A method for automatic sampling of a MIDI-controlled grand piano

JONAS MASKO

Bachelor Thesis
Stockholm, Sweden 2014

TRITA-ICT-EX-2015:71
A method for automatic sampling of a MIDI-controlled grand piano

JONAS MASKO

Examensarbete inom information- och programvarusystem, grundnivå (II121X)

Civilingenjörsutbildning i informationsteknik (CINTE, 300 hp)

KTH, Royal Institute of technology, year 2014

Supervisor Anders Friberg (at CSC)

Examiner Christian Schulte

TRITA-ICT-EX-2015:71

KTH, Royal Institute of Technology
School of Information & Communication Technology

KTH ICT
SE-100 44 Stockholm, Sweden

http://www.kth.se/ict
Sammanfattning

När man skapar musik digitalt så kan det vara önskvärt att kunna använda exakta återgivanden av riktiga instrument, och inte bara digitala simuleringar. För att kunna nyttja dessa återgivningar av instrument så måste äkta ljud spelas in från instrumentet i fråga. Detta tar väldigt lång tid och kan vara svårt att genomföra noggrant.

Denna rapport visar en Java lösning som automatisk kan spela in äkta ljud från ett instrument som kan ta emot och hantera digitala signaler från standarden för kommunikation mellan musikinstrument (MIDI) och presenterar ett exempel på hur Java kan användas som platform för MIDI och ljud.

För att kunna skapa denna lösning så samlades information från olika forum samt dokumentationer online och en lösning provades fram.

Resultatet är en demonstration av ett program som kan samla ljud från vilket instrument som helst som stödjer MIDI-inmatning, men är specifikt designat för att fungera med pianot. Programmet kan utvecklas vidare för att fungera med andra instrument med högre variation, exempelvis violiner, som skulle kunna ta emot mer MIDI-signaler än ett piano.
Abstract

When producing digital music it might be desirable to be able to use accurate representations of actual instruments and not just digital simulations. To acquire these accurate representations, real audio must be recorded from the instrument. These tasks can be very time consuming and difficult to properly control.

This report presents a solution to automate the recording of instruments which can receive and process the signals from the digital standard for musical instrument communication (MIDI) and provides an example of using Java as a platform when it comes to MIDI and audio.

To create the solution, information was gathered from various forums and message boards online along with official and unofficial documentations and put into use in an trial-and-error approach.

The outcome is information on how to use Java with MIDI and audio, a demonstration application which can collect samples from any instrument that supports MIDI, but is designed to work with the grand piano. This application can be developed further to support more advanced instruments, such as violins, which have a lot more variation than the grand piano and can use additional MIDI signals.
# Table of Contents

Chapter 1 - Introduction.................................................................................................................. 7  
  1.1 Background............................................................................................................................ 7  
  1.2 Problem statement.................................................................................................................. 8  
  1.3 Purpose.................................................................................................................................. 8  
  1.4 Goals...................................................................................................................................... 9  
  1.5 Method.................................................................................................................................. 10  
  1.6 Risks, Sustainability & Ethics................................................................................................ 11  
  1.7 Disposition............................................................................................................................. 11  

Chapter 2 - Background.................................................................................................................. 13  
  2.1 Introduction to digital music making..................................................................................... 13  
       2.1.1 Samples (music)............................................................................................................... 13  
       2.1.2 MIDI............................................................................................................................. 14  
  2.2 Kontakt.................................................................................................................................. 16  

Chapter 3 – Audio Programming.................................................................................................. 18  
  3.1 What is digital audio.............................................................................................................. 18  
  3.2 Recording Audio..................................................................................................................... 20  
  3.3 Processing Digital Audio....................................................................................................... 22  
       3.3.1 Converting channels....................................................................................................... 22  
       3.3.2 Silence Detection........................................................................................................... 23  
  3.4 MIDI programming.............................................................................................................. 27  
       3.4.1 MIDI signals.................................................................................................................. 27  
       3.4.2 MIDI and Java ............................................................................................................... 28  

Chapter 4 - Execution................................................................................................................... 29  
  4.1 Design.................................................................................................................................... 29  
       4.1.1 MIDI............................................................................................................................. 29  
       4.1.2 Recording....................................................................................................................... 31  
       4.1.3 Merging MIDI with audio............................................................................................. 32  
       4.1.4 Silence detection........................................................................................................... 34  
       4.1.5 Logic flow....................................................................................................................... 34  
       4.1.6 The channel error........................................................................................................... 35  

Chapter 5 - Results...................................................................................................................... 36  
  5.1 User Interface....................................................................................................................... 36  
       5.1.1 Output & input testing..................................................................................................... 36  
       5.1.2 Sample size and range.................................................................................................... 36  
  5.2 Back end................................................................................................................................ 38  
       5.2.1 MIDI Interface............................................................................................................... 38  
       5.2.2 Audio Processor............................................................................................................ 38  
  5.3 Goals breakdown.................................................................................................................. 38  

Chapter 6 - Evaluation................................................................................................................ 39  
  6.1 Discussion............................................................................................................................. 39  
       6.1.1 Silence detection............................................................................................................. 39  
       6.1.2 The application.............................................................................................................. 41  
       6.1.3 Ethics............................................................................................................................. 41  
  6.2 Further Development.......................................................................................................... 41  
  6.3 Conclusions.......................................................................................................................... 42  

Chapter 7 - Summary.................................................................................................................. 43  
  Summary..................................................................................................................................... 43  
  References................................................................................................................................... 43
Chapter 1

Introduction

1.1 Background

When creating music today, it can all happen in the computer - no real instruments are even needed! The computer generates sound using different Synthesizer applications, often containing a vast array of settings and options for how to digitally create the audio, as seen in fig.1.

However, it might desirable to use genuine sound from real instruments instead of digital simulations. Samplers are applications which use prerecorded audio from actual instruments, allowing the musician to use those in the computer. These prerecorded audio segments are called samples. The process to create these, known as sampling, is extremely time consuming and difficult. This report will look into how to automate this process using instruments which support the standard interface when communicating digitally between musical devices: the Musical Instrument Digital Interface (MIDI).

Figure 1: Micrologue synth for Cubase[10]. The author is the owner of this image.
1.2 Problem statement

1.2.1 Sampling issue
Collecting samples from an acoustic instrument is a very time consuming task, provided that it is done thorough. For example, sampling a piano can be very difficult to achieve by hand:

- The recording environment is ideally isolated in a studio which means that any sound produced by the human performer will also be recorded.
- When sampling different pressure, soft, medium, hard, etc... it is almost impossible to have an equal pressure on the keys, it will only be roughly the same level.
- Extremely time consuming for the human performer, and thus costly to realize.

Note that some instruments, such as electric guitars, have small built-in microphones which pick up the audio and do not need a studio to sample. Most instruments, however, require a studio to properly capture the audio.

1.2.2 Solution
Whatever instrument a person can play, there is probably MIDI output support for it in some way. Only grand pianos have a convenient support for MIDI as of February 2014. However, demos and experiments on other instruments can be found all over the world, ranging from advanced robots to simple amateur level hobby designs; a few examples would be guitars, violins and drums. [1][2][3]

A computer can send output to a MIDI-controlled instrument, in this case a grand piano, and at the same time record the sound it produces. The sound it records can then be edited in a way to match a targeted sampler software.

The target software for this project is Kontakt by Native Instruments. [4]

1.3 Purpose
The purpose of this project is to create a user friendly Java application to automatically sample a MIDI-controlled instrument.
This software is a request from a researcher at The Music Group at Department of Speech, KTH. His work at KTH started with a rule system for music performance run on a mini-computer and using a rule based programming language developed for speech synthesis. To test this computer generated music, a local grand piano is used. However, since it is not always possible to test towards this local grand piano (especially from contributors from other locations), he wants to accurately sample this grand piano and use those samples in the sampler software Kontakt – so that everyone has access to the exact sound the piano produces.

Creating this software will explore the concepts of automatically sampling any MIDI-controlled instrument. The aim of this thesis is to present my research and conclusions of automatic MIDI-controlled sampling, as well as provide additional tutorial resources when it comes to Java audio programming.

I am a musician myself and utilize MIDI to send data to samplers to create music. I am very familiar with this area and know all about the background and goals a musician would need from an application like this. This is why I wanted to carry out this project, and was chosen to do it.

1.4 Goals

The goal of this project is to design a user friendly Java application to automatically sample a MIDI-controlled instrument, and to write a report of how MIDI-controlled instruments automatically can be sampled and how to use Java when it comes to digital audio (which you are reading now).

The software also needs to be multi-platform to work on the 3 major operative systems of Windows, Mac and Linux. This is achieved by using Java as programming language.

The main goals of the software:

- Send MIDI output to a MIDI controlled acoustic instrument.
- Control the MIDI output; be able to specify key index and velocity.
- Record audio from a microphone and save this to a file.
- The recorded audio needs to have a format supported by Kontakt.
- Command-line test client. (irrelevant in case of GUI)
Additional goals of the software, provided there is sufficient time:

- GUI (replace the command line)
- Ability to specify detailed sequences of key indexes and velocity levels to be recorded.
- Measure the decibel of all key indexes and velocity levels and export as a spreadsheet.
- Detailed recording; only record certain notes at certain velocity levels, do not sample the entire instrument.

1.5 Method

This project was conducted alone in a time span of 10 weeks. The software was initially requested by the supervisor to be created in Pure Data[11]. However, due to the author's experience and knowledge of using Java[12] in audio context, Java was chosen as the platform for the application.

To address the important part of this topic (A method for automatic sampling a MIDI-controlled grand piano), knowledge of the area was collected before starting designing the software. There are many good resources to learn about MIDI and digital audio programming. The book Digital Audio with Java, written by Craig A. Lindley[14], can be a good source to read if the prior knowledge of audio and Java is low. As for MIDI, http://www.jsresources.org[17] has lots of example code to get started with MIDI programming in Java.

The design was approached using a trial-and-error method using tutorials found in http://www.jsresources.org[17]. This was due to the fact that a method for automatic sampling is the goal and no applications like this exists as of February 2014.

Audio and MIDI was written as two separate independent program parts, and was linked together using a separate controller part. The separation is due to multiple threads, which is unavoidable when connecting to multiple hardwares. It was quickly discovered that, in order to make the application record audio from the instrument without error, an algorithm which detects when the instrument has stopped producing sound has do be written. This algorithms had to work using only the audio data provided by the audio input.

The application was tested using a SE Electronics X1 microphone connected to an ART Tube MP preamplifier which was connected to a M-Audio Audiophile 2496 soundcard. The instrument used when testing was a Yamaha PSR-275 Keyboard using the default settings when powered on with maximum volume.
1.6 Risks, Sustainability & Ethics

The lack of information on this subject is a large risk for this project. No literature or articles discuss how to effectively automatically record output of a MIDI controlled instrument. There is also a lack of Java applications demonstrating MIDI and audio processing.

This effectively means that much trial-and-error is needed, making it hard to estimate the time for each goal. To account for that, the goals are split into a “main” and “additional” section to make sure a application can be built even if I do not have the time to implement desirable features.

This project provides a demonstration of using computer's to play MIDI instruments in a controlled way. The goal is not to play a MIDI sequence, the goal is to make sure that the MIDI instrument never overlaps any of its own audio. This can be used in more areas than just sampling a MIDI instrument. Since there is research in computer generated music, having the ability to let the computer know at what amplitude a MIDI instrument is providing audio might be relevant. [7]

The ethic issues revolves about music being an art and a very human thing. Replacing the human performer with an advanced computer might not be an idea which too many would agree with. There is already robot music performers out there for entertainment purposes[8]. While computers are not able to create music on their own, researchers are trying to create algorithms which a computer can use to create pop hits[9]. Should we discover an algorithm which is superior to human songwriters, it can mean that human music performers and songwriters will have a hard time to use their talents in a professional way, at least when it comes to pop music. This project is a part of this by adding further experiments and research into removing humans when using instruments.

1.7 Disposition

Chapter 2
This is information in how music is created digitally, and why samplers and samples are needed.

Chapter 3
The results from my trial-and-error method in creating the software will be presented here. Information in how to process audio and MIDI in Java, and technically in any language, can be found here.
Chapter 4
The design of the software will be presented here. This can be seen as putting the theory from chapter 4 into practical use.

Chapter 5
The results from making the software will be presented here.

Chapter 6
Discussing some improvements of the software along with the concept of automatically sampling MIDI-controlled instruments.
Chapter 2

Background

2.1 Introduction to digital music making

This will give an introduction how MIDI and digital music making works. This is explained from a hobby musician with 10 years of experience of writing digital music.

2.1.1 Samples (music)

A sample is a prerecorded sound that can be used as an instrument (or sound in general) in music production. Recording a few bangs on a kettle is as much of a set of samples as recording the entire audio supply from a violin. A common way of utilizing samples is to record some sound from an instrument and processing them digitally to emulate the actual instrument. A few companies, such as East West[13], record a complete set of samples from the acoustic instruments to the level where digital processing will not even be necessary. This also preserves the original acoustic sounds from the instruments, making it ideal when it is desired to compose acoustic music digitally. These libraries of sounds consists of hundreds of gigabytes of audio data.

Samples are usually raw sound data in one (or multiple) audio files. These data files are read by a device or an application called a Sampler. An example of a sampler, and the application used in this project, is Kontakt by Native Instruments[4]. The application uses audio files and maps these onto the virtual keyboard, see fig. 2. Kontakt can then digitally process these audio files in different ways.
2.1.2 MIDI

In order for samplers to receive data, an interface must be used. This interface, Musical Instrument Digital Interface, MIDI, specifies the standard communication between devices, usually a physical instrument or music editing program which provides MIDI output to a sampler software. The MIDI does not transfer any audio. Instead, it transfers instructions. There are a lot of MIDI instructions, and depending on the software (or MIDI receiver in general) it can interpret these differently. There are a few standardized instructions though, such as (translated to English) “play this note” and “stop playing that note”.

When creating digital music using samplers, you never work with the actual audio in your end; you work with the MIDI signals it sends to the samplers of your choice. The difference can be seen in fig. 3.
MIDI can also work the other way. Many instruments which support MIDI output also support MIDI input. This allows a digital software to send an instruction to a physical instrument what to play, and the other way around. This means that a computer can send instructions to a software on the computer as well as to a physical device.
2.2 Kontakt

Kontakt, a software made by Native Instruments[4], is a well known sampler. Unlike other well known samplers, such as East Wests Play software[13], it supports user created content. This effectively means that anyone can create a virtual instrument to use in Kontakt.

This makes it the ideal tool when sampling an instrument of your own. There are many in-depth tutorials for creating Kontakt instruments, but I will demonstrate the initial approach when importing the samples into it. By doing this, the file structure expected by Kontakt will be more clear. In fig.4, an instrument has been created and contain a wave file with samples recorded from a physical keyboard. The samples represents every key ranging from C2 to D#5 at a velocity of 100. The blue keys on the keyboard at the button are the keys which have a sample attached to it. At the moment, it ranges from C2 to G2, rather than C2 to D#5. This is because Kontakt interprets the audio as a single note, rather than a series of notes.

With the click of a few buttons, Kontakt can automatically, using silence detection, split the wave file into samples, as seen in fig.5.
These samples can the automatically be mapped to the keyboard. This can be done for different velocity levels of the samples, but each file must contain a series of notes with the same velocity value. This means that when making an own instrument in Kontakt, all samples can be recorded in succession and stored in a single file (per velocity level).

Figure 5: Auto slice feature of Kontakt, the blue lines indicate samples, on top: all the samples, on the bottom: zoomed in on the 3 leftmost samples. The author is the owner of this image.
Chapter 3

Audio Programming

This is based on my knowledge of audio. This is the same as the theory from Digital Audio with Java[14]

3.1 What is digital audio

Sound is waves traveling through the air by the change of air pressure at given points in time. These changes in the air can be measured. When recording audio into an analog device, such as a tape, the capture device constantly captures audio which is converted into electrical voltage. The changes in voltage can then be transferred onto the tape, continuously. These changes are stored in range from -1 to 1.

Digital recordings does the exact same thing, but the difference is that it is not recording continuously. Instead, it records at given points in time. The difference is very easy to distinguish in fig. 6. The reason why it is not continuous is simply because it is not possible, since capturing continuous audio digitally would require an infinite small timespan between each point in time which is measured.

![Figure 6: Difference in analog and digital audio. The author is the owner of this image.]

So how large is that timespan, known as sample rate? Well, it depends. Expressions such as 44.1kHz, 88kHz and 192kHz can often be seen in an audio context. What this means is how many samples (points) which are captured per second. Human hearing normally ranges between 20hz and 20kHz, meaning that a minimum sample rate would be 40kHz. Why double the sample rate is needed can be explained with Nyquist sampling theorem. Nyquist sampling theorem states that if we would sample a 20kHz sound using 20kHz, it would just make a continuous line, producing no sound. If we are able to capture half
that cycle, using twice the frequency as we would like to hear, we end up with twice the sample rate. [5]

The following series of images shows an audio file opened in Audacity[15]. When zooming in, the sample points can be seen.

![Figure 7: Left: actual zoom, Right: digitally enhanced sample points and a sample rate grid. The author is the owner of this image.](image)

Having a large sample rate improves the accuracy of the audio, but is not the only thing affecting the accuracy. Each sample also has a **bit size** (also known as **sample format**). Just as there is 8-bit graphics, 16-bit graphics and 32-bit graphics. There is 8-bit audio, 16-bit audio, 24-bit audio and 32-bit audio. Other bit formats exist as well, but these 4 are the standardized bit sizes. The audio is ranging from -1 to 1. The following table demonstrates the accuracy between the different bit formats, when storing the arbitrary chosen value of -0.4457848543.

<table>
<thead>
<tr>
<th></th>
<th>8-bit digital audio</th>
<th>16-bit digital audio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric Range (in bits)</td>
<td>0 to 255</td>
<td>-32'766 to 32'767 (signed)</td>
</tr>
<tr>
<td>Stored value</td>
<td>71</td>
<td>-14607 (signed)</td>
</tr>
<tr>
<td>Converted back to -1 to 1</td>
<td>-0.4453125</td>
<td>-0.445783868</td>
</tr>
</tbody>
</table>

The difference between 16-bit and real audio is so small we need 6 decimals to see it in
this case. This will give a quite an accurate representation of the real audio and the difference between this and 24-bit audio is not apparent. 8-bit might seem like a trivial difference as well, with the need of 4 decimals, but is clearly is not if you listen to a 16-bit audio file you convert down to 8-bit.

Another variable in the audio is the **channels**. While not affecting the quality of the audio, digital audio can also have different channels, usually 1 channel (mono) or 2 channel (stereo) is used. Channels can be seen as independent audio sources (having the exact same duration) saved into a single file. Media player applications can be expected to interpret a two channel audio file as two separate sources for audio for the left and right speaker, playing one channel in the right speaker, and one channel in the left. Having 4 channels is expected to be a surround, and data will be sent to speakers at the rear as well. It is possible to have any amount of channels, but only 1 channel (mono) and 2 channel (stereo) should be expected to be fully supported by any application or soundcard.

The size of the audio can be calculated by using the following formula:

\[
\text{Size (in bytes)} = \text{Sample Rate} \times \text{Number of Channels} \times \frac{\text{Sample Format}}{8} \times \text{Time (in seconds)}
\]

This means that 1 second of 44'1kHz, with 16-bit audio, in stereo format, will have a size of 172kb!

To summarize:

- Digital audio is stored by taking the current value of the oscillation of the sound. This is ranging from -1 to 1. This is done many times per second.
- The precision of this stored value depends on how many bits each sample are stored as. The more bits, the more precise the audio will be. This knows as **bit size** or **sample format**.
- The number of samples per second is known as **sample rate**. The minimum sample rate to cover standard human hearing is 44.1kHz.
- Digital audio can have a number of **channels**, usually 1 or 2 (representing mono and stereo audio).

### 3.2 Recording Audio

Now that we know how digital audio works, we can start working with those bits. To process digital audio, a soundcard is needed. These are built-in into modern motherboards. When using applications that require a lot of audio processing, such as
When creating digital music, a stand-alone powerful soundcard is needed. To simply record some audio from a microphone does not require heavy audio processing and can rely on the motherboard soundcard.

To process incoming recorded audio, the soundcard needs a short amount of time to process the audio information. This time span is known as **latency** and the amount of data a soundcard collects before processing it is called a **buffer**. Latency is never larger than ~10ms in professional contexts due to the use of powerful soundcards and processors. The soundcard used for testing, M-Audio Audiophile 2496, has between 3ms and 9.8ms latency, depending on the buffer size of the driver. This figure can be seen from the application for configuring the driver for the soundcard. However, latency be considerably larger in motherboard soundcards. The latency is determined by the soundcard’s **buffer size** combined with processing power.

A buffer is the time the soundcard is allowed to process the audio before releasing it to speakers or the recording software. Having a low buffer size will require a lot more processing power, as it more often needs to process small segments incoming or outgoing audio, while having a large buffer size will save processing power but increase the latency.

When recording from a soundcard, the recording software is reading the soundcard buffer each time the soundcard releases new audio data. When storing or sending this recorded data, it is important that the sample rate, sample format and channels match the format used in the file or other application. The released audio is returned as a byte array, and can be proceed before storing it.

A simple recorder which records audio data and saves this into a .wav-file works like this:

1. Create an empty file.
2. Write the .wav headers to it, specifying the following (details in appendix D):
   1. Sample rate
   2. Sample format
   3. Channels
3. Start streaming data from the soundcard buffer into the file.
Note that the specified sample rate, sample format and channels must match the soundcard for this simple audio recorder. If other audio formats than the one produced by the soundcard are desired, the software must process the audio before storing it in the file, or a 3rd party application must be used to convert the newly stored data after it has been saved. Also note that a wave file has more header data than the sample rate, sample format and channels. There are many online resources on how to build a wave file, but a Java example of this can be seen in appendix D.

### 3.3 Processing Digital Audio

#### 3.3.1 Converting channels

If the format provided by the soundcard is undesirable, the audio can be processed by manipulating the audio. Take a look at fig. 10; the soundcard provided a two channel recording, but the microphone was mono and provided only one channel, resulting in the right channel being completely silent. If we would stream this straight into a wav-file, we would get undesirable results.

If exporting the 16-bit 44.1kHz 2 channel audio file in fig.10 as raw data from audacity, and opening the file in a hex viewer as seen fig.11, a pattern emerges.

![Figure 8: Undesirable channel settings. The author is the owner of this image.](image)

![Figure 9: Raw data of the audio file from figure 6, seen in GHex[16]. The author is the owner of this image.](image)

The first 16 bits, the size of a 16-bit sample, seem to contain no data. The next 16 bits contain data, and the following contains nothing again. This is because the audio is stored in a cycle for each channel. It stores one sample per channel before moving to the next sample.
If we would like to make this into a one channel mono, the simplest approach would be to rewrite the file as a 1 channel audio file and only store the left channel data into it. Instead of saving the entire byte array to the file as a continuous stream, only the left channel data should be considered. Since the data is 16-bit, this means that 2 bytes are stored, and then the next 2 bytes discarded... this repeats until the current audio buffer is processed. Java code for this can be found in appendix A.

This technique is not suited if wanting to merge stereo files which have sound in both channels. The technique to merge stereo to mono is most commonly solved by averaging the left and right channel and saving this into the mono byte array.

### 3.3.2 Silence Detection

It is not always a set amount of seconds needed to record. What if there is need to record audio until it stops? This is the case when sampling instruments.

The key to inspecting when silence is reached is to inspect each *sample point* in the audio. When the amplitude of a series of sample point is 0 it means total silence. Taking some thousand sample points and measuring their average level will determine if silence is reached. Take at look at fig. 12

![Figure 10: Audio reaching silence. The author is the owner of this image.](image_url)

We can clearly see from fig.12 that the audio reach silence at ~4.7ms. If we inspect a set number of samples and find that the last 100 of them has an amplitude of 0, we know...
silence has been reached. However, this will only happen in an ideal situation. In reality, there is always a minor noise. The silence would look more like seen in fig. 13.

![Figure 11: Actual silence; small noise. The author is the owner of this image.](image)

To define the noise as silence, the noise must be measured. This is simply reading a large number of sample points when there is “silence” and calculating the level of the noise. When doing this, it is important to view all samples with regards to their absolute value (this is important when averaging oscillations, or a sine wave would have an average of 0). So what defines the level of the noise, it is average? I found that using the average level from a 10 second segment of noise made it almost impossible for the algorithm to detect silence over my chosen timespan of ~0.2 seconds. I increased the noise average by 10% to account for the minor disturbances in the noise and it worked perfectly. This percentage comes from the following measurements I conducted:

In Audacity, I recorded noise using 5 different noise levels (greatly increasing the input volume) from a SE Electronics X1 microphone connected to a ART Tube MP preamplifier which was connected to a M-Audio Audiophile 2496 soundcard. The noise levels averaged at ~0.01, ~0.05, ~0.10, ~0.20 & ~0.30 (having different input volumes). These recordings were ~10 seconds each. I measured the maximum value of each recording, and searched how many sample points were below different percentage of this maximum point. The louder sample points are the relevant ones since it is those which will create enough volume to prevent silence detection at the plain average. I tested on percentages 1%, 2%, 3%... 100%.
My measurements showed that no matter what the average noise level is, the results are the same:

<table>
<thead>
<tr>
<th>Percentage of max volume level</th>
<th>Percentage of sample points above the max volume level percentage.</th>
</tr>
</thead>
<tbody>
<tr>
<td>50%</td>
<td>~10%</td>
</tr>
<tr>
<td>80%</td>
<td>~1.5%</td>
</tr>
<tr>
<td>90%</td>
<td>~0.7%</td>
</tr>
<tr>
<td>95%</td>
<td>~0.5%</td>
</tr>
<tr>
<td>98%</td>
<td>~0.3%</td>
</tr>
</tbody>
</table>

The fact that 10% of the sample point in above half the volume clearly demonstrates that it is a few large volume spikes in the noise that increase the average noise level. Since I used 200ms as a time frame to detect silence, this would mean that about 9000 sample points will be used when using a sample rate of 44.1kHz. I checked the entire 10 second noise audio files with a buffer size of 9000 (which is 200ms), calculating the average noise of each segment, and then comparing this to the average of the entire 10 second audio file and got a result showing that about 10% of the segments would not pass a silence detection test using the average level, as seen in chart 1. This is also the same as the number of samples above 50% of the max volume level.

![Chart 1: Difference of average level of 200 segments compared to the average level for all of them, which is ~0.0165 for the tested audio file in this chart.](image)
In chart 1, we can see that the loudest segment has an average of ~0.0025, which is 15% of the average of ~0.0165 in chart 1. This means that an increase of 6.6% of the average noise level would result in silence for all segments of this noise audio file. I tested 6.6% increase in average on 10 other recordings, all with an average volume of ~0.05 and between 2 to 5 seconds, but two of these recordings still had a few segments not counted as silence. I decided to round up 6.6% to 10%, which was arbitrarily chosen, and made tests with another 10 recordings. This time, no segment ever came above the “silent” level.

The Java code for calculating the average amplitude of a buffer of 16-bit mono audio can be found in appendix C, but can be summarized to this:

1. Create a list of AudioFormat objects representing the formats you want to work with. Refer to the Java documentation on creating AudioFormats.

2. Test the formats, with the ones you would prefer first, towards a DataLine, see Appendix C.

3. With this supported format, when calculating the volume level, convert every pair of bits into a short, and then calculate the average of all these shorts. Also, make sure to only use the absolute value of each sample, or the average will be invalid, just as when averaging alternating currents and other oscillation-based data.

4. Lastly, divide these by the maximum value of a 16-bit number (these differ depending on if it is signed or unsigned!), and a value between 0 and 1 is returned.

The buffer size should not be too small, especially when calculating the noise at “silent” level. For the noise, I used 5 seconds of measure, which is a total buffer size of 220'500 at 16-bit 44.1kHz. The silence detection has a 0.2 time span, chosen arbitrary.
3.4 MIDI programming

3.4.1 MIDI signals

There are MIDI channel messages and MIDI system messages. System messages are sent globally and is not a concern when working with only one MIDI device. The channel messages are what this section focus on. These messages are sent as a segment of 2 or 3 bytes. A message, seen in hexadecimal, telling a MIDI device to start playing note #70 at 80 velocity looks like this:

0x91 0x46 0x50

The first 4 bits, 9, is telling the MIDI device what signal it is, in this case 9, note on.
The second 4 bits, 1, is the channel to send this message to. In simple systems, this is 1.
The other two bytes are the hexadecimal values of the note index and the velocity value.

A list of all possible MIDI messages can be found in many sources and is not included in this report.

3.4.2 MIDI and Java

Using this information, we can send MIDI instructions by sending bits to the device once the application can stream data towards it. In Java, or rather javax (Official Java extension libraries), there are some MIDI classes available. Let’s take a look at them:

MidiSystem class

This is a class which is used to list all the MIDI devices found by the computers soundcard. Both software and hardware MIDI devices will be listed. In the next chapter, in fig.14, there will be two MIDI devices listed by this class. Both are software MIDI devices, as the soundcard selected by the operative system for Java did not have any MIDI hardware communication possibilities. Appendix B has a method for listing all available MIDI devices from the MidiSystem class.

MidiDevice & Receiver classes

MidiDevice is the Java representation of a MIDI device. The Receiver class has a bit of a misleading name as the name suggests that the class receives data. Is is true that the Receiver does receive MIDI input from the MIDI device, but it can also be used to send MIDI data to the MIDI device. See appendix B for a method of opening a connection to a MIDI device.
**MidiMessage class and subclasses**

To send a message, either the MidiMessage or ShortMessage will do the job. The MidiMessage allows specification of the bytes sent, but has protected access and is only used by its subclasses and takes bytes as a parameter as described in the previous section 3.4.1. The ShortMessage is specified through parameters, using the ShortMessage fields as parameters, which represents each message type, followed by their respective values. An example of using ShortMessage can be seen in appendix B.

The other two other available MIDI messages are MetaMessage and SysexMessage. MetaMessage is not designed to send data to MIDI instruments, and contains data that should be read by a human, such as signatures, copyright info, etc... SysexMessage are the system messages which are broadcasted. Generally you don't need to worry about using these unless you really know you need to.
Chapter 4

Execution

To demonstrate how automatic sample collection on a MIDI-controlled grand piano could work, a demonstration application was written. This application is explained in this chapter.

4.1 Design

Using the gained information on MIDI and audio, an application can be built. This section will describe my approach and findings when designing the application.

4.1.1 MIDI

This is a good end to start in since the MIDI-controlled instrument will produce the audio to work with in latter stages. After completing tutorials on MIDI[17], a simple test application for MIDI can be created. Note that even if the computer is having a soundcard with no MIDI connections, it can be expected to exist at least one software synthesizer on the computer the soundcard can detect. These are often named “General MIDI” or similar. At this stage, it is a good idea to write into the code which of the MIDI devices to use, after initially listing them once, as no user interface is built.

When having a connection to a MIDI device, a MIDI message can be sent to it, as described in previous section 3.4.2. This is where to test what different MIDI parameters will instruct the instrument to do. Since the target instrument is a MIDI-controlled grand piano, the only MIDI signals of interest are velocity and note index. Always read the manual for the target instrument before sending any MIDI instructions. Even if MIDI is describing the standard for musical instrument communication, there is no guarantee that the instrument will respond as expected. For instance, the target grand piano at KTH was not able to play a note with a velocity value over ~110 as this could damage the instrument, as instructed by the supervisor.

When verifying what MIDI messages the instrument would receive, a list of messages is created. Depending on the desired accuracy of the samples put together, different note intervals and velocity values are used. At this stage it might be wise to put some test cases into the list and run then with a few seconds delay between each MIDI message.
If happy with the results, a list describing the desired output should be created. When creating this list, be aware that it should be user configurable. The application should have its MIDI sequence construction function based on parameters. The logic flow of the application will at this point look like this:

Flowchart 1: MIDI logic.

### 4.1.2 Recording

With the MIDI section up and running, it is now possible to record the audio produced by the instrument. When creating this, the MIDI-controlled instrument was played manually. The accuracy of the samples is not relevant so there is no need for computer assistance from the MIDI part at this point, but it is possible to test this along with the MIDI part of the application by starting them as two separate instances.

Setup a basic test to check that the audio input works and save this into a file. In order to record from the audio input, an audio format must be specified. You can expect any sound card to handle the standard 44.1kHz 16-bit stereo (2 channel) format. If the format is invalid Java will throw an exception. To be safe, create a list of all possible audio formats you wish to use (manually) and attempt to create a connection to the
audio data line using these formats, testing one at the time. This will make sure there is no formatting problem when recording. For this design, all usable variations of 44.1kHz was added to an array which the initialization function used when creating the data line. Java's try-catch brackets in combination with a loop is useful when attempting to find a working format for the data line.

When the format is found, data can easily be streamed into a file. While at it, add wave headers to the file before streaming data to it (as this will make it easier to open it in an external audio editor). To emulate recording a sequence of sounds, record for a set amount of time, and then create a new file to stream into. Repeat this a few times to make sure it works.

The logic flow for this looks like this:

*Flowchart 2: Audio Recording Logic.*
4.1.3 Merging MIDI with audio

With a working MIDI part and a working audio recording part, it is now the time to put them together. The trick is to synchronize the creation of a new file with sending the next MIDI message. Since the target application for this audio recording is Kontakt, it should have all notes per velocity level recorded in series. This way there is no need to make sure the recording starts the moment the instrument is played. Should it be needed, timing cannot be relied on due to possible delay between the hardware when receiving instructions. Consider adding silence detection if that is the case, to crop away the initial silence in the recording.

To test this, a sequence of 4 notes with 4 velocity levels was used. The audio part of the application needs a clock which signals the time for the next MIDI note. The MIDI part needs a way of signaling that a sequence is finished and that is now starting playing notes over again with a new velocity value. To make things easier, a controller class could be written to handle these signals the MIDI and audio part wants to send one another. The controller can also function as the object whom receives instructions from the user interface. This design decision is based on experience and is highly recommenced. It is possible to make the MIDI and audio part communicate without a controller part, but this can make the application quite complicated when adding user input.

A class for a MIDI sequence is also created, and the list of MIDI messages are now a list of MIDI sequences. The MIDI sequences reflect each series of notes that should be recorded before storing the audio in a new file. On the following page, flowchart 3, the flow chart for the merged MIDI and audio parts can be seen.
The “initialize MIDI” and “initialize audio” reflects the initialization phase of the earlier flowcharts 1 and 2. Note that the application is now running it's user created threads; the MIDI, the audio recording, the controller and the 5 second timer. Note that the logical question “Sequence finished?” must return “yes” initially, since there is no sequence currently running.

4.1.4 Silence detection

The application logic is now complete. The only issue is the 5 second timer. As previously mentioned, it is impossible to know the duration of each note played. Having a large timespan is also not a viable solution. The final step is to replace this 5 second timer with a silence detector. This silence detector has been described already. The silence detector measurements from 3.3.2 might not work on all hardware, so it is highly recommended to make measurements of your own. It might be a wise idea to have a few seconds extra when silent level is detected...
to prevent overlapping. This project aimed to minimize the file size along with running no risk of the instrument playing notes too fast, overlapping the audio with each other – and thus a lot of effort went into the silence detection algorithm.

Be aware that the instrument might need a short amount of time before producing any audio. The silence detection must either ignore a few seconds after notifying silence, or measure a long segment for silence. The way to deal with this is to have a set of short segment to measure. 9000 sample points were chosen per segment, which reflects ~0.2s in a 44.1kHz audio stream. This short timespan is likely to run into the problem of delayed audio. When silence is detected, the buffer stops streaming data into the file for 4 segments, storing the segments in memory temporarily. Should the all the following segments contain silence as well, silence is reached. Should any of the following segments not count as silent, the temporary buffer is streamed into the file and the segments reset. This prevents unnecessary writing of silence into the file while at the same time accounting for possible hardware delays.

### 4.1.5 Logic flow

When everything is working as expected, a user interface must be created. This user interface must allow the user to specify what notes to record and at what velocity. The user must also be able to choose a MIDI device as well as conduct testing on each device. The final application logic can be seen in the following page.
4.1.6 The channel error

As previously mentioned, it might be an error in the channel settings when recording, depending on the hardware. To account for this, make sure to run the channel correction algorithm from appendix A.
Chapter 5

Result

5.1 User Interface

![User Interface Image](image)

*Figure 12: User interface of the application. The author is the owner of this image.*

5.1.1 Output & input testing

Due to the behavior of different operative systems and audio devices - selecting the recording input line and soundcard is done by the operative system, which is provided to Java. To make sure the desired soundcard is selected, the application can make some basic MIDI & audio testing to compensate for this. Should an input or output not work, this can be corrected in a way suitable from the operative system. The application is designed to lock all other features until it can confirm it has MIDI output (which is checked each time a MIDI device is selected), to prevent confusion.

5.1.2 Sample size and range

Selecting what range to capture is an obvious setting to account for. The MIDI instrument most likely has a limited range, and playing notes which are out of range will not produce any sound and ends up bloating the recorded audio file with silence.
Compressing the samples might also be of interest. Recording every note of every velocity will produce 15'360 samples! Estimating every sample to be ~5 seconds, and providing that the audio settings are 44’100 Hz, stereo 16 bit, the files produced would be **around 13 gB**! Of course, only around 5-8 velocity levels are usually required to get an accurate representation of the instrument. Capturing every note might also be overdoing it, capturing every 3 or 4 notes might suffice in most cases. It all depends on how accurately the instrument should be sampled and if file sizes are an issue. Sampling the entire note range by these settings: 6 velocity levels, every 4th note, 44’100 Hz, stereo 16 bit, would result in a file size of around 28 MB (or 0.028 gB, around 2’000 times smaller than if capturing every note at every velocity)

The length of the sample might vary. A low note with a low velocity produces a very quiet tone which quickly fades – while a note close to the middle with the highest velocity is very loud and long. The automatic silence detection is the ideal tool to use for this. To prevent the silence detection to ignore the really quite notes (as they produce close to no extra dB, which is what is measured by the application), and to prevent the other way around – that the application records too much low audio unnecessarily – limits can be enforced on the recording time.
5.2 Back end

5.2.1 MIDI Interface

The MIDI interface is what sends the output to the MIDI device. The interface never reads any MIDI input, as it will only care about MIDI output during the automated process. The only applied MIDI signals are the velocity and key index.

It grabs hold of a MIDI device and makes sure to notify if a problem should arise. It can only detect MIDI devices which is located on the selected soundcard on the operative system.

Basics of this interface can be found in appendix B.

5.2.2 Audio Processor

The major feature of this is the silence detection. The silence detection is built in in the audio capturing and it constantly measures the dB of a sound buffer. The sound buffer size is determined by the soundcard. The theory concerning silence detection has been explained in Chapter 3. The algorithm for determining the volume level is included in appendix C.
In order to produce files supported by the target application, Kontakt, the audio should be saved as a continuous file with all notes in ascending order. Each velocity level should have its own series of notes.

Should the audio be in stereo format, it will be converted into mono by the procedures described earlier and can be found in appendix A.

### 5.3 Goals breakdown

The main goals of the software:

- **Send MIDI output to a MIDI controlled acoustic instrument.**
  - Examples of this can be seen in appendix B.
- **Control the MIDI output; be able to specify key index and velocity.**
  - The application sends a series of notes, one at the time, to a MIDI device. This is to match the file structure expected by Kontakt.
- **Record audio from a microphone and save this to a file.**
  - It does, and how to save a wave file using an array of bytes can be seen in appendix D.
- **The recorded audio needs to have a format supported by Kontakt.**
  - As previously mentioned, it works.
- **Command-line test client. (irrelevant in case of GUI)**
  - This is deprecated and no longer in use (see GUI)

Additional goals of the software, if time is sufficient:

- **GUI (replace the command line)**
  - As seen in figure 14 in 4.1.
- **Ability to specify detailed sequences of key indexes and velocity levels to be recorded.**
  - Partially; the sequence can be set to sample every "n" keys, but has no "detailed" way of specifying the sequences.
- **Measure the DB of all key indexes and velocity levels as a spreadsheet**
  - Not implemented.
- **Detailed recording; only record certain notes at certain velocity levels, do not sample the entire instrument.**
  - Not implemented.
Chapter 6

Evaluation

6.1 Discussion

6.1.1 Silence detection

The silence detection could have been achieved in many ways. Using the average might seem like a bad idea when about 10% of the samples are loud spikes. Using the median value would easily eliminate any spikes and give a more accurate representation of the level of the noise. However, the median will still detect silence if less than half the segment measured are actual audio, as seen in fig.16. The median could be used to accurately measure complete silence, since it don’t care about larger volume spikes – and then the average could be used to “listen” on the actual recording. But mixing two algorithms is not a good idea in this case as they produce different levels and descriptions of the “silent” level, making it impossible for them to cooperate.

Another method for the silence detection could have been using the max value of the “silence”. This was my initial approach and it worked without issues. However, during one of the test runs of the application, the “max” level was set to an unreasonable volume level. It could have been caused by the hardware, or something in the room producing a relative loud sound. If, by any reason, the volume would suddenly make a large increase above the noise level, it would give an inaccurate description of the max value. The average will not be particularly affected by this short huge increase when measuring a large time span. The minor increment of the average and the potential loss of the ~0.2ms of close to inaudible audio it would result in from recordings can be accepted.

Measuring the “maximum” on a large set of segments, and then using the median on those, could work. I did not consider this approach until after the functioning algorithm based on the average was finished, so I choose not to try to implement it. It is important to note that the silence detection and the noise calculation must use the same algorithm, whatever the choice of algorithm is.
6.1.2 The application

I should have made this into a library from the start, rather than a packaged Java executable. The demonstration application could use those libraries as 3rd party additions. Since my goal was to make it possible to sample the targeted grand piano at KTH department of Speech, music and hearing, I was focused on creating a deliverable rather than an opportunity for the public to use the different parts of the applications individually.

6.1.3 Ethics

As mentioned in chapter 1.6, there are some moral issues when automating music and removing the human element from it. However, sampling is not playing an instrument, or creating music. It is simply a way to collect the audio supply, allowing musicals to use their favorite samplers to create master pieces of their own. Without samples, all acoustic sound have to be recorded in a studio or similar, for each piece of music, every time. When recording an instrument for the target piece of music, a human playing adds the “human touch”, which cannot be described to a computer. However, this “human touch” can also be created by a human using digital manipulation of the original audio, the samples. Creating music using samples does not remove the “human touch”. Essentially, playing an instrument and fiddling with MIDI settings are just two different tools towards the same goal. The choice of tools is individual – but having an entire symphony orchestra available in the hard drive on a computer, with possibility of controlling every aspect of the sound, is an opportunity few musicians would refuse.

6.2 Further Development

To expand the usage of the application from strictly a grand piano to other instruments, more MIDI parameters must be supported. Virtual instruments usually take a lot more input than just the velocity value and key index, those two parameters are just the tip of an iceberg. To enable full control of all MIDI instruments, all available MIDI parameters should be accounted for.

MIDI parameter usage depends on how the MIDI is interpreted by the different instruments, which is unknown to me at this moment. Before adding MIDI parameters, research should be conducted on the different MIDI-controlled instruments to know the purpose of each MIDI parameter. The best way is to allow free control of the MIDI parameters, allowing the user to specify whatever intervals and parameters should be used on what notes. This removes the simple approach in my current design, but it is not impossible to make this application user friendly for those not familiar with programming and computers in general and just want to sample an instrument.
6.3 Conclusions

Creating an application which automatically samples a MIDI-supported instrument is possible to achieve in a platform supported by most common operative systems, as the demo application shows.

The prerequisites to automatically record an instrument is

• The instrument must support MIDI input
• The computers soundcard must support MIDI output
• A microphone must be connected to the computers soundcard.

Depending on the microphone and soundcard, different audio settings are used. This might also cause the recording to incorrectly record only one channel of a stereo track. When developing applications to record audio, always test it on many different settings to make sure it is compatible with most soundcards.

It is impossible to predict the length of each individual sound from the instrument, to account for this, the application must have some kind of silence detection.
Chapter 7

Summary

This thesis have investigated the potential of automatic recording of instruments which support MIDI. An application which demonstrates collection of samples from a MIDI controlled grand piano has been constructed. The application saves all sound in a file structure supported by the sampler Kontakt. The application can be developed into supporting more instruments that grand pianos and target different file structures than those supported by Kontakt. Information on how to use Java with MIDI and audio has been provided, as well as how digital audio works. This thesis should provide sufficient information to continue/start development of automatic MIDI instrument sample collectors, in Java or any other programming language.

References


http://docs.oracle.com/javase/7/docs/api/javax/sound/midi [June 20, 2014]


[10] Official website for Steinberg, creator of Cubase, a program to write digital music in.
[11] Official website for Pure Data, a visual programming language
https://puredata.info/ [Jun 30, 2015]

[12] Official website for Java, a widely used programming language

[13] Official website for East West, a professional sample supplier


[16] Official website for Ghex, a hex viewer for linux https://wiki.gnome.org/Apps/Ghex
[Jun 30, 2015]

[17] Tutorials and examples of using Java in MIDI and audio context
http://www.jsresources.org [Jan 10, 2015]
Appendix A

Stereo to mono conversion in Java, using the left channel

```java
private static byte[] convert16BitStereoLeftChannelTo16BitMono(byte[] audioData) {
    short[] stereo16Bit = byteArrayToShortArray(audioData);
    short[] mono16Bit = new short[stereo16Bit.length / 2];
    for (int i = 0; i < mono16Bit.length; i++) {
        mono16Bit[i] = stereo16Bit[i * 2];
    }
    return shortArrayToByteArray(mono16Bit);
}

public static short[] byteArrayToShortArray(byte[] data) {
    short[] stereo16Bit = new short[data.length / 2];
    for (int i = 0; i < stereo16Bit.length; i++) {
        ByteBuffer bb = ByteBuffer.allocate(2);
        bb.order(ByteOrder.BIG_ENDIAN);
        bb.put(data[i * 2]);
        bb.put(data[i * 2 + 1]);
        stereo16Bit[i] = bb.getShort(0);
    }
    return stereo16Bit;
}

public static byte[] shortArrayToByteArray(short[] data) {
    byte[] stereo8Bit = new byte[data.length * 2];
    for (int i = 0; i < data.length; i++) {
        stereo8Bit[i * 2 + 1] = (byte) (data[i] & 0xff);
        stereo8Bit[i * 2] = (byte) ((data[i] >>> 8) & 0xff);
    }
    return stereo8Bit;
}
```
Appendix B

Basic MIDI setup and usage in Java

```java
private MidiDevice[] getMidiDevices() {
    LinkedList<MidiDevice> deviceList = new LinkedList<MidiDevice>();

    MidiDevice.Info[] midiInfo = MidiSystem.getMidiDeviceInfo();
    for (int i = 0; i < midiInfo.length; i++) {
        try {
            MidiDevice d = MidiSystem.getMidiDevice(midiInfo[i]);
            if (d.getMaxReceivers() != 0) {
                deviceList.add(d);
            }
        } catch (MidiUnavailableException e) {
            //Device is unavailable, ignore it or handle it
        }
    }
    return deviceList.toArray(new MidiDevice[0]);
}

public void setMidiOutDevice(MidiDevice d){
    activeDevice = d;
    try {
        activeReceiver = activeDevice.getReceiver(); //This handles MIDI messages
        activeDevice.open();
    } catch (MidiUnavailableException e) {
        //Handle Error
    }
}

public void playMidiNote(int i, int velocity) {
    if (activeDevice == null)
        return; //Prevent exception if no device is active
    if (velocity < 1)
        velocity = 1; //Prevent exception of invalid velocity data
    if (velocity > 127)
        velocity = 127; //Prevent exception of invalid velocity data
    try {
        ShortMessage playMessage = new ShortMessage();
        playMessage.setMessage(ShortMessage.NOTE_ON, 0, i, velocity);
        activeReceiver.send(playMessage, -1);
    } catch (InvalidMidiDataException ex) {
        //Invalid MIDI Message
    }
}
```
Appendix C

Measure the average volume level of an audio buffer

//Will range from -1 to 1, where 0 is silent
public static double getVolumeLevelFrom16BitSegment(byte[] buffer, AudioFormat supportedFormat)
{
    boolean bigEndian = supportedFormat.isBigEndian();
    boolean signed = (supportedFormat.getEncoding() == AudioFormat.Encoding.PCM_SIGNED);
    if (supportedFormat.getSampleSizeInBits() == 16) {
        double average = 0;
        for (int i = 0; i < buffer.length; i += 2) {
            short level = 0;
            byte hiByte = (bigEndian ? buffer[i] : buffer[i + 1]);
            byte loByte = (bigEndian ? buffer[i + 1] : buffer[i]);
            if (signed) {
                short shortVal = (short) hiByte;
                shortVal = (short) ((shortVal << 8) | (byte) loByte&0xFF);
                level = shortVal;
            }
            else {
                level = (short) ((hiByte << 8) | loByte&0xFF);
            }
            if (level < 0){
                level *= -1;
            }
            average += level;
        }
        return ( (average / (buffer.length / 2)) ) / (signed ? Short.MAX_VALUE : 0xffff);
    }
    return 0; //not 16 bit, ignore
}

public static AudioFormat getBestSupportedFormat() {
    for (int i = 0; i < AUDIO_FORMATS.length; i++) {
        TargetDataLine line;
        DataLine.Info info = new DataLine.Info(TargetDataLine.class, AUDIO_FORMATS[i]);
        if (!AudioSystem.isLineSupported(info)) {
            continue; //Not working, try the next one...
        }
        return AUDIO_FORMATS[i];
    }
    return null;
}
Appendix D

Saving a wave file in Java.

```java
public static boolean save(File f, byte[] data, AudioFormat format) {
    DataOutputStream outFile = new DataOutputStream(new FileOutputStream(f));
    int subChunk2Size = data.length * format.getChannels() * format.getSampleSizeInBits() / 8;

    // write the wav file per the wav file format
    outFile.writeBytes("RIFF"); // 00 - RIFF
    outFile.write(intToByteArray(36 + subChunk2Size), 0, 4); // 04 - how big is the rest of this file?
    outFile.writeBytes("WAVE"); // 08 - WAVE
    outFile.writeBytes("fmt "); // 12 - fmt
    outFile.write(intToByteArray(16), 0, 4); // 16 - size of this chunk
    outFile.write(shortToByteArray((short) 1), 0, 2); // 20 - what is the audio format? 1 for PCM
    outFile.write(shortToByteArray((short) format.getChannels()), 0, 2); // 22 - channel count
    outFile.write(intToByteArray((int) format.getSampleRate()), 0, 4); // 24 - samples per second
    outFile.write(intToByteArray((int) (format.getSampleRate() * format.getChannels() * format.getSampleSizeInBits() / 8)), 0, 4); // 28 - bytes per second
    outFile.write(shortToByteArray((short) format.getChannels() * format.getSampleSizeInBits() / 8), 0, 2); // 32 - # bytes in one sample, for all channels
    outFile.write(shortToByteArray((short) format.getSampleSizeInBits()), 0, 2); // 34 - how many bits in a sample(number)? usually 16 or 24
    outFile.writeBytes("data"); // 36 - data
    outFile.write(intToByteArray(subChunk2Size), 0, 4); // 40 - how big is this data chunk
    outFile.write(data); // 44 - the actual data itself - just a long string of numbers
    outFile.close();
}
```

```java
return true;
```