Real-time Visual Pitch Analysis

Alan Zhou, ayzhou@princeton.edu

Advisor: Robert Dondero

Abstract:

Modern technologies and algorithms currently allow almost everyone with a fundamental understanding of music to interact with it in ways never allowed before. Two relevant technologies to this area of study are notation software, defined as software that allows for notes to be displayed on the screen, and pitch analysis, defined as algorithms that detect sound input from any source and translate that to a hertz value. These pieces of software are pretty common and useful, but they both lack one fundamental aspect each. Music notation software lacks musical analysis capability, as it is only an interface for displaying notes. Pitch analysis lacks musical notation, as the algorithms give only the user information about the sound itself, but does not convert that into musical notes or rhythms. My research and development implements the intersection of these two technologies in a software called MusAid. The core functionality of MusAid is a more dynamic and user-centric feature that reads the notes of a specific song, interprets live feedback from the user, and displays visual feedback to the user as to whether that particular note was sung or played in pitch or out of tune. MusAid is accurate at listening to and giving visual feedback for keyboard and violin input, but is less accurate for voice input. However, this integration still provides significant improvement in usefulness compared to modern commercial software today.

1. Background

As a beginner musician, it is very hard to begin to learn music. First you have to learn the concept of pitch, the idea of what a note represents, rhythm, tempo, dynamics, and all the

intricate workings of a musical piece that make it a musical piece. It seems extremely daunting, and it is. The majority of musical leaners hire a teacher or a professor to teach them the fundamentals, but often finances and time limit the ability of a student to learn. For a casual user then, improving the performance of an instrument then relies on individual motivation and resources outside of a music teacher.

1.1 Music notation software

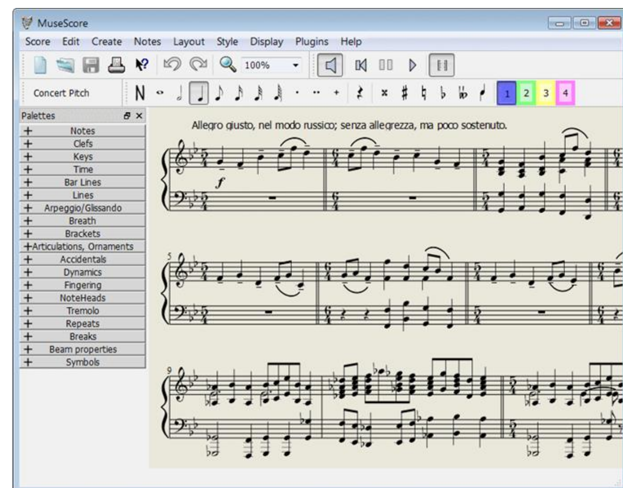

Figure 1. Screenshot of user interface from Musescore, showing some of its features. [1]

The modern approach, then, is to use learning software as a way to either supplement or entirely teach a beginner the musical fundamentals. As stated earlier, there are two major classes of musical software: musical notation software and pitch analysis software. Musical notation software mainly gives people the ability to both display musical notes on the screen and edit those notes as well. Some musical software also give some additional features, such as extraction into a specific file format (.wav, .pdf, .mp3, .midi, etc.) and a music player that plays the music displayed on the screen. The types of files that can be exported vary, from sound files (.wav), to visual files (.pdf), to specialized music files that follow a certain industry standard (.midi). These special files usually contain text describing features of the song, such as tempo, dynamics, and notes. Another format for musical files that is less popular but still used is called abc format, and we'll get to that later.

An example of one of a musical notation software is MuseScore [1]. As listed on their website, their features include creating sheet music with their editor, listening to the score using

computer playback, sharing and printing the score, and extracting the score into specific file formats. For a beginner, this software is useful in terms displaying notes and understanding how musical pieces are written and created. One critical aspect lacking, however, is user-software interaction. As an educational tool for performance, however, this tool is nearly useless. Because the software only takes user input in the form of what note to display, it does not actually analyze the features of the sound if the student were to play the notes. Thus, musical notation cannot help the student improve his or her playing.

1.2 Sound analysis software

On the other hand, sound analysis software has the completely opposite problem. The purpose of analyzing pitch is to provide input on the features of the sound waves it hears using sophisticated algorithms. Some features of sound analysis may include pitch analysis, loudness detection, pitch shifting, and percussion onset detection. An extremely good open-source library that implements all these features in Java is called TarsosDSP, written by Joren Six, located at https://github.com/JorenSix/TarsosDSP [2]. For the purposes of our research, I will mainly be looking at pitch analysis in the form of an example application called a spectrogram, located at http://0110.be/posts/Spectrogram_in_Java_with_TarsosDSP. Figure 2 shows the spectrogram

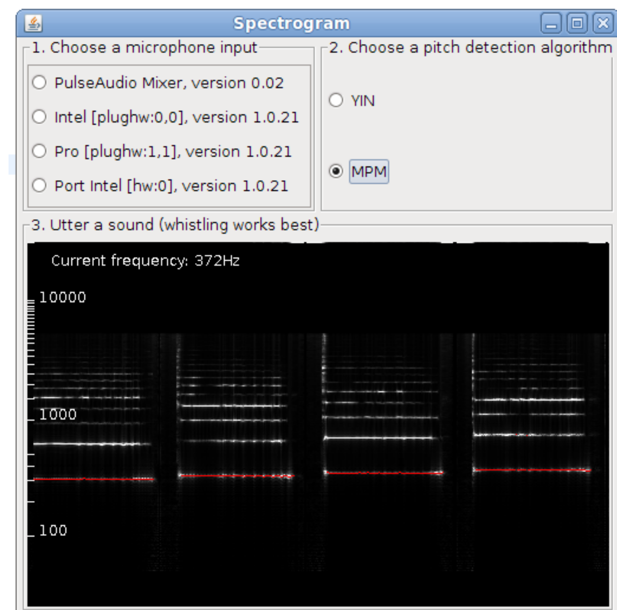

Figure 2. The TarsosDSP spectrogram in action, plotting hertz values over time [2].

listening to live sound input, using an algorithm to determine the frequency at which the sound wave is vibrating, and then plotting the hertz on a scrolling graph.

Figure 3 shows the flow of sound through the Tarsos processing pipeline. Tarsos listens to audio using Java Sound's API, and then the dispatcher chops into different user-customizable size blocks, with or without some overlap between the blocks, again user-customizable. Afterwards, each "event" block is sent to the processor, where some algorithms are run on the event, depending on what you're attempting to get out of the input. In order to determine the pitch of the audio, the processor is a pitch-detection algorithm. If instead, a user wanted to determine the pitch onset of a note, that user would run an onset detection algorithm, etc. At the end, the output would be a return value from the processor, and then the return value can be used however the user likes.

One major shortcoming that a student would have with this kind of tool in general is that although it takes live sound-input and does analysis on it, it is unnecessarily complicated and not necessarily music-related at all. For example, for the spectrogram example that we saw earlier, a beginner student would be unlikely to even recognize what a hertz value is, let alone convert that into pedagogical advice that could help the student improve for the future. Furthermore, the interface for such complicated software makes the underlying assumption that a beginner student knows how to utilize all the specific features for that software, which is simply not a safe assumption. Because of this, there needs to be a combination of the musical notation software and pitch analysis software that takes live sound input

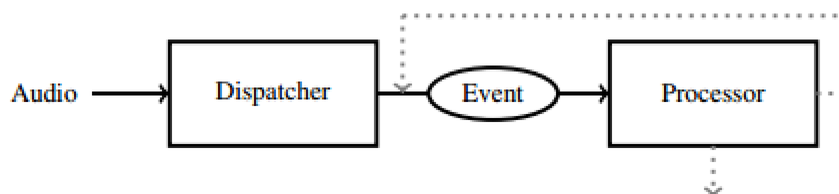

Figure 3. The TarsosDSP pipeline. It takes audio, filters it and chops it up into blocks called events, has a processor that analyzes the event, and produces some form of output [3].

from a user, converts that into something simple a beginner student can understand, and ultimately give the student pedagogical advice that can ultimately improve their playing or singing.

1.3 SingAndSee

The commercial software that comes closest to meeting these criteria is called SingAndSee. (http://www.singandsee.com/) [4]. SingAndSee has two functionalities. One is a spectrogram similar to the spectrogram for TarsosDSP, except that as a commercial software it is a little more aesthetically appealing than Tarsos's interface. Another feature that is has is that instead of plotting hertz, it plots the musical pitch that your audio is vibrating(for example, A flat, E natural, etc.). This solves a lot of the problems that TarsosDSP had and MuseScore had. In terms of musical notation software, it displays exactly what pitch you are singing/playing, so there is at least a minimal musical interface for the user. In terms of pitch analysis software, it uses a pitch detection algorithm to determine what pitch the user is singing/playing and then displays it on the screen.

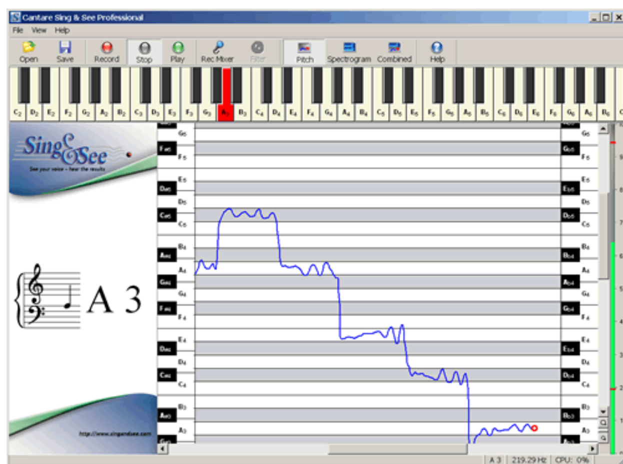

Figure 4. SingAndSee's user interface. The box on the left side is the note estimation, and the right side is the spectrogram [4].

Figure 4 shows the spectrogram over time as well as the pitch estimation in terms of musical notes. As previously mentioned, SingAndSee solves a lot of the problems that MuseScore and TarsosDSP had, but it still is not the perfect educational tool. There are several flaws. One flaw is that SingAndSee does not give visual feedback in relation to a 'correct' note. SingAndSee only tells the user

that they are singing at a certain pitch, while often times users are trying to play a song, and are trying to compare whether their notes are correct or not in relation to the note in the song. Thus, if a user were trying to learn a specific song, then SingandSee would cease to be useful. Another flaw is the spectrogram. To a beginner's eyes, it appears as a bunch of lines, which a bunch of plateaus in the line here and there. To a beginner, this information is essentially useless. Plotting more a more meaningful note graph over time would be significantly more helpful in understanding which notes they were playing correctly and which notes were being played wrongly.

As such, this research and development focuses on bridging the gap between these two technologies and using the result to be a much more effective educational tool for students.

2. Functionality

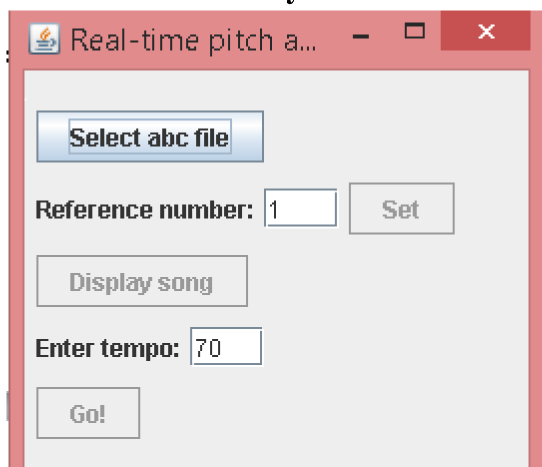

Figure 5. Shows the abc file select button, reference number selection, display song button, tempo selection, and go button.

MusAid is an extension of a music notation library that adds dynamic repainting of the notes on the screen, dynamic pitch analysis, and visual notation feedback

The first feature the user sees is a basic user interface. The user first clicks on the "Select abc file" button, which pops out a file explorer in which the student can select the abc file that the student wants to sing from. As mentioned earlier, an abc file is a specific type of file that describes the features of a song. Currently, MusAid only supports abc files. Additional information about this specific design

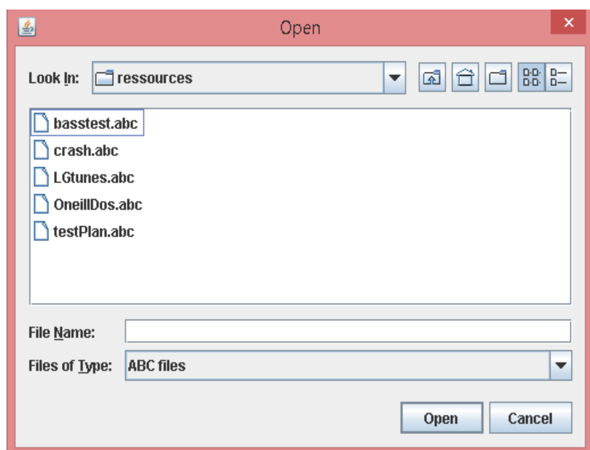

Figure 6. Shows the file chooser that pops up once the user clicks "Select abc file". Only filters for abc files.

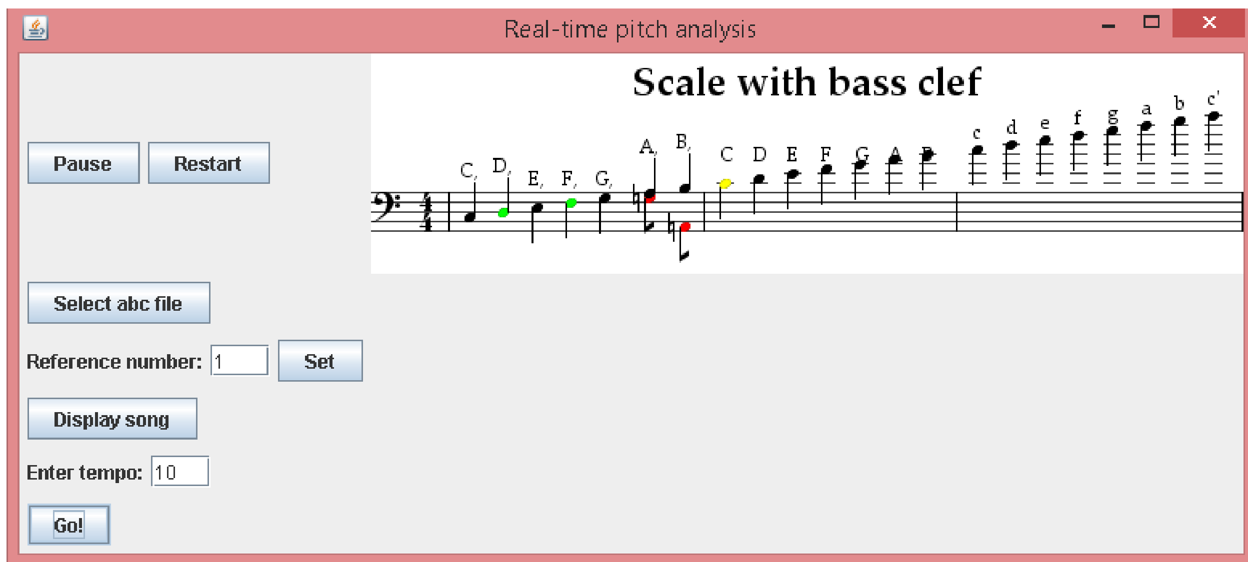

Figure 7. Additional functionality of real-time pitch analysis once song is selected. Has pause and restart button. For the visual feedback, black notes are notes that either have not been played or were not perceived to have heard a specific pitch. Green means note was played in tune, and red means note was out of tune, displaying which note was perceived to have been played.

decision and abc files are included in the design section.

An additional feature is that the buttons are not enabled until the previous step has been concluded. Figure 5 displays the above functionalities of the interface, and Figure 6 shows the pop-up file chooser menu.

After the user selects the song and presses “Display”, the song will be concatenated above the above interface, as shown in Figure 7. These additional features include a pause and a restart button, the song itself, and dynamically updating feedback. As soon as the user presses “Go”, the program will start listening for sound input for the amount of time that the highlighted note lasts. Note that that the current note whose input is being compared to is highlighted to yellow. After waiting and listening to the user for that specific note’s amount of time, the program will move on to the next note, and the user will get immediate feedback on the note that was played. It will appear black if the algorithm did not perceive any specific note to be played, green if the note was in tune, and red if the note was out of tune. Once the song ends, MusAid will stop and the user must click restart in order to load a new song.

3. Design

Real-time visual pitch analysis is an executable JAR that uses two libraries: abc4j and TarsosDSP. The executable main that is run is an addition to the source code of Abc4j. It is located in /src/AlanUI.java of the Abc4j library. Some minor modifications were also made to Note.java and JNote.java of src/notation/Note.java and src/ui/JNote.java, respectively.

The flow of the program is described in Figure 8. Once the student selects the song and presses go, Java sound listens to the student's audio input for the current note that is supposed to be played. The implementation of a pitch-detection algorithm in TarsosDSP will then give a pitch to MusAid which will compare the pitch to the current pitch. Then, Abc4j will dynamically repaint the song as it progresses to give visual feedback to the user.

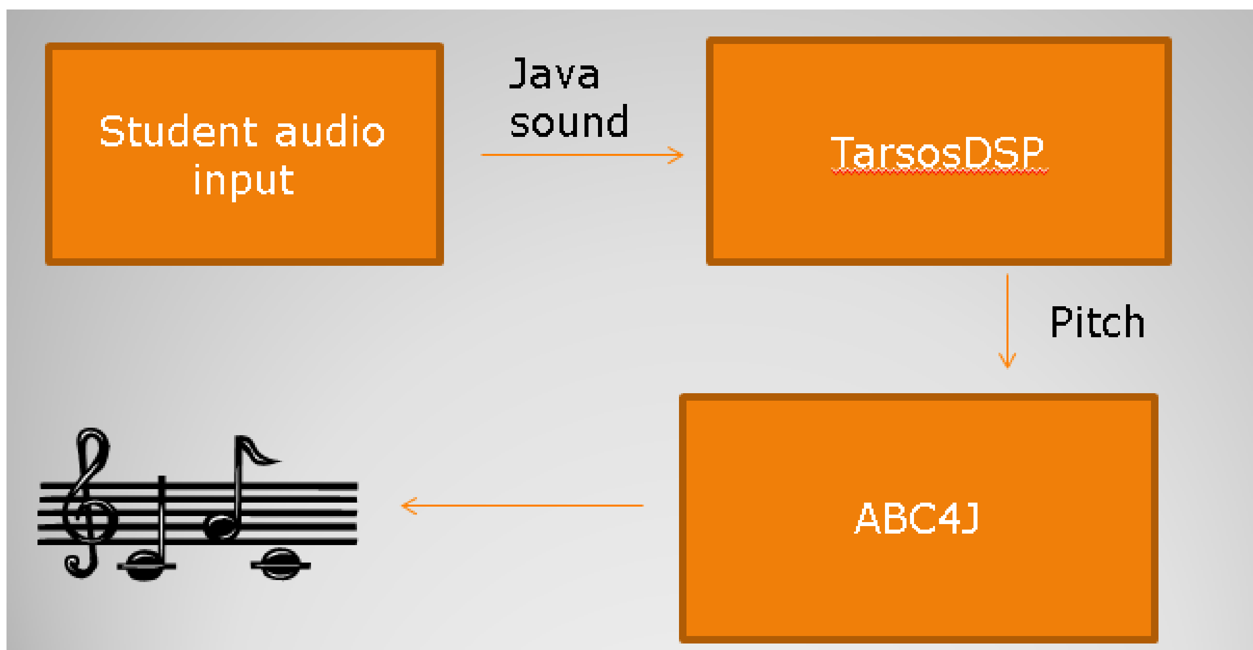

Figure 8. The flow of the program.

3.1 Abc4j

Abc4j (https://code.google.com/p/abc4j/) is a Java library that represents notes, songs, and other musical elements as objects, and eventually displays those representations on the screen using Java Swing [5]. One major advantage of using Abc4j is that it is multifaceted. Not only can it display notes, it can also create musical pieces, edit them, and parse music from an abc file and display on the screen. These features are exactly what real-time visual analysis needs because my software needs to be able to compare a user's input to a specific song, have a method to display specific note inputs onto a screen, and is easy for the user to read and understand. One of the best features of abc4j is that it has its own Swing representation of each musical object. An example of this is the Note class. A note can be represented as an object, but all it is is some data associated with the properties, such as pitch, length, where in the piece it is located, etc. However, abc4j has a Swing representation of the Note object called a JNote. When a method is called to print the entire piece, each musical component is rendered by calling its render method, which works synonymously with other render methods to paint the object representation of a song as a Swing representation.

One disadvantage to using Abc4j is that it is not well-maintained. The last update was made in July 2008, and no modifications or updates to the source code have been made since then. There is also a wiki page (https://code.google.com/p/abc4j/w/list) that describes the basic functionalities of Abc4j, but it if you want to either modify it or want it to perform more than the most basic features, you will have to download and view the source code to perform the modifications. Another disadvantage is that instead of parsing music from a midi file, which is the most popular format for transferring notes, rhythms, dynamics, etc. Instead, Abc4j implements an abc file parser that parses an abc formatted file and converts it into Java objects

```
X:1
T:Shandon bells
%test de commentaire.
K:D

X:2
T:Shandon bells
%test de commentaire.
K:D dorian
abcde

X:3
T:Anderson's
M:C
L:1/8
K:D Major
B|ABdf efdB|AFF2 EDEF|ABdf efdf|afef d3:|!
f|a2fa bafa|a2fd edBd|a2fa bafb|afef defg|!
a2fa bafa|a2fd edBd|AFF2 ABdf|afef d3:|!
```

Figure 9. An example of an Abc formatted file. Note that each file can hold multiple songs, separated by the 'X's.

that represent the song. Abc files are less popular than MIDI files or MusicXML files, the more popular file types, but there is an easy open-source converter called EasyABC located at http://www.nilsliberg.se/ksp/easyabc/ that allows easy conversion from MIDI or MusicXML format to Abc format.

An example of an abc-formatted file is Figure 9. As noted in the caption, each abc-formatted file can hold multiple songs, separated by the Xs. The important parts of a file are X, the song number; T, the title of the song; L; M, the time signature; and then the actual song at the end of each segment of text.

Once abc4j reads the file, abc4j converts each song into Java objects, the two most important of which are a Tune, and a Note. These core object representations are located at /src/abc/notation/ of abc4j's source code. A tune contains all the relevant information about a specific song, including its title, key signature, time signature, as well as a vector of notes that indicate the sequence of notes that a song contains.

Note.java contains many different attributes of a note that will come in handy considering our case, which is iterating through the song and comparing user input with the given note. First, the note has constants that represent the 'height' of a note, a synonym for pitch. However, these constants do not take into accidentals, which are sharps or flaps. Accidentals are represented with a completely different object, an Accidental object, which each note owns an instance of. Another crucial property of a Note object is its length, again represented at a constant. This

property represent the duration of a note that should be played during the song. These two properties form the backbone of the application.

3.2 File Chooser

The vast majority of the application is a class in /src/ called AlanUI.java. In order to enhance Abc4j so that it met the requirements for the project, some additional functions were needed. First of all, there needed to be an interface that allows a user select a specific song and display that on the screen. The current method was to programmatically select a file using Java to programmatically select a specific file, and then programmatically use the parser to parse fthe file into Java objects. Because beginner music students will most likely not know anything about programming, there needed to be a way to hide that complexity from the students. The design decision was to use a 'File Chooser' class implemented by the Java Swing package that allowed students to browse through their file directory and choose the song that they wanted to sing/play. This was implemented by having a JButton in the interface open up to the File Chooser, and then once the user selected an abc file, it would set the current song to be displayed as the song that they selected. A feature that makes this easier is the abc format filterer, implemented as a FileNameExtensionFilter. This allows for the user to only see abc files in the file explorer, making it harder for them to misclick an unacceptably formatted file, which would throw an error.

3.3 Colored Notes

```
{
char[][] chars = valuateInvertedNoteChars();
if (chars[0] != null) // stem
        gfx.drawChars(chars[0], 0, 1, (int)displayPosition.getX(), (int)displayPosition.getY());
if (color != null) {
    previousColor = gfx.getColor();
    gfx.setColor(this.color);
}
if (chars[1] != null) // head
        gfx.drawChars(chars[1], 0, 1, (int)notePosition.getX(), (int)notePosition.getY());
        }
if (color != null)
    gfx.setColor(previousColor);
}
}
```

Figure 10. A section of the renderNoteChars() function in JNote.java.

Another feature that the original abc4j lacked was its inability to render colored notes. Because notes in regular musical pieces were only colored black, there was no need to ever render notes in any other color. Because I needed to give users visual feedback on whether their notes were sung right or wrong, I needed to be able to color notes red, green, and yellow for the current note sung. In order to do this, I modified the infrastructure of Abc4j a bit to allow for colored notes. First of all, I edited Note.java so that it had a variable called ‘color’. Then, I created a constructor so that it would allow easy creation of notes with a certain color.

Second, I edited JNote.java so that it could take advantage of the color property of the note. First, I also created a property in JNote called color, and then had the single constructor of JNote set the color property equal to the color property of the Note that was passed in. However, note that not all Note objects were created with a non-null color. Only Notes that purposely had a color set had a non-null color, otherwise the color variable remained null. Finally, I edited the renderNoteChars() function so that it could take advantage of the color property. Figure 10 shows a section of the renderNoteChars() function. The basic flow is first, check whether the JNote’s color property is not null. If it is, then render it normally, painted black. However, if it does have a color property, then first save the previous color into a variable, and then change the

color to whatever color the JNote has. Then, paint only the head of the note, not the stem. The stem should remain black because I only want to tell the user whether the note was right or wrong and what note they played, and this is represented by the head, not the stem. Once the rendering takes place, then the previous color is restored and it is free to continue painting other notes.

3.4 Dynamic repainting

Originally, Abc4j was very static. The main use case was for reading a file, displaying it on the screen, and then never updating it again. However, our use case requires us to be constantly reading audio input, interpreting it, and then updating the piece to reflect the user's performance.

```
jscore.setTune(tune);
alan.add(jscore);
alan.repaint();
frame.pack();
frame.repaint();
```

Figure 11. The piece of code called to dynamically update the JPanel holding the graphic representation of the piece.

This was done by either adding in a new note or revising an existing note in the current Tune object (song), and then calling the code in Figure 11 to update and repaint the entire piece. First, the code updates JScore (the object representation of the piece) so that it contains the current tune. Then, alan (the reference to the JPanel holding the jscore) is called to add it and then repaint it. Finally, frame (the reference to the overall JFrame holding everything) is called to pack all the songs properly into the frame, and then everything is repainted so that it looks nice.

3.5 Animating the Beat Marker

The original design was to have a vertical line animate and run through the piece, having the user sing/play whichever note the vertical line was hovering over. Figure 12 shows the original proposition. However, there are several reasons why I decided to replace the beat

Figure 12. Shows the original proposition with the moving beat marker, hovering right after the rhy in rhy-thm.

marker. First, because the duration of time the beat marker spends on each note depends on the length of the note itself, the beat marker's speed would have to vary between note to note, and this may be confusing for some users as the beat marker may shift speed often. Second, animation may be choppy because each time I repaint the beat marker, I also have to repaint the entire song, as per the dynamic repainting section above. This may result in choppy animation, and represents another layer of dynamic repainting that could result in worse performance.

Thus, I decided to represent the current note that was being sung as a yellow note, and the duration for which it lasts is dependent on both the tempo and the duration of the note being played. This would do the exact same feature as the beat marker but would be a little less aesthetically appealing, which is a reasonable trade off given the performance and simplicity of the new approach.

This was implemented by first calculating the duration in milliseconds of the time the program would have to listen to the user for.

The formula is `timeToRest = (long) (1000* quarterLength*((Note) currentScoreEl).getDuration()/Note.QUARTER),` and quarterLength is equal to `quarterLength = 60/(double) tempo.` The duration of a quarter note is equal to 60/tempo because tempo is measured in quarter notes per minute. Then, the timeToRest must be equal to the ratio of the duration of the note compared to the quarter note multiplied by how long

the quarter note lasts. The ratio is represented by `((Note) currentScoreEl).getDuration()/Note.QUARTER)` and the quarter length is represented by quarterLength. Finally, multiply by 1000 to get the amount of time to wait in milliseconds. Use this time as a parameter to Thread.sleep(), so the system waits that specific amount of time before continuing.

3.6 Integrating with TarsosDSP

This is the heart of the application. TarsosDSP has an algorithm that determines if a 'chunk' of user input has a pitch, and if it does, then it returns a pitch to the user. This is done by adding in a pitch-detection processor to the dispatcher in the Tarsos flow as shown in section 1.2. The algorithm that I am using is called the Yin algorithm [6].Once the Yin algorithm processor is added, a separate thread starts to listen to user input.

```
int pitch = (int) pitchDetectionResult.getPitch();
int count = 0;
if (pitches.containsKey(pitch)) {
    count = pitches.get(pitch);
}
if (pitch > 50) {
    pitches.put(pitch, count + 1);
}
```

Figure 13. A snippet of the handlePitch() function. This is called whenever a pitch that is non-null is heard.

For each note in the piece, the program will listen to the user for exactly timeToRest milliseconds described in section 3.4. This is implemented with a helper function, listenFor(). Once timeToRest milliseconds passes, the program will close the stream until the next time listenFor() is called. The processor in TarsosDSP that handles the pitch is implemented by implementing the PitchDetectionHandler interface. A PitchDetectionHandler requires a handlePitch() function, which is called whenever the algorithm suspects that there is a pitch. Figure 13 shows a snippet of the implementation of the handlePitch() function. The snippet is called whenever the pitch is not null.

Because the algorithm may find multiple pitches during the duration that the program is listening for user input, there must be a way to determine which pitch the algorithm 'thought' the user meant to sing. The approach that I took was to use a HashMap, and to use the integer value of the pitch and map it to the number of times that the pitch appeared during the duration of that one note. In the code snippet in Figure 13, pitches represents a HashMap, and it first checks whether the pitch exists in the HashMap. If it does, it retrieves the count and then puts the int value of the pitch back in the HashMap with a +1 count. Once the note ends, the HashMap is reset and a new one is used for the next note.

This is the most accurate approach when compared to other approaches, more specifically the average value approach and the latest pitch heard approach for several reasons. First, as a user is ending his or her audio feedback, the note may taper off at the end and then fade, and the algorithm may interpret this as the user singing/playing lower than the user actually is. This ruins the latest pitch heard approach because it will interpret the user as playing/singing a hertz value lower than the actual hertz value. Second, there may be some outliers in the data. Sometimes the algorithm may interpret the user's feedback as an octave higher or lower than the actual hertz value, and so the feedback may be twice or half the actual value. Furthermore, the beginning and the end of user input may have discrepancies with the majority of the user's input because it is tapering on and tapering off as the user begins and stops singing, respectively. Thus, the most often heard approach is effective because it avoids both of these problems because it makes the assumption that the pitch most heard happens the majority of the time, and disregards discrepancies because the number of times that they happen is less than the number of times the actual pitch occurs. Rounding to an integer value is necessary because the pitch is returned as a double, and it would be very unlikely that the user is accurate down to the double value.

3.7 Hertz to Pitch and Note Matching

Once the algorithm's hertz value is calculated from the algorithm, then the corresponding note and accidental is calculated from the hertz value by using the frequencyToPitch() function. The frequencyToPitch() function uses the formula 69 + 12 * Math.*log*(frequency/440.0)/Math.*log*(2) to calculate the corresponding real pitch. 69 on this scale is A*440*, which is the central pitch on this scale. One integer value up or down represents a half-step either up or down.

However, in Note.java, these constants are represented a little differently. A*440* is represented as the constant 9, and although each half-step or half-step is accounted for in the constants, they are represented with the Accidental object, not with a constant. For example, an A flat would be represented with a 9 constant, but its accidental value would be an Accidental.FLAT, not an Accidental.NONE or Accidental.NATURAL. However, a G note would then be represented as a 7, which takes into account the half-step in between the G and the A, but officially there is not an 8 constant representation of an A flat. Thus, in order to get around this I need to check whether both constant representation and the accidental representation match. Figure 14 shows the frequencyToPitch() function, which calculates both the Note constant representation and the accidental representation.

```
private byte[] frequencyToPitch(float frequency) {
    byte[] pitch = new byte[2];
    double rPitch = 69 + 12 * Math.log(frequency/440.0)/Math.log(2);
    pitch[0] = (byte) Math.round(rPitch-60);
    if (accidentalPitches.contains(Integer.valueOf((int) pitch[0]) % 12)) {
        if (keySig.hasOnlySharps()) {
            pitch[0] = (byte) (pitch[0] - 1);
            accidental = Accidental.SHARP;
        }
        else {
            pitch[0] = (byte) (pitch[0]+1);
            accidental = Accidental.FLAT;
        }
    }
    return pitch;
}
```

Figure 14. The frequencyToPitch() function which converts hertz to a Note constant and an accidental. The note constant and the accidental both depend on the current key signature. At the end of the function, pitch[0] contains the pitch constant and the accidental value contains the accidental constant.

First, the function converts the hertz value into a pitch constant as defined by Note.java. AccidentalPitches is a set that contains all the constants in an octave that are accidentals. For example, 8 is an accidenrtal constant as explained above. 6 is another constant because it represents G flat / F sharp, etc. Because there are twelve half-steps in an octave, I can mod the pitch constant by 12 to determine whether it is an accidental constant or not. If it is an accidental constant, I have to determine whether to use the constant above (such as A sharp) or the constant below (such as B flat) because they are synonymous with the same note.

The if statement `keySig.hasOnlySharps()` is true if the key signature has only sharps, which means that I will want to go down one half-step and use Accidental.SHARP. If it is false, then I will want to go up a constant and use Accidental.FLAT. At the end of the function, I return the pitch array, in which the first element is the pitch constant. I use the result to determine whether the algorithmic determined pitch and accidental is equal to the actual note and accidental.

3.8 Displaying Visual Feedback

The dynamic updating of the music is done with a class called dynamicPaintTask, which is an inner class in AlanUI.java. It inherits from a SwingWorker. This task is executed as a

background task because Swing has a main thread called the Event Dispatch Thread. When time-consuming tasks are run on it, the application freezes. Thus, the dynamic repainting is run as a separate thread to prevent this from happening. The doInBackground() function is called whenever an instance of the class is created and execute() is called on that object.

When the doInBackground() function is called, a while loop is called in which it iterates through all the elements in the Vector of the Tune object that represents the song. A vector comprises of many MusicElement objects that represent many different aspects of a song, such as either a note, rest, bar, chords, etc. If it is not a Note object, then I am not interested in it and I go on to the next object in the vector. If it is a rest, I want to wait for the calculated amount of time then go on to the next object. If it is a note, then I want to highlight the note, wait for the specified amount of time, and then check the note and accidental that was calculated using the set algorithm.

Next, I check the note and accidental to see if it matches the current note. For each iteration of the loop, I create a vector. This vector is used as an argument for the construction of a Multinote. A Multinote object represents a chord, or a series of notes meant to be stacked on top of each other. If the note and accidental matches, I color the current note green and add it to the vector to construct a Multinote with. If it does not match, I take the original note, color it black, and add it to the vector. I also take the algorithm's perceived note and accidental, and add it to the vector as well. The Multinote, either composed of a single green note or black and a red note is then queued for the next iteration of the while loop, where it replaces the previous note and is painted there.

4. Testing and Evaluation

Over the course of the semester, I debugged MusAid with the help of the sample abc files included in abc4j's resources folder. For accuracy purposes, I ran tests on basstest.abc, pictured in Figure 15. The reason why basstest.abc was chosen was because first, it provided insight on how well MusAid handles low notes as well as high notes. Second, it provided insight on both keyboard and voice inputs and how accurate in comparison to each other the algorithm was in calculating the actual pitch.

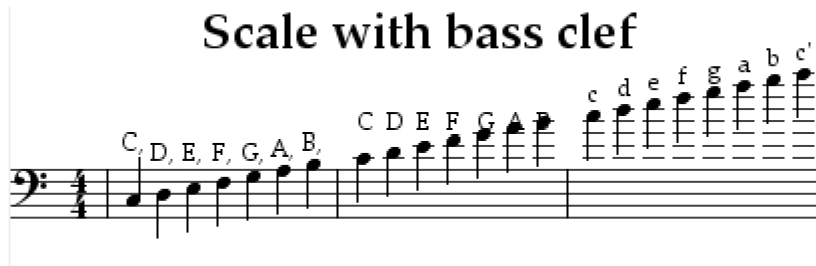

Figure 15. Visualization of basstest.abc with Abc4j

Over the course of 10 tests run with keyboard input provided by MuseScore's synthesizer, accuracy out of 24 notes was at least 21/24 for all 10 tests. Furthermore, the errors were all located in the first three notes. Figure 16 shows an example of one of the runs done. As you can notice, the first three notes have no detected input for them, while the rest of the notes' accuracy was spot-on with the keyboard input. Some anomalies in other runs included one run where the third note was correct, one run where the second note's pitch was interpreted to be one octave above the correct pitch, two runs where the second run was correct, and one run where the second note was incorrect (it matched to a C instead). However, the important thing is that the last 21 notes remained constant and stayed green for all 10 runs. As such, the accuracy for these tests are at least 87.5%.

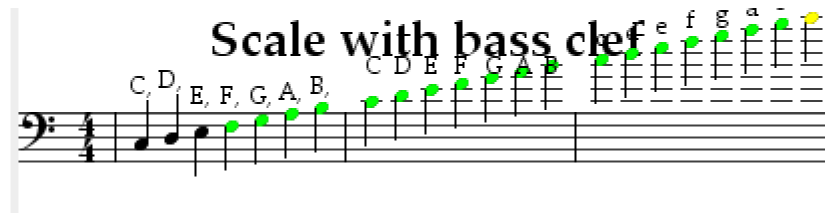

Figure 16. An example of one of the runs done with MuseScore keyboard input

There are several implications as a result of these tests. First, MusAid works better for pitches that are not too deep in the bass clef, and it is significantly more accurate when used for songs in the treble clef. Second, MusAid is consistent when it comes to analyzing user input. If the input is the same as previous input, then most likely the algorithm will give the user the same visual feedback. Third, when it comes to analyzing the errors in that are too deep in the hertz scale, the algorithm may come into difficulties. Either it will not recognize a specific pitch, or it may mark it wrong. This is most likely because the pitch is too low for the algorithm to distinguish from another pitch, and as such there is not a single definitive pitch that was spit out by Tarsos. Thus, the lack of visual feedback.

I also tested voice feedback by singing basstest.abc from the first note to the eighth note, which is my voice range. Note that these tests are EXTREMELY subjective on the accuracy of your pitch, voice range, and tempo sung as well as an extraordinarily wide range of other factors, but it should at the very least provide a baseline of the accuracy of user voice input. Out of ten tests, I managed to sing 4 or more notes out of 8 on seven of those attempts. An example of a run where I got 5/8 correct is Figure 17. Some of the same problems that I noticed in the keyboard runs were also present in the voice runs. In every single one of my runs, at least one of three bottom notes were simply getting dropped, and in over half the attempts, the second note was getting interpreted as a C, not a D. Overall, the accuracy of the algorithm is pretty good, getting over 85% on keyboard inputs and performing decently well on voice inputs, albeit those tests are fairly subjective. The main problem is not that the algorithm is marking my voice input wrong, but instead is just not recognizing that user input is being sung on some notes.

Figure 17. An example of one of my voice runs where the accuracy was 5/8.

These runs were all done manually due to the real-time audio nature of MusAid, so the statistical significance of these tests may not be as significant as expected. They do, however, give us a glimpse as to the accuracy and success of MusAid in achieving its goal, actually helping students improve their playing without the aid of a music teacher. I do believe that functionally, this program has achieved all it has set out to do. It has combined music notation software and pitch analysis software in a user-friendly way that ultimately is accurate enough to give the user feedback on what they sang/played correctly or incorrectly.

4.1 Limitations

There are several limitations that prevent MusAid from being a truly commercially viable product. First of all, the test results really accentuate the dependency on the algorithm used to determine what pitch at which the user input was vibrating. If the algorithm is inaccurate, then no matter how aesthetically appealing the interface is, it will always be giving the user false information. Currently, the YIN algorithm is doing a satisfactory job of analyzing pitches in the high range of the bass clef and all of the treble clef. However, once it gets to around the range of the middle C in the bass clef, the algorithm starts becoming inaccurate. Because MusAid depends on the Tarsos implementation of the YIN algorithm, MusAid is limited to the accuracy of their implementation, rather than anything on my side. Thus, any inaccuracy in the algorithm may have to be dealt with unless I have the time to learn the theory and math behind pitch analysis algorithms in order to either look at their implementation and improve it or try another approach entirely.

On the subject of pitch analysis algorithms, the majority of pitch detection algorithms can only detect non-chord pitches. For example, if two instruments are playing at the same time, then it will be extremely hard to detect which instrument specifically you are trying to identify.

This also applies to background noise. If there is too much noise in the background, then these algorithms will become unable to identify the pitch because there is too much extraneous audio input coming in. This limits the abc files that the user selects to only allow one instrument or voice at a time. However, the default behavior currently is to not throw an error for if there is more than one voice in the song, but to simply allow audio input for the first voice, and ignore the other voices. These limitations are currently based on the current algorithms out there, so this may change in the future when either multi-voice pitch detection algorithms are implemented accurately or if current single-voice algorithms are enhanced to recognize multiple voices.

Another limitation is that there are still some interface features that are slightly buggy, and may decrease usability. One bug is that once the user hits the 'Restart' button and another interface pops up, once the user displays a brand new song, the 'Pause' and 'Restart' buttons don't work anymore. This may be because of the way I am either creating new 'Pause' and 'Restart' buttons or may be because of the way I am creating the new interface, but it is a thing that decreases the commercial viability of the project. Another bug is that sometimes when you click the 'Display' button while a song is currently taking audio input, the song starts over, but it starts listening again at a faster pace, and there are two yellow notes highlighted. The intended functionality was to stop the audio input and to display the new song, but this bug may be because of a concurrency issue.

None of these bugs take away from the core functionality of the product, the integration of the musical notation software and the pitch analysis software, but it does take away from the commercial viability of the product. There is no real aesthetic appeal of the product, partly because my goal for this research was to implement functionality first, and then enhance the aesthetics. Thus, there are only the very basic swing components that make up the product, and it

is not that visually appealing. Furthermore, the interface bugs, although they don't take away from the functionality, do make it annoying to try to manipulate MusAid until it works properly again. Thus, these bugs do make it very hard to sell the product and it is unlikely it will ever be as successful as the other music software.

5. Conclusion

Although MusAid will probably never reach commercial viability in its current state, there are some logical next steps that will take it closer to reaching that goal. First, all the current interface bugs need to be eliminated. This can be done by a lot of rigorous testing of the various modules of MusAid to determine whether the Swing, Abc4j, or Tarsos is responsible for the bugs that I described in the testing and evaluation section. Once these bugs are gone, then comes the accuracy testing. It would be extremely helpful to somehow automate the testing such that I can determine what ranges the algorithm is accurate in listening to and which ranges it's inaccurate in listening to. Then, once I figure out what ranges are faulty, I can do one of two things. The worse option would be to limit the users' input to certain ranges where the algorithm is effective, and throw a warning when the user attempts to analyze singing or playing for an inaccurate range. The better option would be to look at the algorithms, and then fix the code by either coming up with a better algorithmic approach or look specifically as to why it doesn't work for low notes and fix that. Finally, in order to make it commercially viable, there needs to be better design of the layout so that is looks attractive to potential consumers.

Overall, the independent work experience was great in that I got to develop an idea from scratch and then implement it throughout the semester. With my previous projects, they were either all required as a part of a course or part of my work, but this was something that I thought would be legitimately interesting and fun to work on. Furthermore, this was one of the

first projects in which I used an IDE (IntelliJ IDEA) to develop. I had used basic text editors such as Sublime before, but it was really neat to learn about how multi-file projects were built, compiled, and then maintained. Although my project was pretty much one source file, I got to see and understand how the author of Abc4j built his library and designed his code to work neatly together, and it should help a lot in the long run when I am working for an actual company, not just creating one or two files for each of the programs in the COS courses I am taking.

One great thing about this experience is that the project design laid out in the Powerpoint presentation at the beginning of the semester didn't really change as the semester progressed. I had heard stories of how students would have to change their overall project because they run into technical difficulties, their project was too ambitious, lack of work ethic, etc. I am proud to say that even though my project did not reach its 'reach' goals of being commercially viable, I did achieve most of my goals that were laid out in my PowerPoint. I was able to combine music notation and pitch analysis software neatly to produce something that in my research has never been released before. To that end, I would label this project a success.

References

[1] MuseScore. musescore.org. 2014.

[2] TarsosDSP. https://github.com/JorenSix/TarsosDSP/. 2014.

[3] J. Six, O. Cornelis, and M. Leman. TarsosDSP, a Real-Time Audio Processing Framework in Java. In *AES 53rd International Conference*, 2014.

[4] SingAndSee. http://www.singandsee.com/. 2014.

[5] Abc4j. https://code.google.com/p/abc4j/. 2008.

[6] A. de Cheveigne´ and H. Kawahara. YIN, a fundamental frequency estimator for speech and music.J. Acoust. Soc. Am., 111(4):1917–1930, 2002.

Appendix

AlanUI.java

```java
import abc.notation.*;
import abc.parser.TuneBookParser;
import abc.ui.swing.JScoreComponent;
import be.tarsos.dsp.AudioEvent;
import be.tarsos.dsp.io.jvm.JVMAudioInputStream;
import be.tarsos.dsp.pitch.PitchDetectionHandler;
import be.tarsos.dsp.pitch.PitchDetectionResult;
import be.tarsos.dsp.pitch.PitchProcessor;

import javax.sound.sampled.AudioFormat;
import javax.sound.sampled.*;
import javax.swing.*;
import javax.swing.SwingUtilities;
import javax.swing.filechooser.FileNameExtensionFilter;
import java.awt.*;

import java.awt.Container;
import java.awt.event.*;
import java.io.File;

import java.io.IOException;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Vector;

import be.tarsos.dsp.AudioDispatcher;
import be.tarsos.dsp.pitch.PitchProcessor.PitchEstimationAlgorithm;

import java.util.HashMap;


/**
 * Created by Alan on 11/7/2014.
 */
public class AlanUI  extends JPanel implements ActionListener,
PitchDetectionHandler {

    //references to the various Swing elements that are needed
    private JFileChooser chooser;
    private File abcFile;
    private JScoreComponent jscore;
    private JFrame frame;
    private JButton openFile;
    private JButton displaySong;
    private JPanel tempoPanel;
    private JFormattedTextField tempoField;
    private JFormattedTextField referenceButtonField;
    private JButton referenceButton;

    //number of the abc song that we are looking at in the file
```

```java
    private int referenceNumber;

    //more Swing elements
    private JButton goButton;
    private JButton playButton;
    private JButton restartButton;

    //iterator for the Music elements in the song
    private Iterator it;

    //the part of the song that we are looking to record
    private Voice voice;

    //current song
    private Tune tune;

    //reference to this object
    private AlanUI alan;

    //tempo at which to sing
    private int tempo;

    //Java Sound elements
    private AudioDispatcher dispatcher;
    private Mixer currentMixer;
    private TargetDataLine targetLine;

    private boolean playing;

    //which algorithm are we using?
    private PitchEstimationAlgorithm algo;

    //thread that listened to the user input
    private Thread audioThread;

    //current pitch
    private float pitch;

    //current accidental
    private Accidental accidental;

    //count of pitches for current note
    private HashMap<Integer, Integer> pitches = new HashMap<Integer,
Integer>();

    //set of constants that represent accidentals
    private HashSet<Integer> accidentalPitches = new HashSet<Integer>();

    //current keysignature we are using
    private KeySignature keySig;

    //thread for dynamically repainting the song
    private dynamicPaintTask paintTask;

    //constants for placing the buttons
    GridBagConstraints gbc;
```

```
//is song paused
private boolean paused;

public AlanUI () throws LineUnavailableException {
    super();
    alan = this;

    //add accidental pitches
    accidentalPitches.add(1);
    accidentalPitches.add(3);
    accidentalPitches.add(6);
    accidentalPitches.add(8);
    accidentalPitches.add(10);


    //initialize the Swing elements
    openFile = new JButton("Select abc file");
    openFile.addActionListener(this);
    JPanel openFilePanel = new JPanel();
    openFilePanel.add(openFile);

    JLabel refText = new JLabel("Reference number:");
    referenceButtonField = new JFormattedTextField(new Integer(1));
    referenceButtonField.setColumns(3);
    JPanel refPanel = new JPanel();
    referenceButton = new JButton("Set");
    referenceButton.addActionListener(this);
    refPanel.add(refText);
    refPanel.add(referenceButtonField);
    refPanel.add(referenceButton);
    referenceButton.setEnabled(false);


    displaySong = new JButton("Display song");
    displaySong.addActionListener(this);
    displaySong.setEnabled(false);
    JPanel displaySongPanel = new JPanel();
    displaySongPanel.add(displaySong);

    tempoField = new JFormattedTextField(new Integer(70));
    tempoField.setColumns(3);
    tempoPanel = new JPanel();
    JLabel text = new JLabel("Enter tempo:");
    tempoPanel.add(text);
    tempoPanel.add(tempoField);

    goButton = new JButton("Go!");
    goButton.addActionListener(this);
    goButton.setOpaque(true);
    goButton.setEnabled(false);
    JPanel goButtonPanel = new JPanel();
    goButtonPanel.add(goButton);

    setLayout(new GridBagLayout());

    gbc = new GridBagConstraints();
    gbc.gridx = 1;
```

```
gbc.gridy = 1;
gbc.weightx = 1;
gbc.anchor = GridBagConstraints.WEST;

super.add(openFilePanel, gbc);

gbc.gridy++;
super.add(refPanel, gbc);

gbc.gridy++;
super.add(displaySongPanel, gbc);

gbc.gridy++;
super.add(tempoPanel, gbc);

gbc.gridy++;
super.add(goButtonPanel, gbc);


openFile.setAlignmentX(Container.LEFT_ALIGNMENT);
refPanel.setAlignmentX(Container.LEFT_ALIGNMENT);
displaySong.setAlignmentX(Container.LEFT_ALIGNMENT);
tempoPanel.setAlignmentX(Container.LEFT_ALIGNMENT);
goButton.setAlignmentX(Container.LEFT_ALIGNMENT);


chooser = new JFileChooser();
jscore = new JScoreComponent();

//add all the elements to the jpanel
super.add(jscore);

//initialize java sound

float sampleRate = 44100;
int bufferSize = 2048;
int overlap = 1536;

final AudioFormat format = new AudioFormat(sampleRate, 16, 1, true,
        true);
final DataLine.Info dataLineInfo = new DataLine.Info(
        TargetDataLine.class, format);
TargetDataLine line;
line = (TargetDataLine) AudioSystem.getLine(dataLineInfo);
targetLine = line;
final int numberOfSamples = bufferSize;
line.open(format, numberOfSamples);
final AudioInputStream stream = new AudioInputStream(line);

JVMAudioInputStream audioStream = new JVMAudioInputStream(stream);
// create a new dispatcher
dispatcher = new AudioDispatcher(audioStream, bufferSize,
        overlap);

algo = PitchEstimationAlgorithm.YIN;
// add a processor
dispatcher.addAudioProcessor(new PitchProcessor(algo, sampleRate,
```

```
bufferSize, this));

        //start a thread to listen to user
        audioThread = new Thread(dispatcher, "audio dispatching");
        audioThread.start();

    }

    //listens to the user for designated milliseconds
    private void listenFor (long milliseconds) throws InterruptedException {
        targetLine.start();
        Thread.sleep(milliseconds);
        targetLine.stop();
        return;

    }

    //creates the jframe that holds all the elements and displays it
    private void createAndShowGui() {
        frame = new JFrame("MusAid");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(this);
        frame.pack();
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }

    //displays the selected song. Called when the display button is clicked
    private void showSong() {
        TuneBookParser parser = new TuneBookParser();
        TuneBook tb;
        try {
            tb = parser.parse(abcFile);
        }
        catch (IOException e) {
            return;
        }
        tune = tb.getTune(referenceNumber); //retrieve the tune to display
from the tunebook using its reference number.

        //gets all the properties of that song and sets it to proper
variables
        keySig = tune.getKey();
        jscore.setTune(tune);
        voice = tune.getMusic().getFirstVoice();
        it = voice.iterator();

        //enable go button
        goButton.setEnabled(true);

        //add pause & restart buttons
        addPauseRestartButtons();

        //repaint
        // super.add(jscore);
        super.repaint();
        frame.pack();
```

```
        frame.repaint();

        //unpause
        paused = false;
    }


    //does all the work. DoInBackground() is called whenever the user clicks
Go.
    class dynamicPaintTask extends
            SwingWorker {
        @Override
        public Object doInBackground() {

            playing = true;
            playButton.setEnabled(true);
            restartButton.setEnabled(true);

            //chord for the last note
            MultiNote lastNote = null;

            //index of last note
            int lastIndex = -1;

            //how long a quarter note shoud last
            double quarterLength = 60/(double) tempo;

            //loop for the notes in the song
            while(it.hasNext()) {
                //checks if the song is paused
                while (paused) {
                    try {
                        Thread.sleep(100);
                    }
                    catch (InterruptedException e) {};
                }

                //gets current note
                MusicElement currentScoreEl = (MusicElement) it.next();
                //if it's not a note, it's something we don't care about.
skip to next element
                if (!(currentScoreEl instanceof Note)){
                    continue;
                }

                //duration of the note
                long timeToRest = (long) (1000* quarterLength*((Note)
currentScoreEl).getDuration()/Note.QUARTER);
                if (currentScoreEl instanceof Note) {

                    //if it's a rest, wait for the duration and then go on to
the next note
                    if (((Note)currentScoreEl).getHeight() == Note.REST) {
                        try {
                            Thread.sleep(timeToRest);
                            continue;
                        }
```

```
                    catch (Exception e) {

                    }
                }

                //vector for the MultiNote. represents the feedback
                Vector vector = new Vector();

                //index of current note
                int index = voice.indexOf(currentScoreEl);

                //paint current note and last note
                Note note = null;
                try {
                    //paints current note yellow
                    note = (Note) currentScoreEl.clone();
                    note.setColor(Color.YELLOW);
                    voice.addElementAt(note, index);

                    //paints the feedback for the user. Green if correct,
red and black if incorrect
                    if (lastNote != null && lastIndex != -1) {
                        voice.addElementAt(lastNote, lastIndex);
                    }
                }
                catch (Exception e) {
                    e.printStackTrace();
                }

                //redraw
                jscore.setTune(tune);
                alan.add(jscore);
                alan.repaint();
                frame.pack();
                frame.repaint();

                //listen for next note
                try {
                    listenFor((int) timeToRest);
                }
                catch (Exception e) {
                    return null;
                }

                //find most frequently occurring pitch
                findMostFrequentPitch();

                //add original note to feedback
                vector.add(currentScoreEl);
                if (lastNote != null){
                    try {
                        if ((pitch > 60 && pitch < 1000)) {
                            //if user is correct, paint it green
                            if (noteMatches((Note) currentScoreEl)) {

((Note)currentScoreEl).setColor(Color.green);
                            }
```

```
                                //if user is incorrect, leave it black and
add in the user's pitch to the vector
                                else {
                                    Note note1 = new Note(Note.c);
                                    note.setDuration(((Note)
currentScoreEl).getDuration());

note1.setHeight(frequencyToPitch(pitch)[0]);
                                    note1.setAccidental(accidental);
                                    note1.setColor(Color.RED);
                                    vector.add(note1);
                                }

                            }
                            //reset pitch, accidentals, and pitch count
                            pitch = 0;
                            accidental = Accidental.NATURAL;
                            pitches = new HashMap<Integer, Integer>();
                        }
                        catch (Exception e) {
                            e.printStackTrace();
                        }
                    }
                    //create a multinote for the user feedback
                    MultiNote multi = new MultiNote(vector);

                    //save these for painting in the next iteration
                    lastNote =  multi;
                    lastIndex = index;
                }
            }
            playing = false;
            return null;
        }
    }

    //takes frequency in hertz, sets accidental to calculated accidental, and
returns a byte array in which
    //the first element is the pitch constant
    private byte[] frequencyToPitch(float frequency) {
        byte[] pitch = new byte[2];

        //convert to pitch constant
        double rPitch = 69 + 12 * Math.log(frequency/440.0)/Math.log(2);

        //convert to abc4j constants
        pitch[0] = (byte) Math.round(rPitch-60);

        //calculate accidental value
        if (accidentalPitches.contains(Integer.valueOf((int) pitch[0]) % 12))
{
            if (keySig.hasOnlySharps()) {
                pitch[0] = (byte) (pitch[0] - 1);
                accidental = Accidental.SHARP;
            }
            else {
                pitch[0] = (byte) (pitch[0]+1);
```

```
                accidental = Accidental.FLAT;
            }
        }
        return pitch;
    }

    //add pause and restart buttons
    private void addPauseRestartButtons() {
        gbc.gridy = 0;
        playButton = new JButton("Pause");
        playButton.addActionListener(this);
        playButton.setOpaque(true);
        playButton.setEnabled(false);

        restartButton = new JButton("Restart");
        restartButton.addActionListener(this);
        restartButton.setOpaque(true);
        restartButton.setEnabled(false);

        JPanel buttonsPanel = new JPanel();
        buttonsPanel.add(playButton);
        buttonsPanel.add(restartButton);

        super.add(buttonsPanel, gbc);

        buttonsPanel.setAlignmentX(Container.LEFT_ALIGNMENT);

    }

    //determine if the note given matches the calculated note
    private boolean noteMatches(Note note) {
        byte p = frequencyToPitch(pitch)[0];
        Accidental acc = keySig.getAccidentalFor(note.getStrictHeight());
        if (note.getHeight() == p && accidental ==
note.getAccidental(keySig)) {
            return true;
        }
        return false;

    }

    //action handler for the buttons
    public void  actionPerformed (ActionEvent e){
        Object obj = e.getSource();

        //unpauses
        if (paused) {
            if (obj == playButton) {
                paused = false;
                return;
            }
        }
        //pauses
        if (!paused) {
            if (obj == playButton) {
                paused = true;
            }
```

```
        }
        //restarts entire program
            if (obj == restartButton) {
                //remove current components
                Component[] cArray = super.getComponents();
                for (Component c : cArray) {
                    super.remove(c);
                }

                //cancel dynamic painting thread
                paintTask.cancel(true);

                //add completely new references
                openFile = new JButton("Select abc file");
                openFile.addActionListener(this);
                JPanel openFilePanel = new JPanel();
                openFilePanel.add(openFile);

                JLabel refText = new JLabel("Reference number:");
                referenceButtonField = new JFormattedTextField(new
Integer(1));
                referenceButtonField.setColumns(3);
                JPanel refPanel = new JPanel();
                referenceButton = new JButton("Set");
                referenceButton.addActionListener(this);
                refPanel.add(refText);
                refPanel.add(referenceButtonField);
                refPanel.add(referenceButton);
                referenceButton.setEnabled(false);


                displaySong = new JButton("Display song");
                displaySong.addActionListener(this);
                displaySong.setEnabled(false);
                JPanel displaySongPanel = new JPanel();
                displaySongPanel.add(displaySong);

                tempoField = new JFormattedTextField(new Integer(70));
                tempoField.setColumns(3);
                tempoPanel = new JPanel();
                JLabel text = new JLabel("Enter tempo:");
                tempoPanel.add(text);
                tempoPanel.add(tempoField);

                goButton = new JButton("Go!");
                goButton.addActionListener(this);
                goButton.setOpaque(true);
                goButton.setEnabled(false);
                JPanel goButtonPanel = new JPanel();
                goButtonPanel.add(goButton);

                setLayout(new GridBagLayout());

                gbc = new GridBagConstraints();
                gbc.gridx = 1;
                gbc.gridy = 1;
                gbc.weightx = 1;
```

```
        gbc.anchor = GridBagConstraints.WEST;

        super.add(openFilePanel, gbc);

        gbc.gridy++;
        super.add(refPanel, gbc);

        gbc.gridy++;
        super.add(displaySongPanel, gbc);

        gbc.gridy++;
        super.add(tempoPanel, gbc);

        gbc.gridy++;
        super.add(goButtonPanel, gbc);


        openFile.setAlignmentX(Container.LEFT_ALIGNMENT);
        refPanel.setAlignmentX(Container.LEFT_ALIGNMENT);
        displaySong.setAlignmentX(Container.LEFT_ALIGNMENT);
        tempoPanel.setAlignmentX(Container.LEFT_ALIGNMENT);
        goButton.setAlignmentX(Container.LEFT_ALIGNMENT);

        frame.setVisible(false);

        createAndShowGui();

        playing = false;
        paused=true;

    return;
}
//shows the file chooser
if (obj == openFile) {
    FileNameExtensionFilter filter = new FileNameExtensionFilter(
            "ABC files", "abc");
    chooser.setFileFilter(filter);
    int returnVal = chooser.showOpenDialog(this);

    if (returnVal == JFileChooser.APPROVE_OPTION) {
        abcFile = chooser.getSelectedFile();
        referenceButton.setEnabled(true);
        //This is where a real application would open the file.
    } else {
        return;
    }
}
//displays the song
else if (obj == displaySong) {
    this.showSong();
}
//executes the dynamic feedback
else if (obj == goButton) {
    tempo = (Integer) tempoField.getValue();
    paintTask = new dynamicPaintTask();
    paintTask.execute();
}
```

```
        //get the reference number
        else if (obj == referenceButton) {
            referenceNumber = (Integer) referenceButtonField.getValue();
            displaySong.setEnabled(true);
        }

    }

    //this is called whenever a pitch is detected by the yin algorithm
    public void handlePitch(PitchDetectionResult
pitchDetectionResult,AudioEvent audioEvent) {
        if(pitchDetectionResult.getPitch() != -1){
            double timeStamp = audioEvent.getTimeStamp();
            double p = pitchDetectionResult.getPitch();
            int pitch = (int) pitchDetectionResult.getPitch();

            //sets the count properly in the hashtable
            int count = 0;
            if (pitches.containsKey(pitch)) {
                count = pitches.get(pitch);
            }
            if (pitch > 50) {
                pitches.put(pitch, count + 1);
            }
            float probability = pitchDetectionResult.getProbability();
            double rms = audioEvent.getRMS() * 100;
        }
    }

    //calculate most frequently occuring pitch in hashtable
    private void findMostFrequentPitch() {
        int max = 0;

        for (Integer i : pitches.keySet()) {
            if (pitches.get(i) > max) {
                max = pitches.get(i);
                pitch = i;
            }
        }

    }

    //executes the program
    public static void main(String[] args){
        SwingUtilities.invokeLater(new Runnable() {
            public void run(){
                try {
                    new AlanUI().createAndShowGui();
                }
                catch(Exception e) {
                    e.printStackTrace();
                }
            }
        });
    }

}
```