses progresses from IV6 to iv in only 0.5% of cases; consider revising”.
- 3) If this program is ever to be implemented for a non-technical population at the classroom level, it may require a GUI. I originally envisioned a program integrated with some open-source musical notation software (like MuseScore) such that, with the push of a button, the input that is in the score at that moment is checked for errors. I imagine that the integration of my program with a GUI would not be all that complicated; the only real change to the current code would have to be in the `printErrors()` method in `utilities.py`, which would not flag errors in the score rather than print them to the terminal. That is, it is an immediate implication of the design decision to interface with the user through musicXML and romantext formatted data that error checking is entirely abstracted from the method of input. Furthermore, the decision to modularize the code into individual, non-codependent families of checks opens the door to real-time partial chorale validation. In other

words, an online program could practically implement checks as needed according to user input, rather than run the entire program at each change of a note. It is thus conceivably possible to implement a GUI that checks input even as it is being entered.

While there is certainly much that can be done to improve the tool, I expect to see the program utilized in its current state. In particular, Dr. Andrew Lovett of the Princeton Department of Music has expressed his intention to implement the tool in the relevant music theory class at Princeton, and Dr. Michal Cuthbert at MIT has expressed interest in publishing the code as a sample application on music21's website.

## Acknowledgements

I owe deep thanks to Dr. Robert Dondero of the Department of Computer Science at Princeton, who spent a great deal of time helping me to understand the programming challenges associated with this project in his role as the advisor to my independent work. I also owe thanks to Dr. Dmitri Tymoczko of the Department of Music at Princeton, who in his classes inspired my interest in computational music theory and who made himself available by email throughout the semester to answer questions regarding design and implementation. Finally, this project could not have even been attempted without the work of Michael Cuthbert and the music21 team.

## Works Cited

- "Finale Music Notation Software Products for Music Composition." *Finale*. N.p., n.d. Web. 06 Jan. 2014. <<http://www.finalemusic.com/>>.
- Harmonia*. N.p., n.d. Web. 30 Dec. 2013. <<http://camil.music.illinois.edu/software/harmonia/>>.
- "JSBChorales.net: Bach Chorales." *JSBChorales.net: Bach Chorales*. N.p., n.d. Web. 30 Dec. 2013. <<http://www.jsbchorales.net/index.shtml>>.
- "MuseScore." *MuseScore*. N.p., n.d. Web. 06 Jan. 2014. <<http://musescore.org/>>.
- "MusicXML for Exchanging Digital Sheet Music." *MusicXML*. N.p., n.d. Web. 30 Dec. 2013. <<http://www.musicxml.com/>>.
- "Sibelius - the Leading Music Composition and Notation Software." *Sibelius - the Leading Music Composition and Notation Software*. N.p., n.d. Web. 06 Jan. 2014. <[http://www.sibelius.com/home/index\\_flash.html](http://www.sibelius.com/home/index_flash.html)>.
- Tymoczko, Dmitri. "Review of Michael Cuthbert, Music21: A Toolkit for Computer-aided Musicology (<http://web.mit.edu/music21/>)."  
*MTO 19.3: Tymoczko, Review of Cuthbert, Music21*. N.p., June 2013. Web. 30 Dec. 2013. <<http://mtosmt.org/issues/mto.13.19.3/mto.13.19.3.tymoczko.php>>.

## Appendix A: Basic Music Theory

This section will cover the fundamentals of music theory that are necessary to understand this project. I will address the following topics:

1. Intervals
2. Scales and Key
3. Chords

### Intervals

The musical distance that separates two tones is said to be the interval between those tones. Intervals are categorized according to the seven tones in a Pythagorean scale. That is, in the absence of a key signature, we can make the following designations:



These designations are known as interval “numbers”. An interval number by itself, however, leaves ambiguity as to the precise distance between two tones. Consider that a “third” describes both of the following intervals.











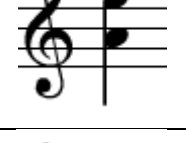

A third




Also a third

For this reason, musicians speak of interval “qualities” as well as numbers. In the example above, the third separating a G and a B would be called a “major” third, while the interval separating a G and a Bb would be called a “minor” third. The following table quantifies intervals by the number of semitones (the distance between two adjacent keys on a piano) they contain. Note that there is redundancy – an augmented fifth is the same as a minor sixth but for notation. Also note that this list is not exhaustive and that more obscure interval designations exist for particular circumstances.



Interval Number	Interval Quality	Semitones	Example
Second	Minor	1	
	Major	2	
Third	Minor	3	
	Major	4	
Fourth	Diminished	4	
	Perfect	5	
	Augmented (tritone)	6	
Fifth	Diminished (tritone)	6	
	Perfect	7	
	Augmented	8	

Sixth	Minor	8	
	Major	9	
Seventh	Minor	10	
	Major	11	
Octave	(Perfect)	12	

## Scales and Key

A “scale” is an unordered collection of tones. For example, the following scale (one version of the “diatonic” scale) contains all of the main notes in the melody of “Let it Be”, by The Beatles, and “Stairway to Heaven” by Led Zeppelin:


A, B, C, D, E, F, G


While in general scales can contain any tones, this scale conveniently contains no accidentals (black keys on the piano).

A “mode” is an ordered scale. That is, a mode is a scale with one note designated as the “tonic” or musical center of the arrangement, and the others by the number of the interval they make with the tonic. The Beatles in “Let it Be”, for example, choose a mode by designating C as the tonic in the scale above, which makes the “key” of the piece C major and provides the bright, happy sound characteristic of major compositions. Led Zeppelin in “Stairway to Heaven” designates A as the tonic, which makes the key A minor and provides the somber, melancholy sound characteristic of minor compositions. While both pieces use the same scale, the method of orienting the scale through choice of mode gives them drastically different tones. Note that

while the terms “scale” and “mode” have different meanings, the term “scale” is frequently substituted for “mode” when the meaning is understood.

While there are tens of scales and hundreds of modes available to composers and musicians, the development of music from the beginning of the Baroque era at the beginning of the 17<sup>th</sup> century until the advent of jazz at the beginning of the 19<sup>th</sup> century was primarily concerned with the major and minor modes of the diatonic scale. Familiarity with these two modes will suffice for the purposes of this project. Use the following chart to facilitate an understanding of the minor and major modes. Note that the mode is named by its tonic, so that a major scale with a tonic of F# would be called “F# major”.

<b>Major</b>	Example: 						
Scale Degree	1 (tonic)**	2 (supertonic)	3 (mediant)	4 (subdominant)	5 (dominant)	6 (submediant)	7 (leading tone)
Interval from preceding tone *	m2	M2	M2	m2	M2	M2	M2
Interval from tonic	N/A	M2	M3	P4	P5	M6	M7

<b>Minor</b>	Example: 						
Scale Degree	1 (tonic)	2 (supertonic)	3 (mediant)	4 (subdominant)	5 (dominant)	6 (submediant)	7 (subtonic)

Interval from preceding tone	M2	M2	m2	M2	M2	m2	M2
Interval from tonic	N/A	M2	m3	P4	P5	m6	m7


\* When notating interval qualities, “M” is shorthand for major, “m” for minor, and “P” for perfect. It will also be useful to recognize “d” for diminished and “A” for augmented.




\*\* Special names are given to each of the scale degrees. It is important to recognize tonic, subdominant, dominant, and leading tone, as they are often used in context of chords in a scale (see below). Supertonic, mediant, submediant, and the minor subtonic are more obscure terms.

## Chords

So far we have covered the basic theoretical elements of musical melody, explaining how notes are chosen from a subset of the available tones called a scale. Like melody, harmony is chosen out of a scale, the difference being that instead of selecting single notes to form a tune, multiple notes are taken to form a chord. As a guitar player, you may already be familiar with the rules for the construction of chords. In order to understand the Bach style, however, we will need to understand how chords fit into the context of a key.

The most fundamental chords involve three tones: the base, the third, and the fifth. Such chords are called “triads”. The following chart gives the names and rules for construction of the four types of triads:

Name	Interval from base to 3 <sup>rd</sup>	Interval from base to 5 <sup>th</sup>	Example
Major	Major	Perfect	F major (F): 

<b>Minor</b>	Minor	Perfect	D minor (Dm): 
<b>Diminished</b>	Minor	Diminished	E diminished (E <sup>o</sup> ): 
<b>Augmented</b>	Major	Augmented	A augmented (A <sup>+</sup> ): 

Note the shorthand conventions for chord notation. Major chords are notated by a capital letter (D major = “D”, G sharp major = “G#”), minor chords with an “m” (A minor = “Am”), diminished chords with a small circle (B diminished = “B<sup>o</sup>”) and augmented chords with a small plus (E flat augmented = “Eb<sup>+</sup>”).

The order of tones in a chord does not change its identity. That is, all of the following are C major chords:



C



C<sup>6</sup>



C<sup>6</sup><sub>4</sub>

We distinguish between such chords by defining chord “inversions”. A chord in “root position” has the base at the bottom, as is notated normally (ex. first chord above). A chord in “first inversion” has the third at the bottom, and is notated with a small 6 (ex. second chord above). A chord in “second inversion” has the fifth at the bottom, and is notated with a <sup>6</sup><sub>4</sub> (ex. third chord above).

Now, consider a simple C major scale, with each scale degree turned into a chord by adding a third and a fifth, as appropriate in the scale.



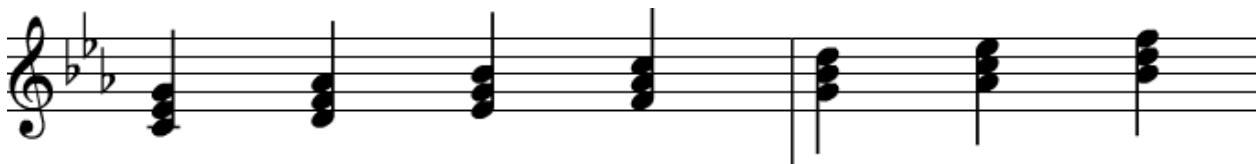
Scale Degree:    1            2            3            4            5            6            7

Chord:            C major    D minor    E minor    F major    G major    A minor    B diminished

This pattern of chord qualities will always be true of chords in a major scale. That is, the chords will always follow the pattern: major, minor, minor, major, major, minor, diminished. We can thus generalize across keys and use the following “Roman numeral notation” to describe chords:

Scale Degree	Example in C major	Roman Numeral Designation
1	C major	I
2	D minor	ii
3	E minor	iii
4	F major	IV
5	G major	V
6	A minor	vi
7	B diminished	vii <sup>o</sup>

A similar pattern and set of Roman numeral designations can be derived for minor:



Scale Degree:    1            2            3            4            5            6            7

Chord:            C minor    D diminished    Eb major    F minor    G minor    Ab major    Bb major

Scale Degree	Example in C minor	Roman Numeral Designation
1	C minor	i

2	D diminished	ii <sup>o</sup>
3	Eb major	III
4	F minor	iv
5	G minor (major)*	v (V)
6	Ab major	VI
7	Bb major	VII

\* In minor keys, the fifth is often made major by sharpening the seventh degree of the scale. See the discussion of chord progressions below.

Roman numeral notation allows us to analyze musical compositions. Consider the following Roman numeral analysis of an excerpt from a Strauss piano piece.

The image shows a musical score for a piano piece by Strauss, marked "Langsam" and "mp". The score is in D major (two sharps) and 2/4 time. The analysis below the score identifies the chords for measures 1 through 8:

- Measure 1: I
- Measure 2: vi<sup>6</sup><sub>s</sub> (I with added 6th?)
- Measure 3: III<sup>+</sup>7
- Measure 4: ii<sup>6</sup>
- Measure 5: V<sup>7</sup>
- Measure 6: viio<sup>7</sup><sub>o</sub>
- Measure 7: III<sup>+</sup>7
- Measure 8: V

Ignore symbols above which have not been covered – we seek here to cover only those elements of Roman numeral analysis relevant to the project at hand.

## Appendix B: Romantext Notation

The purpose of this section is to provide a functional description of Dmitri Tymoczko's romantext format for roman numeral analysis. While romantext format is yet to be documented in an easy to find place, it has the benefit of being extremely simple.

Romantext notation assigns each measure to a single line. For each measure, each chord is enumerated by its beat within the measure followed by its roman numeral. We also notate keys and key changes with upper case letters for major keys and lower case letters for minor keys. Designation of a key must occur only at the beginning of a piece and at a key change. Consider the following phrase:



In romantext format, we could write its roman numeral analysis as

m0 b1 C: I b2 V b3 I b4 V

m1 b1 V b2 I

The following list enumerates other details in the romantext format:

1. Fractional beats are notated with a decimal after the beat number. For example, a chord occurring on the fifth eighth note in a 4/4 measure would be denoted with b2 . 5; one occurring on the fifth sixteenth note would be denoted with b1 . 25.
2. First inversion is notated as expected: a first inversion I chord would be I6. Other inversions (which involve more than one number) are notated with a "/". For example, a second inversion I chord would be I6/4, and a third inversion V7 chord would be V4/2.
3. Borrowed chords are also represented with "/". For example, a first inversion D7 chord in the key of C is V6/5/V.
4. Diminished chords are represented with an "o", as in v i i o6. Augmented chords are represented with a "+".
5. One may include multiple versions of an analysis. For example, the following line occurs in Mia Tsui's analysis of BWV 151.5:



m5 I b2 V4/3 b3 I6 b4 vi

m5var1 I b2 V4/3 b3 I6 b4 vi b4.5 IV6

- Notes may be included in the analysis. They appear on their own lines and begin with the designation "Note:".

I found it easiest to learn this form of notation simply by inspection. For that purpose, I've included here an analysis of Bach's 24<sup>th</sup> Chorale, by Professor Tymoczko.

Composer: J. S. Bach  
BWV: 415  
Title: Valet will ich dir geben  
Analyst: Dmitri Tymoczko  
Proofreader: Hamish Robb  
Note: please email corrections to dmitri@princeton.edu

Time Signature: 4/4

Form: chorale

m0 b4 D: I

Note: viio6-IV6-I is a variant of "V-IV6/4-I" which appears at multiple points in the chorales

m1 I b2 I6 b2.5 I b3 IV b4 viio6

m2 IV6 b2 I b3 I || b4 I

m3 viio7/vi b2 vi b3 viio7/V b4 V

m4 I :|| b4 I

m5 I b2 vi b3 V6/V A: V6 b3.5 V7 b4 vi7 b4.5 viio

m6 I b2 V6/vi b3 vi || b4 vi

m7 viio6 b2 I6 b3 V b4.5 V7

m7var1 viio6 b2 I6 b3 V b3.5 ii6/5 b4 V b4.5 V7

Note: Riemenschneider has a first-inversion chord on beat 4 of m. 8

m8 I || b4 I6 D: V6

m8var1 I || b4 I6 D: V6

m9 I b2 I6 b2.5 V7/IV b3 IV b3.5 |vii/o4/3 b4 I6 b4.5 e: ii/o2

m10 V6/5 b2 i b3 V || b4 G: V6/5

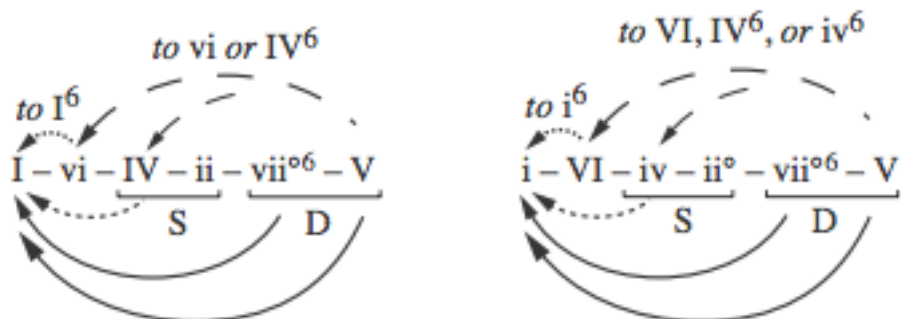
m11 I b2 iii D: vi b3 ii6/5 b3.5 ii7 b4 V b4.5 V7

m12 I

## Appendix C: Bach Rules of Harmony

The purpose of this section is to outline the harmonic rules implemented in the program. This information is included here for lack of space in the body of the paper itself.

At the basic level, the Bach style employs *functional harmony*, which is a set of rules governing two-chord sequences. In particular, the following charts outline permitted progressions in major and minor respectively:



These diagrams are taken from Professor Dmitri Tymoczko's notes for MUS105, an introductory music theory class taught at Princeton. A progression may move rightward by any amount towards the V chord, but may regress leftward only by the arrows. The first two chords comprise the "tonic" harmonic area, while the subsequent pairs of chords are called the "subdominant" and "dominant" regions, as indicated. The Bach chorales, and subsequent works that employ some derivative of this functional harmony, can be said (very) generally to repeatedly move from the tonic region to the dominant region, sometimes passing through the subdominant region, before returning to the tonic.

The program checks that the rules of functional harmony are followed in the given chorale, both in that movements from chord to chord are valid and also in that chords are of the correct modality (ex.  $ii^0$  instead of  $ii$  in minor,  $IV$  instead of  $iv$  in major). The program also makes the following related validations:

1. Illegal single-chord inversions are avoided. In particular, the second degree chord in minor does not appear in root position. In both major and minor, the sixth degree chord must be in root position, and the seventh degree chord must appear in first inversion.
2. When  $vi$  resolves to the tonic, that tonic chord must be in first inversion.
3. In minor,  $V$  goes to  $IV^6$  or  $vi^6$ , rather than  $IV$  or  $vi$ .

## Appendix D: Testing Log

The following is a list of the mistakes inserted into the testing chorales, as described above:

1. Parallel Intervals: Taken from #10 (ALL FOUND)
  - P5 TA in measure 2 beginning
  - PU SA in measure 4 beginning
  - P8 BA in measure 5 beginning
  - P8 TS in measure 6 whole measure
  - P5 TA in measure 7 beginning
  - PU SA in measure 10 beginning
2. Ranges: Taken from #6 (ALL FOUND)
  - Sop measure 1

Sop measure 2  
Alto measure 3  
Alto measure 4  
Ten measure 5  
Ten measure 6  
Bass measure 7  
Bass measure 7

3. Part Dist: Taken from #22 (ALL FOUND)

SA measure 1  
AT measure 2  
SA measure 3  
AT measure 5  
SA measure 6  
AT measure 7

4. Voice Crossings: Taken from #46 (ALL FOUND)

AT measure 1  
SA measure 2  
AT measure 3  
AT measure 4  
SA measure 5  
BT measure 7  
BT measure 8

5. Tritones: Taken from #16 (ALL FOUND)

A measure 2 (twice)  
S measure 3 (twice)  
S measure 5 (twice)  
B measure 5  
T measure 6  
A measure 8  
B measure 10  
B measure 11  
T measure 12

6. RepeatedBass:

measure 1  
measure 3  
measure 4  
measure 6  
measure 7  
measure 8  
measure 9

7. Harmony: Taken from #45

m0: Major/Minor i (m)  
m1: Major/Minor iv (m)  
m2: Illegal root position ii in minor  
m3: Illegal root position ii in minor  
m3: Illegal V to iv6  
m4: Illegal first inversion vi  
m5: Illegal root position vii  
m6: Major/Minor ii7 (M)  
m7: vi to i instead of i6 NOT FOUND  
m8: illegal V/V to i  
m11: Tolerate proceeding by own subdominant  
m11: Illegal V/V to V/II

The following is the output of the program on those chorales:

```
Cals-MacBook-Pro:chassis calpeyser$ python choraleAnalyzer.py TestChorales/parallels.xml  
TestChorales/parallels.txt
```

music21: Certain music21 functions might need these optional packages: matplotlib, numpy, scipy; if you run into errors, install it by following the instructions at <http://mit.edu/music21/doc/html/installAdditional.html>

-----Chorale Analyzer-----

Scanning for parallel unisons

Parallel unison found!

Measure Number: 4

First Voice:

First Note: D in octave 5 Quarter Note

Second Note: C in octave 5 Eighth Note

Second Voice:

First Note: D in octave 5 Quarter Note

Second Note: C in octave 5 Quarter Note

Parallel unison found!

Measure Number: 10

First Voice:

First Note: B in octave 4 Quarter Note

Second Note: A in octave 4 Quarter Note

Second Voice:

First Note: B in octave 4 Quarter Note

Second Note: A in octave 4 Quarter Note

-----  
Scanning for parallel octaves

Parallel octave found!

Measure Number: 6

First Voice:

First Note: A in octave 4 Half Note

Second Note: B in octave 4 Quarter Note

Second Voice:

First Note: A in octave 3 Half Note

Second Note: B in octave 3 Quarter Note

Parallel octave found!

Measure Number: 6

First Voice:

First Note: B in octave 4 Quarter Note

Second Note: C in octave 5 Quarter Note

Second Voice:

First Note: B in octave 3 Quarter Note

Second Note: C in octave 4 Quarter Note

Parallel octave found!

Measure Number: 4

First Voice:

First Note: F in octave 4 Quarter Note

Second Note: C in octave 4 Quarter Note

Second Voice:

First Note: F in octave 3 Quarter Note

Second Note: C in octave 3 Quarter Note

Parallel octave found!

Measure Number: 5

First Voice:

First Note: C in octave 4 Quarter Note

Second Note: D in octave 4 Eighth Note

Second Voice:

First Note: C in octave 3 Quarter Note

Second Note: D in octave 3 Quarter Note

-----

Scanning for parallel fifths

Parallel fifth found!

Measure Number: 2

First Voice:

First Note: A in octave 4 Quarter Note

Second Note: G in octave 4 Eighth Note

Second Voice:

First Note: D in octave 4 Quarter Note

Second Note: C in octave 4 Quarter Note

Parallel fifth found!

Measure Number: 7

First Voice:

First Note: G in octave 4 Quarter Note

Second Note: A in octave 4 Quarter Note

Second Voice:

First Note: C in octave 4 Quarter Note

Second Note: D in octave 4 Eighth Note

-----

Checking ranges

Bass out of range in measure 13

-----

Checking distances between parts

Interval between tenor and alto greater than an octave in measure 10

Interval between tenor and alto greater than an octave in measure 10

Interval between tenor and alto greater than an octave in measure 10

Interval between tenor and alto greater than an octave in measure 10

---

Checking for voice crossings

Alto/Soprano voice crossing in measure 4

Bass/Tenor voice crossing in measure 6

---

Checking for tritones

Tenor makes tritone in measure 7

Tenor makes tritone in measure 9

---

Checking for repeated bass notes

Repeated bass note (E3) in measure 2

---

Verifying that Roman Numeral Analysis in fact reflects the score

Incorrect chord root in measure 4 beat 2.5

Incorrect chord root in measure 9 beat 3.5

---

Validating chord progressions in Roman Numeral Analysis

Measure 2: In a minor key, a chord on scale degree 7 must be diminished

Measure 2: In a minor key, a chord on scale degree 6 must be major

Measure 2: The sixth degree chord cannot appear in first inversion.

Measure 6: In a minor key, a chord on scale degree 7 must be diminished

Measure 7: In a minor key, a chord on scale degree 7 must be diminished

Measure 7: The seventh degree chord must appear in first inversion.

Measure 9: In a minor key, a chord on scale degree 5 must be major

Measure 2: Invalid progression from VII6 to III.

Measure 7: Invalid progression from VII to iv6.

Measure 9: Invalid progression from v to III.

Measure 12: In minor, V goes to iv6, not iv

Measure 12: Invalid progression from VI to III.

-----

-----Analysis Complete-----

Cals-MacBook-Pro:chassis calpeyser\$ python choraleAnalyzer.py TestChorales/ranges.xml TestChorales/ranges.txt

music21: Certain music21 functions might need these optional packages: matplotlib, numpy, scipy; if you run into errors, install it by following the instructions at <http://mit.edu/music21/doc/html/installAdditional.html>

-----Chorale Analyzer-----

Scanning for parallel unisons

No parallel unisons found

-----

Scanning for parallel octaves

No parallel octaves found

-----

Scanning for parallel fifths

Parallel fifth found!

Measure Number: 3

First Voice:

First Note: F in octave 4 Quarter Note

Second Note: E in octave 5 Dotted Quarter Note

Second Voice:

First Note: B-flat in octave 2 Quarter Note

Second Note: A in octave 2 Quarter Note



Parallel fifth found!

Measure Number: 7

First Voice:

First Note: G in octave 4 Quarter Note

Second Note: F in octave 4 Quarter Note

Second Voice:

First Note: C in octave 3 Quarter Note

Second Note: B-flat in octave 1 Quarter Note

---

Checking ranges

Soprano out of range in measure 2

Soprano out of range in measure 3

Alto out of range in measure 4

Alto out of range in measure 5

Tenor out of range in measure 6

Bass out of range in measure 7

Bass out of range in measure 8

---

Checking distances between parts

Interval between alto and soprano greater than an octave in measure 3

Interval between alto and soprano greater than an octave in measure 3

Interval between alto and soprano greater than an octave in measure 3

Interval between alto and soprano greater than an octave in measure 3

Interval between tenor and alto greater than an octave in measure 4

Interval between tenor and alto greater than an octave in measure 4

Interval between alto and soprano greater than an octave in measure 5

---

Checking for voice crossings

Alto/Soprano voice crossing in measure 2

Alto/Soprano voice crossing in measure 4

Alto/Soprano voice crossing in measure 4

Tenor/Alto voice crossing in measure 5

Tenor/Alto voice crossing in measure 6

Tenor/Alto voice crossing in measure 6

Bass/Tenor voice crossing in measure 8

Bass/Tenor voice crossing in measure 8

-----

Checking for tritones

No tritones found

-----

Checking for repeated bass notes

No repeated bass notes

-----

Verifying that Roman Numeral Analysis in fact reflects the score

No chord found for roman numeral in measure 0 beat 4

No chord found for roman numeral in measure 1 beat 1

No chord found for roman numeral in measure 1 beat 2

No chord found for roman numeral in measure 1 beat 3

No chord found for roman numeral in measure 2 beat 2.5

Incorrect chord root in measure 3 beat 2.5

No chord found for roman numeral in measure 3 beat 3.5

No chord found for roman numeral in measure 5 beat 2

No chord found for roman numeral in measure 5 beat 2.7

No chord found for roman numeral in measure 5 beat 3

No chord found for roman numeral in measure 5 beat 3.5

Incorrect chord root in measure 6 beat 1.5

No chord found for roman numeral in measure 7 beat 4.7

-----  
Validating chord progressions in Roman Numeral Analysis

Measure 7: In a major key, a chord on scale degree 2 must be minor

Measure 5: Invalid progression from ii6 to vi.  
-----

-----Analysis Complete-----

Cals-MacBook-Pro:chassis calpeyser\$ python choraleAnalyzer.py TestChorales/PartDist.xml  
TestChorales/PartDist.txt

music21: Certain music21 functions might need these optional packages: matplotlib, numpy, scipy; if you run into errors, install it by following the instructions at <http://mit.edu/music21/doc/html/installAdditional.html>

-----Chorale Analyzer-----

Scanning for parallel unisons

No parallel unisons found  
-----

Scanning for parallel octaves

No parallel octaves found  
-----

Scanning for parallel fifths

No parallel fifths found  
-----

Checking ranges

Bass out of range in measure 3

Alto out of range in measure 5

Bass out of range in measure 5

Bass out of range in measure 15  
-----

Checking distances between parts

Interval between alto and soprano greater than an octave in measure 1

Interval between tenor and alto greater than an octave in measure 2

Interval between tenor and alto greater than an octave in measure 2

Interval between alto and soprano greater than an octave in measure 3

Interval between tenor and alto greater than an octave in measure 5

Interval between alto and soprano greater than an octave in measure 6

Interval between tenor and alto greater than an octave in measure 7

Interval between tenor and alto greater than an octave in measure 7

-----

Checking for voice crossings

Tenor/Alto voice crossing in measure 3

Alto/Soprano voice crossing in measure 5

Tenor/Alto voice crossing in measure 6

Alto/Soprano voice crossing in measure 7

-----

Checking for tritones

Tenor makes tritone in measure 6

-----

Checking for repeated bass notes

No repeated bass notes

-----

Verifying that Roman Numeral Analysis in fact reflects the score

Incorrect chord root in measure 7 beat 2.5

-----

Validating chord progressions in Roman Numeral Analysis

Measure 6: In a minor key, a chord on scale degree 2 must be diminished

Measure 6: In a minor key, the second degree chord must appear in first inversion

Measure 7: In a minor key, a chord on scale degree 5 must be major

Measure 11: The seventh degree chord must appear in first inversion.

Measure 12: Invalid progression from V to iii.

-----

-----Analysis Complete-----

Cals-MacBook-Pro:chassis calpeyser\$ python choraleAnalyzer.py TestChorales/VoiceCrossing.xml  
TestChorales/VoiceCrossing.txt

music21: Certain music21 functions might need these optional packages: matplotlib, numpy, scipy; if you run into errors, install it by following the instructions at <http://mit.edu/music21/doc/html/installAdditional.html>

-----Chorale Analyzer-----

Scanning for parallel unisons

No parallel unisons found

-----

Scanning for parallel octaves

No parallel octaves found

-----

Scanning for parallel fifths

Parallel fifth found!

Measure Number: 7

First Voice:

First Note: E in octave 4 Quarter Note

Second Note: F-sharp in octave 4 Eighth Note

Second Voice:

First Note: A in octave 2 Quarter Note

Second Note: B in octave 3 Eighth Note

-----

Checking ranges

Tenor out of range in measure 2

Tenor out of range in measure 4

Tenor out of range in measure 5

Bass out of range in measure 8

Bass out of range in measure 9

-----

Checking distances between parts

Interval between tenor and alto greater than an octave in measure 6

---

Checking for voice crossings

Tenor/Alto voice crossing in measure 2

Tenor/Alto voice crossing in measure 2

Alto/Soprano voice crossing in measure 3

Alto/Soprano voice crossing in measure 3

Tenor/Alto voice crossing in measure 4

Tenor/Alto voice crossing in measure 4

Tenor/Alto voice crossing in measure 5

Tenor/Alto voice crossing in measure 5

Alto/Soprano voice crossing in measure 6

Bass/Tenor voice crossing in measure 8

Bass/Tenor voice crossing in measure 8

Bass/Tenor voice crossing in measure 9

Bass/Tenor voice crossing in measure 9

---

Checking for tritones

No tritones found

---

Checking for repeated bass notes

No repeated bass notes

---

Verifying that Roman Numeral Analysis in fact reflects the score

No chord found for roman numeral in measure 0 beat 4

No chord found for roman numeral in measure 1 beat 1

No chord found for roman numeral in measure 1 beat 2

No chord found for roman numeral in measure 1 beat 3

No chord found for roman numeral in measure 1 beat 3.5

Incorrect chord root in measure 3 beat 4.5

Incorrect chord root in measure 5 beat 2.5

No chord found for roman numeral in measure 5 beat 3.5

No chord found for roman numeral in measure 6 beat 1.5

Incorrect inversion in measure 6 beat 2.5

Incorrect chord root in measure 6 beat 4.5

Incorrect chord root in measure 7 beat 1.5

Incorrect inversion in measure 7 beat 2.5

No chord found for roman numeral in measure 7 beat 3.5

Incorrect chord root in measure 7 beat 4.5

Incorrect chord root in measure 8 beat 1.5

Incorrect chord root in measure 8 beat 2.5

-----

#### Validating chord progressions in Roman Numeral Analysis

Measure 1: The seventh degree chord must appear in first inversion.

Measure 7: Invalid progression from IV to iii6.

-----

-----Analysis Complete-----

Cals-MacBook-Pro:chassis calpeyser\$ python choraleAnalyzer.py TestChorales/harmony.xml  
TestChorales/harmony.txt NoParallels

music21: Certain music21 functions might need these optional packages: matplotlib, numpy, scipy; if you run into errors, install it by following the instructions at <http://mit.edu/music21/doc/html/installAdditional.html>

-----Chorale Analyzer-----

#### Checking ranges

Bass out of range in measure 1

Bass out of range in measure 3

Bass out of range in measure 6

Bass out of range in measure 8

-----

Checking distances between parts

Interval between alto and soprano greater than an octave in measure 11

Interval between alto and soprano greater than an octave in measure 11

---

Checking for voice crossings

Bass/Tenor voice crossing in measure 3

Tenor/Alto voice crossing in measure 12

Tenor/Alto voice crossing in measure 12

---

Checking for tritones

No tritones found

---

Checking for repeated bass notes

No repeated bass notes

---

Verifying that Roman Numeral Analysis in fact reflects the score

Incorrect dominant seventh identification, or some part of dominant seventh chord not present in measure 2 beat 1.5

Incorrect dominant seventh identification, or some part of dominant seventh chord not present in measure 3 beat 2.5

Incorrect dominant seventh identification, or some part of dominant seventh chord not present in measure 6 beat 3.5

Incorrect dominant seventh identification, or some part of dominant seventh chord not present in measure 9 beat 3.5

Incorrect chord root in measure 11 beat 2.5

---

Validating chord progressions in Roman Numeral Analysis

Measure 0: In a minor key, a chord on scale degree 1 must be minor

Measure 1: In a minor key, a chord on scale degree 6 must be major

Measure 2: In a minor key, a chord on scale degree 2 must be diminished

Measure 2: In a minor key, the second degree chord must appear in first inversion



Measure 3: In a minor key, a chord on scale degree 2 must be diminished

Measure 3: In a minor key, the second degree chord must appear in first inversion

Measure 4: The seventh degree chord must appear in first inversion.

Measure 4: In a minor key, a chord on scale degree 6 must be major

Measure 4: The sixth degree chord cannot appear in first inversion.

Measure 5: In a major key, a chord on scale degree 7 must be diminished

Measure 5: The seventh degree chord must appear in first inversion.

Measure 6: In a major key, a chord on scale degree 2 must be minor

Measure 8: The seventh degree chord must appear in first inversion.

Measure 10: The seventh degree chord must appear in first inversion.

Measure 12: In a minor key, a chord on scale degree 7 must be diminished

Measure 12: The seventh degree chord must appear in first inversion.

Measure 12: The seventh degree chord must appear in first inversion.

Measure 13: In a minor key, the second degree chord must appear in first inversion

Measure 13: In a minor key, a chord on scale degree 1 must be minor

Measure 2: Invalid progression from V7 to ii.

Measure 3: In minor, V goes to IV6, not iv6 or IV

Measure 3: Invalid progression from V7 to ii.

Measure 3: Invalid progression from ii to i6.

Measure 4: Invalid progression from viio to vi6.

Measure 7: Invalid progression from IV to vi.

Measure 8: Invalid progression from V/V to i6.

Measure 11: Invalid progression from V/V to V/II.

Measure 11: Invalid progression from V/II to i.

-----

-----Analysis Complete-----

## Appendix E: Code

This section contains the Python code used to implement the chorale checker. This code will be made available online.

### LEVEL 1 MODULES

#### choraleAnalyzer.py

```
import os
import sys
from music21 import *
from checkParallels import *
from checkZeroOrder import *
from checkVoiceLeading import *
from checkHarmony import *
from utilities import *
import subprocess, glob

# there must be at least one command line argument
if len(sys.argv) < 2:
    print "Required input: musicXML file."
    exit();

print("-----Chorale Analyzer-----");

# list for error storage
assignList();

# parse and store MusicXML file
chorale = converter.parse(sys.argv[1]);
```

```
# parse and store Roman Numeral Analysis
harmonyFlag = False;
if len(sys.argv) > 2:
    rntext = converter.parse(sys.argv[2]);
    harmonyFlag = True;

parallelsFlag = True;
if len(sys.argv) == 4:
    parallelsFlag = False;

if parallelsFlag:
    checkParallels(chorale)

checkZeroOrder(chorale);
checkVoiceLeading(chorale);
if (harmonyFlag == True):
    checkHarmony(chorale, rntext);

# read output
errorTracker = errorTracker();
# utilities.py has already taken care of writing to 'errors'.
# now we read
outputLines = retrieveList();
for line in outputLines:
    errorTracker.processError(line);

# Count measures
chordification = chorale.chordify();
measureCounter = 0;
```

```

for i in range(len(chordification)):
    if str(type(chordification[i])) == "<class 'music21.stream.Measure'>":
        measureCounter += 1;

# compute and print Bachness
print("Bachness Score: " +
str(errorTracker.computeBachness(measureCounter)));

print("-----Analysis Complete-----");

```

## LEVEL 2 MODULES

### checkParallels.py

```

# Function checks for parallels by calling relevant functions.
# Prints

from checkParallelFifths import *
from checkParallelOctaves import *
from checkParallelUnisons import *
from utilities import printErrors

def checkParallels(chorale):

    # find parallel unisons

    printErrors(checkParallelUnisons, chorale, "Scanning for parallel
unisons", "No parallel unisons found");

    # find parallel octaves

    printErrors(checkParallelOctaves, chorale, "Scanning for parallel
octaves", "No parallel octaves found");

    # find parallel fifths

```

```
    printErrors(checkParallelFifths, chorale, "Scanning for parallel fifths",
"No parallel fifths found");
```

### **checkZeroOrder.py**

```
# Function performs zero-th order analysis. Checks each chord for internal
# violations of the style.
```

```
from checkRanges import *
from checkCloseness import *
from checkVoiceCrossing import *
from utilities import printErrors
```

```
def checkZeroOrder(chorale):
```

```
    # slices is a list of VerticalSlice objects, each of which
    # contains one chord of the chorale. Note that in each slice,
    # object, the parts are numbered as follows: 0-soprano, 1-alto,
    # 2-tenor, 3-bass
```

```
    slices = theoryAnalysis.theoryAnalyzer.getVerticalSlices(chorale);
```

```
    # check ranges
```

```
    printErrors(checkRanges, slices, "Checking ranges", "All parts in
range");
```

```
    # check part distances
```

```
    printErrors(checkCloseness, slices, "Checking distances between parts",
"All parts sufficiently close together");
```

```
    # check voice crossings
```

```
    printErrors(checkVoiceCrossing, slices, "Checking for voice crossings",
"No voice crossings found");
```

### **checkVoiceLeading.py**

```
# Function performs voice leading analysis. Checks each chord for internal
# violations of the style.
```

```

from checkTritones import *
from checkRepeatedBass import *
from utilities import printErrors

def checkVoiceLeading(chorale):

    chordification = chorale.chordify();
    slices = theoryAnalysis.theoryAnalyzer.getVerticalSlices(chorale);

    # check for tritones
    printErrors(checkTritones, slices, "Checking for tritones", "No tritones
found");

    # check for repeated bass tones
    printErrors(checkRepeatedBass, chordification, "Checking for repeated
bass notes", "No repeated bass notes");

    # could add check2leap, checkDimInt

```

### **checkHarmony.py**

```

# Function performs basic harmonic analysis. Checks each chord for internal
# violations of the style.
from music21 import *
from verifyRNA import verifyRNA
from checkProgression import *
from utilities import printErrors

def checkHarmony(chorale, rntext):

    # slices is a list of VerticalSlice objects, each of which
    # contains one chord of the chorale. Note that in each slice,
    # object, the parts are numbered as follows: 0-soprano, 1-alto,
    # 2-tenor, 3-bass

```

```

slices = theoryAnalysis.theoryAnalyzer.getVerticalSlices(chorale);

# roman is a stream object which contains the roman numeral analysis
# inputed by the student
rom = rntext[1];

choraleAndRoman = [chorale, rom];

printErrors(verifyRNA, choraleAndRoman, "Verifying that Roman Numeral
Analysis in fact reflects the score", "Roman Numeral Analysis appears to be
correct");

printErrors(checkProgression, rom, "Validating chord progressions in
Roman Numeral Analysis", "Chord Progression Valid");

```

### LEVEL 3 MODULES

#### checkParallelUnisons.py

```

# Funicton checks for parallel unisons. Returns an error message to
# be printed, if any.
from music21 import *

def checkParallelUnisons(chorale):
    # output is a list of strings which give error messages
    output = [];

    # identifyParallelUnisons() is a music21 method which stores all
    # parallel unisons in VLQTheoryResult objects
    theoryAnalysis.theoryAnalyzer.identifyParallelUnisons(chorale);

    # get list, put in parallelUnisons
    parallelUnisons = chorale.analysisData['ResultDict']['parallelUnisons'];

    # iterate through list, produce error message for each

```

```

# VLQTheoryResult object. Note: A VLQTheoryResult object
# contains a VoiceLeadingQuartet object, which can be used like
# in other methods.
if not parallelUnisons:
    return output;

for unison in parallelUnisons:
    output.append('Parallel unison found!' +
                  '\n Measure Number: ' +
str(unison.vlq.v1n1.measureNumber) +
                  '\n First Voice: ' +
+
                  '\n    First Note: ' + str(unison.vlq.v1n1.fullName)
+
                  '\n    Second Note: ' + str(unison.vlq.v1n2.fullName)
+
                  '\n Second Voice: ' +
+
                  '\n    First Note: ' + str(unison.vlq.v2n1.fullName)
+
                  '\n    Second Note: ' + str(unison.vlq.v2n2.fullName)
+
                  '\n')
    return output;

```

### **checkParallelFifths.py**

```

# Function checks for parallel fifths. Returns an error message to
# be printed, if any.
from music21 import *

def checkParallelFifths(chorale):
    # output is a list of strings which give error messages
    output = [];

    # parallelFifths is a list of VoiceLeadingQuartet objects, each
    # which gives a pair of intervals in illegal relationship to

```



```

    # eachother.

    parallelFifths =
theoryAnalysis.theoryAnalyzer.getParallelFifths(chorale);

    # iterate through list, produce error message for each
VoiceLeadingQuartet
    # object.

    if not parallelFifths:
        return output;

    for fifth in parallelFifths:
        output.append('Parallel fifth found!' +
            '\n Measure Number: ' + str(fifth.v1n1.measureNumber) +
            '\n First Voice: ' +
            '\n     First Note: ' + str(fifth.v1n1.fullName) +
            '\n     Second Note: ' + str(fifth.v1n2.fullName) +
            '\n Second Voice: ' +
            '\n     First Note: ' + str(fifth.v2n1.fullName) +
            '\n     Second Note: ' + str(fifth.v2n2.fullName) +
            '\n')

    return output;

```

### **checkParallelOctaves.py**

```

# Function checks for parallel octaves. Returns an error message to
# be printed, if any.
from music21 import *

def checkParallelOctaves(chorale):
    # output is a list of strings which give error messages
    output = [];

    # parallelFifths is a list of VoiceLeadingQuartet objects, each

```

```

    # which gives a pair of intervals in illegal relationship to
    # eachother.

    parallelOctaves =
theoryAnalysis.theoryAnalyzer.getParallelOctaves(chorale);

    # iterate through list, produce error message for each
VoiceLeadingQuartet
    # object.

if not parallelOctaves:
    return output;

for octave in parallelOctaves:
    output.append('Parallel octave found!' +
+
        '\n Measure Number: ' + str(octave.v1n1.measureNumber)
+
        '\n First Voice: ' +
        '\n     First Note: ' + str(octave.v1n1.fullName) +
        '\n     Second Note: ' + str(octave.v1n2.fullName) +
        '\n Second Voice: ' +
        '\n     First Note: ' + str(octave.v2n1.fullName) +
        '\n     Second Note: ' + str(octave.v2n2.fullName) +
        '\n')

return output;

```

### **checkTritones.py**

```

# Funciton checks for tritones. Returns an error message to
# be printed, if any.
from music21 import *

intA = interval.Interval('A4');

```

```
intB = interval.Interval('D5');
intC = interval.Interval('A-4');
intD = interval.Interval('D-5');
intervalList = [intA, intB, intC, intD];

def failedSopranoTritone(firstChord, secondChord):
    if interval.Interval(firstChord[0], secondChord[0]) in intervalList:
        return True;
    else:
        return False;

def failedAltoTritone(firstChord, secondChord):
    if interval.Interval(firstChord[1], secondChord[1]) in intervalList:
        return True;
    else:
        return False;

def failedTenorTritone(firstChord, secondChord):
    if interval.Interval(firstChord[2], secondChord[2]) in intervalList:
        return True;
    else:
        return False;

def failedBassTritone(firstChord, secondChord):
    if interval.Interval(firstChord[3], secondChord[3]) in intervalList:
        return True;
    else:
```

```

        return False;

def checkTritones(slices):
    # output is a list of strings which give error messages
    output = [];

    # iterate until second to last chord. Check chord with subsequent chord
    for count in range(0, len(slices) - 2):
        firstChord = slices[count].getChord();
        secondChord = slices[count + 1].getChord();

        if (failedSopranoTritone(firstChord, secondChord)):
            output.append('Soprano makes tritone in measure ' +
                str(firstChord[0].measureNumber));

        if (failedAltoTritone(firstChord, secondChord)):
            output.append('Alto makes tritone in measure ' +
                str(firstChord[1].measureNumber));

        if (failedTenorTritone(firstChord, secondChord)):
            output.append('Tenor makes tritone in measure ' +
                str(firstChord[2].measureNumber));

        if (failedBassTritone(firstChord, secondChord)):
            output.append('Bass makes tritone in measure ' +
                str(firstChord[3].measureNumber));

    return output;

```

### **checkRepeatedBass.py**

```

# Function checks for repeated bass notes. Returns an error message to
# be printed, if any.
from music21 import *

def checkRepeatedBass(chordification):

```

```

# output is a list of strings which give error messages
output = [];

# We create a list of measures. While were at it, find time
# signature
measures = [];
for i in range(len(chordification)):
    if str(type(chordification[i])) == "<class
'music21.stream.Measure'>":
        measures.append(chordification[i]);
timeSignature = measures[1].bestTimeSignature().numerator;

# We now create a list of notes, maintaining measureNum
# as a property. Also maintain beat in bt property.
bassNotes = [];
for m in range(len(measures)):
    for n in range(len(measures[m])):
        if str(type(measures[m][n])) == "<class 'music21.chord.Chord'>":
            bassNote = measures[m][n][-1];
            bassNote.measureNum = m;
            bassNote.bt = measures[m][n].beat + timeSignature*m;
            bassNotes.append(bassNote);

# Go note to note, and identify repeated bass notes
for i in range(len(bassNotes) - 1):
    firstNote = bassNotes[i];
    secondNote = bassNotes[i+1];
    if firstNote.pitch == secondNote.pitch:
        if firstNote.bt + firstNote.duration.quarterLength <
secondNote.bt:

```

```
        output.append("Repeated bass note (" + str(secondNote.pitch)
+ ") in measure " + str(secondNote.measureNum));
```

```
    return output;
```

### **checkRanges.py**

```
# Verifies that each part is inside of expected range
```

```
from music21 import *
```

```
from utilities import removeDuplicates
```

```
def failedSopranoRange(chord):
```

```
    if chord[0].pitch > pitch.Pitch('g5'):
```

```
        return True;
```

```
    elif chord[0].pitch < pitch.Pitch('c4'):
```

```
        return True;
```

```
    else:
```

```
        return False;
```

```
def failedAltoRange(chord):
```

```
    if chord[1].pitch > pitch.Pitch('d5'):
```

```
        return True;
```

```
    elif chord[1].pitch < pitch.Pitch('g3'):
```

```
        return True;
```

```
    else:
```

```
        return False;
```

```
def failedTenorRange(chord):
```

```
    if chord[2].pitch > pitch.Pitch('g4'):
```

```
        return True;
```

```

elif chord[2].pitch < pitch.Pitch('c3'):
    return True;
else:
    return False;

def failedBassRange(chord):
    if chord[3].pitch > pitch.Pitch('c4'):
        return True;
    elif chord[3].pitch < pitch.Pitch('f2'):
        return True;
    else:
        return False;

def checkRanges(slices):
    # output is a list of strings which give error messages
    output = [];

    # since VerticalSlice is not represented in a measure, use
    # bass note to determine measure
    for slice in slices:
        chord = slice.getChord();
        if (failedSopranoRange(chord)):
            output.append('Soprano out of range in measure ' +
                          str(chord[3].measureNumber));
        if (failedAltoRange(chord)):
            output.append('Alto out of range in measure ' +
                          str(chord[3].measureNumber));
        if (failedTenorRange(chord)):
            output.append('Tenor out of range in measure ' +

```

```

        str(chord[3].measureNumber));
    if (failedBassRange(chord)):
        output.append('Bass out of range in measure ' +
            str(chord[3].measureNumber));

    return removeDuplicates(output);

```

### **checkCloseness.py**

# Verifies that distance between parts is not too large

```
from music21 import *
```

# Distance between bass and soprano cannot be greater than

# a 12th

```
#def failedTenorSopranoDistance(chord):
```

```
#     if interval.Interval(chord[3], chord[1]).cents >
interval.Interval('P12').cents:
```

```
#         return True;
```

```
#     else:
```

```
#         return False;
```

# Distance between alto and soprano cannot be greater than an octave

```
def failedAltoSopranoDistance(chord):
```

```
    if interval.Interval(chord[1], chord[0]).cents >
interval.Interval('P8').cents:
```

```
        return True;
```

```
    else:
```

```
        return False;
```

```
def failedTenorAltoDistance(chord):
```

```
    if interval.Interval(chord[2], chord[1]).cents >
interval.Interval('P8').cents:
```



```

        return True;
    else:
        return False;

def checkCloseness(slices):
    # output is a list of strings which give error messages
    output = [];

    # since VerticalSlice is not represented in a measure, use
    # chord notes to determine measure
    for slice in slices:
        chord = slice.getChord();
        #         if (failedTenorSopranoDistance(chord)):
        #             output.append('Interval between tenor and soprano greater than a
12th in measure ' + str(chord[3].measureNumber));
        if (failedAltoSopranoDistance(chord)):
            output.append('Interval between alto and soprano greater than an
octave in measure ' + str(chord[1].measureNumber));
        if (failedTenorAltoDistance(chord)):
            output.append('Interval between tenor and alto greater than an
octave in measure ' + str(chord[2].measureNumber));

    return output;

```

### **checkVoiceCrossing.py**

```

# Verifies that there are no voice crossings

```

```

from music21 import *

```

```

def failedBassTenorCross(chord):
    if chord[3].pitch > chord[2].pitch:
        return True;
    else:

```



```
    if (failedAltoSopranoCross(chord)):
        output.append('Alto/Soprano voice crossing in measure '
                      + str(chord[1].measureNumber));

return output;
```

### **verifyRNA.py**

```
# Verifies correctness of a roman numeral analysis
```

```
from music21 import *
```

```
def compareRomanToChord(rom, ch):
```

```
    if (ch.root().name != rom.root().name):
```

```
        return "Incorrect chord root";
```

```
    elif (ch.inversion() != rom.inversion()):
```

```
        return "Incorrect inversion";
```

```
    elif (ch.quality != rom.quality and ch.quality != 'other'):
```

```
        return "Incorrect chord quality";
```

```
    elif (ch.isDominantSeventh() != rom.isDominantSeventh()):
```

```
        return "Incorrect dominant seventh identification, or some part of  
dominant seventh chord not present";
```

```
    else:
```

```
        return "No error";
```

```
def verifyRNA(choraleAndNum):
```

```
    # output is a list of strings which give error messages
```

```
    output = [];
```

```
    # decompose input
```

```
    chorale = choraleAndNum[0];
```

```
    num = choraleAndNum[1];
```

```

# obtain chord breakdown
chordification = chorale.chordify();

# obtain the type of a RomanNumeral object, for type checking
SampleRomanNumeral = roman.RomanNumeral('I');
TypeRomanNumeral = type(SampleRomanNumeral);

# obtain the type of a chord object, for type checking
TypeChord = chord.Chord;

# create numeral list as follows:
# 0: measure.beat as number
# 1: RomanNumeral object
numList = [];
for omeasure in range(len(num)):
    for romanNumeral in range(len(num[omeasure])):
        if type(num[omeasure][romanNumeral]) == TypeRomanNumeral:
            entry = [];
            time = num[omeasure][romanNumeral].measureNumber +
(0.1)*num[omeasure][romanNumeral].beat;
            time = round(time, 4);
            entry.append(time);
            entry.append(num[omeasure][romanNumeral]);
            numList.append(entry);

# create chord list as follows
# 0: measure.beat as number
# 1: Chord object
chordList = [];
for omeasure in range(len(chordification))[1:]:

```

```

for ochord in range(len(chordification[omeasure])):
    if type(chordification[omeasure][ochord]) == TypeChord:
        entry = [];
        time = chordification[omeasure][ochord].measureNumber +
(0.1)*chordification[omeasure][ochord].beat;
        # round to cutoff extra decimal values
        time = round(time, 4);
        # add to chordList
        entry.append(time);
        entry.append(chordification[omeasure][ochord]);
        chordList.append(entry);

# I define a correct roman numeral analysis as one in which each
# numeral lines up with an appropriate chord. Verify by going
# by going through numList. Use a boolean to make sure there
# is a chord for each roman numeral
for i in range(len(numList)):
    numeralPair = numList[i]
    foundChord = False;
    for j in range(len(chordList)):
        chordPair = chordList[j];
        if (chordPair[0] == numeralPair[0]):
            foundChord = True;
            # One chord case:
            if (j == len(chordList) - 1 or i == len(numList) - 1 or
numList[i+1][0] <= chordList[j+1][0]):
                err = compareRomanToChord(numeralPair[1], chordPair[1]);
                if (err != "No error"):
                    spl = str(chordPair[0]).split('.');
                    if len(spl[1]) > 1:
                        spl[1] = spl[1][0] + "." + spl[1][1];

```

```

        output.append(err + " in measure " + spl[0] + "
beat " + spl[1]);

        # Two chord case:
        else:
            err = compareRomanToChord(numeralPair[1], chordPair[1]);
            err2 = compareRomanToChord(numeralPair[1],
chordList[j+1][1]);
            if (err != "No error" and err2 != "No error"):
                spl = str(chordPair[0]).split('.');
                if len(spl[1]) > 1:
                    spl[1] = spl[1][0] + "." + spl[1][1];
                    output.append(err + " in measure " + spl[0] + "
beat " + spl[1]);

            if (foundChord == False):
                spl = str(numeralPair[0]).split('.');
                if len(spl[1]) > 1:
                    spl[1] = spl[1][0] + "." + spl[1][1];
                output.append("No chord found for roman numeral in measure " +
spl[0] + " beat " + spl[1]);

    return output;

```

### **checkProgression.py**

```

# Funciton checks for repeated bass notes. Returns an error message to
# be printed, if any.
from music21 import *
from progressionChecker import *

def checkProgressionPairs(rom, dim = 2):
    # output is a list of strings which give error messages
    output = [];

```

```

# rom1D is a one dimensional list of roman numerals
if (dim == 1):
    rom1D = rom;

if (dim == 2):
    rom1D = [];
    for x in range(len(rom)):
        for y in range(len(rom[x])):
            if type(rom[x][y]) == roman.RomanNumeral:
                rom1D.append(rom[x][y]);

# create a progressionChecker object with the right starting mode
checker = progressionChecker(rom1D[0].key.mode);

# Pairs of chords: iterate until second to last chord.
for i in range(len(rom1D) - 1):
    # if modulation, recursively call checkProgression
    if (rom1D[i].key != rom1D[i+1].key):
        return output + checkProgressionPairs(rom1D[i+1:], 1);
    else:
        checker.setChords(rom1D[i], rom1D[i+1]);
        err = checker.isValid()
        if (len(err) != 0):
            for errorString in err:
                output.append(errorString);

return output;

```

```

def checkProgressionSingles(rom, dim = 2):
    # output is a list of strings which give error messages
    output = [];

    # rom1D is a one dimensional list of roman numerals
    if (dim == 1):
        rom1D = rom;

    if (dim == 2):
        rom1D = [];
        for x in range(len(rom)):
            for y in range(len(rom[x])):
                if type(rom[x][y]) == roman.RomanNumeral:
                    rom1D.append(rom[x][y]);

    # create a progressionChecker object with the right starting mode
    checker = progressionChecker(rom1D[0].key.mode);

    # Pairs of chords: iterate until second to last chord.
    for i in range(len(rom1D)):
        # if modulation, recursively call checkProgression
        if (i < len(rom1D) - 1):
            if (rom1D[i].key != rom1D[i+1].key):
                return output + checkProgressionSingles(rom1D[i+1:], 1);

```



```

        checker.setChord(rom1D[i]);
        err = checker.isValid()
        if (len(err) != 0):
            for errorString in err:
                output.append(errorString);

    return output;

```

```

# return errors in roman numeral analysis. Covers only those errors which
# can be determined from the roman numeral analysis itself.

```

```

def checkProgression(rom):
    return checkProgressionSingles(rom) + checkProgressionPairs(rom);

```

### **progressionChecker.py**

```

# A progression checker contains the data for harmonic rules, and provides
# methods for validating chord sequences.

```

```

from music21 import *

```

```

class progressionChecker:

```

```

    MIdest = [True , True , True , True , True , True , True ];
    Miidest = [False, True , False, False, True , False, True ];
    Miiidest = [True , False, True , True , True , True , False];
    MIVdest = [True , True , False, True , True , False, True ];
    MVdest = [True , False, False, True , True , True , False];
    Mvidest = [True , True , False, True , True , True , True ];
    Mviidest = [True , False, False, False, True , False, True ];

```

```

midest = [True , True , True , True , True , True , True ];
miidest = [False, True , False, False, True , False, True ];
mIIIdest = [True , False, True , True , True , True , False];
mivdest = [True , True , False, True , True , False, True ];
mVdest = [True , False, False, True , True , True , False];
mVIdest = [True , True , False, True , True , True , True ];
mviidest = [True , False, False, False, True , False, True ];

# gives the appropriate modality for each scale degree

Mmodalities = ['major', 'minor', 'minor', 'major', 'major', 'minor',
'diminished'];

mmodalities = ['minor', 'diminished', 'major', 'either', 'major',
'major', 'diminished']

# we first define a couple of simple checks. These in particular
# return strings, not booleans. Some are mode specific.

# One chord checks

def modalityErr_Major(self):

    if self.num.quality != self.Mmodalities[self.num.scaleDegree - 1] and
self.Mmodalities[self.num.scaleDegree - 1] != 'either':

        return ('Measure ' + str(self.num.measureNumber) + ': In a major
key, a chord on scale degree ' + str(self.num.scaleDegree) + ' must be ' +
str(self.Mmodalities[self.num.scaleDegree - 1]));

def modalityErr_Minor(self):

    if self.num.quality != self.mmodalities[self.num.scaleDegree - 1] and
self.mmodalities[self.num.scaleDegree - 1] != 'either':

        return ('Measure ' + str(self.num.measureNumber) + ': In a minor
key, a chord on scale degree ' + str(self.num.scaleDegree) + ' must be ' +
str(self.mmodalities[self.num.scaleDegree - 1]));

def twoInversion_Minor(self):

```

```

    if self.num.scaleDegree == 2:
        if self.num.inversion() != 1:
            return('Measure ' + str(self.num.measureNumber) + ': In a
minor key,the second degree chord must appear in first inversion');

def sixInversion_MajorMinor(self):
    if self.num.scaleDegree == 6:
        if self.num.inversion() == 1:
            return('Measure ' + str(self.num.measureNumber) + ': The
sixth degree chord cannot appear in first inversion. ');

def sevenInversion_MajorMinor(self):
    if self.num.scaleDegree == 7:
        if self.num.inversion() != 1:
            return('Measure ' + str(self.num.measureNumber) + ': The
seventh degree chord must appear in first inversion. ');

# Two chord checks
def sixToOne_MajorMinor(self):
    if self.num1.scaleDegree == 6 and self.num2.scaleDegree == 1 and not
self.of1 and not self.of2:
        if self.num2.inversion() != 1:
            return ('Measure ' + str(self.num2.measureNumber) + ': When
vi resolves to the tonic, it must be I6');

def fiveTo_Minor(self):
    if self.num1.scaleDegree == 5 and not self.of1:
        if self.num2.scaleDegree == 4 and not self.of2:
            if self.num2.quality == 'minor' or self.num2.inversion != 1:
                return ('Measure ' + str(self.num2.measureNumber) + ': In
minor, V goes to IV6, not iv6 or IV');

```

```

        if self.num2.scaleDegree == 6 and not self.of2:
            if self.num2.inversion() != 1:
                return ('Measure ' + str(self.num2.measureNumber) + ': In
minor, V goes to vi6, not vi');

def pairValid_MajorMinor(self):
    out = True;
    # Check each of four cases
    # Case 1: Diatonic chord to diatonic chord
    if not self.of1 and not self.of2:
        out = self.dest[self.base1.scaleDegree -
1][self.base2.scaleDegree - 1];

    # Case 2: Diatonic chord to borrowed chord
    elif not self.of1 and self.of2 != None:
        if self.of2.quality == 'major' or self.of2.quality == 'minor':
            out = True;
        else:
            out = False;

    # Case 3: Borrowed chord to diatonic chord
    elif self.of1 != None and not self.of2:
        if self.of1.scaleDegree == self.base2.scaleDegree:
            if self.base1.scaleDegree == 5 or self.base1.scaleDegree ==
7:
                out = True;
            # deceptive progression
            elif (self.base1.scaleDegree + self.of1.scaleDegree ==
self.base2.scaleDegree + 7):
                out = True;
            else:
                out = False;

    # Case 4: Borrowed chord to borrowed chord

```

```

elif self.of1 != None and self.of2 != None:
    # Subcase 1: both chords are borrowed from the same key
    if self.of1.scaleDegree == self.of2.scaleDegree:
        # same chord
        if self.base1.scaleDegree == self.base2.scaleDegree:
            out = True;
        # preceded by own subdominant
        elif (self.base1.scaleDegree == 4 or self.base2.scaleDegree
== 2) and (self.base2.scaleDegree == 5 or self.base2.scaleDegree == 7):
            out = True;
        # deceptive progression
        elif self.base1.scaleDegree + self.of1.scaleDegree + 1 ==
self.base2.scaleDegree + self.of2.scaleDegree:
            out = True;
        else:
            out = False;
    # Subcase 2: the chords are borrowed from different keys
    else:
        out = False;

if out == False:
    return ('Measure ' + str(self.num1.measureNumber) + ': Invalid
progression from ' + str(self.num1.figure) + ' to ' + str(self.num2.figure) +
'.');

```

```

# there are two types of progressionCheckers - major and minor. They
# have an array of allowed simple progressions, and a list of mode
# specific checks to be run.

```

```

def __init__(self, mode):

```

```

    assert(mode == "major" or mode == "minor")

    self.mode = mode;

    if mode == "major":

        self.dest = [self.MIdest, self.Miideest, self.Miiideest,
self.MIVdest, self.MVdest, self.Mvidest, self.Mviideest];

        self.singleChecks = [self.modalityErr_Major,
self.sixInversion_MajorMinor, self.sevenInversion_MajorMinor];

        self.doubleChecks = [self.sixToOne_MajorMinor,
self.pairValid_MajorMinor];

    if mode == "minor":

        self.dest = [self.mideest, self.miideest, self.miiideest,
self.mivdest, self.mVdest, self.mVideest, self.mviideest];

        self.singleChecks = [self.modalityErr_Minor,
self.twoInversion_Minor, self.sixInversion_MajorMinor,
self.sevenInversion_MajorMinor];

        self.doubleChecks = [self.sixToOne_MajorMinor, self.fiveTo_Minor,
self.pairValid_MajorMinor];

# setChords is meant to set up the progressionChecker for use on a pair
# of chords, in order to verify the progression.  If isValid is called
# after a setChords call, the validity of the pair will be determined
def setChords(self, numA, numB):

    # tell the object that it is of type TwoChord
    self.checkerType = "TwoChord";

    # first we must store the roman numerals
    self.num1 = numA;
    self.num2 = numB;

    # now must parse into base and of
    if not self.num1.secondaryRomanNumeral:

```

```

        self.base1 = self.num1;
        self.of1 = None;
    else:
        breakdown = self.num1.figure.split("/") +
self.num1.secondaryRomanNumeral.figure);
        self.base1 = roman.RomanNumeral(breakdown[0]);
        self.of1 =
roman.RomanNumeral(self.num1.secondaryRomanNumeral.figure);
        if not self.num2.secondaryRomanNumeral:
            self.base2 = self.num2;
            self.of2 = None;
        else:
            breakdown = self.num2.figure.split("/") +
self.num2.secondaryRomanNumeral.figure);
            self.base2 = roman.RomanNumeral(breakdown[0]);
            self.of2 =
roman.RomanNumeral(self.num2.secondaryRomanNumeral.figure);

def setChord(self, num):

    # tell the object that it is of type OneChord
    self.checkerType = "OneChord";

    # first we must store the roman numeral
    self.num = num;

    # now must parse into base and of
    if not self.num.secondaryRomanNumeral:
        self.base = self.num;
        self.of = None;
    else:
        breakdown = self.num.figure.split("/") +
self.num.secondaryRomanNumeral.figure);

```

```

        self.base = roman.RomanNumeral(breakdown[0]);
        self.of =
roman.RomanNumeral(self.num.secondaryRomanNumeral.figure);

# tells if whatever was put into the progressionChecker is valid.
# If a single chord was put in, validates it. If two chords were put
# in, validates the transition.
def isValid(self):
    output = [];
    if self.checkerType == "OneChord":
        for function in self.singleChecks:
            err = function();
            if err != None:
                output.append(err);
    elif self.checkerType == "TwoChord":
        for function in self.doubleChecks:
            err = function();
            if err != None:
                output.append(err);

    return output;

```

## OTHER MODULES

### Utilites.py

```
# module contains utilities
```

```
# assign list of errors
```

```
def assignList():
    global errorList;
    errorList = [];
```

```
# get list of errors
```



```

def retrieveList():
    return errorList;

# removes duplicates from a list
def removeDuplicates(errors):
    seen = [];
    for error in errors:
        if error not in seen:
            seen.append(error);
    return seen;

# function takes as an argument a function which returns error messages.
# the messages are then printed
def printErrors(errorGenerator, arg, initialMes, clearMes):
    print(" " + initialMes);
    messages = errorGenerator(arg);
    if not messages:
        print(" " + clearMes);
    else:
        for error in messages:
            print(error);
            errorList.append(error);
    print("-----");

# An errorTracker is a data structure meant to store the types and number
# of errors occuring over several chorale checks.
class errorTracker:

```

```

def __init__(self):
    # Parallel Intervals
    self.P1 = 0;
    self.P5 = 0;
    self.P8 = 0;
    # Ranges
    self.bassRange = 0;
    self.tenorRange = 0;
    self.altoRange = 0;
    self.sopRange = 0;
    # Closeness
    self.altSopInt = 0;
    self.tenAltInt = 0;
    # Voice Crossing
    self.BTVC = 0;
    self.TAVC = 0;
    self.ASVC = 0;
    # Tritones
    self.tritone = 0;
    # Repeated Bass
    self.repBass = 0;
    # Chord progressions
    self.inversion = 0;
    self.majorMinor = 0;
    self.invalidProg = 0;

    self.bachness = [];

    # For each of the 17 error types, average violations per measure
    # in the chorales

```

```
self.bachViolations = [0.000994036, 0.21868787, 0.001988072,  
0.081510934, 0.005964215, 0.000994036, 0.000994036, 0.021868787, 0.07554672,  
0.040755467, 0.093439364, 0.028827038, 0.047713718, 0.034791252, 0.03777336,  
0.078528827, 0.093439364]
```

```
# The average violations per measure in this chorale
```

```
self.inputViolations = [];
```

```
def printState(self):
```

```
    print('P1: ' + str(self.P1));
```

```
    print('P5: ' + str(self.P5));
```

```
    print('P8: ' + str(self.P8));
```

```
    print('bassRange: ' + str(self.bassRange));
```

```
    print('tenorRange: ' + str(self.tenorRange));
```

```
    print('altoRange: ' + str(self.altoRange));
```

```
    print('sopRange: ' + str(self.sopRange));
```

```
    print('altSopInt: ' + str(self.altSopInt));
```

```
    print('tenAltInt: ' + str(self.tenAltInt));
```

```
    print('BTVC: ' + str(self.BTVC));
```

```
    print('TAVC: ' + str(self.TAVC));
```

```
    print('ASVC: ' + str(self.ASVC));
```

```
    print('tritone: ' + str(self.tritone));
```

```
    print('repBass: ' + str(self.repBass));
```

```
    print('inversion: ' + str(self.inversion));
```

```
    print('majorMinor: ' + str(self.majorMinor));
```

```
    print('invalidProg: ' + str(self.invalidProg));
```

```
    print self.bachness;
```

```
# takes an error and updates local variables. Works by basic
```

```
# string manipulation
```

```
def processError(self, err):
```

```
    sp = err.split();
```

```
if len(sp) < 3:
    return
firstWord = sp[0];
secondWord = sp[1];
thirdWord = sp[2];
lastWord = sp[-1];
# Parallel Unison
if secondWord == 'unison':
    self.P1 = self.P1 + 1;
# Parallel Fifth
elif secondWord == 'fifth':
    self.P5 = self.P5 + 1;
# Parallel Octave
elif secondWord == 'octave':
    self.P8 = self.P8 + 1;
# Bass Range
elif firstWord == 'Bass' and secondWord == 'out':
    self.bassRange = self.bassRange + 1;
# Tenor Range
elif firstWord == 'Tenor' and secondWord == 'out':
    self.tenorRange = self.tenorRange + 1;
# Alto Range
elif firstWord == 'Alto' and secondWord == 'out':
    self.altoRange = self.altoRange + 1;
# Soprano Range
elif firstWord == 'Soprano' and secondWord == 'out':
    self.sopranoRange = self.sopranoRange + 1;
# Alto Soprano Closeness
elif firstWord == 'Interval' and thirdWord == 'alto':
    self.altSopInt = self.altSopInt + 1;
```

```
# Tenor Alto Closeness
elif firstWord == 'Interval' and thirdWord == 'tenor':
    self.tenAltInt = self.tenAltInt + 1;

# Bass Tenor Voice Crossing
elif firstWord == 'Bass/Tenor':
    self.BTVC = self.BTVC + 1;

# Tenor Alto Voice Crossing
elif firstWord == 'Tenor/Alto':
    self.TAVC = self.TAVC + 1;

# Alto Soprano Voice Crossing
elif firstWord == 'Alto/Soprano':
    self.ASVC = self.ASVC + 1;

# Tritone
elif thirdWord == 'tritone':
    self.tritone = self.tritone + 1;

# Repeated Bass
elif firstWord == 'Repeated':
    self.repBass = self.repBass + 1;

# Inversion
elif lastWord == 'inversion':
    self.inversion = self.inversion + 1;
elif lastWord == 'I6':
    self.inversion = self.inversion + 1;

# Major/Minor
elif lastWord == 'major' or lastWord == 'minor':
    self.majorMinor = self.majorMinor + 1;

# Invalid progression
elif thirdWord == 'Invalid':
    self.invalidProg = self.invalidProg + 1;
elif firstWord == 'Bachness':
```

```

        self.bachness.append(thirdWord);

def computeBachness(self, numberOfMeasures):
    self.inputViolations = [self.P1, self.P5, self.P8, self.bassRange,
self.tenorRange, self.altoRange, self.sopRange, self.altSopInt,
self.tenAltInt, self.BTVC, self.TAVC, self.ASVC, self.tritone, self.repBass,
self.inversion, self.majorMinor, self.invalidProg];

    for i in range(len(self.inputViolations)):
        self.inputViolations[i] =
float(self.inputViolations[i])/numberOfMeasures;

    ratios = [];

    for i in range(len(self.bachViolations)):
        ratios.append(self.inputViolations[i]/self.bachViolations[i]);

    bachness = 0;

    for i in ratios:
        bachness += i;

    return bachness/17;

```

### **testBach.py**

```

import sys
import os

from utilities import *

import subprocess, glob

if len(sys.argv) < 2:
    print "Usage: python testBach.py [number of chorales]";
    exit();

pFlag = True;

if len(sys.argv) == 3:
    pFlag = False;

```

```

numberOfChorales = int(sys.argv[1]);
output = [];

for i in range(numberOfChorales):
    num = i + 1;
    if num < 10:
        txt = "00" + str(num);
    elif num < 100:
        txt = "0" + str(num);
    elif num < 1000:
        txt = str(num);
    c = "Tymo/XMLChorales/riemenschneider" + txt + ".xml";
    b = "Tymo/BachChorales/riemenschneider" + txt + ".txt";
    input = "python chassis.py " + c + " " + b;
    print input;

    # Code taken from stack overflow post
    # http://stackoverflow.com/questions/15786637/python-subprocess-store-
    # each-line-of-output-in-a-list
    globpattern = 'path/to/*.*'
    if pFlag:
        cmd = ['python', 'choraleAnalyzer.py', c, b];
    else:
        cmd = ['python', 'choraleAnalyzer.py', c, b, 'noParallels'];
    cmd.extend(glob.glob(globpattern))
    proc = subprocess.Popen(cmd,stdout=subprocess.PIPE)
    outputlines = filter(lambda x:len(x)>0,(line.strip() for line in
proc.stdout))

```

```
output = output + outputlines;

errorTracker = errorTracker();
for line in output:
    errorTracker.processError(line);

errorTracker.printState();
```