

A System for Computational Analysis of Music

Greg Cipriano

ABSTRACT:

Note-based analysis of music can unlock the mysteries of why a piece sounds like it does, but is a time consuming process requiring a skilled analyst. While computers will never supplant the creativity required in such a field, they can perform many of the more tedious tasks much faster than a human. Two such tasks are key (or tonal-center) discovery and chord elucidation, or the process of finding chord structures in a song. Presented here are algorithms for both, along with descriptions of their relative strengths and weaknesses. Also included is a program called "MIDIStat", written in Java Swing, that implements these algorithms and provides various other statistics on a given MIDI composition.

INTRODUCTION

I once got in an argument with an acquaintance over the idea of tonality. He asserted that the notes forming a musical system are a constant – that cultures pick and choose from various patterns within this system, but that the system itself never changes. Within the framework of what a note is – a wave propagating through space with a fundamental frequency and numerous overtones determining its timbre, mood and color – this meant that the collection of fundamental frequencies and their relationships between one another (often expressed as ratios of one note to another) do not change. The notes you can play on a piano, he said, were it.

This I found an utterly abhorrent idea and, without benefit of actual proof, argued that he simply couldn't be right. The notes we choose as more pleasant to listen to would not have just been handed down from the gods, but must have grown out of our own evolutionary predilections. Thus, it seemed to me, that since many cultures evolved in isolation, their musical systems should reflect this.

Entirely after the fact, my proof came easily: In many purely African musical forms, there exists a 'blue' note between the major and minor thirds. Both jazz and blues adapted this idea to the Western tonal system, but not without difficulty: contemporary blues pianists cannot hit a true 'blue' note, so most approximate by hitting both surrounding notes at the same time.

MUSICAL PRIMARIES

In a sense, I was arguing that different mathematical models for music exist down to the most fundamental level: the notes being played. Others have argued that the same is true for rhythm. In *Early Jazz* by Gunther Schuller, the author explains that tribal music in Africa makes extensive use of polyrhythms, or series of rhythms layered on top of each other.

Tribal music can carry polyrhythms to an extreme: in some tribal songs, a dozen parts could be playing simultaneously, with their measure bars lining up only in what seems a coincidental manner. Yet the rhythms aren't at all random. They have a definite structure, one Schuller says may be completely foreign to Westerners but no less legitimate.

Here again, the main differences occur at a mathematical level, which then should be the logical place to start any analysis. But before the advent of computational systems, this analysis was either painfully repetitious or superficial.

WHAT TO ANALYZE?

The structure of music can be broken down into a number of individual parts, each autonomously analyzable. Of them, the concepts of note, scale, chord and rhythm form a synergetic whole in music. All rely on one another.

Nevertheless, a definite hierarchy exists – one starting with pitch frequencies at the bottom forming notes, which form scales, from which chords are pulled out of, and ending at the top with progressions of those chords.

In other words, a scale is merely a subset of the larger set of available notes (which themselves are a subset of the set of available pitch frequencies).

A chord is rarely thought of outside of the context of the scale it's played on. There again, a chord can be thought of as a rhythmically congruent set of notes taken out of the framework of a scale. Granted, some atonal chords avoid basing themselves on a particular scale, but then they only skip one level in the musical hierarchy; for even atonal chords are a misnomer. They must somehow use notes.

Rhythm occupies a position separate from this hierarchy, but like everything else rhythm is subservient to tonality: including percussive sounds as specialized forms of notes, rhythm is useless without a note to be played.

Lastly, while alternate models for note ratios between pitch frequencies exist throughout the world, the Western variation has been settled upon as a standard in computational analysis. It is this variation that serves as the basis for the rest of this paper. Thus the theme of this paper will be that notes are the atoms of music and provide context for every conceivable type of analysis.

TWO IMPORTANT STATISTICS: KEY DISCOVERY AND CHORD FINDING

While the realm of music analysis is very broad, encompassing everything from intervallic studies to composer identification to actual music synthesis using derived style (such as that done by EMI – Experiments in Musical Intelligence, by David Cope), this paper will focus primarily on two important facets of this analysis: key discovery and chord finding.

The former involves the finding of a song's dominant key, while the latter involves the rule-based mapping of simultaneously sounding tones to a set of known chords. Both are well known problems that have been solved with only partial success. And both can provide meaningful statistics when done properly that provide a basis for quantifying a composer's style.

KEY DISCOVERY

Crucial to the study of any piece is elucidation of the key. In written music this is easy, as the notational requirements provide for easily spotting the key in a manuscript. Key finding is easy because lines in staves only encode for the 'white' notes on a piano. So any of the 'black' notes must come from modifying them – either by sharpening or flatting.

A key is shown, then, by marking certain lines in the staves with sharps or flats at the outset of the piece. Then, whenever the performer encounters notes on those lines, unless counteracted with a natural sign before the note in the same measure, they are

modified accordingly – either up in the case of a sharp, or down in the case of a flat. In this manner, all notes that would otherwise exist on the ‘C’ scale are changed to accommodate other scales.

Unfortunately in the computer realm, finding a key is not as easy. MIDI – existing strictly as a compact means to convey performance related data – does not provide any reliable means of storing a song’s key, or for that matter, its chord information and phrase structure (more on this later). It doesn’t need to, because while the staff in sheet music requires a key to show performers when to sharp and flat notes, notes in MIDI are represented as-is. Each note in the chromatic scale is represented by a different number, which ranges from 0 for the lowest C in the MIDI scale, to 12 for the next highest C, and upward. Each C is given a 0 (modulo the 12 chromatic notes between two Cs), each F# is given a 6, each E a 4 and so on.

Thus storing a key is unessential to the computer, whose only task is playing those notes without understanding of structure. So while a MIDI file does in fact allow a key to be stored within its meta-data fields, it is up to the encoder to give that key accurately. And most MIDI programs don’t make this easy.

In addition, using analytical means to discover in what key a piece is written provides further insight into the piece itself because that analysis takes into account the piece in its entirety. This may be useful because many pieces start in a key, and then traverse through several different modes of that key (or even change keys entirely) before returning. A piece that behaves in this way then has many local keys and modes which change the overall tonal-center in mysterious ways. No other reliable means exists to quantify this overarching ‘key’ of a piece outside of computational analysis.

VECTOR THEORY

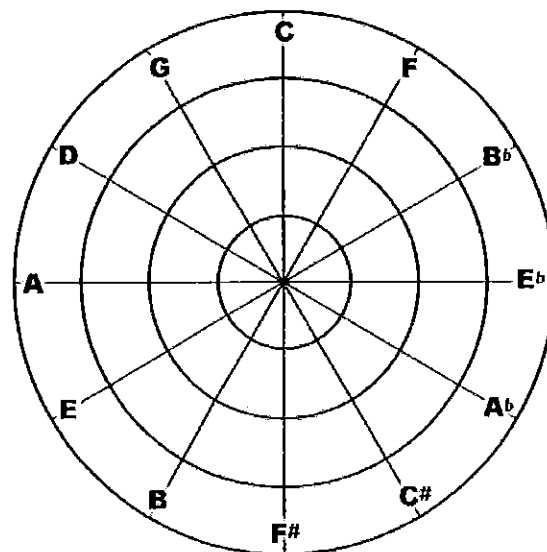
Vector analysis depends greatly on the theories proposed by R.M. Mason in his book “Modern Methods of Music Analysis Using Computers”. In it, he describes notes as vectors existing in a 2D Cartesian plane. This plane is partitioned into discrete sections by uniformly distributed rays emanating from the origin outward.

A note, then, describes a vector in two ways: the note’s tone influences its respective vector’s direction (more on this later), and the amount of time the note sounds influences its length. In this manner, a piece can be converted into the vector sum of all its notes, with the beginning of the song located at the origin.

This sum, he asserts, ends up pointing more or less to the tonal center of the piece. If this piece does not wander too far from its true key for too long, the tonal-center shows what key the piece is in. Though a bit abstract, the fundamental property of notes that solidifies Mason’s theory and makes it worth pursuing is rooted in a simple idea – the circle of fifths.

CIRCLE OF FIFTHS

In major key harmony, the key of ‘C’ is a great place to start learning to play the piano, as it encompasses only the white notes. These notes are what the staff, if no flats and sharps are present, defaults to. So C major is said to have no sharps or flats. On this continuum, F major has only one flat (Bb), Bb major has two (Bb and Eb),



and so on. The complete list: (C, F, A#/Bb, D#/Eb, G#/Ab, C#/Db, F#/Gb, B, E, A, D, G, C). Note that A#/Bb, D#/Eb, G#/Ab, etc are considered enharmonic: they are two different ways to labeling the same note. Under this consideration F#/Gb Major has alternately 6 sharps, or 6 flats, depending on your perspective. Also note that while moving right through this list increases the flatness of the key, it also decreases the sharpness, again depending on perspective.

After listing this progression an obvious pattern appears: the next key on the list is the fifth note on the scale of its previous key. (C major's scale is CDEFG...) Another feature worth noting is that the progression wraps back on itself, forming a 'circle'. Thus the name 'circle of fifths.'

USE OF CIRCLE IN VECTOR THEORY

Mason takes advantage of another well-known property of this circle: every key has as its members the notes that form the root of the 5 keys before it and 1 key after it. So one way of finding the notes in Bb major is to look at this list – (ADGCFBbEb).

Considering this progression, his method was to overlay this circle over a Cartesian plane, orienting the circle itself with C major pointing upward, and inscribing the rest clockwise around the circle. This circle represents the keys for any given series of notes – if the notes tend to form a line straight up, they can be considered as having the key of C as their tonal center.

But for the actual directions, note that in the list for Bb major, C appeared in the center of the list. Mason describes this by saying that the note C legitimizes the key of Bb,

and in general the second note of any major scale legitimizes the scale itself, since these tones form the average of the 180-degree arc (within the circle) that contains the notes for the key.

So in his model, to find the actual direction of a vector for a given note, he overlaid another circle of fifths, with this one having D point straight upward (as D legitimizes C). So to find the vector for any given note, one merely finds the angle represented by the note on this circle, and gives it an appropriate magnitude scaled according to how long that note sounded. Adding these vectors and then comparing the sum to the previous circle gives the key.

A MINOR POINT

This circle, however, is only really good for songs written in a major key. Songs based in minor keys tend to frustrate the system, which either returns the relative major or an arbitrarily random key. In the latter case, key discovery has failed, while in the former, a bit more analysis can uncover exactly which minor key the author intended.

How? The relative minor to a major key, by definition, contains the exact same notes as the major but begins (or in other words has its root) a minor third lower. In this case, Mason contends, the relative minor to a major key shares all of the same notes, and so should be subtend almost the same arc as that of its major. The question, then, is how should this relationship relate to Mason's vector circle?

Here he makes a (seemingly arbitrary) decision. In the case of C major, he says, its relative minor, A, shares more tonally with G than with F (though it shares all of its notes

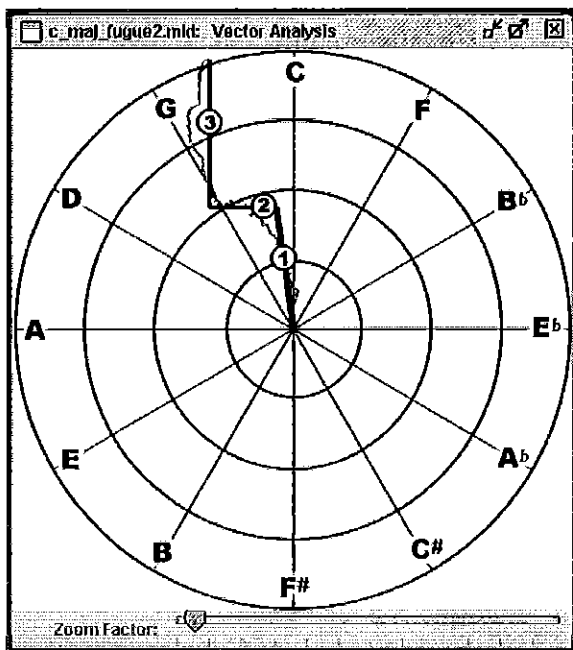
with C). Using this logic, his vector graph is then subdivided once more with minor keys 15 degrees counter-clockwise from their relative major.

USING LESS COMPUTATIONAL METHODS OF VECTOR ANALYSIS

Songs that change key can be visually (though usually not automatically) discovered by looking at the path the notes take in space. Looking at Bach's fugue in C Major, the vector begins by pointing in the key of C major, but upon modulation veers leftward towards D, and then resumes its upward momentum.

Here, the resultant vector ends at least 15 degrees counter-clockwise of C, even though the majority of the tune is in C. Thus the overarching tonality is not precisely C, but rather A minor – a development that Bach may or may not have intended, but

nevertheless is distinctly present in this piece.

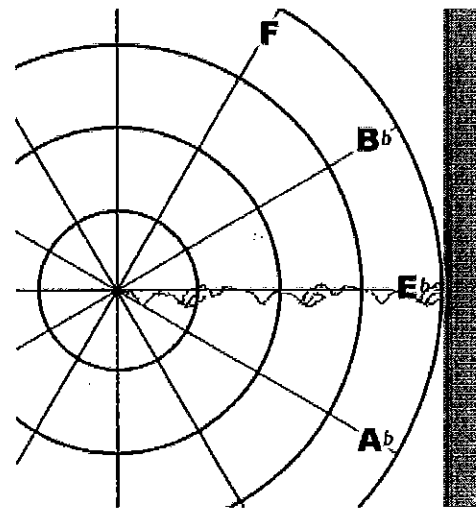


A number of possibilities to further automate this process exist outside the scope of this paper, and are not implemented in this version of MIDISat. One of them is curve fitting: the vector path itself could be fitted by lines within a user-definable degree of accuracy.

In this example, if the accuracy was set fairly low, the curve might be given three lines: one for the initial vertical progression,

a second approximating the leftward movement, and a third for the finishing vertical progression. The accuracy would determine how many lines fitted to the data, with a higher accuracy giving more lines, as it would follow wiggly vectors more closely. Results could be presented in a table, or compared against chord-determining techniques (see below).

“Cherokee” is a song that more dramatically shows these possibilities. Its form follows a standard model for jazz songs: a chorus, then a bridge, repeating back and forth. In Cherokee, though, the bridge runs through chords whose roots traverse the circle of fifths. The vector model, one would assume, should show a loop when this circle takes place – which it does exactly.



CHORD DISCOVERY

Equally as critical as finding the key in a song is the notion of finding the chord progression for a song. The theory goes: every tonal song moves through a progression of chords which serves as a foundation to the song itself. The choice of notes at each point in time is directly related to what chord the composer has in mind for that point. And that choice of notes is directly related to the sound of the piece.

The underlying chords may change quickly (as evidenced by Charlie Parker’s “Giant Steps”) as the song moves on, or may stick around as long as desired. Whichever

choice, the effect is very noticeable: a passage whose underlying chord does not change doesn't seem to 'move on' – the song seems to stand still until another chord takes its place.

These foundation chords may be explicitly stated (as in a full chord being played) or implicitly stated (as in notes being played in a scale related to a chord). But nevertheless they are omnipresent, and fundamental to any serious analysis.

Though fundamental, only a handful of sources discuss algorithms for taking music and reconstructing the chord pattern this music follows. The most recent ones look for some way of using Digital Signal Processing to find chords in a waveform-sampled music. For various reasons – discussed below – the following discussion is tailored to the use of MIDI-sampled music. And so this methodology is of little use.

Root progression seemed a particularly good feature to study, partly because objective procedures for determining roots have been formulated, and partly because it is unlikely that a composer would consciously manipulate root progression, especially a composer of linear counterpoint or of allegedly atonal music. (Youngblood, "Root Progression and Composer Identification", 172)

The problem of fitting chords to music had been addressed (not entirely to my satisfaction) in older books, including a number of papers compiled in The Computer and Music. In it, the various methods described fall into a number of categories, and indeed several of the papers contained within this book use different, not entirely compatible means to achieve the desired end.

THE INTERVAL CHECKING METHOD

In “Root Progression and Composer Identification”, by Joseph Youngblood, the algorithm he presented to find a chord went like this: First take a measure and identify its constituent structures (what they call a collection of notes). Take these structures and reduce them into their pitch classes – effectively flattening the various tones into a one octave range. Once this is done, the structure is compared with the immediately preceding valid structure, and if exactly the same, removed. (175)

Analysis was done measure-by-measure because of the format of his data structures – punch cards. In a more modern variant, included in MIDISat, chords are not lumped into measures, but rather sampled at equally spaced time intervals, usually dividing the measure. This allows for a bit more flexibility in placement, as a chord can extend over a measure.

Any valid chords found are then broken down into a set of intervals between each note. The best interval – which was explained poorly, but which could mean either the largest interval, or more likely the most tonally pleasant – was taken as signifying the chord, and the root of this interval was taken as the root of the chord itself. If no best intervals were found, or if the best interval happened to be a tritone, then the chord was considered ambiguous and stored for comparison with the next valid structure.

As this went, chord possibilities were reduced by assuming that a composer would always choose (whether consciously or otherwise) to move the root of the chord as little

as possible between changes. In the vast majority of cases, they found, this turned out to be true.

Unfortunately, the few outside cases tend to confuse this process. But as there is then (1) inherent ambiguity in these cases and so (2) no known way to resolve the ambiguity algorithmically, the chords found were considered good enough.

A more glaring deficiency in this method of analysis was that YoungBlood's assumptions made in identifying a single chord were often wrong. Especially in more complicated chords, as in those including upper structures beyond the root triad, the largest interval doesn't necessarily include the root of the chord.

For example, in an inverted F major 7, the notes may be C - E - F - A. The 'best' interval is certainly not the EF interval, since a minor second introduces dissonance to the chord. So the computer would record FA and CE as interval candidates for the root. Say this chord progresses to C major (a very likely move). Then the C in the previous chord would be deemed closest to C major, and be assigned as the root of F major 7 – an obviously false positive by the computer.

So the following chord analysis algorithm takes only one piece from this study – that finding the minimum intervallic distance between successive chords is a valid way to reduce the number of possible roots for a chord. Into this, MIDISat incorporates a second, arguably simpler algorithm, mentioned in "Music Style Analysis by Computer", by A. James Gabura.

ANOTHER METHOD FOR FINDING A ROOT

In his paper, Gabura talks about a method for finding the root of a chord by searching for the largest arc on a circle of fifths made by each set of tones in a chord. This method produces nicer results than the one before it, but he still concedes some flaws:

This method will not produce a correct root for the dominant seventh chord ... or for the diminished or augmented triads... it is much more practical, it has been found, simply to list all possible chords and their roots in a table. Then a simple look-up can be performed whenever a root is needed. (253)

HOW CHORD TYPES ARE FOUND

The following algorithm for finding chord roots and types combines the best parts of the two algorithms presented. A Chord object contains most of the known chord types, sorted in order from most complex to most simple – complexity determined as a function of how ‘high’ the upper structures go from the root.

Two observations come into play in a practical implementation. First, simple chords can (and should) be extracted from more complex, but unknown or exotic, chords. Second, in a typical song, the melody *may* in fact contradict the chord playing below it. A prudent method would then not strive to match chords to notes exactly, but merely check that subsets of the notes played from a specific chord.

Using both of these observations, MIDISat's chord checking algorithm works in the following manner:

- (1) Take the simultaneously played notes and move them into a one-octave range. This range is represented internally as a 12-bit number (or mask), with a bit encoding each note in an octave – 1 as played and 0 as not played.
- (2) Compare the mask to every known chord, from most complex to most simple. If the mask includes 1s in all the places the chord does, then it matches. So include this chord in the list of possible chords that the mask could represent. Comparison is short-circuited once a match is found, since higher-order chords include lower-order chords within them. Note the rotation of the mask, since that determines the root of the chord (as the stored chord types are all C chords).
- (3) Rotate the mask, then, repeating step (2) once for each note in a chord. In this manner, only one chord of each root can possibly be found – the most 'comprehensive' type.

Note: Exact matches are given special consideration, as they are more likely to contain a true chord. If an exact match exists, then after the searching phase is over, all non-exact matches are removed.

- (4) If after phase one, a chord is still ambiguous (more than one likely chord was found), then this chord is compared to the previous one, and the root with the smallest interval wins.

Again note that if no previous chord exists, then an ambiguous chord is compared against the key found using vector analysis. This method is predicated on the idea that the root of the first chord in a song will match the song's tonal center (denoted as key by vector analysis) . If this doesn't turn out to be possible (either the chord in the first measure is indiscernible, or else no matching chord with that root was found), the first chord will at least keep a minimal distance from the root.

CONCLUSION

Though the field of musical analysis by computer appears to be burgeoning, relatively few programs exist to facilitate such analysis.

From here, a number of possibilities exist for further use of these algorithms. Using MIDISat, one can analyze all of the nuances of a composition, correlate the statistics on a group of songs and even identify the 'style' of this work as described in purely mathematical terms. Though MIDISat does not perform these functions, this style may allow a musicologist to narrow down the potential authors of a piece to a select few, or to definitively state that a piece exhibits traits of Bach, Brahms, etc.

Musicology doesn't have to be tedious, and as demonstrated by MIDISat, can even be mechanized in unexpected ways. Though some errors will always exist in both chord finding and key finding, a researcher can augment their own skills with these tools.

APPENDIX A: THE PROGRAMMING OF MIDISTAT

THE BEGINNINGS OF A MUSIC ANALYSIS PROGRAM

Included in this paper, MIDISat demonstrates the capabilities computational tools can give to music analysts, allowing one to read in a MIDI file and perform structural analysis on it. Using the above philosophy, it considers notes as the fundamental building block of a song – essentially converting the MIDI data into a large array of Note objects, each knowing its start time, duration, pitch and velocity.

This conversion is not automatic. In fact, MIDI data is encoded as a stream of events, some of which correspond to audible notes, others to events (patch changes, pitch wheel). Depending on the format (MIDI 0 or MIDI 1), these events could either explicitly specify a note going on or going off, or just specify a velocity – with a note going completely off when its velocity is set to 0. In this case, a zeroed velocity is treated exactly as if a note-off event occurred.

Java does almost nothing to ameliorate this situation. While it could provide a means to coerce note data from the raw MIDI data, it instead places each event intact into Java's objectified structure.

What MIDISat does to perform the necessary conversion, then, is to create a Vector (an 'unbounded' array – an unfortunate choice of naming in Java, as it's completely unrelated to the mathematical notion of a vector) to hold note-on events. Whenever a note-on event occurs, a new NoteOnEvent object is added to the end of the Vector. This object contains information about the note's pitch, track, start time, and

velocity. And whenever a note-off event occurs, the Vector is scanned from the beginning until it finds a matching (of the same pitch and track) note-on event, which is then removed. Finally, a new Note object is added to the Note array encapsulating this information.

This predicates on several assumptions: that the MIDI input is well-formed, and that for every note-on event, there is a matching note-off event and vice-versa. Its note building algorithm is fault-tolerant in that unmatched events are ignored, and overlapping notes (those where two note-on events occur on the same note in the same channel before their matching note-off events) are treated in the expected way: any note-off event is assumed to correspond to the first as-yet-unmatched note-on event.

This is done for all events, so what is left is an array of Notes, which are extremely lean and better suited for analysis.

WHY MIDISTAT USES MIDI

MIDISat began as a project to read in WAV files, or any type of digitally sampled sound files that happen to be music. It could then run through the file looking for large blocks of a single frequency (large enough to be a likely note) and filing those away in a form similar to above description

I immediately ran into difficulty not because the files contained too little information, but because I was unable to account for a fundamental property of musical sounds: the overtone series.

THE OVERTONE SERIES

The notes heard when a harp string is plucked, or a violin played are not merely composed of the pure frequency the brain thinks it hears. Rather, each note is made of a fundamental tone (usually the most prominent) which determines the overriding pitch of the note, and a number of frequencies of various pitches and degrees of loudness above that tone. This series of tones are unique to the instrument being played, and are what make a piano sound like a piano, and not a clarinet, even though both are playing what is heard as the same pitch.

Unfortunately, it is this series that thwarted all attempts to pull apart sampled audio: whenever MIDISat tried to encode a note, junk notes coming from the overtone series appeared. Further, any noise or imperfections in the recording including drum beats, coughs or audience sounds, contributed to a completely unusable data set.

Some packages that purport to accomplish this feat rely on a separate audio file that contains a characteristic sample of the instrument used. In this manner, they pattern match between what is contained in the sample file and what appears in the music. This gets around the problem of phantom notes from the overtone series, but incurs problems of its own; it can only analyze simple, homogenous songs. Any other noise damages the translation. And a song with multiple instruments, or instruments that radically change timbre, will not turn out right.

MIDI

The MIDI format circumvents all of these problems, but unfortunately is inadequate to capture subtle nuances from a performance. This is less of a problem than expected, though, since the goals of this paper are not to distinguish between performers, but rather to collect statistics on a composer, using only his body of works and independent of interpretation of and improvisation on (performance-based or otherwise) those works.

WHAT MIDISTAT CAN DO

MIDISat can load in MIDI files, and perform statistical analysis on those files. The results of that analysis can be shown in a number of graphs, and in a table of the most important data. MIDISat makes careful use of the tracks in that MIDI files, allowing for tracks to be independently analyzed, or excluded from analysis of the song (very useful when a song contains a drum track, which contains pitches that don't correspond to tones but rather a type of drum).

MIDISat can make a fairly good guess at the key (to the nearest major, anyway) of a song or track. It can also make a fairly good guess at the chords forming a song, and analyze statistics on those chords.

Specifics on each of its features are detailed in Appendix B.

WHAT MIDISTAT CAN'T DO (BUT I WANT IT TO)

So far, there seems to be a completely reliable way for a computer to identify phrase structure without any a priori knowledge of the song it's analyzing. University Professor Gary Vercase and I have carefully considered this in the context of "Eronel" – a composition by Thelonious Monk.

"Eronel" has everything a jazz piece could hope for to aid analysis: repetitive phrasing, fairly simple meter and a quarter-note laden bass line which spells out the tune's scale, and so chord structure. But despite these assets, Eronel confounded all attempts at algorithmic description.

Using rests to delineate phrases seemed at first promising, since in the songs' opening tag, phrases were separated by rest. But unfortunately, this idea doesn't apply universally: in many bars, the print version of the song describes an unbroken group of notes as being split into two phrases – one for the ascending half, and the other for a descending half. A computer could run through analyzing melodic movement (indeed MIDISat can do that.) But as evidenced by the printed score, this plan fails equally often.

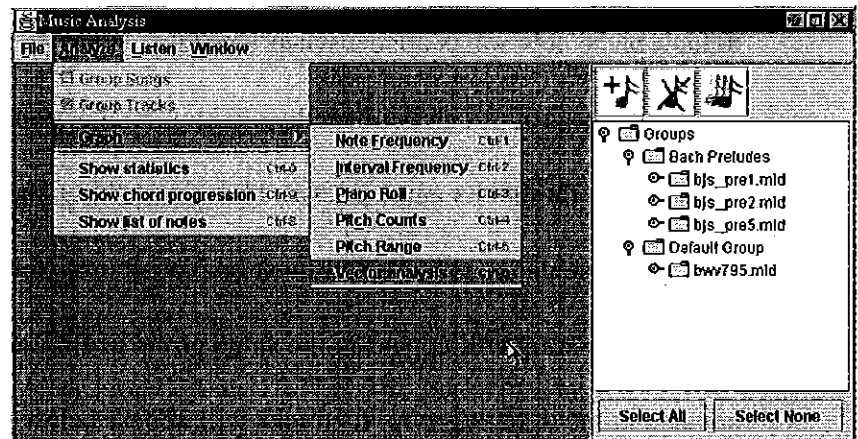
Nevertheless, since the phrase structure in effect gives a performer cues as to how loud to play, reconstruction of phrases could work in reverse; a carefully coded MIDI file could contain loudness cues for an analysis engine. Unfortunately, most music archives don't pay enough attention to loudness, encoding it either as an afterthought or not at all.

APPENDIX B: HOW TO USE MIDISTAT

MIDISat uses Java Swing extensively for its Graphical User Interface. To that end, all components are laid out in a hopefully intuitive manner. Starting the program is accomplished by typing:

```
java -jar MIDISat.jar
```

on the command line.



Once it starts, there are three components worth noting. The first is the menu bar on top. All functions allowing loading and analyzing MIDI files are accessible through the various menus.

The second is the panel on the right. On this are three buttons, representing (1) adding a song, (2) removing a song or group and (3) adding a new group. These directly relate to the tree displayed below them.

Finally, there is the desktop (the large purple region), where graphs are placed in their own internal frames.

THE TREE

The tree has three levels. First are groups (which can contain any number of songs), then the songs themselves, then the individual tracks of those songs. Upon startup, a default group is created, which is special in that it cannot be deleted, and is the default place new songs appear when loaded.

Other groups can be made by clicking on the 'add group' button. Any group can be renamed, though its name doesn't change anything about how it functions.

When a song is loaded, it must be placed in a group. If anything in the tree is selected, the song will appear in the same group as that selection. Otherwise the song appears in the default group. Its tracks will appear below the song, and each can be enabled and disabled by clicking on that track. When a track is disabled, its notes will not count in analysis of its song or the group that contains it.

Songs can be dragged from group to group, allowing the user to manipulate the song list under a given group.

HOW TO ANALYZE SONGS

Any of the entities on the tree can be analyzed in the same fashion: groups of songs, an individual song, or any track in a song. This is performed by selecting at least one item in the tree, and choosing the method of analysis from the menu (more on this later).

Note that analysis is fully dependent on the set of songs/tracks below the item being analyzed, and that enabling or disabling a track or moving a song to another group takes immediate effect on its parents.

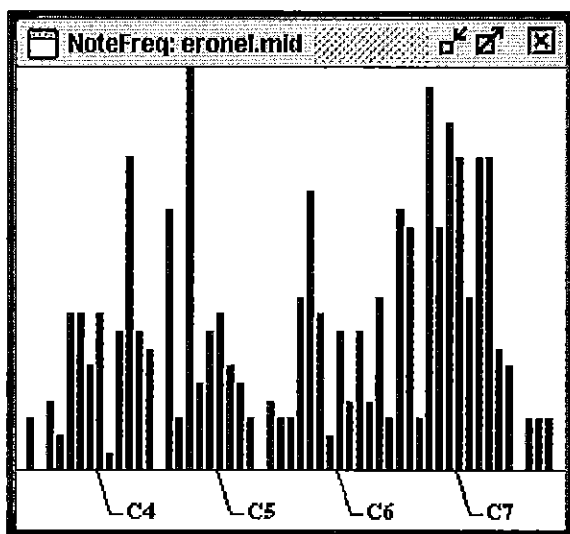
Also note that if a selected item has no enabled tracks under it, no analysis can be performed, and an error will sound. This happens when one attempts to analyze a group that has no songs, a song that has no enabled tracks, or a group of songs with no enabled tracks.

GRAPH-BASED ANALYSIS

MIDISat allows for a number of visual representations of the underlying musical structure, be it group, song or track. The term 'graph' is used to describe these visual representations, which include note frequency, note interval graphs, as well as piano roll, pitch count, pitch range and vector visualizations. The first two fit into the category of static graphs, whose data fits easily into a frame, and so require neither scrolling nor zooming.

The next three are scrollable graphs – their data exists on a sheet that is as wide as the song is long. This data can be scrolled across using a scroll bar and also adjusted for size using a slider just below that bar.

Finally, vector visualization doesn't need scrolling exactly, but includes a slider to allow for zooming (which really just increases/decreases the scale of each note vector).



NOTE FREQUENCY GRAPH

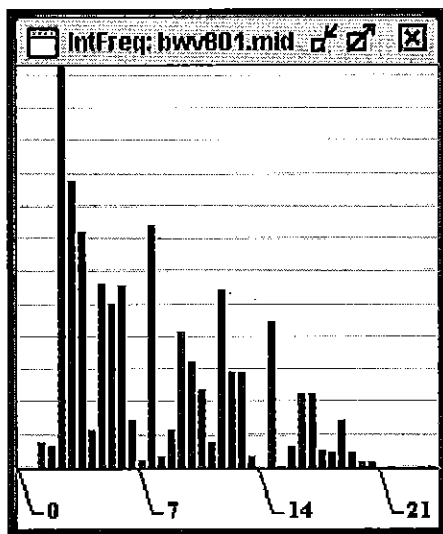
Note frequency analysis involves running through the list of notes in the structure, dropping each note in a separate bin based on its pitch. The resulting graph of these bins form a histogram of the tones in a song, much in the same way a

histogram in a drawing program shows the number of pixels of each brightness value.

Using this graph, one can visually determine both the tonal center and the underlying key of a song. In the most simple of songs – those that stick to a single key and refrain from modulating – the note frequency graph shows very quickly which notes form the key, as they will have the tallest bars. For example, a song in the key of C should have bars of negligible or no height on the C#, D#, F#, G#, and A# bars.

Even in less well-behaved songs, the majority of notes should still stick to the key, and so this graph remains valuable.

Also, the majority of songs, especially in classical and pop forms, form a bell-shaped curve upon note-frequency inspection. This curve, then, has at its highest peak the tonal center of the song.

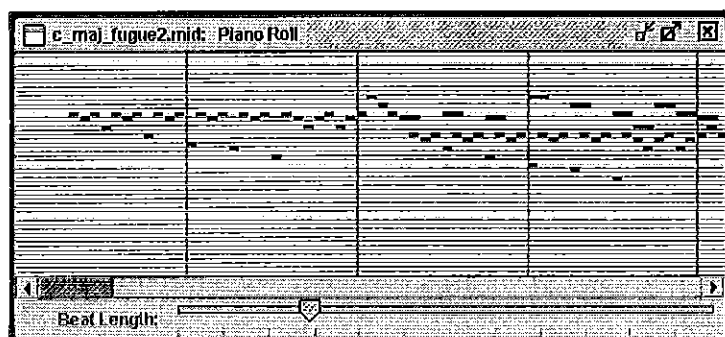


NOTE INTERVAL GRAPH

The note interval graph works similarly to the note frequency graph, only instead of note pitches, the intervals between simultaneously sounding notes are thrown into bins based on their size.

In songs that have multiple voices playing simultaneously, certain intervals usually appear as 'favored' intervals. In major harmony, those intervals include the major 3, perfect 5 and perfect 6. In minor harmony, the minor 3 would replace the major 3. This graph, then, can facilitate the discovery of the 'type' of key used in the majority of the song.

The same caveats apply here as in note-frequency analysis: A song that is poorly behaved, either modulating frequently, or having a great deal of both accidentals and 'out' tones can foul up this graph, invalidating any results it provides. But then those songs defy conventional theory in general, and so are not good candidates for automatic, computer-based analysis.



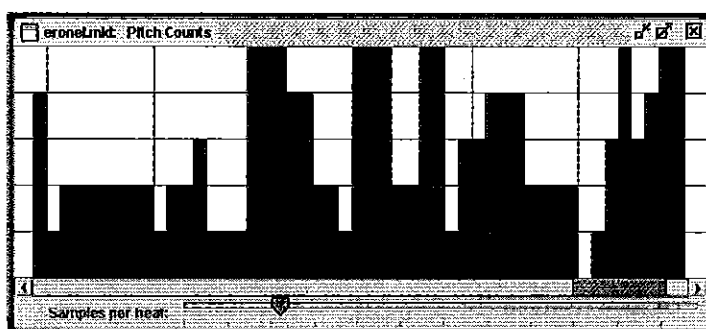
PIANO ROLL GRAPH

This graph is a bit out of place in the list of graphs, as it doesn't analyze some facet of a

song. Instead, its function is to show the notes in a song in a form similar to the tape an

old-fashioned player piano. In this respect, its use is primarily to both serve as a reference for other graphs and to display what ordinarily would be put on a staff in a homogenous manner.

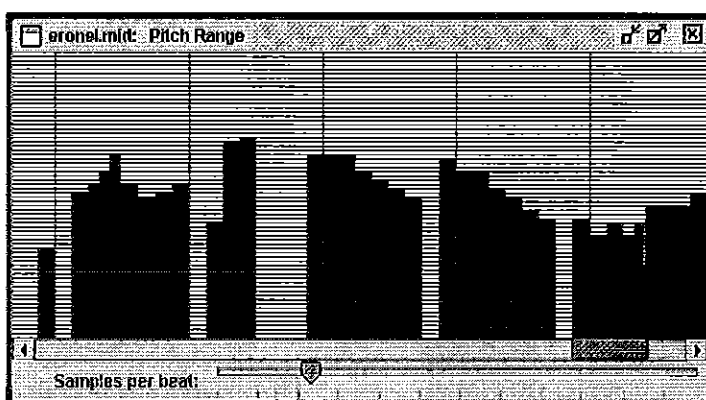
Finding contrapuntal motion on this graph is very easy, as is noticing the range of pitches the piece roams through at any point in the song.



PITCH COUNT GRAPH

This graph samples the song at evenly spaced intervals of time, counting how many

notes are sounding in each interval. The resulting graph shows the complexity of the piece at any instant, as well as the song's change in complexity over time.



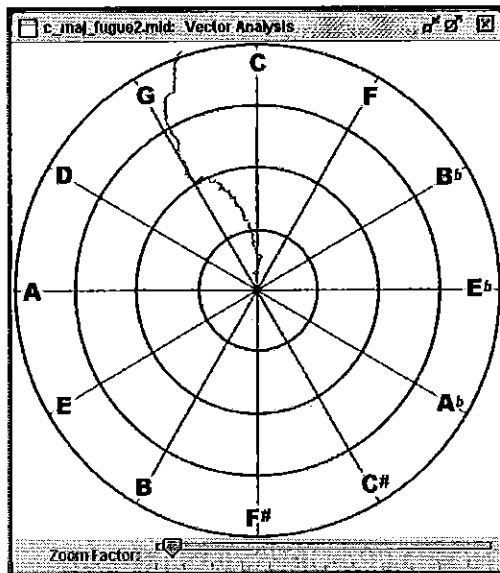
PITCH RANGE GRAPH

The pitch range graph is similar to the Pitch Count Graph in that it samples the data over evenly spaced intervals. But

where the pitch count graph shows how many pitches are sounding over each time slice, the pitch range graph shows how large of an interval exists between the lowest and highest of those pitches.

Of course, if only one pitch (or none at all) is sounding at a specific time, the pitch graph will have a range of zero. But for other cases, the pitch range graph can show the contrapuntal movement (be it oblique, contrary, or otherwise) between the outer voices. This can be quite useful when analyzing a composers propensity toward opening up voices (bringing them further apart) or closing them.

Also, quick jumps or discontinuities in the graph signify a composer that prefers less fluidity in his or her music, a characteristic of several styles, including atonal.

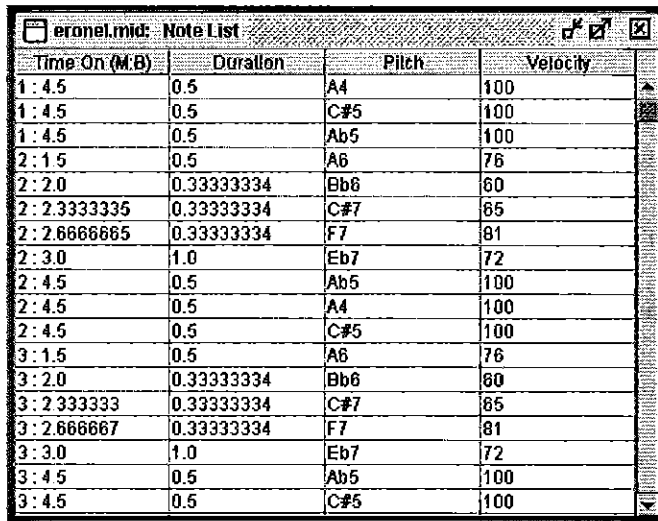


VECTOR ANALYSIS GRAPH

This graph shows the Major circle of fifths and a tone path in red superimposed on top of it in the manner described in 'Vector Theory' above.

DATA ANALYSIS

In addition to the graphs available, MIDISat also includes three other ways of looking at a song's data: a Note List Frame, a Chord Progression Frame and a Statistics Frame. All use Swing-based tables (or JTables) to show their data.



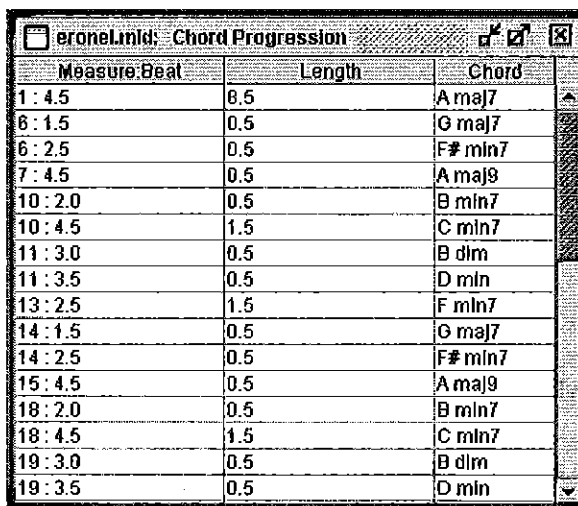
Time On (M:B)	Duration	Pitch	Velocity
1 : 4.5	0.5	A4	100
1 : 4.5	0.5	C#5	100
1 : 4.5	0.5	Ab5	100
2 : 1.5	0.5	A6	76
2 : 2.0	0.33333334	Bb6	60
2 : 2.3333335	0.33333334	C#7	65
2 : 2.6666665	0.33333334	F7	81
2 : 3.0	1.0	Eb7	72
2 : 4.5	0.5	Ab5	100
2 : 4.5	0.5	A4	100
2 : 4.5	0.5	C#5	100
3 : 1.5	0.5	A6	76
3 : 2.0	0.33333334	Bb6	60
3 : 2.3333333	0.33333334	C#7	65
3 : 2.6666667	0.33333334	F7	81
3 : 3.0	1.0	Eb7	72
3 : 4.5	0.5	Ab5	100
3 : 4.5	0.5	C#5	100

NOTE LIST FRAME

The Note List Frame shows the same set of data as the Piano Roll Graph, but in a different form. Each Note contained in the Note array of an Analyzer occupies a separate row of the table, while the

columns show when the note started, how long it played, the pitch of the note, and its velocity.

This frame then provides one more way to view the notes in a song – with an emphasis more on finding exact starting and stopping times as well as checking the pitch of a note, rather than the more broad picture provided by a Piano-Roll Graph.



Measure:Beat	Length	Chord
1 : 4.5	8.5	A maj7
6 : 1.5	0.5	G maj7
6 : 2.5	0.5	F# min7
7 : 4.5	0.5	A maj9
10 : 2.0	0.5	B min7
10 : 4.5	1.5	C min7
11 : 3.0	0.5	B dim
11 : 3.5	0.5	D min
13 : 2.5	1.5	F min7
14 : 1.5	0.5	G maj7
14 : 2.5	0.5	F# min7
15 : 4.5	0.5	A maj9
18 : 2.0	0.5	B min7
18 : 4.5	1.5	C min7
19 : 3.0	0.5	B dim
19 : 3.5	0.5	D min

CHORD PROGRESSION FRAME

This frame shows the results of MIDISat's chord analysis algorithm on a song, with each chord on a separate row. In this frame, columns represent the chord's start time, length, and root/type.

erone1.mid: Statistics	
Statistic	Value
Number of notes:	419
Range:	53 notes
Highest note:	A7
Lowest note:	F3
Mean Note:	A5
Standard pitch deviation (from mean):	14 notes
Mean Note (weighted):	F5
Standard pitch deviation (from weighted mean):	14 notes
Most often sounded note:	A4
Number of times it sounded:	23
Highest number of simultaneous voices:	5
Largest interval:	47 notes
Most frequent interval:	10 notes
Number of times it sounded:	47
Approximate Key:	C maj
Key stored in file:	C maj

STATISTICS FRAME

This frame holds on to the myriad statistics collected for an individual song, track or entire group. Some of the more obvious statistics presented here are:

Number of notes: the length of the Note array.

Highest Note: the highest pitch played throughout the notes array

Range: the distance, in notes, between the highest note and lowest note.

Most often sounded note & how many times it sounded: goes without saying.

Highest number of simultaneous voices: the maximum number of notes sounding at any given time throughout the song.

Largest Interval: the largest distance between simultaneously sounding notes.

Most frequent interval: the interval that sounded most often between simultaneous notes.

Also, a number of less obvious but useful statistics are presented here:

Mean note: the average note played rounded to the nearest note. This ignores how long the notes played, factoring only their pitch.

Standard pitch deviation: The deviation from the mean note. Gives a fair description of the composer's most comfortable range in a song (or track).

Weighted mean note: Similar to mean note, but this takes into account how long a note played, weighting longer notes more than short notes. This method is arguably more accurate in determining the tonal mean in a song, but both are included for comparison.

Weighted standard pitch deviation: Same as standard pitch deviation, but using the weighted mean note rather than the unweighted.

Approximate key: takes the resultant vector from the Vector Analysis graph, and prints out the major key associated with the 30 degree region that that vector ended up in.

Key stored in file: Prints out the key included in the metadata of the MIDI file. As discussed previously, this key may or may not be accurate, and is just printed here for informational purposes.

APPENDIX C: GLOSSARY

ATONAL - A section of music or chord lacking a tonal center, in which all tones carry equal importance.

CHROMATIC SCALE - A scale that includes every playable note (eg every note on a piano). Note: a chromatic scale generally doesn't have a root, since transposing the scale has no effect on its tone set.

CONTRAPUNTAL MOTION - describes the various ways counterpoint can move between lines.

COUNTERPOINT - The combination of two or more melodic lines played against one another. Counterpoint's horizontal structure is built upon competing melodic lines, rather than chords.

FLAT - A sign signifying the decrease of a note's pitch by one half step.

HALF STEP - Like a whole step, a half step may be abbreviated as H and represents a distance of one tone. (i.e., C# is a whole step above C).

INTERVAL - In music, the distance between two notes. Starting from a unison (an interval of 0 half steps), musical intervals proceed upward by: Minor 2, Major 2, Minor 3, Major 3, Perfect 4, Tritone, Perfect 5, Minor 6, Major 6, Minor 7, Major 7, Octave.

INVERSIONS - The different forms of chords created by changing which chord member is played as the lowest note. In this paper, all inversions of a chord are considered isomorphic to one another.

KEY - The tonal center of a piece, based on the tonic of the scale most present in a piece.

MAJOR KEY - A key that, starting from its root, proceeds upward by WWHWWWH

(see whole step, half step). **META-DATA FIELD** - A field in MIDI that defines extra information beyond that needed for a performance. Meta-Data can define a key signature, time signature, as well as composer information and raw text.

MIDI - or (Musical Instruments Digital Interface) is the generic name for a number of protocols defining how digital musical devices (including sound cards, keyboards, etc.) talk to each other. MIDI comes in several flavors: MIDI 0, in which all events are grouped into one track, MIDI 1, in which up to 16 tracks can be defined with individual instruments, and MIDI 2, which can store multiple songs in a single file. MIDISat cannot currently analyze MIDI 2 files .

MINOR KEY - Exists in many variations, but 'minor' usually implies pure minor, which proceeds upward by WHWWHWW.

MODE - A scale pattern made up of a set of whole and half steps.

MODULATION – A change of keys within a song, modifying the tonal center.

NATURAL SIGN - A natural note is one that has not been raised or lowered from its named pitch. A natural sign signifies that a previously sharped or flatted note should instead play a natural note. On a piano, naturals are the white keys.

OCTAVE - The distance between a note and the closest note of the same name, either above or below. Almost always, an octave is measured by 12 half steps.

RELATIVE MAJOR - Noting that the major key steps are a rotation of the minor key steps, a relative major is the major key which contains the same notes as a given minor key (eg. C major is A minor's relative major).

RELATIVE MINOR - see RELATIVE MAJOR.

ROOT - the most fundamental note of a chord, often the bass note, which usually contains the other members of the chord in its overtones.

ROOT TRIAD - The first three notes of a chord that is constructed by proceeding upward by either major or minor thirds. For instance, a C major triad would consist of C (the root), E and G.

SCALE - A set of notes, as a subset of the chromatic scale, that defines a diatonic (or fully natural) tonality centered on a tonic note.

SHARP - A sign signifying the increase of a note's pitch by one half step.

STAFF - The five horizontal lines upon which sheet music is written.

TONAL CENTER (or TONIC) - The foundation of the scale or melody in a song or subsection of a song.

TONIC - The note defined as a starting position for a scale, the root.

TRITONE - An interval one half-step larger than a perfect 4. The tritone is special in that (1) it's note vectors point in exact opposite directions on a circle of 5ths, and (2) the tritone is a fully ambiguous interval: the notes forming a tritone are always the same distance apart, regardless of inversion.

WHOLE STEP - Sometimes abbreviated as W when used in conjunction with key, a whole step represents a chromatic distance of two tones (i.e., D is a whole step above C).

WORKS CITED

- Crane, Frederick and Fiehler, Judith. "Numerical Methods of Comparing Musical Styles." The Computer and Music. Ed. Harry B. Lincoln. New York: Cornell University Press, 1970.
- Gabura, A. James. "Music Style Analysis by Computer." The Computer and Music. Ed. Harry B. Lincoln. New York: Cornell University Press, 1970.
- Mason, R.M. Modern Methods of Music Analysis Using Computers. Peterborough, NH: Transcript Printing Company, 1985.
- Morse, Raymond William. "Use of Microcomputer Graphics to Aid in the Analysis of Music." Diss. University of Oregon, 1985.
- Youngblood, Joseph. "Root Progressions and Composer Identification." The Computer and Music. Ed. Harry B. Lincoln. New York: Cornell University Press, 1970.

APPENDIX E: SOURCE CODE

CODE LAYOUT

The code layout of MIDISat is fairly simple. The entry point is located in `MidiStat.java`. This file builds the main frame of the GUI, and adds several subcomponents: A `TreePanel`, and a `JDesktopPane`.

The `TreePanel` contains a tree managed by the `TreeManager`, which controls file loading, removing, and the finer points of managing a `JTree`. On this tree, each node (besides the root) is represented by a type of `Analyzer`, which supports generic analysis on a series of `Notes`. The first level below the root contains only `GroupAnalyzers`, the second level only `SongAnalyzers` and the third and final level only `TrackAnalyzers`.

Onto the `JDesktopPane`, MIDISat adds `JInternalFrames` when a user initiates certain actions. These frames are: `ChordProgressionFrame`, `StatisticsFrame`, `NoteListFrame`, and the `GraphFrame`.

In the latter, a number of different `Graph` types (all of which must implement the `MusicGraph` interface) are plugged in. These form the following hierarchy:

```
MusicGraph (interface)
|
|--- IntervalFreqGraph
|--- NoteFreqGraph
|--- ScrollBarGraph (abstract)
|       |
|       |--- PianoRollGraph
|       |--- DataGraph (abstract)
|               |
|               |--- PitchCountGraph
|               |--- PitchRangeGraph
|--- VectorAnalysisGraph
```

All `MusicGraphs` (and most `Frames`) consume an `Analyzer`, using the various methods `Analyzers` provide to construct their display. None depend on any specific type of `Analyzer`, so this allows any groups of songs, one song, or a sub-component of a song to be displayed without prejudice.

Finally, `Notes` and `Chords` are atomic representations of their names. A `Note` contains pitch, start, duration, and velocity. A `Chord` is either determined (in which case it contains a root and upper structures) or undetermined (in which case it contains all possible roots each of their upper structures).


```

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.sound.midi.*;
import javax.swing.*;
import javax.swing.plaf.metal.MetalLookAndFeel;
import java.beans.PropertyVetoException;

/**
 * MidiStat.java <p>
 *
 * The entry point of this program, MidiStat sets up
 * all of the menubars, the main frame and all major GUI components
 * within that frame.
 *
 * @author      Greg Cipriano
 */

public class MidiStat {
    private String metal    = "javax.swing.plaf.metal.MetalLookAndFeel";
    private String motif    = "com.sun.java.swing.plaf.motif.MotifLookAndFeel";
    private String windows  = "com.sun.java.swing.plaf.windows.WindowsLookAndFeel";
    private JCheckBoxMenuItem groupSongs;
    private JCheckBoxMenuItem groupTracks;
    private JDesktopPane desktop;
    private JFrame frame;
    private JMenuItem playSelected;
    private JMenuItem pauseSelected;
    private JMenuItem resumeSelected;
    private JMenuItem stopSelected;
    private MusicPlayer musicPlayer;
    private TreeManager treeMan;

    private int frameCounter = 0;

    /**
     * The entry point for the whole application. This just
     * constructs an instance of the class.
     */

    public static void main(String[] args) {
        new MidiStat();
    }

    /**
     * This method makes sure the look and feel of the component
     * is set before setting up the frame.
     */

    public MidiStat() {
        try {
            UIManager.setLookAndFeel(metal);
        } catch (Exception e) {}
        musicPlayer = new MusicPlayer(new SongEndedListener(this));
        setUpFrame();
    }

    /**
     * This creates the frame, the desktop, and the tree panel and
     * adds the latter to the former using a JSplitPane. Basically,
     * this prepares GUI components.
     */

    private void setUpFrame() {
        frame = new JFrame("MidiStat");
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        setUpMenuBar();
        // Split the frame in two, placing the desktop on the left and
        // the tree on the right.

        Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
        desktop = new JDesktopPane();
        desktop.setPreferredSize(new Dimension(screenSize.width - 270, screenSize.height - 150));

        JSplitPane p = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
                                      desktop, new TreePanel(treeMan = new TreeManager()));
        p.setDividerSize(2);
        frame.setContentPane(p);
        frame.pack();

        // Center the frame in the screen.

        Dimension frameSize = frame.getSize();
        int x = (screenSize.width - frameSize.width) / 2;
        int y = (screenSize.height - frameSize.height) / 2;
        frame.setLocation(x, y);
        frame.show();
    }

```

```

)

/**
 * Well... This sets up the menu bar. Each menu item is given its own personal
 * anonymous ActionListener, which responds to events generated by that item. Also,
 * the createMenuItem(...) method is used extensively to simplify things a bit
 * (and, quite frankly, to make sure I don't forget to attach events and names to
 * each menu item.)
 */

private void setUpMenuBar() {
    JMenuBar menuBar= new JMenuBar();
    JMenu fileMenu = new JMenu("File");
    fileMenu.setMnemonic(KeyEvent.VK_F);
    fileMenu.getAccessibleContext().setAccessibleDescription(
        "File-related menu options");

    fileMenu.add(createMenuItem("Load", "Loads a MIDI file...", KeyEvent.VK_L,
        KeyEvent.VK_L, ActionEvent.CTRL_MASK,
        new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                treeMan.addSongs();
            }
        }));
    fileMenu.add(createMenuItem("Remove", "Removes selected MIDI file...", KeyEvent.VK_R,
        KeyEvent.VK_R, ActionEvent.CTRL_MASK,
        new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                treeMan.removeSelectedNodes();
            }
        }));
    fileMenu.addSeparator();
    fileMenu.add(createMenuItem("Exit", "Exit MidiStat...", KeyEvent.VK_X,
        KeyEvent.VK_X, ActionEvent.CTRL_MASK,
        new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.exit(0);
            }
        }));

    JMenu analyzeMenu = new JMenu("Analyze");
    analyzeMenu.setMnemonic(KeyEvent.VK_A);
    analyzeMenu.getAccessibleContext().setAccessibleDescription(
        "Full set of Visualization/Analyzing algorithms");

    groupSongs = new JCheckBoxMenuItem("Group Songs", false);
    //groupSongs.addItemListener(musicListener);
    groupTracks = new JCheckBoxMenuItem("Group Tracks", true);
    //groupTracks.addItemListener(musicListener);
    groupTracks.setEnabled(false);
    groupSongs.setEnabled(false);
    analyzeMenu.add(groupSongs);
    analyzeMenu.add(groupTracks);
    analyzeMenu.addSeparator();

    JMenu graphMenu = new JMenu("Graph");
    graphMenu.setMnemonic(KeyEvent.VK_G);
    graphMenu.add(createMenuItem("Note Frequency",
        "Draws a histogram of note frequencies in selected piece",
        KeyEvent.VK_F, KeyEvent.VK_1, ActionEvent.CTRL_MASK,
        new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                showGraphs(new NoteFreqGraph());
            }
        }));
    graphMenu.add(createMenuItem("Interval Frequency",
        "Draws a histogram of intervallic frequencies in selected piece",
        KeyEvent.VK_I, KeyEvent.VK_2, ActionEvent.CTRL_MASK,
        new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                showGraphs(new IntervalFreqGraph());
            }
        }));
    graphMenu.add(createMenuItem("Piano Roll",
        "Draws a representation of the piece ala a piano roll.",
        KeyEvent.VK_P, KeyEvent.VK_3, ActionEvent.CTRL_MASK,
        new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                showGraphs(new PianoRollGraph());
            }
        }));
    graphMenu.add(createMenuItem("Pitch Counts",
        "Draws number of simultaneous pitches at each timestep.",
        KeyEvent.VK_C, KeyEvent.VK_4, ActionEvent.CTRL_MASK,
        new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                showGraphs(new PitchCountGraph());
            }
        }));
    graphMenu.add(createMenuItem("Pitch Range",
        "Draws the pitch range within each timestep.",

```



```

        KeyEvent.VK_R, KeyEvent.VK_5, ActionEvent.CTRL_MASK,
        new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                showGraphs(new PitchRangeGraph());
            }
        }
    ));
graphMenu.add(createMenuItem("Vector analysis",
    "Draws a 'vector' view of the piece.",
    KeyEvent.VK_V, KeyEvent.VK_6, ActionEvent.CTRL_MASK,
    new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            showGraphs(new VectorAnalysisGraph());
        }
    }
));

analyzeMenu.add(graphMenu);
analyzeMenu.addSeparator();
analyzeMenu.add(createMenuItem("Show statistics",
    "Shows statistics on a song/track",
    KeyEvent.VK_S, KeyEvent.VK_0, ActionEvent.CTRL_MASK,
    new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            Analyzer[] nodes = treeMan.getSelectedNodes();
            for (int i = 0; nodes != null && i < nodes.length; i++) {
                if (nodes[i].containsChildren()) {
                    StatisticsFrame sf = new StatisticsFrame(nodes[i]);
                    addInternalFrame(sf);
                } else {
                    Toolkit.getDefaultToolkit().beep();
                }
            }
        }
    }
));
analyzeMenu.add(createMenuItem("Show chord progression",
    "Shows guessed chord progression (experimental)",
    KeyEvent.VK_C, KeyEvent.VK_9, ActionEvent.CTRL_MASK,
    new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            Analyzer[] nodes = treeMan.getSelectedNodes();
            for (int i = 0; nodes != null && i < nodes.length; i++) {
                if (nodes[i].containsChildren()) {
                    ChordProgressionFrame cpf =
                        new ChordProgressionFrame(nodes[i]);
                    addInternalFrame(cpf);
                } else {
                    Toolkit.getDefaultToolkit().beep();
                }
            }
        }
    }
));
analyzeMenu.add(createMenuItem("Show list of notes",
    "Shows an list of notes, ordered from first to last",
    KeyEvent.VK_L, KeyEvent.VK_8, ActionEvent.CTRL_MASK,
    new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            Analyzer[] nodes = treeMan.getSelectedNodes();
            for (int i = 0; nodes != null && i < nodes.length; i++) {
                if (nodes[i].containsChildren()) {
                    NoteListFrame nlf =
                        new NoteListFrame(nodes[i]);
                    addInternalFrame(nlf);
                } else {
                    Toolkit.getDefaultToolkit().beep();
                }
            }
        }
    }
));

JMenu windowMenu = new JMenu("Window");
windowMenu.setMnemonic(KeyEvent.VK_W);
windowMenu.add(createMenuItem("Close All Windows",
    "Closes all windows",
    KeyEvent.VK_A, KeyEvent.VK_F4, ActionEvent.CTRL_MASK,
    new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            JInternalFrame[] jif = desktop.getAllFrames();
            for (int i = 0; i < jif.length; i++) {
                jif[i].setVisible(false);
                desktop.remove(jif[i]);
                jif[i].dispose();
            }
        }
    }
));
windowMenu.add(createMenuItem("Minimize All Windows",
    "Minimizes all windows",
    KeyEvent.VK_M, KeyEvent.VK_F5, ActionEvent.CTRL_MASK,
    new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            try {
                JInternalFrame[] jif = desktop.getAllFrames();
                for (int i = 0; i < jif.length; i++) {

```

```

        jif[i].setIcon(true);
    }
    } catch(PropertyVetoException pre) {
        System.err.println("Problem minimizing window!");
    }
}
));
windowMenu.add(createMenuItem("Restore All Windows",
    "Restores all windows",
    KeyEvent.VK_R, KeyEvent.VK_F6, ActionEvent.CTRL_MASK,
    new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            try {
                JInternalFrame[] jif = desktop.getAllFrames();
                for (int i = 0; i < jif.length; i++) {
                    jif[i].setIcon(false);
                }
            } catch(PropertyVetoException pre) {
                System.err.println("Problem restoring window!");
            }
        }
    }
));
windowMenu.add(createMenuItem("Cascade",
    "Arranges all windows in a cascade",
    KeyEvent.VK_C, KeyEvent.VK_F7, ActionEvent.CTRL_MASK,
    new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            frameCounter = 0;
            JInternalFrame[] jif = desktop.getAllFrames();
            for (int i = 0; i < jif.length; i++) {
                int start = 5 + (frameCounter++ % 10) * 25;
                jif[i].setLocation(start, start);
            }
        }
    }
));

JMenu listenMenu = new JMenu("Listen");
listenMenu.setMnemonic(KeyEvent.VK_L);
listenMenu.add(playSelected = createMenuItem("Play selected item",
    "Plays the selected track/song or queues entire group.",
    KeyEvent.VK_P, KeyEvent.VK_P, ActionEvent.ALT_MASK,
    new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            Analyzer a = treeMan.getSelectedNode();
            musicPlayer.start(a);
            pauseSelected.setEnabled(true);
            stopSelected.setEnabled(true);
        }
    }
));
listenMenu.add(pauseSelected = createMenuItem("Pause",
    "Pauses the currently playing song (if any).",
    KeyEvent.VK_A, KeyEvent.VK_A, ActionEvent.ALT_MASK,
    new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            musicPlayer.pause();
            resumeSelected.setEnabled(true);
            pauseSelected.setEnabled(false);
        }
    }
));
listenMenu.add(resumeSelected = createMenuItem("Resume",
    "Resumes the currently playing song (if any).",
    KeyEvent.VK_R, KeyEvent.VK_R, ActionEvent.ALT_MASK,
    new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            musicPlayer.resume();
            resumeSelected.setEnabled(false);
            pauseSelected.setEnabled(true);
        }
    }
));
listenMenu.add(stopSelected = createMenuItem("Stop",
    "Stops playing songs.",
    KeyEvent.VK_S, KeyEvent.VK_S, ActionEvent.ALT_MASK,
    new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            musicPlayer.stop();
        }
    }
));
pauseSelected.setEnabled(false);
resumeSelected.setEnabled(false);
stopSelected.setEnabled(false);
menuBar.add(fileMenu);
menuBar.add(analyzeMenu);
menuBar.add(listenMenu);
menuBar.add(windowMenu);
frame.setJMenuBar(menuBar);
}

/**
 * Constructs a JMenuItem with the specified title, popup text, quickkeys
 * and ActionListener.
 */

```

```

private JMenuItem createMenuItem(String title, String popup,
                                int quickkey, int keyevent, int actionevent,
                                ActionListener a) {
    JMenuItem menu = new JMenuItem(title, quickkey);
    menu.setAccelerator(KeyStroke.getKeyStroke(keyevent, actionevent));
    menu.getAccessibleContext().setAccessibleDescription(popup);
    menu.addActionListener(a);
    return menu;
}

/**
 * Called by the MusicPlayer when a song stops, or when the user
 * decides. This disables the pause, resume and stop controls.
 */

public void songStopped() {
    musicPlayer.stop();
    pauseSelected.setEnabled(false);
    resumeSelected.setEnabled(false);
    stopSelected.setEnabled(false);
}

/**
 * Called whenever an internal frame should show up.
 * This method makes sure frames are staggered on the screen
 * (Specifically, with 10 spots, each 25 pixels apart).
 */

public void addInternalFrame(JInternalFrame jif) {
    int start = 5 + (frameCounter++ % 10) * 25;
    jif.setLocation(start, start);
    desktop.add(jif);
    jif.setVisible(true);
    jif.pack();
}

/**
 * Called whenever a graph needs to be shown. This is passed in
 * a type of music graph. If nodes are selected, for each one,
 * a new MusicGraph (of the given type) is constructed and
 * given the Analyzer associated with that node.
 */

private void showGraphs(MusicGraph mg) {
    Analyzer[] nodes = treeMan.getSelectedNodes();
    for (int i = 0; nodes != null && i < nodes.length; i++) {
        Analyzer node = nodes[i];
        if (node.containsChildren()) {
            MusicGraph newGraph = mg.createMusicGraph(node);
            GraphFrame gf = new GraphFrame(newGraph);
            addInternalFrame(gf);
            newGraph.drawGraph(true);
        } else {
            Toolkit.getDefaultToolkit().beep();
        }
    }
}
}

```

```

import javax.sound.midi.*;
import java.util.*;

/**
 * Analyzer.java <p>
 *
 * This abstract class implements many of the most important analysis
 * features itself for subclasses, based on generic properties of all
 * Analyzers (that is, all Analyzers contain a Note array which can be
 * accessed through getNotes()).
 *
 * @author      Greg Cipriano
 */

abstract public class Analyzer {
    protected Chord givenKey;

    protected float songLength;
    protected int meanPitch, stdPitchDeviation;
    protected int weightedMeanPitch, weightedStdPitchDeviation;
    protected int maxSimultaneousVoices, mostOftenSoundedNote;
    protected int countOfMostOftenSoundedNote;
    protected int highestNote, lowestNote, largestInterval;
    protected int mostFrequentInterval;
    protected int countOfMostFrequentInterval;

    /**
     * Called whenever notes are changed. This method calls
     * other methods which initialize various statistics.
     */
    protected void analyzeNotes() {
        analyzeNoteMean();
        analyzeWeightedNoteMean();
        analyzeRanges();
        getVoiceDensity(1);
        getNoteFrequencies();
        getIntervalCounts();
    }

    /**
     * Describes whether the Notes[] array is invalid and
     * needs to be rebuilt.
     */

    protected boolean notesInvalid = true;

    /**
     * Returns a String representing the key stored
     * in the MIDI file associated with this Analyzer,
     * if it exists. If not, return "None given".
     */

    public String getGivenKey() {
        if (givenKey == null) { return "None given"; }
        else {
            return givenKey.toString();
        }
    }

    /**
     * Returns the highest number of voices playing simultaneously
     * at any point in the song.
     * @see    #getVoiceDensity(int).
     */

    public int getMaxSimultaneousVoices() { return maxSimultaneousVoices; }

    /**
     * Returns the note played most often in the song.
     *
     * @see    #getNoteFrequencies()
     */

    public int getMostOftenSoundedNote() { return mostOftenSoundedNote; }

    /**
     * Returns how many times the most often played note
     * sounded throughout the song.
     *
     * @see    #getNoteFrequencies()
     */

    public int getCountOfMostOftenSoundedNote() { return countOfMostOftenSoundedNote; }

    /**
     * Returns the average (statistical mean) note over all
     * the notes in the Note array. This doesn't take into
     * account the length of each note.
     *
     * @see    #analyzeNoteMean()
     */

```

```

public int getMeanPitch() { return meanPitch; }

/**
 * Returns the standard deviation from the mean over
 * all notes.
 *
 * @see      #analyzeNoteMean()
 * @see      #getMeanPitch()
 */

public int getStdPitchDeviation() { return stdPitchDeviation; }

/**
 * Returns the average (statistical mean) note played,
 * over the entire Note array, weighted according
 * to how long that note played.
 *
 * @see      #analyzeWeightedNoteMean()
 */

public int getWeightedMeanPitch() { return weightedMeanPitch; }

/**
 * Returns the standard deviation from the weighted mean over
 * all notes.
 *
 * @see      #analyzeWeightedNoteMean()
 * @see      #getWeightedMeanPitch()
 */

public int getWeightedStdPitchDeviation() { return weightedStdPitchDeviation; }

/**
 * Returns the highest note contained in the Note array.
 *
 * @see      #analyzeRanges()
 */

public int getHighestNote() { return highestNote; }

/**
 * Returns the lowest note contained in the Note array.
 *
 * @see      #analyzeRanges()
 */

public int getLowestNote() { return lowestNote; }

/**
 * Returns the range of notes over the Note array.
 * Equivalent to getHighestNote() - getLowestNote() + 1.
 *
 * @see      #analyzeRanges()
 * @see      #getHighestNote()
 * @see      #getLowestNote()
 */

public int getRange() { return (getHighestNote() - getLowestNote() + 1); }

/**
 * Returns the largest interval existing between
 * simultaneously playing notes in the Note array.
 *
 * @see      #getIntervalCounts()
 */

public int getLargestInterval() { return largestInterval; }

/**
 * Returns the most frequently played interval between
 * simultaneous notes.
 *
 * @see      #getIntervalCounts()
 */

public int getMostFrequentInterval() { return mostFrequentInterval; }

/**
 * Returns how often the most frequently played interval
 * was played.
 *
 * @see      #getMostFrequentInterval()
 * @see      #getIntervalCounts()
 */

public int getCountOfMostFrequentInterval() { return countOfMostFrequentInterval; }

/**
 * Returns the length of the song, in beats.
 *
 * @see      #analyzeRanges()
 */

```

```

public float getSongLength() { return songLength; }

/**
 * Returns the number of notes contained in the 'notes' array.
 * If the array happens to be null, this returns 0;
 */
public int getNumNotes() {
    Note[] notes = getNotes();
    if (notes == null) return 0;
    else {
        return notes.length;
    }
}

/**
 * Get an array of chords, sampled a fixed number (4)
 * of times per beat.
 */
public Chord[] getChords() {
    Note[] notes = getNotes();
    float samplesPerBeat = 4f;
    int numSamples = (int) (getSongLength() * samplesPerBeat);
    int[] count = new int[numSamples];

    // This may be too tricky...
    // Keep note data in a bit field (integer)

    for (int i = 0; i < getNumNotes(); i++) {
        int start = (int) (notes[i].getTimeOn() * samplesPerBeat);
        int end = (int) (notes[i].getTimeOff() * samplesPerBeat);
        int pitch = notes[i].getPitch() % 12;

        for (int j = start; j < end && j < numSamples; j++) {
            if ((count[j] >> pitch) % 2 == 0) {
                count[j] += 1 << pitch;
            }
        }
    }

    // Now weed out duplicate and undetermined chords.
    // Also, for chords that don't know their identity,
    // reduce possibilities based on previous chord.

    Vector progression = new Vector();
    int progSize = 0;
    float sampleWidth = 1 / samplesPerBeat;
    for (int i = 0; i < numSamples; i++) {
        Chord c = new Chord(count[i], i / samplesPerBeat, sampleWidth);
        if (c.getType() != Chord.NOTACHORD) {
            if (progSize > 0) {
                Chord oldChord = (Chord) progression.elementAt(progSize - 1);
                if (c.getType() == Chord.UNKNOWN) {
                    c.reducePossibilities(oldChord);
                }
                if (oldChord.equals(c)) {
                    float diff = c.getStart() - oldChord.getEnd();
                    oldChord.inclLength(diff + sampleWidth);
                } else {
                    progression.add(c);
                    progSize++;
                }
            } else {
                // First real chord found. So reduce possibilities
                // based on the key found using vector analysis.
                c.reducePossibilities(getKeyVector());
                progression.add(c);
                progSize++;
            }
        }
    }

    // Convert the vector back into an array and return;

    int numChords = progression.size();
    Chord[] p = new Chord[numChords];
    for (int i = 0; i < numChords; i++) {
        p[i] = (Chord) progression.elementAt(i);
    }
    return p;
}

/**
 * Returns an integer representing the key in the range
 * from 0 to 12, 0 == C major.
 */
public int getKeyVector() {
    double[] position = new double[2];

```

```

    Note[] notes = getNotes();
    for (int i = 0; i < notes.length; i++) {
        notes[i].addVector(position, 1);
    }
    double xPos = position[0];
    double yPos = position[1];
    double angle = Math.atan(yPos / xPos);

    if (xPos < 0) {
        angle = Math.PI / 2 - angle;
    } else {
        angle = 3 * Math.PI / 2 - angle;
    }

    int quadrant = (int) (6 * angle / Math.PI + Math.PI / 6);
    int key = (quadrant * 7) % 12;
    return key;
}

/**
 * Returns the range of notes the song goes through, sampled at
 * 'interval' times per beat. If absoluteCount is true, this
 * array contains two values for each sample: the lowest note,
 * followed by the highest note. Otherwise, the array contains
 * just one value: the range (or difference between these values).
 */

public int[] getPitchRange(int interval, boolean absoluteCount) {
    Note[] notes = getNotes();
    int width = (int) (getSongLength() * interval);
    int[] lowNote = new int[width];
    for (int i = 0; i < width; i++) { lowNote[i] = 512; }
    int[] highNote = new int[width];
    for (int i = 0; i < width; i++) { highNote[i] = -1; }
    for (int i = 0; i < notes.length; i++) {
        int start = (int) (notes[i].getTimeOn() * interval);
        int end = (int) (notes[i].getTimeOff() * interval);
        int pitch = notes[i].getPitch();
        for (int j = start; j < end && j < width; j++) {
            if (pitch > highNote[j]) { highNote[j] = pitch; }
            if (pitch < lowNote[j]) { lowNote[j] = pitch; }
        }
    }
    if (absoluteCount) {
        int [] r = new int[2 * width];
        for (int i = 0; i < width; i++) {
            r[2 * i] = lowNote[i];
            r[2 * i + 1] = highNote[i];
        }
        return r;
    } else {
        for (int i = 0; i < width; i++) {
            highNote[i] -= lowNote[i];
        }
        return highNote;
    }
}

/**
 * Counts the number of times each note sounds. Returns
 * an array of the size of the range containing this info.
 * This method also initializes 'mostOftenSoundedNote',
 * and 'countOfMostOftenSoundedNote'.
 */

public int[] getNoteFrequencies() {
    Note[] notes = getNotes();
    int[] noteCount = new int[getRange()];
    countOfMostOftenSoundedNote = 0;
    for (int i = 0; i < getNumNotes(); i++) {
        int pitch = notes[i].getPitch() - lowestNote;
        noteCount[pitch]++;
        if (noteCount[pitch] > countOfMostOftenSoundedNote) {
            mostOftenSoundedNote = notes[i].getPitch();
            countOfMostOftenSoundedNote = noteCount[pitch];
        }
    }
    return noteCount;
}

/**
 * Returns how many notes are playing simultaneously, sampled at
 * 'interval' times per beat. This method also initializes
 * 'maxSimultaneousVoices'.
 */

public int[] getVoiceDensity(int interval) {
    Note[] notes = getNotes();
    int width = (int) (getSongLength() * interval);
    int[] count = new int[width];

```

```

maxSimultaneousVoices = 0;
for (int i = 0; i < getNumNotes(); i++) {
    int start = (int) (notes[i].getTimeOn() * interval);
    int end = (int) (notes[i].getTimeOff() * interval);
    for (int j = start; j < end && j < width; j++) {
        count[j]++;
        if (count[j] > maxSimultaneousVoices) {
            maxSimultaneousVoices = count[j];
        }
    }
}
return count;
}

/**
 * Runs through the array of Notes collecting the number of times
 * a specific interval (counted when two notes are sounding simultaneously)
 * occurs. This method also initializes 'largestInterval',
 * 'mostFrequentInterval' and 'countOfMostFrequentInterval'.
 *
 * @returns an array containing these intervals.
 */
public int[] getIntervalCounts() {
    Note[] notes = getNotes();
    int[] intervalCount = new int[128];
    largestInterval = 0;
    countOfMostFrequentInterval = 0;
    for (int i = 0; i < getNumNotes(); i++) {
        float start = notes[i].getTimeOn();
        float end = notes[i].getTimeOff();
        for (int j = i + 1; j < notes.length && notes[j].getTimeOn() < end; j++) {
            int ip = notes[i].getPitch();
            int jp = notes[j].getPitch();
            int interval = (ip > jp) ? ip - jp : jp - ip;
            if (interval > largestInterval) { largestInterval = interval; }
            intervalCount[interval]++;
            if (intervalCount[interval] > countOfMostFrequentInterval) {
                mostFrequentInterval = interval;
                countOfMostFrequentInterval = intervalCount[interval];
            }
        }
    }
    return intervalCount;
}

/**
 * analyzeNoteMean() finds both the mean and the standard
 * deviation of the pitches of all notes. These statistics are
 * stored in local variables: 'meanPitch' and 'stdPitchDeviation'.
 */
protected void analyzeNoteMean() {
    Note[] n = getNotes();
    int numNotes = getNumNotes();

    // 'zero' local statistics
    meanPitch = 0;
    stdPitchDeviation = 0;

    for (int i = 0; i < numNotes; i++) {
        meanPitch += n[i].getPitch();
    }

    // meanPitch and stdPitchDeviation are valid
    // only if the number of notes is greater than 1.
    // (NOTE: meanPitch still works if numNotes == 1),
    // it just doesn't need further calculation
    // (meanPitch = n[0].pitch and stdPitchDeviation = 0)
    if (numNotes > 1) {
        double preciseMean = (double) meanPitch / (double) numNotes;
        double devSum = 0;
        for (int i = 0; i < numNotes; i++) {
            devSum += Math.pow((double) n[i].getPitch() - preciseMean, 2);
        }
        meanPitch = (int) preciseMean;
        stdPitchDeviation = (int) Math.sqrt(devSum / (numNotes - 1));
    }
}

/**
 * analyzeWeightedNoteMean() finds both the mean and the standard
 * deviation of the pitches of all notes. These statistics are
 * stored in local variables: 'weightedMeanPitch' and
 * 'weightedStdPitchDeviation'.
 */
protected void analyzeWeightedNoteMean() {
    Note[] n = getNotes();
    int numNotes = getNumNotes();

    // 'zero' local statistics

```



```

        weightedMeanPitch = 0;
        weightedStdPitchDeviation = 0;

        // temporary floats...
        float tempSum = 0;
        float lengthSum = 0;

        for (int i = 0; i < numNotes; i++) {
            tempSum += n[i].getPitch() * n[i].getDuration();
            lengthSum += n[i].getDuration();
        }

        // same sort of thing goes for weightedNoteMean, etc
        // as in regular noteMean.
        if (numNotes > 1) {
            double preciseMean = (double) tempSum / (double) lengthSum;
            double devSum = 0;
            for (int i = 0; i < numNotes; i++) {
                devSum += Math.pow((double) n[i].getPitch() - preciseMean, 2);
            }
            weightedMeanPitch = (int) preciseMean;
            weightedStdPitchDeviation = (int) Math.sqrt(devSum / (numNotes - 1));
        }
    }

    /**
     * analyzeRanges() finds the lowest and highest notes in the song,
     * as well as its total length (in beats). These statistics are
     * stored in local variables: 'lowestNote', 'highestNote' and
     * 'songLength'.
     */
    protected void analyzeRanges() {
        Note[] n = getNotes();

        // 'zero' local statistics.
        lowestNote = 512;
        highestNote = 0;
        songLength = 0;

        for (int i = 0; i < getNumNotes(); i++) {
            int pitch = n[i].getPitch();
            if (pitch > highestNote) { highestNote = pitch; }
            if (pitch < lowestNote) { lowestNote = pitch; }

            float noteEnd = n[i].getTimeOff() + 1;
            if (noteEnd > songLength) { songLength = noteEnd; }
        }
    }

    /**
     * To be filled in by subclasses, returns the notes
     * contained in the Analyzer.
     */
    abstract public Note[] getNotes();

    /**
     * Returns the shortest representation of the Analyzer...
     * This will appear on the JTree Node associated
     * with this Analyzer.
     */
    abstract public String toString();

    /**
     * Returns a detailed String describing the Analyzer...
     * titles most internal frames.
     */
    abstract public String getFullTitle();

    /**
     * Returns a Sequence for this Analyzer, so MusicPlayer
     * can play the song. Defined in subclasses.
     */
    abstract public Sequence getSequence();

    /**
     * Returns whether this Analyzer contains other enabled
     * Analyzers subordinate to it.
     * Example: GroupAnalyzers contain SongAnalyzers,
     * which contain TrackAnalyzers.
     */
    abstract public boolean containsChildren();
}

```

```

import java.util.*;

/**
 * Chord.java <p>
 *
 * Encapsulates a chord, which knows its root,
 * type, starting time and length.
 *
 * @author      Greg Cipriano
 */

public class Chord {
    private Vector possibleChords = new Vector();
    private boolean exactMatchFound = false;

    private float length;
    private float start;
    private int rootPitch;
    private int type;

    public static final int MAJ3 = 1 + 16;
    public static final int MIN3 = 1 + 8;
    public static final int MAJ = MAJ3 + 128;
    public static final int MIN = MIN3 + 128;
    public static final int AUG = MAJ3 + 256;
    public static final int DIM = MIN3 + 64;
    public static final int MAJ6NO5 = MAJ3 + 512;
    public static final int MIN6NO5 = MIN3 + 512;
    public static final int MAJ7NO5 = MAJ3 + 2048;
    public static final int MIN7NO5 = MIN3 + 1024;
    public static final int DOM7NO5 = MAJ3 + 1024;
    public static final int MAJ6 = MAJ + 512;
    public static final int MIN6 = MIN + 512;
    public static final int MAJ7 = MAJ + 2048;
    public static final int MIN7 = MIN + 1024;
    public static final int DOM7 = MAJ + 1024;
    public static final int FULLDIM7 = DIM + 512;
    public static final int HALFDIM7 = DIM + 1024;
    public static final int MAJ9NO5 = MAJ7NO5 + 4;
    public static final int DOM9NO5 = DOM7NO5 + 4;
    public static final int MIN9NO5 = MIN7NO5 + 4;
    public static final int MAJ9 = MAJ7 + 4;
    public static final int DOM9 = DOM7 + 4;
    public static final int MIN9 = MIN7 + 4;

    /**
     * Signifies that the notes given in the notemask
     * do not form a legal chord, among the known chord types.
     */

    public static final int NOTACHORD = -1;

    /**
     * Signifies that the notes given map to more than one
     * chord, and that reducePossibilities() needs to be called.
     *
     * @see      #reducePossibilities(int)
     */

    public static final int UNKNOWN = -2;

    /**
     * The list of legal chords this chord can map to, in order
     * from smallest to largest. Chords will be tried in reverse order,
     * with more 'full' chords the first have matching attempted.
     */

    private static int[] chordList = { MAJ, MIN, AUG, DIM, MAJ6NO5, MIN6NO5,
                                         MAJ7NO5, MIN7NO5, DOM7NO5, MAJ6, MIN6,
                                         MAJ7, MIN7, DOM7, FULLDIM7, HALFDIM7,
                                         MAJ9NO5, DOM9NO5, MIN9NO5,
                                         MAJ9, DOM9, MIN9};

    /**
     * Entry point for testing the Chord class.
     */

    public static void main(String[] args) {
        int mask = 0;
        for (int i = 0; i < args.length; i++) {
            mask += (int) Math.pow(2, Double.parseDouble(args[i]));
        }
        Chord c = new Chord(mask, 0, 1);
        System.out.println(c);
    }

    /**
     * Explicitly construct a chord with the given root, type,
     * starting time and length.
     */

```

```

public Chord(int root, int t, float st, float clength) {
    rootPitch = root % 12;
    type = t;
    start = st;
    length = clength;
}

/**
 * Constructor for the chord. This uses a notemask, and
 * tries to match that mask to one of the known patterns
 * by rotating and comparing. If a match is found, the amount
 * this was rotated determines the root.
 */
public Chord(int notemask, float st, float clength) {
    type = notemask;
    for (int i = 0; i < 12; i++) {
        addIfMatch(type, i);
        type = (type * 2) % 4095; //Rotate mask
    }
    int numFound = possibleChords.size();
    if (numFound > 0) {
        if (exactMatchFound) {
            for (int i = 0; i < possibleChords.size(); i++) {
                PossibleChord p = (PossibleChord) possibleChords.elementAt(i);
                if (!p.exactMatch) {
                    possibleChords.remove(p);
                }
            }
            numFound = possibleChords.size();
        }
        if (numFound > 1) {
            type = UNKNOWN;
        } else {
            PossibleChord p = (PossibleChord) possibleChords.elementAt(0);
            rootPitch = p.root;
            type = p.type;
        }
    } else {
        type = NOTACHORD;
    }
    start = st;
    length = clength;
}

/**
 * Compares the mask to all known chord types, starting with
 * higher order chords. This checks if given mask INCLUDES
 * the notes of a given chord, and records also whether the
 * mask is EXACTLY the same as that chord. Those that match
 * exactly will be given preference.
 */
private void addIfMatch(int mask, int rotation) {
    // Traverse the list of legal chords from largest (most full)
    // to smallest. If a match is made, return immediately --
    // since large chords are more accurate and include small
    // chords.
    for (int i = chordList.length - 1; i >= 0; i--) {
        // Check if mask contains all of chordList's notes
        // and possibly more
        if ((mask & chordList[i]) == chordList[i]) {
            int possRoot = (12 - rotation) % 12;

            boolean exactMatch = (mask == chordList[i]);
            exactMatchFound |= exactMatch;

            possibleChords.add(new PossibleChord(possRoot, chordList[i], exactMatch));
            //System.out.println(chordList[i] + " " + Note.toNote(possRoot));
            //System.out.println();
            return;
        }
    }
}

/**
 * Returns a string representation of this chord.
 */
public String toString() {
    String out = Note.toNote(rootPitch, false);
    if (type == MAJ) { out += " maj"; }
    else if (type == MIN) { out += " min"; }
    else if (type == MAJ7) {
        type == MAJ7NO5) { out += " maj7"; }
        type == MIN7NO5) { out += " min7"; }
    }
    else if (type == DOM7) {
        type == DOM7NO5) { out += " dom7"; }
    }
    else if (type == DIM) { out += " dim"; }
}

```

```

    else if (type == AUG) { out += " aug"; }
    else if (type == MAJ6 || type == MAJ6NO5) { out += " maj6"; }
    else if (type == MIN6 || type == MIN6NO5) { out += " min6"; }
    else if (type == FULLDIM7) { out += " dim7"; }
    else if (type == HALFDIM7) { out += " 1/2 dim7"; }
    else if (type == MAJ9 || type == MAJ9NO5) { out += " maj9"; }
    else if (type == DOM9 || type == DOM9NO5) { out += " dom9"; }
    else if (type == MIN9 || type == MIN9NO5) { out += " min9"; }
    else if (type == UNKNOWN) { out = "TOO MANY CHOICES!!!"; }
    else if (type == NOTACHORD) { out = "ERROR: NOT A CHORD!!"; }
    else {
        out = "ERROR: UNKNOWN TYPE";
    }
    return out;
}

/**
 * Returns the root of the chord, hopefully after
 * that root has been uniquely determined.
 */

public int getRootPitch() { return rootPitch; }

/**
 * Returns the type of chord this is, among the
 * possibilities.
 */

public int getType() { return type; }

/**
 * Returns the length of this chord, in beats.
 */

public float getLength() { return length; }

/**
 * Returns when this chord came into effect, in beats.
 */

public float getStart() { return start; }

/**
 * Returns when this chord ends. Equivalent to
 * getStart() + getLength().
 */

public float getEnd() { return getStart() + getLength(); }

/**
 * Reduce the possibilities given the last chord.
 * Hands off the chords root to reducePossibilities(int).
 */

public void reducePossibilities(Chord lastChord) {
    reducePossibilities(lastChord.getRootPitch());
}

/**
 * Reduce the possible chord down to one probable chord
 * based on the distance the root moves from the last chord.
 * This is predicated on the assumption that a composer
 * will move the root as little as possible between
 * consecutive chords...
 */

public void reducePossibilities(int oldRoot) {
    int bestRoot = 0;
    int bestType = 0;
    int bestDistance = 1000;
    for (int i = 0; i < possibleChords.size(); i++) {
        PossibleChord pc = (PossibleChord) possibleChords.elementAt(i);
        int possRoot = pc.root;
        int possType = pc.type;
        int distance = (possRoot > oldRoot) ? possRoot - oldRoot
            : oldRoot - possRoot;

        // account for wrapping of chromatic scale
        if (distance > 6) { distance = 12 - distance; }

        if (distance < bestDistance) {
            bestRoot = possRoot;
            bestType = possType;
            bestDistance = distance;
        }
    }
    rootPitch = bestRoot;
    type = bestType;
}

/**

```

```

* Overrides equals() in Object. Here, a chord equals
* another chord when it has the same root pitch and
* the same type.
*/

public boolean equals(Object c) {
    if (c instanceof Chord) {
        return (getRootPitch() == ((Chord) c).getRootPitch() &&
            getType() == ((Chord) c).getType());
    }
    return false;
}

/**
 * A similar chord was found, so coalesce the two
 * by increasing this chord's length.
 */

public void incLength(float l) { length += l; }

/**
 * Convenience method to set the root pitch of this
 * chord.
 */

public void setRootPitch(int p) { rootPitch = p; }

/**
 * A simple structure to hold on to this chord's
 * possibilities...
 */

private class PossibleChord {

    /** one possible root for this chord */
    public int root;

    /** the type of this chord, as one of the known types */
    public int type;

    /** if this chord matched EXACTLY one of the known patterns */
    public boolean exactMatch;

    /**
     * Constructs a chord with the given root, type and
     * match quality.
     */

    private PossibleChord(int r, int t, boolean b) {
        root = r;
        type = t;
        exactMatch = b;
    }
}

```

```

import java.awt.*;
import javax.swing.*;
import javax.swing.table.*;

/**
 * ChordProgressionFrame.java <p>
 *
 * ChordProgressionFrame shows a list of the guessed chords
 * in a song, assuming a maximum of 2 per measure with
 * duplicates removed.
 *
 * @author      Greg Cipriano
 */

public class ChordProgressionFrame extends JFrame {

    /**
     * Constructs the frame, setting up all of the cells of the
     * JTable, adding that table to a scroll pane and that pane
     * to the frame itself. If there are no recognizable chords
     * in the song, add a label with an error message (so user
     * at least gets some feedback that chord elucidation failed.)
     */

    public ChordProgressionFrame(Analyzer a) {
        super(a.getFullTitle() + ": Chord Progression", true, true, true, true);
        a.analyzeNotes();
        Chord[] chords = a.getChords();
        JComponent c = null;
        if (chords.length != 0) {
            String[][] columnValues = new String[chords.length][3];
            for (int i = 0; i < chords.length; i++) {
                String start = ((int) chords[i].getStart() / 4 + 1) + " : "
                    + (chords[i].getStart() % 4 + 1);
                columnValues[i][0] = start;
                columnValues[i][1] = Float.toString(chords[i].getLength());
                columnValues[i][2] = chords[i].toString();
            }
            c = createTable(columnValues);
        } else {
            c = new JLabel("No Chords found!", JLabel.CENTER);
        }
        setContentPane(new JScrollPane(c));
        setPreferredSize(new Dimension(360, 260));
        pack();
    }

    /**
     * Creates a JTable with the entries denoted by columnValues and the
     * column headers: "Starting Measure", "Length", and "Chord".
     * Other parameters -- including making it uneditable, and making
     * the TableModel specific -- are also added to this JTable.
     */

    private JTable createTable(final String[][] columnValues) {
        final String[] columnNames = { "Measure:Beat", "Length", "Chord" };

        TableModel dataModel = new AbstractTableModel() {
            public String getColumnName(int col) {
                return columnNames[col];
            }
            public int getColumnCount() { return columnValues[0].length; }
            public int getRowCount() { return columnValues.length; }
            public boolean isCellEditable(int row, int col) { return false; }
            public Object getValueAt(int row, int col) {
                return columnValues[row][col];
            }
        };
        JTable table = new JTable(dataModel);
        table.getColumnModel().getColumn(2).setMaxWidth(85);
        return table;
    }
}

```

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * DataGraph.java <p>
 *
 * DataGraph is an abstract graph set up to draw integral data.
 * It leaves up to sub-classes to define (1) what that data
 * is and (2) what its maximum value is. Minimum is assumed to be 0.
 *
 * @author      Greg Cipriano
 */

public abstract class DataGraph extends ScrollBarGraph {
    private final String[] titles = { "Samples per beat" };
    private final int[] defaultValues = { 30 };
    private final int PREFERREDWIDTH = 500;
    private final int PREFERREDHEIGHT = 200;
    private final int OFFSET = 12;
    private JPanel view;
    private JPanel innerPane;
    private JLabel label;
    private Image offscreenI;
    private Graphics backpage;
    private ImageIcon icon;
    private int beatSize;

    /**
     * Stores the Analyzer this graph works with, protected
     * so that only subclasses can get at it.
     */
    protected Analyzer analyzer;

    /**
     * The maximum value this data set attains, used
     * for providing scale in the graph.
     */
    protected int maxValue;

    /**
     * The data to be rendered.
     */
    protected int[] data;

    /**
     * Empty constructor. Doesn't set up anything. Used when an
     * object class is needed.
     */
    public DataGraph() {}

    /**
     * Real constructor. Takes an Analyzer, and sets up the
     * view's scrollbar and panel.
     */
    public DataGraph(Analyzer a) {
        analyzer = a;
        analyzer.analyzeNotes();
        innerPane = new JPanel();
        innerPane.setPreferredSize(
            new Dimension(PREFERREDWIDTH, PREFERREDHEIGHT + OFFSET));
        innerPane.addComponentListener(this);

        scrollbar = new JScrollBar(JScrollBar.HORIZONTAL);
        setVariable(0, defaultValues[0], false);
        view = new JPanel(new BorderLayout());
        scrollbar.setBlockIncrement(PREFERREDWIDTH - 20);
        scrollbar.addAdjustmentListener(this);
        view.add(innerPane, BorderLayout.CENTER);
        view.add(scrollbar, BorderLayout.SOUTH);
    }

    public String[] getVariableTitles() { return titles; }
    public int[] getDefaultValues() { return defaultValues; }
    public JComponent getView() { return view; }

    /**
     * Draws the graph, creating the offscreen image and graphics contexts
     * if they haven't been created yet.
     */
    public void drawGraph(boolean repaintIcon) {
        int start = scrollbar.getValue();
        int height = PREFERREDHEIGHT + OFFSET;
    }

```

```

int width = innerPane.getSize().width;
int end = start + width;

// Set up the back buffer if necessary, making it the size of
// the screen, so if the frame holding this graph is resized,
// it's still big enough.
if (offscreenI == null) {
    int screenWidth = Toolkit.getDefaultToolkit().getScreenSize().width;
    setupBackBuffers(screenWidth, height);
}

backPage.setColor(Color.white);
backPage.fillRect(0, 0, width, height);
backPage.setColor(Color.red);

// Draw data as vertical lines
int scalar = (height - OFFSET) / maxValue;
for (int i = 0; i < width && (i + start) < data.length; i++) {
    int h = data[i + start] * scalar;
    backPage.drawLine(i, height, i, height - h);
}

// 4 quarter-notes per measure
int measureSize = 4 * beatSize;

// Draw measure dividing lines. Also draw measure numbers.
backPage.setColor(new Color(100, 100, 100));
int measureNumber = start / measureSize + 1;
for (int i = measureSize - (start % measureSize); i < end + measureSize; i += measureSize) {
    backPage.drawLine(i, 0, i, height);
    backPage.drawString("" + (measureNumber++), i - measureSize / 2, OFFSET - 2);
}
backPage.drawLine(0, OFFSET, width, OFFSET);

// Draw horizontal lines dividing view according to possible data values.
backPage.setColor(Color.black);
for (int i = height; i > OFFSET; i -= scalar) {
    backPage.drawLine(0, i, width, i);
}

if (repaintIcon) {
    // A hack to get the icon to paint REALLY fast. :)
    icon.paintIcon(label, label.getGraphics(),
        0, (innerPane.getSize().height - height) / 2);
}

/**
 * Create offscreen images, graphics contexts
 * for the data graph.
 */

public void setupBackBuffers(int width, int height) {
    offscreenI = view.createImage(width, height);
    innerPane.setBackground(Color.gray);
    innerPane.setLayout(new BorderLayout());
    innerPane.add(label = new JLabel(icon = new ImageIcon(offscreenI)), BorderLayout.WEST);
    innerPane.validate();

    backPage = offscreenI.getGraphics();
    backPage.setFont(new Font("Serif", Font.BOLD, 11));
}

/**
 * Sets the variable at 'varnum' with the value. If redraw
 * is set, repaint the graph to reflect this change. This
 * also updates the scrollbar's extent and position.
 */

public void setVariable(int varNum, int value, boolean redraw) {
    if (varNum == 0) {
        int newPosition = 0;
        if (beatSize != 0) {
            newPosition = (scrollbar.getValue() * value) / beatSize;
        }
        beatSize = value;
        scrollbar.setValues(newPosition, innerPane.getSize().width, 0, data.length);
    }
    if (redraw) { drawGraph(true); }
}
}

```



```

import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * GraphFrame.java <p>
 *
 * GraphFrame is an JInternalFrame that contains
 * a MusicGraph and a number of horizontal JSlider bars
 * below it -- based on what values the MusicGraph wants
 * to listen to.
 *
 * @author      Greg Cipriano
 */

public class GraphFrame extends JInternalFrame implements ChangeListener {
    private MusicGraph mg;
    private int numSliders;
    private JSlider[] sliders;

    /**
     * Construct the frame according to the given MusicGraph...
     * Add as many scrollbars as the graph has controllable variables,
     * querying the graph for both default values and the title for each.
     */

    public GraphFrame(MusicGraph mg) {
        super(mg.toString(), true, true, true, true);
        this.mg = mg;
        String[] optionsTitles = mg.getVariableTitles();
        int[] values = mg.getDefaultValues();
        JComponent view = mg.getView();

        if (optionsTitles == null) {
            // Here, no scrollbars are needed, so
            // just make the frame hold just a scrollpane.
            setContentPane(new JScrollPane(view));
        } else {
            numSliders = values.length;
            sliders = new JSlider[numSliders];

            JPanel options = new JPanel();
            options.setLayout(new BoxLayout(options, BoxLayout.Y_AXIS));
            for (int i = 0; i < numSliders; i++) {
                JLabel title = new JLabel(" " + optionsTitles[i] + ": ");
                sliders[i] = new JSlider(JSlider.HORIZONTAL, 1, 120, values[i]);
                sliders[i].setMajorTickSpacing(10);
                sliders[i].setPaintTicks(true);
                sliders[i].addChangeListener(this);
                JPanel oneOption = new JPanel();
                oneOption.setLayout(new BorderLayout());
                oneOption.add(title, BorderLayout.WEST);
                oneOption.add(sliders[i], BorderLayout.CENTER);
                options.add(oneOption);
            }

            JPanel content = new JPanel();
            content.setLayout(new BorderLayout());
            content.add(view, BorderLayout.CENTER);
            content.add(options, BorderLayout.SOUTH);
            setContentPane(content);
        }
    }

    /**
     * Listen to a change in the slider bar, updating the
     * MusicGraph to that change.
     */

    public void stateChanged(ChangeEvent e) {
        Object source = e.getSource();
        for (int i = 0; i < numSliders; i++) {
            if (source == sliders[i]) {
                mg.setVariable(i, sliders[i].getValue(), true);
            }
        }
    }
}

```

```
import java.util.*;
import javax.sound.midi.*;
```

```
/**
 * GroupAnalyzer.java <p>
 *
 * Manages a group of songs (SongAnalyzers), providing analysis
 * on these. This class provides facilities to add and remove
 * SongAnalyzers from its internal Vector, and also has various
 * methods which provide analysis capabilities above that provided
 * by the base Analyzer class.
 *
 * @author      Greg Cipriano
 */
```

```
public class GroupAnalyzer extends Analyzer {
    private String groupName;
    private Vector songAnalyzers = new Vector();
    private Note[] notes;
```

```
/**
 * Initializes a GroupAnalyzer, which must have a name
 * and which initially contains no SongAnalyzers.
 */
```

```
public GroupAnalyzer(String name) { setName(name); }
```

```
/**
 * Changes the name of this Group. When this GroupAnalyzer
 * is placed in a JTree, this name is what appears in the
 * tree's view.
 */
```

```
public void setName(String name) { groupName = name; }
```

```
/**
 * Returns a boolean as to whether this group contains
 * at least one active song (a song with at least one
 * active track).
 */
```

```
public boolean containsChildren() {
    for (int i = 0; i < songAnalyzers.size(); i++) {
        if (((SongAnalyzer) songAnalyzers.elementAt(i)).containsChildren()) {
            return true;
        }
    }
    return false;
}
```

```
/**
 * Adds a SongAnalyzer to this objects vector.
 * This should invalidate the notes, requiring them to be
 * rebuilt.
 */
```

```
public void addSongAnalyzer(SongAnalyzer sa) {
    songAnalyzers.add(sa);
    sa.setGroupAnalyzer(this);
    invalidate();
}
```

```
/**
 * Removes a SongAnalyzer to this objects vector. If
 * sa is null, nothing should happen. Otherwise, this
 * should invalidate the notes, requiring them to be
 * rebuilt.
 */
```

```
public void removeSongAnalyzer(SongAnalyzer sa) {
    songAnalyzers.remove(sa);
    invalidate();
}
```

```
/**
 * Called to force Note array to be recomputed the next
 * time it's accessed.
 */
```

```
protected void invalidate() { notesInvalid = true; }
```

```
/**
 * Construct a Note array if necessary by gathering up
 * all the notes referenced by all SongAnalyzers underneath
 * this GroupAnalyzer.
 */
```

```
public Note[] getNotes() {
    if (notesInvalid) {
        int totalLength = 0;
        int counter = 0;
```

```

        for (int i = 0; i < songAnalyzers.size(); i++) {
            totalLength += ((SongAnalyzer) songAnalyzers.elementAt(i)).getNumNotes();
        }
        if (totalLength == 0) {
            notes = null;
            return notes;
        }
        notes = new Note[totalLength];

        for (int i = 0; i < songAnalyzers.size(); i++) {
            SongAnalyzer sAnal = (SongAnalyzer) songAnalyzers.elementAt(i);
            Note[] n = sAnal.getNotes();
            int numNotes = sAnal.getNumNotes();
            for (int j = 0; j < numNotes; j++) {
                notes[counter++] = n[j];
            }
        }
        Arrays.sort(notes); // Make sure the notes are in
                           // order of time.
        notesInvalid = false;
    }
    return notes;
}

public String toString() { return groupName; }
public String getFullTitle() { return "Song Group: " + toString(); }

/**
 * A GroupAnalyzer shouldn't play a song... it'd just be a mess.
 * So return null.
 */
public Sequence getSequence() { return null; }

/**
 * Return the standard deviation between the mean pitches
 * in all SongAnalyzers underneath this GroupAnalyzer.
 */
public int getGroupMeanPitchDeviation() {
    int sumPitch = 0;
    for (int i = 0; i < songAnalyzers.size(); i++) {
        sumPitch += ((SongAnalyzer) songAnalyzers.elementAt(i)).getMeanPitch();
    }

    double preciseMean = (double) sumPitch / (double) songAnalyzers.size();
    double devSum = 0;
    for (int i = 0; i < songAnalyzers.size(); i++) {
        devSum += Math.pow((double) ((SongAnalyzer) songAnalyzers.elementAt(i)).getMeanPitch()
                           - preciseMean, 2);
    }
    return (int) Math.sqrt(devSum / (songAnalyzers.size() - 1));
}
}

```

```

import java.awt.*;
import javax.swing.*;

/**
 * IntervalFreqGraph.java <p>
 *
 * Draws a graph representing the number of times
 * each interval occurs throughout the song, starting on
 * the left with a unison, and working right towards larger
 * intervals.
 *
 * @author      Greg Cipriano
 */

public class IntervalFreqGraph implements MusicGraph {
    private Analyzer analyzer;
    private JPanel view;
    private final int noteWidth = 5;
    private final int PREFERREDHEIGHT = 200;
    private final int OFFSET = 30;

    /**
     * Empty constructor. Doesn't set up anything. Used when an
     * object class is needed.
     */

    public IntervalFreqGraph() {}

    /**
     * Real constructor... takes an Analyzer, and sets up
     * the panel that will soon be drawn upon.
     */

    public IntervalFreqGraph(Analyzer a) {
        analyzer = a;
        analyzer.analyzeNotes();
        view = new JPanel();
        view.setPreferredSize(new Dimension(
            (a.getLargestInterval() + 2) * noteWidth, PREFERREDHEIGHT + OFFSET));
    }

    /**
     * Returns the title of this graph, using the
     * shortened version of the Analyzer's title.
     */

    public String toString() { return "IntFreq: " + analyzer.toString(); }

    /**
     * Factory for creating IntervalFreqGraphs.
     */

    public MusicGraph createMusicGraph(Analyzer a) {
        return new IntervalFreqGraph(a);
    }

    public String[] getVariableTitles() { return null; }
    public int[] getDefaultValues() { return null; }
    public JComponent getView() { return view; }

    /**
     * Actually draw the graph. First interval counts
     * are fetched from the analyzer, then statistics
     * needed to scale the data are also fetched. Graphics
     * contexts and images are created, then drawn upon.
     */

    public void drawGraph(boolean repaintIcon) {
        int[] intervalCount = analyzer.getIntervalCounts();
        int largestInterval = analyzer.getLargestInterval();
        int countOfMostFrequentInterval = analyzer.getCountOfMostFrequentInterval();

        int width = view.getSize().width;
        int height = view.getSize().height;
        float scalar = (float) (height - OFFSET) / countOfMostFrequentInterval;

        // Set up graphics contexts -- once, since drawGraph() is only
        // ever called once.
        Image offscreenI = view.createImage(width, height);
        Graphics backpage = offscreenI.getGraphics();
        backpage.setColor(Color.white);
        backpage.fillRect(0, 0, width, height);

        // If interval counts are high, reduce the number of times
        // interval lines are drawn.

        int step = 1;
        if (countOfMostFrequentInterval > 50) { step = 10; }
        if (countOfMostFrequentInterval > 200) { step = 50; }
    }
}

```

```

// Draw horizontal interval lines across graph
backPage.setColor(new Color(200, 200, 200));
for (int i = 1; i < countOfMostFrequentInterval; i += step) {
    int h = height - OFFSET - (int) (i * scalar);
    backPage.drawLine(0, h, width, h);
}

// Now draw the data
backPage.setColor(Color.red);
for (int i = 1; i <= largestInterval; i++) {
    int h = (int) (intervalCount[i] * scalar);
    backPage.fillRect(noteWidth * (i + 1), height - OFFSET - h,
        noteWidth - 1, h);
}

// Draw the lines and numbers on bottom.
backPage.setColor(Color.black);
backPage.drawLine(0, height - OFFSET, width, height - OFFSET);
backPage.setFont(new Font("Serif", Font.BOLD, 11));
int octave = 0;
for (int i = 0; i < width; i += 12 * noteWidth) {
    backPage.drawLine(i, height - OFFSET, i + 7, height - 8);
    backPage.drawLine(i + 7, height - 8, i + 12, height - 8);
    backPage.drawString("" + (octave++) * 7, i + 15, height - 5);
}

// And finally lay everything out
view.setLayout(new BorderLayout());
view.setBackground(Color.gray);
view.add(new JLabel(new ImageIcon(offscreenI)), BorderLayout.CENTER);
view.validate();
}

```

```

/**
 * setVariable does nothing here, since there are no
 * variables to change.
 */

```

```

public void setVariable(int varNum, int value, boolean redraw) {}

```

```

import java.io.File;
import javax.swing.filechooser.*;

/**
 * MidiFilter.java <p>
 *
 * An extension of FileFilter to make the dialog
 * box ignore files other than those ending with ".mid".
 *
 * @author      Greg Cipriano
 */

public class MidiFilter extends FileFilter {
    /**
     * Update accept(File) to filter Files out if
     * they don't conform to the ".mid"
     * pattern.
     */

    public boolean accept(File f) {
        if(f != null) {
            if(f.isDirectory()) { return true; }

            String extension = "";
            String filename = f.getName();
            int i = filename.lastIndexOf('.');
            if(i>0 && i<filename.length()-1) {
                extension = filename.substring(i+1).toLowerCase();
            }
            return (extension.equals("mid"));
        }
        return false;
    }

    public String getDescription() {
        return "Midi files (*.mid)";
    }
}

```

```

import java.awt.*;
import javax.swing.*;

/**
 * MusicGraph.java <p>
 *
 * An interface to a graph, including the most important
 * functions that a graph must support.
 *
 * @author      Greg Cipriano
 */
public interface MusicGraph {

    /**
     * Should return a string representation of the Graph,
     * Declared here to force this behaviour.
     */
    public String toString();

    /**
     * A factory method allowing any subclass to automatically
     * create MusicGraphs of its type upon request.
     */
    public MusicGraph createMusicGraph(Analyzer a);

    /**
     * Returns an array of Strings representing the titles for
     * variables... these will appear at the bottom of the frame this
     * graph is placed in (if not null).
     */
    public String[] getVariableTitles();

    /**
     * Gets the default values associated with each variable.
     * This array must be the same length as the array returned by
     * getVariableTitles().
     */
    public int[] getDefaultValues();

    /**
     * Return the JComponent that will be placed in the frame.
     */
    public JComponent getView();

    /**
     * Draws the graph onto the view, optionally repainting.
     */
    public void drawGraph(boolean repaintIcon);

    /**
     * Sets a variable (represented by varNum) to value. If redraw is true,
     * drawGraph() will also be called to update the view to the change.
     */
    public void setVariable(int varNum, int value, boolean redraw);
}

```

```

import javax.sound.midi.*;

/**
 * MediaPlayer.java <p>
 *
 * MediaPlayer performs one task: it allows a controller (MidiStat.java)
 * to start, stop, pause and resume a MIDI file player -- called sequencer.<BR>
 * JMF takes care of all the details.
 *
 * @author      Greg Cipriano
 */

public class MediaPlayer implements Runnable {
    private Thread thread;
    private Sequence sequence;
    private Sequencer sequencer;
    private boolean songDone = false;
    private boolean paused = false;

    /**
     * Construct the Music player, creating a sequencer, and registering
     * the given SongEndedListener's intention to listen to MetaEvents
     * generated by the current sequencer.
     */

    public MediaPlayer(SongEndedListener sel) {
        try {
            sequencer = MidiSystem.getSequencer();
            sequencer.addMetaEventListener(sel);
        } catch (MidiUnavailableException e) {
            e.printStackTrace();
            System.err.println("Bad news... no MIDI!");
        }
    }

    /**
     * Entry point for the song playing Thread. This starts
     * the sequencer playing.
     */

    public void run() {
        if (sequence != null) {
            sequencer.start();
        }
    }

    /**
     * Starts the song playing, stopping a previous song if it
     * was either playing or paused.
     */

    public void start(Analyzer m) {
        if (!songDone) { stop(); }
        try {
            sequence = m.getSequence();
            if (sequence != null) {
                sequencer.open();
                sequencer.setSequence(sequence);
                songDone = false;
                thread = new Thread(this);
                thread.setPriority(Thread.MAX_PRIORITY);
                thread.start();
            }
        } catch (Exception e) {
            e.printStackTrace();
            System.err.println("Can't Start...");
        }
    }

    /**
     * Pause the currently playing song (if any).
     */

    public void pause() {
        if (sequence != null) {
            sequencer.stop();
        }
    }

    /**
     * Resume the currently paused song.
     */

    public void resume() {
        if (sequence != null) {
            sequencer.start();
        }
    }

    /**
     * Stop the song, if it was playing. This also nullifys

```



```
* both the sequence itself and the thread dedicated to playing it.  
*/
```

```
public void stop() {  
    if (thread != null) {  
        thread.interrupt();  
    }  
    thread = null;  
    sequence = null;  
    if (sequencer.isOpen()) {  
        sequencer.stop();  
        sequencer.close();  
    }  
    songDone = true;  
}
```

```
,
```

```

import java.util.*;

/**
 * Note.java <p>
 *
 * An object representing one note, which
 * has a starting time, duration, velocity and pitch.
 *
 * @author      Greg Cipriano
 */

public class Note implements Comparable {

    /** Each note in a standard octave as a string */
    private final static String[] notes = { "C", "C#", "D",
        "Eb", "E", "F", "F#", "G", "Ab", "A", "Bb", "B" };

    /** A good place to keep precomputed angles */
    private static double[] directions;

    /** The time when this note comes on (in beats) */
    private float timeOn;

    /** How long this note sounds (in beats) */
    private float duration;

    /** How hard this note was hit */
    private int velocity;

    /** The pitch (represented as an int) */
    private int pitch;

    // Initialize the directions array... This just sets
    // up 12 evenly spaced points (x, y) around the
    // unit circle
    static {
        directions = new double[24];
        for (int i = 0; i < 6; i++) {
            double angle = (double) (Math.PI * i) / 6;
            directions[2 * i] = Math.cos(angle);
            directions[2 * i + 1] = -Math.sin(angle);
            directions[2 * i + 12] = -directions[2 * i];
            directions[2 * i + 13] = -directions[2 * i + 1];
        }
    }

    /**
     * Construct a note, started at time t, of duration d,
     * velocity v and pitch p.
     */

    public Note(float t, float d, int v, int p) {
        timeOn = t;
        duration = d;
        velocity = v;
        pitch = p;
    }

    /**
     * Returns the time this note came on.
     */

    public float getTimeOn() { return timeOn; }

    /**
     * Returns how long this note is.
     */

    public float getDuration() { return duration; }

    /**
     * Returns when this note ends.
     */

    public float getTimeOff() { return timeOn + duration; }

    /**
     * Returns this note's pitch.
     */

    public int getPitch() { return pitch; }

    /**
     * Returns this note's velocity (loudness).
     */

    public int getVelocity() { return velocity; }

    /**
     * Implements compareTo() in the comparable interface (Makes sorting work).
     * This is implemented so that the note's start time sets up its

```

```

    * relative order.
    */

    public int compareTo(Object o) {
        Note n2 = (Note) o;
        if (timeOn < n2.getTimeOn()) { return -1; }
        else if (timeOn > n2.getTimeOn()) { return 1; }
        else { return 0; }
    }

    /**
     * A string representation of this note. <P>
     * Returns: timeOn + " " + duration + " " + getNote() + " " + velocity
     */

    public String toString() {
        return (timeOn + " " + duration + " " + getNote() + " " + velocity);
    }

    /**
     * Equivalent to calling Note.toNote(this.pitch);
     */

    public String getNote() { return Note.toNote(pitch); }

    /**
     * Adds the vector representing this note to the
     * given array in place.
     */

    public void addVector(double[] array, double scalar) {
        int p = ((pitch % 12 + 7) * 7) % 12;
        array[0] += (double) duration * scalar * Note.directions[2 * p];
        array[1] += (double) duration * scalar * Note.directions[2 * p + 1];
    }

    /**
     * Equivalent to calling Note.toNote(pitch, false)
     */

    public static String toNote(int pitch) {
        return toNote(pitch, true);
    }

    /**
     * Converts the given pitch into a String representing this note.
     * For example: 60 == "C5", 68 == "G#5", etc.
     * If returnOctave is true, append the octave to the end of the string.
     */

    public static String toNote(int pitch, boolean returnOctave) {
        String r = Note.notes[pitch % 12];
        if (returnOctave) {
            return r + pitch / 12;
        } else {
            return r;
        }
    }
}

```

```

/**
 * NoteFreqGraph.java <p>
 *
 * Draws a graph representing the number of times
 * each pitch was played, starting at the lowest
 * note contained in the Analyzer on the left
 * and moving towards the highest on the right.
 *
 * @author      Greg Cipriano
 */

public class NoteFreqGraph implements MusicGraph {
    private Analyzer analyzer;
    private JPanel panel;
    private final int OFFSET = 30;
    private final int PREFERREDHEIGHT = 200;
    private final int noteWidth = 5;

    /**
     * Empty constructor. Doesn't set up anything. Used when an
     * object class is needed.
     */

    public NoteFreqGraph() {}

    /**
     * Real constructor... takes an Analyzer, and sets up
     * the panel that will soon be drawn upon.
     */

    public NoteFreqGraph(Analyzer a) {
        analyzer = a;
        analyzer.analyzeNotes();
        panel = new JPanel();
        panel.setPreferredSize(new Dimension(
            (analyzer.getRange() + 2) * noteWidth, PREFERREDHEIGHT + OFFSET));
    }

    /**
     * Returns the title of this graph, using the
     * shortened version of the Analyzer's title.
     */

    public String toString() { return "NoteFreq: " + analyzer.toString(); }

    /**
     * Factory for creating NoteFreqGraphs.
     */

    public MusicGraph createMusicGraph(Analyzer a) {
        return new NoteFreqGraph(a);
    }

    public String[] getVariableTitles() { return null; }
    public int[] getDefaultValues() { return null; }
    public JComponent getView() { return panel; }

    /**
     * Actually draw the graph. First note frequency counts
     * are fetched from the analyzer, then statistics
     * needed to scale the data are also fetched. Graphics
     * contexts and images are created, then drawn upon.
     */

    public void drawGraph(boolean repaintIcon) {
        int[] noteCount = analyzer.getNoteFrequencies();
        int highCount = analyzer.getCountOfMostOftenSoundedNote();
        int lowestNote = analyzer.getLowestNote();

        int width = panel.getSize().width;
        int height = panel.getSize().height;

        // Create graphics contexts and the offscreen image
        Image offscreenI = panel.createImage(width, height);
        Graphics backpage = offscreenI.getGraphics();
        backpage.setColor(Color.white);
        backpage.fillRect(0, 0, width, height);

        // Actually draw the data
        backpage.setColor(Color.red);
        float scalar = (float) (height - OFFSET) / highCount;
        for (int i = 0; i < noteCount.length; i++) {
            int h = (int) (noteCount[i] * scalar);
            backpage.fillRect(noteWidth * (i + 1), height - OFFSET - h,
                noteWidth - 1, h);
        }
    }
}

```

```

// Draw the lines and note names (the C's of each scale)
// on the bottom
backPage.setColor(Color.black);
backPage.drawLine(0, height - OFFSET, width, height - OFFSET);
backPage.setFont(new Font("Serif", Font.BOLD, 11));

// Find the position of the lowest C relative to the lowest note.
int lowestC = ((13 - lowestNote % 12) % 12) * noteWidth;
int lowestOctave = lowestNote / 12 + 1;
for (int i = lowestC; i < width; i += 12 * noteWidth) {
    backPage.drawLine(i, height - OFFSET, i + 7, height - 8);
    backPage.drawLine(i + 7, height - 8, i + 13, height - 8);
    backPage.drawString("C" + (lowestOctave++), i + 15, height - 5);
}

// Finally lay out and components
panel.setLayout(new BorderLayout());
panel.setBackground(Color.gray);
panel.add(new JLabel(new ImageIcon(offscreenI)), BorderLayout.CENTER);
panel.validate();
}

/**
 * setVariable does nothing here, since there are no
 * variables to change.
 */

public void setVariable(int varNum, int value, boolean redraw) {}
}

```

```
import java.awt.*;
import javax.swing.*;
import javax.swing.table.*;
```

```
/**
 * NoteListFrame.java <p>
 *
 * NoteListFrame uses a JTable to show a list of all notes in the song,
 * ordered by time of occurrence in each row. Columns are the properties
 * of a note: time on, duration, pitch and velocity.
 *
 * @author      Greg Cipriano
 */
```

```
public class NoteListFrame extends JInternalFrame {
```

```
/**
 * Constructs the frame, setting up all of the cells of the
 * JTable, adding that table to a scroll pane and that pane
 * to the frame itself.
 */
```

```
public NoteListFrame(Analyzer a) {
    super(a.getFullTitle() + ": Note List", true, true, true, true);
    Note[] notes = a.getNotes();
    String[][] columnValues = new String[notes.length][4];
    for (int i = 0; i < notes.length; i++) {
        String timeOn = ((int) notes[i].getTimeOn() / 4 + 1) + " : "
            + (notes[i].getTimeOn() % 4 + 1);
        columnValues[i][0] = timeOn;

        columnValues[i][1] = Float.toString(notes[i].getDuration());
        columnValues[i][2] = Note.toNote(notes[i].getPitch());
        columnValues[i][3] = Integer.toString(notes[i].getVelocity());
    }
    setContentPane(new JScrollPane(createTable(columnValues)));
    setPreferredSize(new Dimension(450, 300));
    pack();
}
```

```
/**
 * Creates a JTable with the entries denoted by columnValues and the
 * column headers: "Time On", "Duration", "Pitch" and "Velocity".
 * Other parameters -- including making it uneditable, and
 * making the TableModel specific -- are added to this JTable.
 */
```

```
private JTable createTable(final String[][] columnValues) {
    final String[] columnNames = { "Time On (M:B)", "Duration", "Pitch", "Velocity" };

    TableModel dataModel = new AbstractTableModel() {
        public String getColumnName(int col) {
            return columnNames[col];
        }
        public int getColumnCount() { return columnValues[0].length; }
        public int getRowCount() { return columnValues.length; }
        public boolean isCellEditable(int row, int col) { return false; }
        public Object getValueAt(int row, int col) {
            return columnValues[row][col];
        }
    };
    JTable table = new JTable(dataModel);
    return table;
}
```

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * PianoRollGraph.java <p>
 *
 * PianoRollGraph is a scrollable graph that shows the music
 * in a form similar to an old-fashioned player piano roll.
 * The roll puts the lowest notes at the bottom, with higher
 * notes above. For ease of identification, the 'C' in each octave
 * is colored gray in the background.
 *
 * @author      Greg Cipriano
 */

public class PianoRollGraph extends ScrollBarGraph {
    private final String[] titles = { "Beat Length" };
    private final int[] defaultValues = { 30 };
    private final int PREFERREDWIDTH = 500;
    private final int NOTEHEIGHT = 3;
    private final int OFFSET = 12;
    private int beatSize;
    private Analyzer analyzer;
    private JPanel view;
    private JPanel innerPane;
    private Image background;
    private Image offscreenI;
    private Graphics backpage;
    private Note[] notes;
    private JLabel label;
    private ImageIcon icon;

    /**
     * Empty constructor. Doesn't set up anything. Used when an
     * object class is needed.
     */

    public PianoRollGraph() {}

    /**
     * Real constructor. Takes an Analyzer, and sets up the
     * view's scrollbar and panel.
     */

    public PianoRollGraph(Analyzer a) {
        analyzer = a;
        analyzer.analyzeNotes();
        notes = analyzer.getNotes();
        innerPane = new JPanel();
        innerPane.setPreferredSize(new Dimension(PREFERREDWIDTH,
                                                    (analyzer.getRange() + 3) * NOTEHEIGHT + OFFSET));
        innerPane.addComponentListener(this);

        scrollbar = new JScrollBar(JScrollBar.HORIZONTAL);
        setVariable(0, defaultValues[0], false);
        view = new JPanel(new BorderLayout());
        scrollbar.setBlockIncrement(PREFERREDWIDTH - 20);
        scrollbar.addAdjustmentListener(this);
        view.add(innerPane, BorderLayout.CENTER);
        view.add(scrollbar, BorderLayout.SOUTH);
    }

    /**
     * Factory for creating PianoRollGraphs
     */

    public MusicGraph createMusicGraph(Analyzer a) {
        return new PianoRollGraph(a);
    }

    /**
     * Returns a string representation of this PianoRollGraph.
     */

    public String toString() {
        return analyzer.getFullTitle() + ":  Piano Roll";
    }

    public String[] getVariableTitles() { return titles; }
    public int[] getDefaultValues() { return defaultValues; }
    public JComponent getView() { return view; }

    /**
     * Implements drawGraph() in the MusicGraph interface
     * to draw the Piano Roll. Here, if the offscreen buffer
     * hasn't been set up yet, do that, too.
     */

    public void drawGraph(boolean repaintIcon) {
        int width = innerPane.getSize().width;

```

```

int height = (analyzer.getRange() + 3) * NOTEHEIGHT + OFFSET;    // height of piano roll
int start = scrollbar.getValue();
int end = start + width;
int lowestNote = analyzer.getLowestNote();

// Set up the back buffer if necessary, making it the size of
// the screen, so if the frame holding this graph is resized,
// it's still big enough.
if (offscreenI == null) {
    int screenWidth = Toolkit.getDefaultToolkit().getScreenSize().width;
    setupBackBuffers(screenWidth, height);
}
backPage.drawImage(background, 0, 0, view);

// 4 quarter-notes per measure
int measureSize = 4 * beatSize;

// draw measure numbers and lines
backPage.setColor(new Color(100, 100, 100));
int measureNumber = start / measureSize + 1;
for (int i = measureSize - (start % measureSize); i < end + measureSize; i += measureSize) {
    backPage.drawLine(i, 0, i, height);
    backPage.drawString("" + (measureNumber++), i - measureSize / 2, OFFSET - 2);
}

// Now draw all of the notes as red bars
backPage.setColor(Color.red);
for (int i = 0; i < notes.length; i++) {
    int startTime = (int) (notes[i].getTimeOn() * beatSize);
    if (startTime < end) {
        int length = (int) (notes[i].getDuration() * beatSize);
        int endTime = startTime + length - start;
        if (endTime > 0) {
            int h = height - (notes[i].getPitch() - lowestNote + 2) * NOTEHEIGHT;
            startTime = (startTime < start) ? 0 : startTime - start;
            length = endTime - startTime;
            backPage.fillRect(startTime, h, length, NOTEHEIGHT);
        }
    } else {
        i = notes.length;
    }
}

if (repaintIcon) {
    // A hack to get the icon to paint REALLY fast. :)
    icon.paintIcon(label, label.getGraphics(),
        0, (innerPane.getSize().height - height) / 2);
}
}

/**
 * Create the graphics contexts and images necessary to draw
 * this graph. This includes predrawn note-separating lines, as
 * well as the more ordinary double-buffering images.
 */

public void setupBackBuffers(int width, int height) {
    offscreenI = view.createImage(width, height);
    innerPane.setBackground(Color.gray);
    innerPane.setLayout(new BorderLayout());
    innerPane.add(label = new JLabel(icon = new ImageIcon(offscreenI)), BorderLayout.WEST);
    innerPane.validate();

    backPage = offscreenI.getGraphics();
    backPage.setFont(new Font("Serif", Font.BOLD, 11));

    background = view.createImage(width, height);
    Graphics bg = background.getGraphics();

    bg.setColor(Color.white);
    bg.fillRect(0, 0, width, height);

    // Paint the lines separating each note
    bg.setColor(new Color(140, 140, 140));
    for (int i = OFFSET; i < height; i += NOTEHEIGHT) {
        bg.drawLine(0, i, width, i);
    }

    // Paint the gray bars representing the 'C's for each octave
    bg.setColor(new Color(220, 220, 220));
    int lowestNote = analyzer.getLowestNote();
    int low = lowestNote % 12;
    int lowestC;
    if (low == 0) {
        // C is the lowest note... special case
        lowestC = height - 2 * NOTEHEIGHT;
    } else {
        lowestC = height - (14 - lowestNote % 12) * NOTEHEIGHT;
    }
    for (int i = lowestC; i > OFFSET; i -= 12 * NOTEHEIGHT) {
        bg.fillRect(0, i, width, NOTEHEIGHT);
    }
}

```



```

    }
}

/**
 * Sets the variable at 'varnum' with the value. If redraw
 * is set, repaint the graph to reflect this change. This
 * also updates the scrollbar's extent and position.
 */
public void setVariable(int varNum, int value, boolean redraw) {
    if (varNum == 0) {
        int newPosition = 0;
        if (beatSize != 0) {
            newPosition = (scrollbar.getValue() * value) / beatSize;
        }
        beatSize = value;
        int songLength = (int) (analyzer.getSongLength() * beatSize);
        scrollbar.setValues(newPosition, innerPane.getSize().width, 0, songLength);
    }
    if (redraw) { drawGraph(true); }
}
}

```

```

/**
 * PitchCountGraph.java <p>
 *
 * A variant of a data graph who's data is
 * the number of pitches playing per time period
 * (sampled according to the value of the JSlider).
 *
 * @author      Greg Cipriano
 */

public class PitchCountGraph extends DataGraph {

    /**
     * Empty... does nothing.
     */

    public PitchCountGraph() {}

    /**
     * The maximum value this graph can attain is
     * stored in a.getMaxSimultaneousVoices()
     */

    public PitchCountGraph(Analyzer a) {
        super(a);
        maxValue = analyzer.getMaxSimultaneousVoices();
    }

    /**
     * A PitchCountGraph factory.
     */

    public MusicGraph createMusicGraph(Analyzer a) {
        return new PitchCountGraph(a);
    }

    /**
     * Return a string representation, so the frame this
     * graph is placed in can be titled correctly.
     */

    public String toString() {
        return analyzer.getFullTitle() + ": Pitch Counts";
    }

    /**
     * Extends the setVariable() method to reanalyze voice density
     * data due to a change in sampling frequency.
     */

    public void setVariable(int varNum, int value, boolean redraw) {
        if (varNum == 0) {
            data = analyzer.getVoiceDensity(value);
            super.setVariable (varNum, value, redraw);
        }
    }
}

```

```

/**
 * PitchRangeGraph.java <p>
 *
 * A variant of a data graph who's data is
 * the range formed by simultaneously playing pitches.
 * (sampled according to the value of the JSlider).
 * This range is, of course, 0 if only one pitch is
 * playing in any given sample.
 *
 * @author      Greg Cipriano
 */

public class PitchRangeGraph extends DataGraph {

    /**
     * Empty... does nothing.
     */

    public PitchRangeGraph() {}

    /**
     * The maximum value this graph can attain is
     * stored in a.getRange()
     */

    public PitchRangeGraph(Analyzer a) {
        super(a);
        maxValue = analyzer.getRange();
    }

    /**
     * A PitchRangeGraph factory.
     */

    public MusicGraph createMusicGraph(Analyzer a) {
        return new PitchRangeGraph(a);
    }

    /**
     * Return a string representation, so the frame this
     * graph is placed in can be titled correctly.
     */

    public String toString() {
        return analyzer.getFullTitle() + ":  Pitch Range";
    }

    /**
     * Extends the setVariable() method to reanalyze pitch range
     * data due to a change in sampling frequency.
     */

    public void setVariable(int varNum, int value, boolean redraw) {
        if (varNum == 0) {
            data = analyzer.getPitchRange(value, false);
            super.setVariable (varNum, value, redraw);
        }
    }
}

```

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * ScrollBarGraph.java <p>
 *
 * An abstract type of Music Graph that fills in the basic
 * framework for supporting a scrollbar and the events
 * that go along with it.
 *
 * @author      Greg Cipriano
 */

public abstract class ScrollBarGraph implements MusicGraph,
                                                AdjustmentListener,
                                                ComponentListener {

    protected JScrollBar scrollbar;

    /**
     * Fired from the scrollbar. Just redraw the graph to reflect the change.
     */

    public void adjustmentValueChanged(AdjustmentEvent e) { drawGraph(true); }

    /**
     * Called whenever the view is resized. This readjusts the scrollbar,
     * and calls for a redraw (but not repaint, since resizing a window
     * will automatically call for a repaint.)
     */

    public void componentResized(ComponentEvent e) {
        scrollbar.setValues(scrollbar.getValue(),
                            ((Component) e.getSource()).getSize().width,
                            0,
                            scrollbar.getMaximum());
        drawGraph(false);
    }

    public void componentHidden(ComponentEvent e) {}
    public void componentShown(ComponentEvent e) {}
    public void componentMoved(ComponentEvent e) {}

    /**
     * Force subclasses to deal with back-buffers in a
     * homogenous way. This method is where they should
     * create graphics contexts and images needed
     * for a refreshable scroll pane.
     */

    public abstract void setupBackBuffers(int width, int height);
}

```

```
import java.io.*;
import java.util.*;
import javax.sound.midi.*;
```

```
/**
 * SongAnalyzer.java <p>
 *
 * A SongAnalyzer represents an individual song,
 * and contains as many TrackAnalyzers as that song
 * contained tracks (with valid notes).
 *
 * @author      Greg Cipriano
 */
```

```
public class SongAnalyzer extends Analyzer {
    private int midiType;
    private int numTracks = 0;
    private GroupAnalyzer group;
    private Sequence song;
    private String fileName;
    private Vector trackAnalyzers;
    private Note[] notes;
    private Vector metaData = new Vector();

    /**
     * Entry point for testing this class.
     */

    public static void main(String[] args) {
        try {
            SongAnalyzer m = new SongAnalyzer(new File(args[0]));
            m.printMessages();
            m.printNotes();
        } catch (Exception e) {
            e.printStackTrace();
            System.out.println("Invalid File: " + args[0] + "!");
        }
    }

    /**
     * Constructor for a Song Analyzer... takes in a (hopefully valid)
     * file name and builds up all tracks.
     */

    public SongAnalyzer(File f) {
        if (f != null) {
            try {
                song = MidiSystem.getSequence(f);
                fileName = f.getName();
                midiType = MidiSystem.getMidiFileFormat(f).getType();
                addTracks(song.getTracks());
            } catch (InvalidMidiDataException e) { e.printStackTrace(); }
            catch (IOException e) { e.printStackTrace(); }
        }
    }

    public String toString() { return fileName; }
    public String getFullTitle() { return fileName; }

    /**
     * Return the tracks (TrackAnalyzers) contained in this SongAnalyzer.
     */

    public Vector getTrackAnalyzers() { return trackAnalyzers; }

    /**
     * A convenience method that returns how many tracks this song
     * contains...
     */

    public int getNumTracks() {
        return getTrackAnalyzers().size();
    }

    /**
     * Returns the sequence contained in the File passed into this
     * SongAnalyzer.
     */

    public Sequence getSequence() { return song; }

    /**
     * Let this SongAnalyzer know which group it's contained in,
     * just in case it needs to tell the group about its status.
     */

    public void setGroupAnalyzer(GroupAnalyzer g) { group = g; }

    /**
     * Return all notes contained in all active trackAnalyzers, lumped
     * into one notes array and sorted according to start times.
     */
}
```

```

*/
public Note[] getNotes() {
    if (notesInvalid) {
        int totalLength = 0;
        int counter = 0;
        for (int i = 0; i < getNumTracks(); i++) {
            TrackAnalyzer tAnal = (TrackAnalyzer) trackAnalyzers.elementAt(i);
            if (tAnal.getActive()) {
                totalLength += ((TrackAnalyzer) trackAnalyzers.elementAt(i)).getNumNotes();
            }
        }

        if (totalLength == 0) { return null; }

        notes = new Note[totalLength];

        for (int i = 0; i < getNumTracks(); i++) {
            TrackAnalyzer tAnal = (TrackAnalyzer) trackAnalyzers.elementAt(i);
            if (tAnal.getActive()) {
                Note[] n = tAnal.getNotes();
                for (int j = 0; j < n.length; j++) {
                    notes[counter++] = n[j];
                }
            }
        }
        Arrays.sort(notes);
        notesInvalid = false;
    }
    return notes;
}

/**
 * Returns whether at least one of its tracks
 * is active. This is a valid method because tracks
 * always contain at least one note.
 */

public boolean containsChildren() {
    for (int i = 0; i < getNumTracks(); i++) {
        if (((TrackAnalyzer) trackAnalyzers.elementAt(i)).getActive()) {
            return true;
        }
    }
    return false;
}

/**
 * Called to force Note array to be recomputed the next
 * time it's accessed.
 */

protected void invalidate() {
    notesInvalid = true;
    group.invalidate();
}

/**
 * Print debug info about this MIDI file.
 */

public void printMessages() {
    System.out.println("Midi Type: " + midiType);
    System.out.println("Resolution: " + song.getResolution());
    System.out.println("Length (ms): " + song.getMicrosecondLength());
    float divtype = song.getDivisionType();
    if (divtype == Sequence.PPQ) {
        System.out.println("Division Type: PPQ");
    } else if (divtype == Sequence.SMPTE_24) {
        System.out.println("Division Type: SMPTE_24");
    } else if (divtype == Sequence.SMPTE_25) {
        System.out.println("Division Type: SMPTE_25");
    } else if (divtype == Sequence.SMPTE_30) {
        System.out.println("Division Type: SMPTE_30");
    } else if (divtype == Sequence.SMPTE_30DROP) {
        System.out.println("Division Type: SMPTE_30DROP");
    }
    System.out.println();
    Track[] t = song.getTracks();
    for (int i = 0; i < t.length; i++) {
        System.out.println("----- " + i + " -----");
        int s = t[i].size();
        for (int j = 0; j < s; j++) {
            MidiEvent me = t[i].get(j);
            MidiMessage mm = me.getMessage();
            long eventTime = me.getTick();

            System.out.print(eventTime);

            byte[] message = mm.getMessage();
            for (int k = 0; k < mm.getLength(); k++) {

```

```

        System.out.print(" " + (int)(message[k] & 0xFF));
    }
    System.out.println();
}
System.out.println();
}
}

/**
 * Tell each contained TrackAnalyzer to print its notes out to
 * standard output (for debugging).
 */

public void printNotes() {
    for (int i = 0; i < getNumTracks(); i++) {
        ((TrackAnalyzer) trackAnalyzers.elementAt(i)).printNotes();
        System.out.println();
    }
}

/**
 * Add all tracks into this SongAnalyzer. If midiType = 0, there
 * is only one track, otherwise loop through them, calling
 * createTrackAnalyzers on each.
 *
 * @see #createTrackAnalyzers(Track)
 */

private void addTracks(Track[] t) throws InvalidMidiDataException {
    if (midiType == 0) {
        trackAnalyzers = createTrackAnalyzers(t[0]);
    } else if (midiType == 1) {
        trackAnalyzers = new Vector();
        int numTracks = t.length;
        for (int i = 0; i < numTracks; i++) {
            trackAnalyzers.addAll(createTrackAnalyzers(t[i]));
        }
    } else {
        throw new InvalidMidiDataException();
    }
}

/**
 * Separate all tracks contained in the individual
 * JMF Track, creating a new TrackAnalyzer for each.
 * This returns a vector containing them all.
 */

private Vector createTrackAnalyzers(Track t) {
    float resolution = (float) song.getResolution();

    // Create 16 vectors, which can hold individual
    // tracks ferreted out of the one Track (not all
    // will be used, though).
    Vector[] v = new Vector[16];
    for (int i = 0; i < 16; i++) { v[i] = new Vector(); }

    Vector noteOn = new Vector();

    int vel = 0;
    for (int i = 0; i < t.size(); i++) {
        // Get all info about this current event,
        // including time, pitch, and various messages...
        MidiEvent me = t.get(i);
        long eventTime = me.getTick();
        MidiMessage mm = me.getMessage();
        int eventType = mm.getStatus() / 16;
        int voice = mm.getStatus() % 16;
        byte[] message = mm.getMessage();
        int pitch = toInt(message[1]);
        if (eventType == 9) {
            vel = toInt(message[2]);
        }

        // Add note on events, or deal with a note going off by
        // creating a Note capturing this info.

        if (vel != 0 && eventType == 9) {
            noteOn.add(new NoteOnEvent(pitch, voice, vel, eventTime));
        } else if (eventType == 8 || (eventType == 9 && vel == 0)) {
            for (int j = 0; j < noteOn.size(); j++) {
                NoteOnEvent noe = (NoteOnEvent) noteOn.elementAt(j);
                if (noe.pitch == pitch && noe.voice == voice) {
                    float start = noe.time / resolution;
                    float dur = (eventTime - noe.time) / resolution;
                    v[voice].add(new Note(start, dur, noe.velocity, pitch));
                    noteOn.remove(noe);
                    j = noteOn.size();
                }
            }
        } else if (eventType == 15) {

```

```

// Add metadata to a vector to give to trackAnalyzers.
// They then can use this information to reconstruct the correct
// tempo and (sometimes) the correct patch, as well as other
// miscellaneous data.
metadata.add(me);

```

```

// Here, pitch is actually the metadata type... not pitch.
if (pitch == 89 && givenKey == null) {

```

```

    // The MIDI file has encoded what it thinks the key is.
    // First comes the key, in terms of the number of sharps and flats
    // convert this to a note value...
    // Second comes the keys tonality (Major or Minor).
    // Create a chord from this info.
    int key = (int) (message[3] * 7 + 96) % 12;

```

```

    if (message[4] == 0) {
        givenKey = new Chord(key, Chord.MAJ, 0, 0);
    } else {
        givenKey = new Chord(key, Chord.MIN, 0, 0);
    }
}

```

```

// Now that all notes have been found, create TrackAnalyzers for
// each non-empty track.

```

```

Vector r = new Vector();
for (int i = 0; i < 16; i++) {
    if (v[i].size() > 0) {
        Object[] o = v[i].toArray();
        Note[] n = new Note[o.length];
        for (int j = 0; j < o.length; j++) { n[j] = (Note) o[j]; }
        Arrays.sort(n);
        TrackAnalyzer ta = new TrackAnalyzer(n, this, ++numTracks, metaData);
        r.add(ta);
    }
}
return r;
}

```

```

/**
 * Converts the signed byte contained in MIDI into an int
 * that Java can more easily deal with.
 */

```

```

private int toInt(byte b) { return (int)(b & 0xFF); }

```

```

/**
 * A quick class/struct that is used for creating TrackAnalyzers.
 * This does little more than encapsulate a NoteOn event.
 */

```

```

private class NoteOnEvent {
    public int pitch;
    public int voice;
    public int velocity;
    public long time;

    NoteOnEvent(int p, int v, int vel, long t) {
        pitch = p;
        voice = v;
        velocity = vel;
        time = t;
    }
}

```



```

import javax.sound.midi.*;

/**
 * SongEndedListener.java <p>
 *
 * A class listening in on the song. All this does at the
 * moment is check on whether the song has finished playing.
 * If so, MidiStat wants to know so it can update
 * its menu items to reflect the new status.
 *
 * @author      Greg Cipriano
 */

public class SongEndedListener implements MetaEventListener {
    MidiStat midiStat;

    public SongEndedListener(MidiStat ms) { midiStat = ms; }

    /**
     * Called when a meta event occurs... if the type is 47, the song
     * has just finished, so let the music analyzer know.
     */

    public void meta(MetaMessage m) {
        if (m.getType() == 47) {
            midiStat.songStopped();
        }
    }
}

```

```
import java.awt.*;
import javax.swing.*;
import javax.swing.table.*;
```

```
/**
 * StatisticsFrame.java <p>
 *
 * StatisticsFrame uses a JTable to show various statistical properties
 * of its given Analyzer. Further, if the analyzer is a GroupAnalyzer,
 * several more statistics are performed that relate specifically to the
 * combined statistics of the group.
 *
 * @author      Greg Cipriano
 */
```

```
public class StatisticsFrame extends JInternalFrame {
```

```
/**
 * Constructs the frame, setting up all of the cells of the
 * JTable, adding that table to a scroll pane and that pane
 * to the frame itself.
 */
```

```
public StatisticsFrame(Analyzer a) {
    super(a.getFullTitle() + ": Statistics", true, true, true, true);
    a.analyzeNotes();
    if (a instanceof GroupAnalyzer) {
        String[][] columnValues = { { "Number of notes:", Integer.toString(a.getNumNotes()) },
            { "Range:", Integer.toString(a.getRange()) + " notes" },
            { "Highest note:", Note.toNote(a.getHighestNote()) },
            { "Lowest note:", Note.toNote(a.getLowestNote()) },
            { "", "" },
            { "Mean Note:", Note.toNote(a.getMeanPitch()) },
            { "Standard pitch deviation (from mean):", Integer.toString(
                a.getStdPitchDeviation()) + " notes" },
            { "", "" },
            { "Mean Note (weighted):", Note.toNote(a.getWeightedMeanPitch()) },
            { "Standard pitch deviation (from weighted mean):",
                Integer.toString(a.getWeightedStdPitchDeviation()) + " notes" },
            { "", "" },
            { "Most often sounded note:", Note.toNote(
                a.getMostOftenSoundedNote()) },
            { "Number of times it sounded:", Integer.toString(
                a.getCountOfMostOftenSoundedNote()) },
            { "Highest number of simultaneous voices:",
                Integer.toString(
                    a.getMaxSimultaneousVoices()) },
            { "", "" },
            { "Largest interval:", Integer.toString(
                a.getLargestInterval()) + " notes" },
            { "Most frequent interval:", Integer.toString(
                a.getMostFrequentInterval()) + " notes" },
            { "Number of times it sounded:", Integer.toString(
                a.getCountOfMostFrequentInterval()) },
            { "", "" },
            { "Approximate Key:", Note.toNote(
                a.getKeyVector(), false) + " major" },
            { "", "" },
            { "Mean pitch (per song) deviation:",
                Integer.toString(
                    ((GroupAnalyzer) a).getGroupMeanPitchDeviation()) } };
        setContentPane(new JScrollPane(createTable(columnValues)));
    } else {
        String[][] columnValues = { { "Number of notes:", Integer.toString(a.getNumNotes()) },
            { "Range:", Integer.toString(a.getRange()) + " notes" },
            { "Highest note:", Note.toNote(a.getHighestNote()) },
            { "Lowest note:", Note.toNote(a.getLowestNote()) },
            { "", "" },
            { "Mean Note:", Note.toNote(a.getMeanPitch()) },
            { "Standard pitch deviation (from mean):", Integer.toString(
                a.getStdPitchDeviation()) + " notes" },
            { "", "" },
            { "Mean Note (weighted):", Note.toNote(a.getWeightedMeanPitch()) },
            { "Standard pitch deviation (from weighted mean):",
                Integer.toString(a.getWeightedStdPitchDeviation()) + " notes" },
            { "", "" },
            { "Most often sounded note:", Note.toNote(
                a.getMostOftenSoundedNote()) },
            { "Number of times it sounded:", Integer.toString(
                a.getCountOfMostOftenSoundedNote()) },
            { "Highest number of simultaneous voices:",
                Integer.toString(
                    a.getMaxSimultaneousVoices()) },
            { "", "" },
            { "Largest interval:", Integer.toString(
                a.getLargestInterval()) + " notes" },
            { "Most frequent interval:", Integer.toString(
                a.getMostFrequentInterval()) + " notes" },
            { "Number of times it sounded:", Integer.toString(
                a.getCountOfMostFrequentInterval()) },
            { "", "" },
```

```

        { "Approximate Key:", Note.toNote(
            a.getKeyVector(), false) + " maj" },
        { "Key stored in file:", a.getGivenKey() } });
    setContentPane(new JScrollPane(createTable(columnValues)));
}
setPreferredSize(new Dimension(360, 260));
pack();
}

/**
 * Creates a JTable with the entries denoted by columnValues and the
 * column headers: "Statistic" and "Value". Other parameters -- including
 * making it uneditable, and making the TableModel specific -- are
 * added to this JTable.
 */
private JTable createTable(final String[][] columnValues) {
    final String[] columnNames = { "Statistic", "Value" };

    TableModel dataModel = new AbstractTableModel() {
        public String getColumnName(int col) {
            return columnNames[col];
        }
        public int getColumnCount() { return columnValues[0].length; }
        public int getRowCount() { return columnValues.length; }
        public boolean isCellEditable(int row, int col) { return false; }
        public Object getValueAt(int row, int col) {
            return columnValues[row][col];
        }
    };
    JTable table = new JTable(dataModel);
    table.getColumnModel().getColumn(1).setMaxWidth(70);
    return table;
}

```

```

import java.util.*;
import javax.sound.midi.*;

/**
 * TrackAnalyzer.java <p>
 *
 * A TrackAnalyzer is the 'smallest' type of Analyzer. It contains
 * only a single immutable track as both an array of Notes and
 * a Java Media Frameworks Sequence.
 *
 * @author      Greg Cipriano
 */

public class TrackAnalyzer extends Analyzer {
    private boolean active = true;
    private int trackNumber;
    private Sequence song;
    private Sequence singleTrack;
    private Note[] notes;
    private Vector metaData;
    private Analyzer parent;

    /**
     * Construct a TrackAnalyzer, giving it its notes, its parent
     * Analyzer and its track number.
     */

    public TrackAnalyzer(Note[] n, Analyzer a, int number, Vector md) {
        notes = n;
        song = a.getSequence();
        trackNumber = number;
        parent = a;
        metaData = md;
    }

    /**
     * Return its full title: "parentname -- trackname"
     */

    public String getFullTitle() {
        return parent.toString() + " -- " + toString();
    }

    /**
     * Returns a short description of this track.
     */

    public String toString() { return "Track #" + trackNumber; }

    /**
     * Returns encapsulated notes.
     */

    public Note[] getNotes() { return notes; }

    /**
     * Return the Sequence representing this track, creating one
     * if necessary. This sequence is an abstraction of the MIDI
     * sequence, and is required for the JMF to be able to play
     * this track.
     */

    public Sequence getSequence() {
        if (singleTrack == null) {
            int resolution = song.getResolution();
            try {
                singleTrack = new Sequence(song.getDivisionType(),
                                           resolution);
                Track t = singleTrack.createTrack();

                for (int i = 0; i < metaData.size(); i++) {
                    t.add((MidiEvent) metaData.elementAt(i));
                }

                for (int i = 0; i < notes.length; i++) {
                    ShortMessage sm1 = new ShortMessage();
                    sm1.setMessage(ShortMessage.NOTE_ON, 1,
                                   notes[i].getPitch(), notes[i].getVelocity());
                    long startTime = (long) (notes[i].getTimeOn() * resolution);
                    MidiEvent start = new MidiEvent(sm1, startTime);
                    t.add(start);

                    ShortMessage sm2 = new ShortMessage();
                    sm2.setMessage(ShortMessage.NOTE_OFF, 1,
                                   notes[i].getPitch(), notes[i].getVelocity());
                    long endTime = (long)
                        ((notes[i].getTimeOff()) * resolution);
                    MidiEvent end = new MidiEvent(sm2, endTime);
                    t.add(end);
                }
            } catch (InvalidMidiDataException e) {}
        }
    }
}

```

```

    }
    return singleTrack;
}

/**
 * Overrides getGivenKey in Analyzer to provide
 * the Song's key (instead of the track's).
 */
public String getGivenKey() {
    return parent.getGivenKey();
}

/**
 * Prints out all notes in the track.
 */
public void printNotes() {
    System.out.println(this);
    for (int i = 0; i < notes.length; i++) {
        System.out.println(notes[i]);
    }
}

/**
 * Sets whether or not this track should be
 * included in its song's analysis.
 */
public void setActive(boolean b) {
    active = b;
    ((SongAnalyzer) parent).invalidate();
}

/**
 * Returns whether this track should be included
 * in a song's analysis.
 */
public boolean getActive() { return active; }

/**
 * Switch this track's active status.
 */
public void changeActive() { setActive(!getActive()); }

/**
 * Returns whether this is analyzable... always true.
 */
public boolean containsChildren() { return true; }
}

```

```

import javax.swing.*;
import javax.swing.tree.*;
import javax.swing.event.*;
import java.io.File;
import java.util.*;
import java.awt.*;
import java.awt.event.*;

/**
 * TreeManager.java <p>
 *
 * TreeManager not only Manages a JTree and its associated Model (allowing only specific
 * operations), it also listens to mouse events from the former.
 *
 * @author      Greg Cipriano
 */

public class TreeManager implements MouseListener, MouseMotionListener {
    private int newGroupCounter = 1;
    private CustomJTree tree;
    private CustomTreeModel treeModel;
    private DefaultMutableTreeNode root;
    private DefaultMutableTreeNode defaultGroup;
    private DefaultMutableTreeNode lastParent;
    private DefaultMutableTreeNode originalParent;
    private DefaultMutableTreeNode draggedNode;
    private Toolkit toolkit = Toolkit.getDefaultToolkit();
    private boolean dragIsLegal = false;           // true when a legal drag
                                                    // is happening

    /**
     * Constructs the this TreeManager, creating a new TreeModel,
     * adding the root node and a default GroupAnalyzer. Listeners
     * are added at this point, too.
     */

    public TreeManager() {
        root = new DefaultMutableTreeNode("Groups");
        treeModel = new CustomTreeModel(root);
        defaultGroup = addGroup("Default Group");

        final CustomRenderer customRend = new CustomRenderer();

        tree = new CustomJTree(treeModel);
        tree.setCellRenderer(customRend);
        tree.setEditable(true);
        tree.setInvokesStopCellEditing(true);
        tree.getSelectionModel().setSelectionMode
            (TreeSelectionMode.DISCONTIGUOUS_TREE_SELECTION);
        tree.setShowsRootHandles(true);
        tree.addMouseListener(this);
        tree.addMouseMotionListener(this);
    }

    /**
     * Return the JTree, so others can use it.
     */

    public JTree getTree() { return tree; }

    /**
     * This pops up a JFileChooser, where the user can select which
     * MIDI song(s) to open. These are encapsulated in SongAnalyzer
     * files, and added to the tree using addSongAnalyzer().
     * @see MidiFilter
     */

    public void addSongs() {
        JFileChooser chooser = new JFileChooser(".") + File.separator + "MIDI";
        MidiFilter midiFilter = new MidiFilter();
        chooser.setMultiSelectionEnabled(true);
        chooser.addChoosableFileFilter(midiFilter);
        chooser.setFileFilter(midiFilter);

        int returnVal = chooser.showOpenDialog(tree);
        if(returnVal == JFileChooser.APPROVE_OPTION) {
            File[] files = chooser.getSelectedFiles();
            for (int i = 0; i < files.length; i++) {
                addSongAnalyzer(new SongAnalyzer(files[i]));
            }
            selectNone();
        }
    }

    /**
     * Adds A SongAnalyzer at the last selected position. If no position
     * is selected, it is added to the default group. This SongAnalyzer is
     * selected, and expanded to show its tracks.
     */

    private void addSongAnalyzer(SongAnalyzer sAnalyzer) {

```

```

    if (sAnalyzer.toString() != null) {
        DefaultMutableTreeNode parent = null;
        TreePath parentPath = tree.getSelectionPath();

        if (parentPath == null || parentPath.getLastPathComponent() == root) {
            parent = defaultGroup;
        } else {
            parent = (DefaultMutableTreeNode) (parentPath.getPathComponent(1));
        }
        DefaultMutableTreeNode songNode = new DefaultMutableTreeNode(sAnalyzer);

        treeModel.insertNodeInto(songNode, parent, parent.getChildCount());
        ((GroupAnalyzer) parent.getUserObject()).addSongAnalyzer(sAnalyzer);
        Vector ta = sAnalyzer.getTrackAnalyzers();
        for (int i = 0; i < ta.size(); i++) {
            DefaultMutableTreeNode track =
                new DefaultMutableTreeNode((TrackAnalyzer) ta.elementAt(i), false);
            treeModel.insertNodeInto(track, songNode, i);
        }
        TreePath tp = new TreePath(songNode.getPath());
        tree.addSelectionPath(tp);
        tree.scrollPathToVisible(tp);
    }
}

/**
 * Adds a GroupAnalyzer with title "name" into the TreeModel, returning
 * a DefaultMutableTreeNode if anyone needs it.
 */
public DefaultMutableTreeNode addGroup(String name) {
    GroupAnalyzer ga = new GroupAnalyzer(name);
    DefaultMutableTreeNode dn = new DefaultMutableTreeNode(ga);
    treeModel.insertNodeInto(dn, root, 0);
    return dn;
}

/**
 * Add an unnamed group ("New Group" and a counter) to tree.
 */
public void addGenericGroup() {
    addGroup("New Group " + (newGroupCounter++));
}

/**
 * Returns an array of Analyzers contained in each selected node.
 * If any node doesn't contain an Analyzer (like the root) then
 * it isn't added.
 */
public Analyzer[] getSelectedNodes() {
    Vector nodes = new Vector();
    TreePath[] currentSelections = tree.getSelectionPaths();

    for (int i = 0; currentSelections != null && i < currentSelections.length; i++) {
        DefaultMutableTreeNode node = (DefaultMutableTreeNode)
            currentSelections[i].getLastPathComponent();
        Object nodeInfo = node.getUserObject();
        if (nodeInfo instanceof Analyzer) {
            nodes.add(nodeInfo);
        }
    }
    int size = nodes.size();
    Analyzer[] a = new Analyzer[size];
    for (int i = 0; i < size; i++) { a[i] = (Analyzer) nodes.get(i); }
    return a;
}

/**
 * Returns the Analyzer contained in the selected node (or the last
 * selected node if more than one are selected). Returns null if
 * the node is the root, or under exceptional circumstances (which shouldn't
 * happen.)
 */
public Analyzer getSelectedNode() {
    TreePath currentSelection = tree.getSelectionPath();
    DefaultMutableTreeNode node = (DefaultMutableTreeNode)
        currentSelection.getLastPathComponent();
    Object nodeInfo = node.getUserObject();
    if (nodeInfo instanceof Analyzer) { return (Analyzer) nodeInfo; }
    else return null;
}

/**
 * Removes all nodes currently selected (except the root node and
 * the default group). This requires some updating of Analyzers, depending
 * on what was removed.
 */

```

```

public void removeSelectedNodes() {
    boolean doBeep = false;
    TreePath[] currentSelections = tree.getSelectionPaths();
    if (currentSelections != null) {
        for (int i = 0; i < currentSelections.length; i++) {
            DefaultMutableTreeNode currentNode = (DefaultMutableTreeNode)
                (currentSelections[i].getLastPathComponent());
            DefaultMutableTreeNode parent = (DefaultMutableTreeNode)
                (currentNode.getParent());
            if (parent != null && currentNode != defaultGroup) {
                Object removed = currentNode.getUserObject();
                if (removed instanceof TrackAnalyzer) {
                    return;
                } else if (removed instanceof SongAnalyzer) {
                    GroupAnalyzer oldGroup = (GroupAnalyzer) parent.getUserObject();
                    oldGroup.removeSongAnalyzer((SongAnalyzer) removed);
                }
                treeModel.removeNodeFromParent(currentNode);
            } else { doBeep = true; }
        }
    }
    if (doBeep) { toolkit.beep(); }
}

/**
 * Iterates across all nodes, selecting each.
 */

public void selectAll() {
    Enumeration e = root.breadthFirstEnumeration();
    while (e.hasMoreElements()) {
        DefaultMutableTreeNode p = (DefaultMutableTreeNode)
            e.nextElement();
        TreePath tp = new TreePath(p.getPath());
        tree.addSelectionPath(tp);
    }
}

/**
 * Removes all selection paths from the tree.
 */

public void selectNone() { tree.removeSelectionPaths(tree.getSelectionPaths()); }

/**
 * This is called when a mouse was pressed on the tree and sets up for a drag
 * (and lets mouseDragged() and mouseReleased() do work) if the mouse was
 * pressed on a node containing a SongAnalyzer. Otherwise if the mouse was
 * pressed on a TrackAnalyzer node, change its active status and notify
 * the tree.
 */

public void mousePressed(MouseEvent e) {
    TreePath tp = tree.getClosestPathForLocation(e.getX(), e.getY());
    DefaultMutableTreeNode tn = (DefaultMutableTreeNode)
        tp.getLastPathComponent();
    Object o = tn.getUserObject();
    if (o instanceof SongAnalyzer) {
        dragIsLegal = true;
        draggedNode = tn;
        originalParent = lastParent = (DefaultMutableTreeNode) (tn.getParent());
        tree.setCursor(new Cursor(Cursor.MOVE_CURSOR));
    } else if (o instanceof TrackAnalyzer) {
        Rectangle rect = tree.getPathBounds(tp);

        //Check if click is EXACTLY on a X-checkbox...
        //if so, change status

        if (e.getX() < rect.x + 16 && e.getX() > rect.x) {
            ((TrackAnalyzer) o).changeActive();
            tree.treeDidChange();
        }
    }
}

/**
 * As a drag occurs, this dynamically changes the TreeModel whenever
 * a the mouse enters a TreePath it wasn't in before -- moving the node
 * there.
 */

public void mouseDragged (MouseEvent e) {
    if (dragIsLegal) {
        TreePath tp = tree.getClosestPathForLocation(e.getX(), e.getY());
        if (tp.getPathCount() > 1) {
            DefaultMutableTreeNode potentialParent =
                (DefaultMutableTreeNode) (tp.getPathComponent(1));
            if (potentialParent != lastParent) {
                treeModel.removeNodeFromParent(draggedNode);
                treeModel.insertNodeInto(draggedNode, potentialParent,
                    potentialParent.getChildCount());
            }
        }
    }
}

```



```

        tree.expandPath(new TreePath(potentialParent.getPath()));
        lastParent = potentialParent;
    }
}

/**
 * When a drag is complete (if it was legally initiated), this
 * updates both the GroupAnalyzer that the SongAnalyzer came from and
 * the GroupAnalyzer that the SongAnalyzer should go to.
 */

public void mouseReleased(MouseEvent e) {
    if (dragIsLegal) {
        if (originalParent != lastParent) {
            GroupAnalyzer oldGroup = (GroupAnalyzer) originalParent.getUserObject();
            oldGroup.removeSongAnalyzer((SongAnalyzer) draggedNode.getUserObject());
            GroupAnalyzer targetGroup = (GroupAnalyzer) lastParent.getUserObject();
            targetGroup.addSongAnalyzer((SongAnalyzer) draggedNode.getUserObject());
        }
        dragIsLegal = false;
        tree.setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
    }
}

public void mouseEntered(MouseEvent e) {}
public void mouseExited(MouseEvent e) {}
public void mouseMoved(MouseEvent e) {}
public void mouseClicked(MouseEvent e) {}

/**
 * A custom class that allows only Group Analyzers to have their name
 * changed on the tree.
 */

protected class CustomJTree extends JTree {
    CustomJTree(DefaultTreeModel t) { super(t); }

    /**
     * The overridden method -- returns if the object contained
     * in the selected cell is a GroupAnalyzer.
     */

    public boolean isPathEditable(TreePath path) {
        DefaultMutableTreeNode node = (DefaultMutableTreeNode) path.getLastPathComponent();
        return (node.getUserObject() instanceof GroupAnalyzer);
    }
}

/**
 * A DefaultTreeModel will automatically change the Object in its structure
 * when a JTree cell is done being edited. Unfortunately, it forgets what type
 * this cell was and overwrites it with a String of the name the cell was renamed
 * to... So this changes that behavior to rename the GroupAnalyzer with said String,
 * without replacing it.
 */

protected class CustomTreeModel extends DefaultTreeModel {
    CustomTreeModel(DefaultMutableTreeNode n) { super(n); }

    /**
     * The overridden method.
     */

    public void valueForPathChanged(TreePath path, Object newValue) {
        DefaultMutableTreeNode aNode = (DefaultMutableTreeNode) path.getLastPathComponent();
        GroupAnalyzer ga = (GroupAnalyzer) aNode.getUserObject();
        ga.setName((String) newValue);
        nodeChanged(aNode);
    }
}

/**
 * A custom renderer that supports a tree icon based on
 * the state of the cell it
 */

protected class CustomRenderer extends DefaultTreeCellRenderer {
    ImageIcon active;
    ImageIcon inactive;

    public CustomRenderer() {
        active = new ImageIcon("images/active.gif");
        inactive = new ImageIcon("images/inactive.gif");
    }

    /**
     * Called when the cell needs repainted. This overrides the
     * default renderer, checks activity on the track analyzer

```

```
* (if that is the object represented by this cell) and
* paints a respective icon determined by that track's state.
*/
```

```
public Component getTreeCellRendererComponent(
    JTree tree,
    Object value,
    boolean sel,
    boolean expanded,
    boolean leaf,
    int row,
    boolean hasFocus) {

    super.getTreeCellRendererComponent(tree, value, sel, expanded, leaf,
                                         row, hasFocus);
    Object node = ((DefaultMutableTreeNode) value).getUserObject();
    if (node instanceof TrackAnalyzer) {
        if (((TrackAnalyzer) node).getActive()) {
            setIcon(active);
        } else {
            setIcon(inactive);
        }
    }
    return this;
}
```

```
    }
}
```

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * TreePanel.java <p>
 *
 * A TreePanel is a panel that includes three buttons on top
 * and two on the bottom for managing a tree (which is in the
 * center.)
 *
 * @author      Greg Cipriano
 */

public class TreePanel extends JPanel {

    /**
     * Construct a Tree Panel, passing in a TreeManager and hooking
     * it to each constructed button. These include: Add Song, Delete Song/Track/Group,
     * Create Group as well as Select All and Select none.<BR>
     * The treeMan controls a JTree. This added to a JScrollPane, which is also added to
     * this panel.
     */

    public TreePanel(final TreeManager treeMan) {
        JToolBar topToolBar = new JToolBar();
        JPanel bottomPanel = new JPanel();

        topToolBar.add(createButton("images/add.gif",
            "Click Here to add song(s) to the list.",
            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    treeMan.addSongs();
                }
            }
        ));

        topToolBar.add(createButton("images/remove.gif",
            "Click Here to remove song(s), tracks(s) or group(s) from the list.",
            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    treeMan.removeSelectedNodes();
                }
            }
        ));

        topToolBar.add(createButton("images/addgroup.gif",
            "Click Here to create a new Group.",
            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    treeMan.addGenericGroup();
                }
            }
        ));

        topToolBar.setFloatable(false);

        JButton selectAll = new JButton("Select All");
        selectAll.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                treeMan.selectAll();
            }
        });

        JButton selectNone = new JButton("Select None");
        selectNone.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                treeMan.selectNone();
            }
        });

        bottomPanel.add(selectAll);
        bottomPanel.add(selectNone);

        setLayout(new BorderLayout());
        add(topToolBar, BorderLayout.NORTH);
        add(new JScrollPane(treeMan.getTree()), BorderLayout.CENTER);
        add(bottomPanel, BorderLayout.SOUTH);
    }

    /**
     * Convenience method, so I can create buttons with a minimum of code.
     */

    private JButton createButton(String iconFileName, String title, ActionListener a) {
        JButton b = new JButton(new ImageIcon(iconFileName));
        b.setToolTipText(title);
        b.addActionListener(a);
        return b;
    }
}

```

```

import java.awt.*;
import javax.swing.*;

/**
 * VectorAnalysisGraph.java <p>
 *
 * A VectorAnalysisGraph is where vector analysis
 * results are drawn...
 *
 * @author      Greg Cipriano
 */

public class VectorAnalysisGraph implements MusicGraph {
    private final String[] titles = { "Zoom Factor" };
    private final int[] defaultValues = { 5 };
    private final int GRAPHWIDTH = 400;
    private final int GRAPHHEIGHT = 400;
    private Analyzer analyzer;
    private Image offscreenI;
    private Graphics backpage;
    private JPanel view;
    private JLabel label;
    private ImageIcon icon;
    private double scalar;

    // This image is static so it only need be loaded once.
    private static Image circle = new ImageIcon("images/circle.gif").getImage();

    /**
     * Empty constructor. Doesn't set up anything. Used when an
     * object class is needed.
     */

    public VectorAnalysisGraph() {}

    /**
     * Real constructor... takes an Analyzer, and sets up
     * the panel that will soon be drawn upon.
     */

    public VectorAnalysisGraph(Analyzer a) {
        analyzer = a;
        analyzer.analyzeNotes();
        setVariable(0, defaultValues[0], false);
        view = new JPanel(new BorderLayout());
        view.setPreferredSize(new Dimension(GRAPHWIDTH, GRAPHHEIGHT));
    }

    /**
     * Returns a string representation of this graph.
     */

    public String toString() {
        return analyzer.getFullTitle() + ":  Vector Analysis";
    }

    /**
     * A VectorAnalysisGraph factory.
     */

    public MusicGraph createMusicGraph(Analyzer a) {
        return new VectorAnalysisGraph(a);
    }

    public String[] getVariableTitles() { return titles; }
    public int[] getDefaultValues() { return defaultValues; }
    public JComponent getView() { return view; }

    /**
     * This draws the graph, creating graphic contexts if needed.
     */

    public void drawGraph(boolean repaintIcon) {
        Note[] notes = analyzer.getNotes();

        // Create ordinary graphic context and image
        // for double-buffering.

        if (offscreenI == null) {
            offscreenI = view.createImage(GRAPHWIDTH, GRAPHHEIGHT);
            backpage = offscreenI.getGraphics();
            view.setBackground(Color.gray);
            view.add(label = new JLabel(icon = new ImageIcon(offscreenI)));
            view.validate();
        }

        // Draw background image first
        backpage.drawImage(circle, 0, 0, view);
        backpage.setColor(Color.black);
    }

```

```

// Then draw the vectors themselves
backPage.setColor(Color.red);
double[] position = new double[2];
double[] oldPosition = new double[2];
oldPosition[0] = position[0] = GRAPHWIDTH / 2;
oldPosition[1] = position[1] = GRAPHHEIGHT / 2;
for (int i = 0; i < notes.length; i++) {
    notes[i].addVector(position, scalar);
    backPage.drawLine((int) position[0], (int) position[1],
                      (int) oldPosition[0], (int) oldPosition[1]);
    oldPosition[0] = position[0];
    oldPosition[1] = position[1];
}

// A hack to get the icon to paint REALLY fast. :)
icon.paintIcon(label, label.getGraphics(),
               (view.getSize().width - GRAPHWIDTH) / 2,
               (view.getSize().height - GRAPHHEIGHT) / 2);
}

/**
 * Called when zoom slider has changed. This updates the 'scalar'
 * variable, and then redraws the graph.
 */

public void setVariable(int varNum, int value, boolean redraw) {
    if (varNum == 0) {
        scalar = (double) (30 * value) / analyzer.getSongLength();
        if (redraw) { drawGraph(true); }
    }
}
}

```