Bachelor Informatica

UvA Universiteit van Amsterdam

Real-Time Score Following on a Mobile Device

Elte Hupkes

June 11, 2012

Supervisor(s): R.G. Belleman

Signed:

Abstract

Playing music from a paper score or sheet can be a cumbersome process, as one needs his or her hands both for playing the instrument and flipping pages. Tablet devices present a platform well suited for displaying musical notation, and have become increasingly popular over the past few years. This thesis proposes a prototype "score following" application that runs on such a device. Feature vectors constructed from audio spectograms are modelled using a Hidden Markov Model to find the most likely position in audio reference data. A training mode is used to link this reference data to a visual notation, thereby removing the need for structural information and leveraging the wide availability of music sheets and tablature online.

Contents

CHAPTER 1

Introduction

Playing music from sheets or tablature has always held a classic problem for artists: there is only so much information that can fit on a single sheet of paper. Displaying music on a digital screen or tablet as of yet provides no solution to this problem, as screen "real estate" on such devices is limited, and more often than not the musician's interaction to scroll or flip the page is still required when his or her hands are already otherwise engaged. Applications that scroll the screen content in time do exist, but this solution is insufficient in many situations as it cannot deal with tempo changes or even going back to a previous part that is repeated. Professionally, often an extra person is employed alongside the performer solely to flip pages at the desired times.

1.1 Research goal and requirements

This thesis aims to find a solution to the described problem by implementing an application that automatically scrolls a visual representation of a performance through time, while "listening" to this performance as it is played in real-time. This type of application is ideal to run on a tablet device, which are becoming more and more widely used.

Developing this type of application involves the development of an algorithm that matches performance audio to known reference data. Despite rapid progress in the field of mobile computing, tablet devices are still limited in terms of memory availability and processing power. The used algorithm will thus have to be low on processor utilization as well as having a low memory footprint. Depending on the device, audio hardware may also be lacking. Matching will have to be robust enough to deal with low quality and potentially noisy signals. The following questions are therefore addressed in this thesis:

- Is it possible to devise an algorithm that can match real-time recorded audio data to reference data?
- Can this algorithm be made robust? Robust meaning tolerant to performer mistakes and low quality audio input.
- Can it be made fast enough to deal with the limitations of mobile device hardware?

Existing score following applications often rely on the availability of symbolic music files that contain information about the aural as well as the visual representation of a work. There is however a much larger set of music tablature [1], scores and sheets available in formats that do not have this information, such as text and images. [2] Being able to use these files would greatly enhance the accessibility of the application.

1.2 Outline

This thesis presents the details of a prototype tablet application that performs the described task. Chapter 2 shows an overview of the existing research in this field and its use to this application, after which chapter 3 discusses the methods employed by this thesis. Details about the implementation of these methods are presented in chapter 4. In chapter 5 the performance of this implementation in both accuracy and speed is measured. Conclusions about this data are drawn in chapter 6. Finally, chapter 7 outlines improvements and recommendations for future work.

1.3 Basic notions

Throughout this thesis several terms relating to the field of music theory are used. An excellent overview of these terms is given in [15], this section provides a summary of what is presented there with some additions.

Pitch A property of human auditory perception that determines how “high” a tone sounds, and thus allows for an ordering of sound.

Chroma Human pitch perception is periodic, resulting in multiple frequencies being perceived as having the same “color”, or *chroma*.

Octave The distance between two tones with the same chroma is always a whole number of octaves. The difference between two neighbouring octaves is always double (or halve) the frequency.

Pitch class Two tones with the same chroma are said to be in the same pitch class.

Tuning system An octave can be arbitrarily divided into a fixed number of pitch classes, this division is called a tuning system. The most commonly used tuning system, and the one assumed in this thesis, is *twelve-tone equal temperament* (12-TET). This system divides each octave into twelve pitch classes with an equal ratio between the frequencies of consecutive tones (or an equal distance on a logarithmic scale). Tuning requires a reference or base tone, usually *A 440* with a 440Hz frequency is used.

Timbre The “tone color” or “quality” of a sound played by a specific instrument. When a tone is played on an instrument it commonly produces several frequencies in the same pitch class alongside the main frequency. Which frequencies are emitted and with what intensity largely determines the timbre of the instrument.

[1] A musical notation format indicating instrument fingering, often used for guitar music.

[2] A well-known website for guitar tablature, `http://www.911tabs.com`, contains 4,413,579 guitar “tabs” at the time of this writing. Over a thousand new tabs are added every day.

CHAPTER 2

Related work

The field of music recognition holds many contributions applicable to this research. Areas of interest include feature extraction and comparison. Various sources describe methods for song recognition through music fingerprinting [12, 27]. Although not directly applicable to estimating position *within* a song, they do identify characteristics of audio signals that can be used as such. These methods tend to settle on using information from audio spectograms over other techniques such as Mel Frequency Cepstral Coefficients (MFFC) (prominently used in speech recognition) [19], spectral flatness / sharpness [6] and others, as noted by Haitsma [12]. He also notes the frequencies relevant to the human auditory system.

The topics of particular interest to this research fall in the field of audio alignment, a problem which has two sides: "off-line" and "on-line" matching [8]. Off-line matching is the process of aligning complete performances, whereas on-line matching deals with comparing real-time input to reference information. In the former field attempts are made to align score representations of music with existing performances [7], a process called *score performance matching* or *score alignment* [4]. The latter applies more directly to this thesis and is commonly denoted *score following*. Generally these fields differ in their approach to the one taken in this thesis by relying on the availability of symbolic reference information (such as MIDI[1]-files).

2.1 Feature extraction

Although score following does not provide the flexibility of being able to look into the future, similar matching techniques can be used. As a feature vector, a *chroma* based representation of windows of audio data is proposed [1, 7, 14]. More specifically a twelve-item *chroma vector* is used, concentrating the spectral information of each pitch class, thereby eliminating specific sound features such as timbre. The implementation details of this feature vector vary, with the original implementation using logarithmic magnitudes [1] whereas later implementations use linear values [7]. Other research utilizes frequency and energy from the input signal [4].

Several sources use structural information provided by the symbolic representation of a reference file in their feature vectors [4, 9, 13, 21]. This limits the possible uses to works for which a symbolic representation is available, making it less useful for this thesis.

[1]A symbolic audio specification [22].

2.2 Audio matching

Existing sources present several matching and alignment techniques. A simple example is the "strict matcher" which forces note order and is thus not very tolerant to mistakes [9]. Alignment methods attempt to find maximum likelihood paths through similarity matrices [2], usually employing a Discrete Time Warp (DTW) algorithm [7, 14]. For on-line matching, Hidden Markov Models (HMMs) are proposed [4, 21]. These provide more flexible training options, although as Dannenberg and Hu [7] point out discrete time warping is actually a form of HMM. In [10] both techniques are combined, using the DTW to guide the HMM. Once again these techniques often rely on the availability of strucural information, but are generally applicable otherwise.

2.3 Mobile applications

Several mobile applications exist in the field of score following, most notably for Apple's iOS platform. The first is *Tonara* [3], which judging by its promotional videos works great, but uses a proprietary reference file format, limiting the available music to files added by the developers. The second is *Autoflip* [4] which works with any PDF file by requiring the user to play and flip the virtual page at the appropriate time. Both of these applications are closed source, so little information about their workings is available besides the listed functionality.

[2]These matrices show the similarity of a reference feature vector with respect to every input feature vector, for every point in time.

[3]`http://www.tonara.com`

[4]`http://www.velvetmatter.com/`

CHAPTER 3

Methods

3.1 General approach

This thesis proposes a solution to the problem using a score following application on a mobile device. It removes the requirement of having symbolic information available by working in two modes:

- A **training mode** in which users view or select a visual representation of the music, in this case a set of image files. They can connect this visual data to audio features by playing through an audio file and marking corresponding points in the visual representation. The application remembers features corresponding to the provided audio, allowing it to jump to the correct position when these features are “heard” in playing mode.
- A **playing mode** in which users can play along with a visual representation of music that has been previously linked to feature information in training mode.

The training mode is similar to the approach taken by the Autoflip application mentioned in section 2.3, but differs in that multiple reference points per page can be chosen, allowing for a “smoother” transition to each point. The application relies on feature extraction and position matching techniques derived from the ones listed in section 2, which are explained in the following sections.

3.2 Feature extraction: The chroma vector

The feature vector used in this thesis is a chroma vector similar to the one proposed in [1, 7, 14]. While reference audio may have similar musical contents to user provided playback, its specific sound features can differ a lot. A chroma vector may do a good job at eliminating these differences. The approach applied here uses a logarithmic magnitude scale to match the roughly exponential amplitude perception [25] of the human auditory system. It differs from the original chroma vector in that it sums the intensities of all corresponding chroma bins instead of averaging them. This results in higher peaks for active frequencies. Details about the implementation of the chroma vector can be found in section 4.1.

3.2.1 Extracting frequencies: The Fourier Transform

Sound is determined by oscillations of air pressure in a medium, usually air. These variations can be measured, showing amplitude as a function of time. Using a *Fourier transform* [3], this function can be transformed to a frequency domain representation of the signal, known as a *frequency spectrum* or *spectogram.* The time/amplitude function is continuous, but can be stored in digital systems using a method called Pulse-code modulation (PCM)[20]. This representation is alias-free as long as the sampling frequency is at least twice the highest frequency contained in the signal (this a lower boundary known as the *Nyquist rate*) [23]. Converting such data to a frequency spectrum uses a discrete form of the Fourier transform, aptly named the *discrete Fourier transform* (DFT) [24]. This transform is given by:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi \frac{k}{N} n}. \tag{3.1}$$

Where $x_0, ..., x_{N-1}$ is the amplitude data, and X_k is a complex number displaying the intensity and phase of each frequency component. [1]

The DFT assumes that its finite input data is an integer number of periods of a continuous waveform. For the purpose of this thesis this is generally not the case. The truncation of waveforms at the sides results in incorrect values in the amplitude/frequency spectrum, known as *spectral leakage.* Figures 3.1a and 3.1b show an example of this effect. Spectral leakage can be minimized by applying a *window function* to the data [16], as figure 3.1c illustrates. This "smoothes" the sides of the input signal, thereby reducing the effects of the truncation.

To create the 12-item chroma vector, each value in the spectogram is assigned to its corresponding chroma bin. This process is detailed in section 4.1.

3.3 Position matching: Hidden Markov Model

To robustly match the position of a feature vector the application uses a modified Hidden Markov Model (HMM). In this type of statistical model, a system is modelled as a discrete set of states which are assumed to adhere to the *Markov property*, meaning that the probability of transitioning to another state depends only on the current state, not on the past. These *transition probabilities* between states are a known property of the model. The state of the system cannot be directly observed (hence the term *hidden*), but in each state the system emits information with a certain probability. Probabilities corresponding to these emissions are also a known property of the model, denoted with *emission probabilities.* Transition and emission probabilities are generally obtained through training using a large dataset of known states and emissions. Knowing both the transition and emission probabilities enables determining a maximum likelihood system state given a set of emissions.

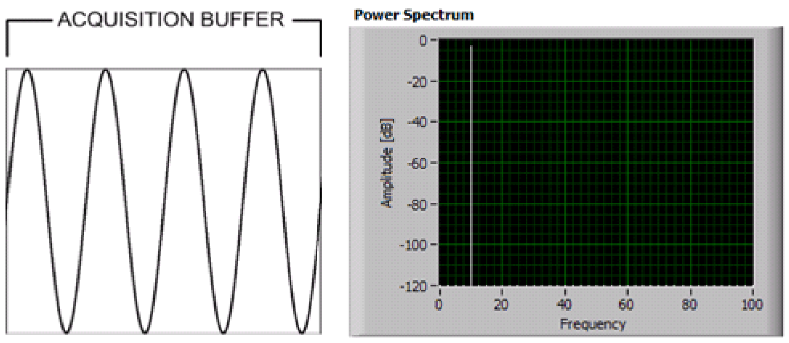
For the model applied in this thesis the state of the system corresponds to a position in the reference audio data. For any position (or state) Q, given a new input vector v, the probability of being in that position is given by:

$$P(Q|A, v) = P_{transition}(A, Q) \times P_{emission}(Q, v) \tag{3.2}$$

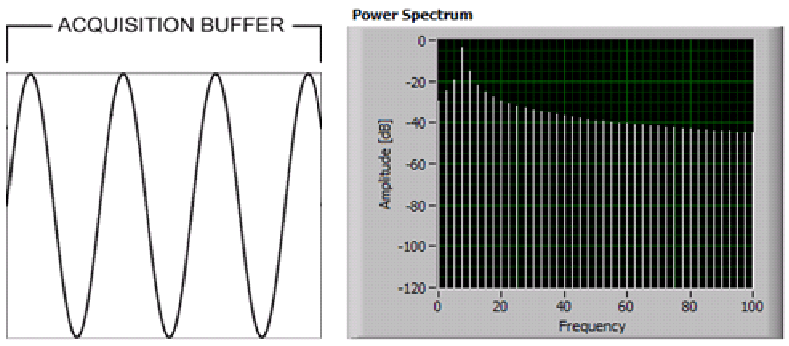
Where A is the active state. This state is not known, rather there is a set S of candidate active states. The probability is thus given by the maximum likelihood as a result of being in any of these states:

$$P(Q|v) = \arg\max_{A \in S} \left(P(A) \times P(Q|A, v)\right) \tag{3.3}$$

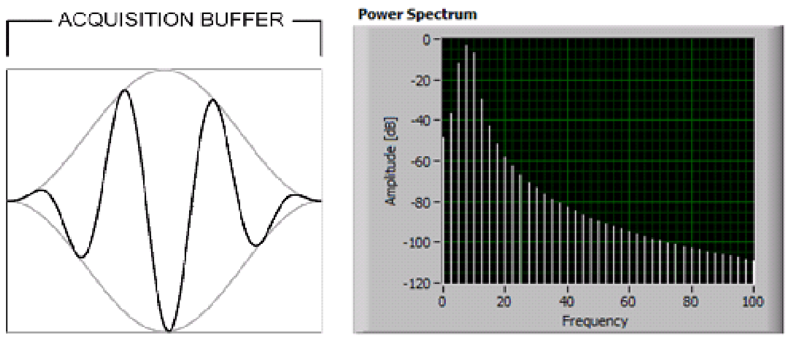
[1]Source: `http://en.wikipedia.org/wiki/Discrete_Fourier_transform`



(a) A frequency spectrum result of an FFT taken from a signal with an integer number of periods.



(b) A frequency spectrum result of an FFT taken from a signal with a truncated period. There is substantial spectral leakage.



(c) The result of applying a window function to the buffer in figure 3.1b. The spectral leakage is greatly reduced.

Figure 3.1: An illustration of how window functions influence a Fourier Transform. Source: [16]

Where $P(A)$ denotes the probability of A being the active state.

The $P_{emission}$ function returns the emission probability of the given vector from the given state, which is not a trained feature but rather inversely proportional to the *Euclidean distance* [11] between an input and a reference feature vector. The transition probability $P_{transition}$ is time dependant, and is modelled after a normal distribution [28] around the expected position:

$$P_{transition}(A, B) \sim \mathcal{N}(\mu, \sigma^2) \tag{3.4}$$

This property is *learning*: the expected position μ and standard deviation of the distribution σ are determined by previous state transitions, adapting to performance speed and previous mistakes. This allows for increased matching accuracy as the performance progresses. Note that the set of possible emissions is infinite with a continuous emission probability, which is unusual for a HMM.

CHAPTER 4

Implementation

The implemented code consists of three different components:

- A platform-independent library component, which contains all the relevant matcher logic, such as feature extraction and position matching.
- An application component, which is the prototype application designed to run on a mobile device.
- An experimentation component, which can be used to simulate running the matcher using audio files for both reference and input data.

All components are implemented using the Java programming language [1] inside the Eclipse IDE [2].

4.1 Feature extraction

The first step of feature extraction is running audio data through a *short time Fourier transform* (STFT) [17]. This algorithm uses a sliding window across the available audio data with two relevant parameters:

- The **window size** w determines the number of samples in each data window.
- The **hop size** h determines the number of samples between the start of each window.

If the hop size is smaller than the window size, data windows use overlapping data. The implementation used here uses $w = h = 0.25$, as was used in [14]. Other sizes can be configured with the constraint $0 < h \leq w$, meaning no data can be discarded. The Android operating system fills the microphone buffer in the background on a different thread, so computation never blocks I/O.

Data is requested from either the device microphone or the experimenter in sizes of h samples.This data is pushed onto the end of a buffer of length w, removing the oldest samples that no longer

[1] `http://www.java.com`
[2] `http://www.eclipse.org`

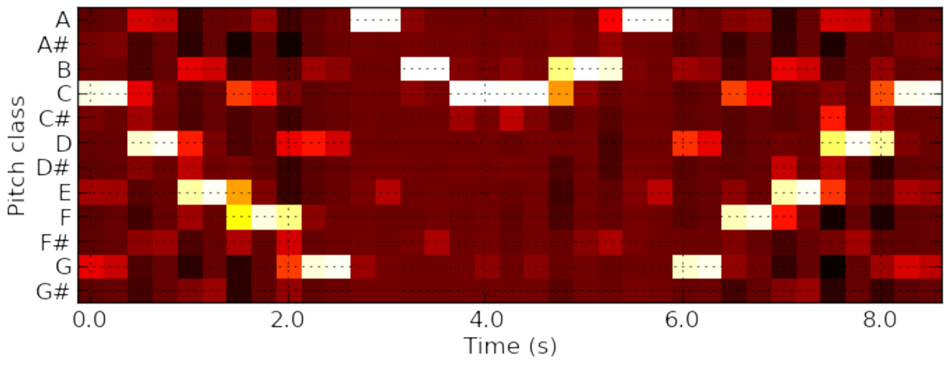


Figure 4.1: A list of chroma-vectors with $w = h = 0.25s$ extracted from a C-major scale played up and down, represented as a heat map. Lighter colors indicate higher intensities.

fit in the buffer. A circular buffer implementation is used for efficiency. To minimize spectral leakage, the data in the buffer is windowed using a *Hann window* [16] before running it through the STFT. To perform the Fourier transform the *JTransforms* library [3] is used, which is a Fast Fourier Transform (FFT) [5] implemented in pure Java. Unlike several other FFTs, this implementation improves on the default $O(N^2)$ complexity even for input lengths that are not a power of two.

In a process similar to [1], each bin of the FFT's output is assigned to one of the twelve chroma bins, summing the magnitudes of corresponding bins. The frequency of the ith bin of a FFT corresponds to:

$$frequency(s, i) = \frac{1}{2} \times \frac{s * i}{N} \tag{4.1}$$

Where s is the sample rate, and N denotes the number of samples. The corresponding bin for a frequency f is found using:

$$frequencyBin(f) = ||12 \times \log_2 \frac{f}{440}|| \bmod 12 \tag{4.2}$$

To create the final chroma vector, the 10-logarithm of the values in each of the twelve resulting bins is taken, after which the values are normalized so that the minimum value in the vector is 0 and the maximum is 1.

Algorithm 1 shows the feature extraction process in pseudo-code. An example of the result can be found in figure 4.1.

4.2 Position matching

To match the position of input feature vectors to reference feature vectors, the application uses a dynamic programming algorithm very similar to the Viterbi algorithm [26]. It tracks a list of all possible states, or positions in time, and the probability of being in these states. These probabilities are normalized to 1 to prevent them from converging to 0 over time. When a new input feature vector becomes available, transition probabilities are calculated for every possible transition from states in this list. From these probabilities, a new list is constructed of all possible states and their maximum probability. From this list, the state with the highest probability is chosen as the current state. Algorithm 2 outlines the basics of the matching algorithm. Its

[3]`http://sites.google.com/site/piotrwendykier/software/jtransforms`

Algorithm 1 Feature extraction

```
$b \leftarrow$ data buffer of length $w$ (the window size)
$d \leftarrow$ input data
$s \leftarrow$ the sample rate
$insert(d \rightarrow b)$
$dw \leftarrow hannWindowed(b)$
$fft \leftarrow doFFT(dw)$
$v \leftarrow$ [0, 0, ..., 0] vector of length 12
for $i = 1 \rightarrow w$ do
    $f \leftarrow frequency(s, i)$
    $bin \leftarrow frequencyBin(f)$
    $v_{bin} \leftarrow v_{bin} + dw_i$
end for
for $i = 1 \rightarrow 12$ do
    $v_i \leftarrow \log_{10} v_i$
end for
for $i = 1 \rightarrow 12$ do
    $v_i \leftarrow v_i - \min v / (\max v - \min v)$
end for
```

computational complexity is $O(|S|^2)$ for each run, where S is the set of reference states. For T input feature vectors, this equals the complexity of the Viterbi algorithm, $O(T \times |S|^2)$.

The actual implementation limits transitions for every state to a window of 6 seconds before and after the state, which should be enough to encompass most tempo variations and errors while limiting the search area. To prevent pursuing very unlikely paths, states with a probability below an ϵ value of 0.000001 (after normalization) are discarded. In addition, the state list is truncated to contain only those states within 10.0 seconds of the estimated position. This limits the transition window to a maximum of 16 seconds before and after the estimated active state.

Algorithm 2 Position matching

```
$S \leftarrow$ set of states
$C \leftarrow$ map of current potential states $\rightarrow$ probabilities
$R \leftarrow$ empty map of states $\rightarrow$ probabilities
$v \leftarrow$ input feature vector
for all states $s \in C$ do
    for all states $t \in S$ do
        $p \leftarrow P(s) \times P_{transition}(s, t) \times P_{emission}(s, v)$
        if $t \notin R$ or $p > R_t$ then
            $R_t \leftarrow p$
        end if
    end for
end for
$C \leftarrow R$
for all states $s \in C$ do
    $C_s = \frac{C_s}{\max R}$
    if $C_s < \epsilon$ then
        delete $C_s$
    end if
end for
$currentState \leftarrow \arg\max S$
```

The probability of a transition from a state at time s to a state at time t is given by a normal distribution:

$$P_{transition}(s,t) = \frac{1}{\sigma\sqrt{2\pi}}\, e^{-\frac{(t-(s+\mu))^2}{2\sigma^2}} \tag{4.3}$$

Initially $\mu = h$, meaning playback is assumed to have the same speed as the reference data and progresses at one hop size per received data frame. The standard deviation σ is initiated at 2.0 seconds, which should be sufficient to keep up with different playback speeds when performance starts. After more than 20 state transitions (corresponding to 5 seconds of playback) have been made, the mean and standard deviation of the time differences corresponding to these transitions are used as the parameters for the distribution. To remain tolerant to mistakes, the standard deviation has a lower boundary of 1.0 second.

The emission probability of a state s and a feature vector v is calculated using:

$$P_{emission}(s,v) = 1 - (distance(v_s, v)/\theta) \tag{4.4}$$

Where $distance(a,b)$ returns the Euclidean distance between two vectors and v_s is the feature vector corresponding to the state s. The spreading variable θ determines the relative impact of the distance on the probability. A larger θ spreads emission probabilities over a smaller area, which results in a narrower band of probabilities that diverge slower. The current implementation uses $\theta = 20$.

4.2.1 Audio detection

In real life situations a performer might pause or wait before starting to play. To prevent having to deal with random-noise feature vectors resulting from these silences, the application employs a simple audio detection algorithm. It works by comparing the sound intensity of each data window to a threshold, below which data is ignored. This threshold is currently set to 8 decibel (dB), which was chosen through observational analysis of several examples. Because device microphones might be calibrated differently and environment noise is variable, the sound intensity in decibel needs to be estimated. This is done by keeping track of the minimum window intensity (where window intensity is the *Root mean square* (RMS) [2] of its data) and using this as the decibel reference level. This approach requires at least one frame of "silence" at the start of each performance.

4.3 Mobile application

The mobile application is implemented to run on the Android platform [4]. For development the Android SDK [5] plugin for the Eclipse IDE was used. The application is targeted at the 4.x platform version, the latest version at the time of this writing. It is intended to be ran on a tablet device. Two important components (called "Activities" in Android) can be identified in the application: a training mode and a playing mode.

4.3.1 Training mode

In training mode, the user first selects one or more image files from the device's storage, which are displayed in a vertical list scaled to the width of the screen. Extra images can be added at any time. Tapping an image reveals an options menu, with options to remove the image, or move it up or down. Additionally, an audio file can be selected to use as the input source.

[4]`http://www.android.com`

[5]`http://developer.android.com/sdk/index.html`

When the “Start” button is tapped, training mode begins. In this mode, the application saves a feature vector for every 0.25 seconds of audio data. If an input audio file was previously selected, its data frames are analysed directly while playing through the device’s speaker. Microphone input is used if no audio file was specified. If the user taps the screen in this mode, a position marker is associated with the last recorded feature vector and added to the screen at the indicated position. Audio detection is active in training mode, allowing a user to stop playing while marking an active position. When training is finished, the user taps the stop button, after which the training can be saved to a file on the device’s storage.

4.3.2 Playing mode

In playing mode, the user can play along with a set of images previously trained with feature vectors and position markers. These training files can be loaded from the device’s storage, and are displayed in a manner similar to training mode: images listed vertically with numbered circles identifying the marker positions. To determine the active marker, the current position is compared to a list of positions associated to the markers. The marker with the highest associated position that is still lower than the active position is marked as “active”, and lights up green. If the last matched marker’s position was higher than the current position, the application chooses the marker with a position closest to the estimated position as the active marker. This prevents quick jumps back and forth between markers when matches are found around their position. The screen is scrolled to center the active marker vertically, as far as the total size of the images allows this. Markers can be hidden using an option in the top bar.

Note that because past data is analysed, the position matcher always lags $h + f + c$ seconds behind the actual performance, where h is the hop size, f the time it takes to construct a feature vector and c the time used for calculating a new position. However, when adding the position markers $h + f$ is also used for data analysis, leaving the lag for uncorrected matching at c. The current scrolling mechanism does not account for this lag. An example of the playing mode’s interface is depicted in figure 4.2.

A “demo mode” is included that uses an audio file as its input source, bypassing the microphone entirely. Just like in training mode, data frames from this audio file are passed to the analyzer as they are played through the device’s speaker.

4.4 Experiments

In order to test the quality of the matching algorithm, an experimentation framework is included with the application that allows isolated testing with fixed audio files. Performing such a test requires three files:

- An audio file to use as reference input, to simulate training.
- An input audio file of a performance.
- An annotation file that matches points in time of the reference file to points in time of the input file.

The experimentation framework reads data from the audio files as PCM-frames and offers it to the analyzer interface of the matcher as if it were microphone data. Audio files are read using the Java Media Framework [6] with the MP3SPI [7] plugin.

[6]http://www.oracle.com/technetwork/java/javase/tech/index-jsp-140239.html
[7]http://www.javazoom.net/mp3spi/mp3spi.html

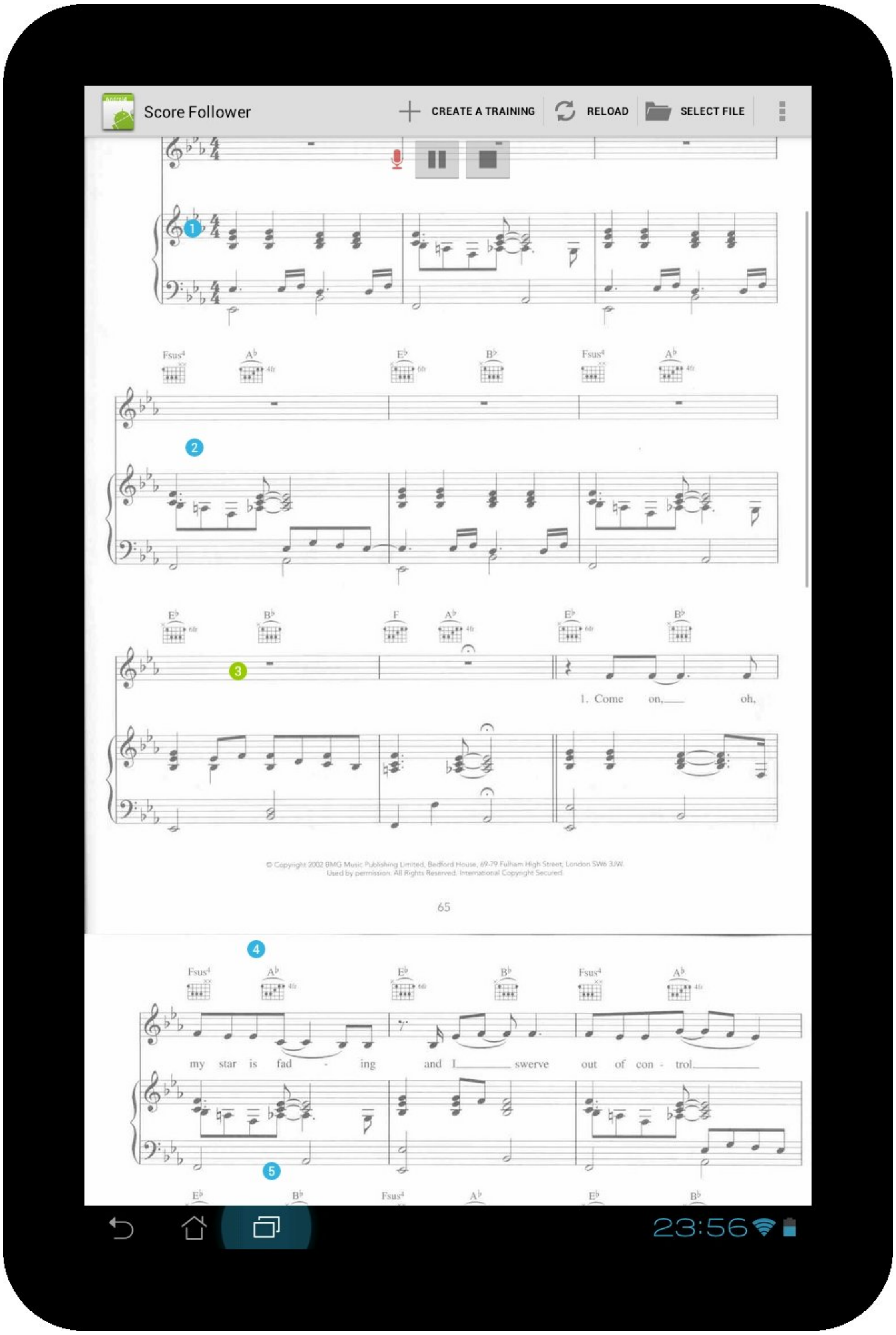


Figure 4.2: A screenshot of the tablet application in playing mode. The screen shows two consecutive images of a music score, overlain with position markers that were added in training mode. It is scrolled to center the highlighted marker.

A test is performed in two steps. First, a reference file containing feature vectors is created from the reference audio file. This uses the training mode as it would on the device, adding feature vectors for non-silent audio frames. No position markers are added to this reference file, since they are relevant for visual representation only. Note that audio detection is active in experimentation mode, so feature vectors might not be stored at equal time intervals. Since the actual time data is required to test matching quality, this data is stored alongside each feature vector. The created reference file is then used in playing mode to match against the performance audio file. In this situation frames of performance audio data are given to the matcher, which estimates their corresponding position in the reference file. The experimenter then reports the difference between these estimations and their correct positions, which are derived from the annotation file. This file has a simple two-column format, which looks as follows:

```
2.14    1.87
3.94    3.87
5.69    5.87
7.51    7.87
9.36    9.87
11.20  11.87
13.03  13.87
14.91  15.87
```

Every first column contains a position (in seconds) in the input file, with its corresponding position in the reference file in the second column. An annotation file contains only a finite number of points in time. The position of a point that cannot be matched exactly to one of the input points is linearly interpolated between the nearest two known points. Statistics are only generated for points for which this information is present, i.e. for those points that are after the first and before the last annotated point.

The experimentation program uses only the library component and can therefore run on a regular computer. Since data points can be presented to the analyzer as fast as they can be read from disk, experiments can be performed much faster than real-time.

CHAPTER 5

Results

Two distinct types of performance are relevant to this application: computational performance, which will here simply be denoted "performance" and matching performance, which will be called "matching quality". This chapter addresses both of these performance measurements.

5.1 Performance

The aim of the performance test is to determine the speed at which positions can be estimated on a mobile device. A benchmarking component is included with the mobile application to test this performance. This component aims to provide a worst-case scenario for computation. Several parameters are modified to this end:

- The audio threshold is set to 0, so all input is registered as being audio. This results in a maximum number of audio frames.
- Feature vectors get a constant emission probability, resulting in all paths having an equal likelihood. Because all path probabilities are equal, no paths will be discarded.

The benchmark runs the matching algorithm, for each data frame measuring the time from filling the microphone buffer to calculating a feature vector, as well as the total time from filling the buffer to the estimation of a corresponding reference position. The reference file used for the benchmark contains 2400 unit feature vectors. Because the emission probability is constant, the content of these vectors is actually irrelevant. Data is requested from the microphone in frames of 0.25 seconds ($w = h = 0.25$), at a sample rate of 44100Hz. The benchmark runs for 10 minutes, equalling 2400 of such data frames, which corresponds to the length of the reference file.

The hardware used for this benchmark is an ASUS Eee Pad Transformer Prime (TF201) [1], running Android 4.0.3. The tablet was running in performance mode, with WiFi disabled and only default processes running. Figure 5.1 shows the combined results of running this benchmark 4 times.

[1] `http://eee.asus.com/eeepad/transformer-prime/features/`

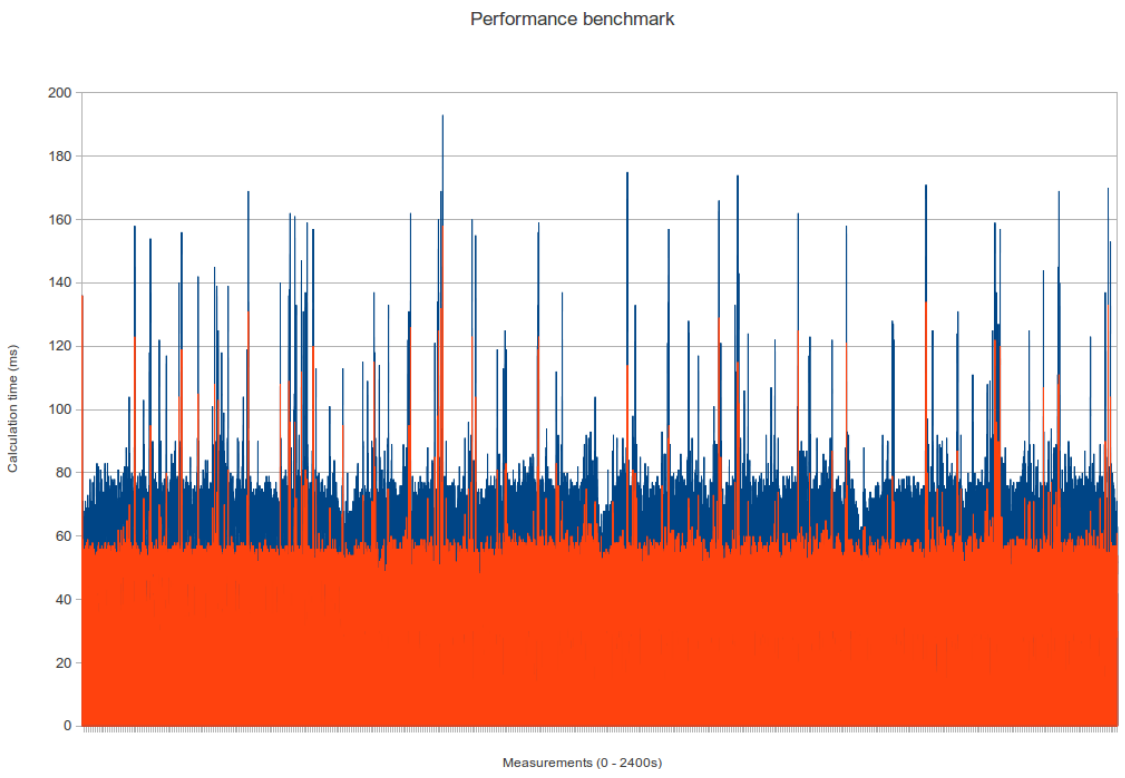
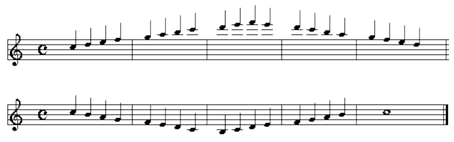


Figure 5.1: Combined results from running the benchmark described in section 5.1 4 times. The blue (dark) area shows the total calculation time, the orange (light) area the time taken for feature vector calculation. On average, feature calculation took 43ms, versus 63ms for the total calculation. Minimum and maximum position estimation times were 25ms and 193ms respectively, with a standard deviation of 14ms.


(a) Scale in C Major

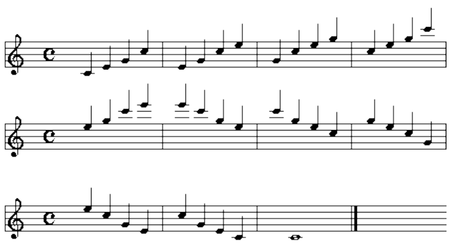

(b) Arpeggio in C Major

Figure 5.2: A scale and arpeggio in C Major used to test the matching algorithm. Source: `http://www.saxlessons.com/scalesandarpeggios.htm`

5.2 Matching quality

The experimentation framework as described in section 4.4 is used to test the quality of the matching algorithm. For each test, this program returns information about the alignment errors in seconds, as well as the "success rate". This rate is defined as the percentage of estimations with an error less than 0.5 seconds from the correct state. In order to closely match a real performance, both the reference files and the input files in this test test were recorded using the device microphone on the ASUS Transformer Prime. The default Sound Recorder application shipping with the Transformer Prime was used for recording, which uses a compressed audio format called 3GP [2]. These files were then converted to MP3 [3] for use with the Java Media Framework. At the time of this writing applications aimed at recording audio in an uncompressed PCM format seem to experience problems with either Android 4.0 or the Transformer Prime device. Attempts at recording in one of these formats therefore failed. [4] The used recordings are listed in table 5.1. They are mostly existing pieces and replications of these pieces that differ in performer and instrument. Table 5.2 shows the results of the test.

A perhaps better illustration of the matching quality can be given by looking at "following graphs", which show the input and output times for both the correct and the estimated time. Figure 5.3 shows these graphs for the performed tests.

[2]`http://www.3gpp.org/ftp/Specs/html-info/26244.htm`

[3]`http://tools.ietf.org/html/rfc3003`

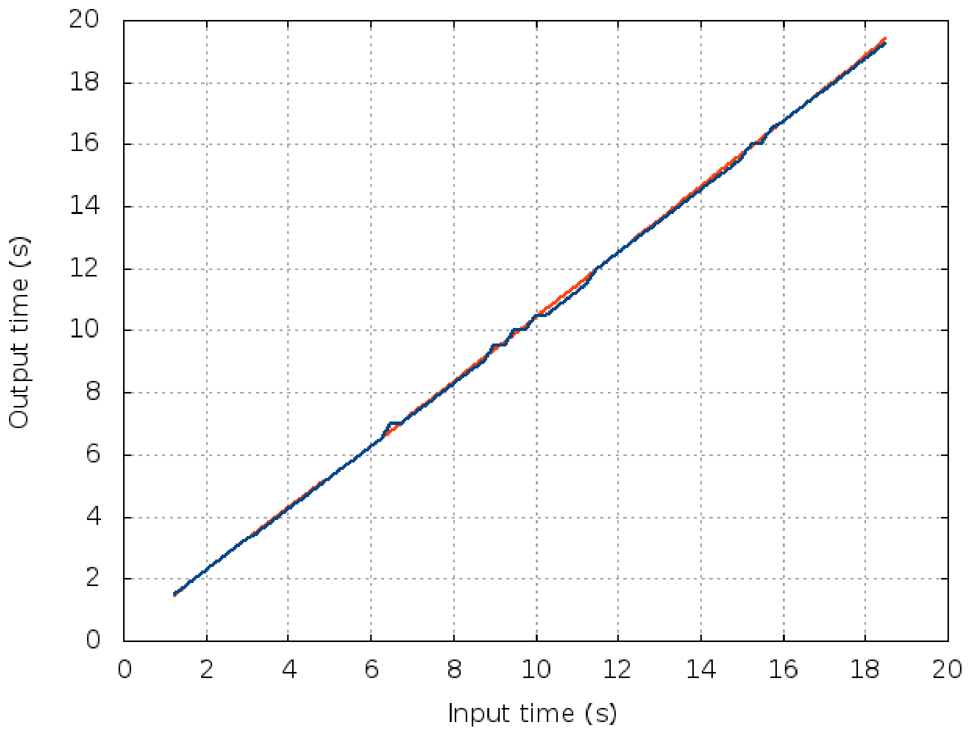
[4]The applications attempted were "Hertz", "Sound Recorder" (another one than the one used, albeit sharing its name), "Tape-a-Talk" and "Voice Recorder", all freely available from the Google Play Store (`https://play.google.com/store`). All of these applications produced corrupt files which either had artefacts or would not play at all.

Name	**Reference**	**Performance**	**Annotated length (s)**
Scale in C Major	A synthesized piano playing the scale depicted in figure 5.2a.	The author of this thesis playing the same scale on an electric piano.	17.33
Arpeggio in C Major	A synthesized piano playing an arpeggio in C Major, as depicted in figure 5.2b.	The author of this thesis playing the same scale on an electric piano.	15.94
Mary Had A Little Lamb (Piano)	A synthesized guitar playing this famous nursery rhyme. Includes both the melody and a base tone.	The author playing this piece on an electric piano.	12.32
Mary Had A Little Lamb (Guitar)	Same as above	The author playing this piece on an acoustic guitar.	12.77
Amsterdam	Studio performance of this song by Coldplay.	The author of this thesis performing the song using an electric piano and vocals.	260.81
Somebody That I Used To Know	Studio performance of this song by Gotye.	Cover of this song found on YouTube (`http://youtu.be/d9NF2edxy-M`) played back at 110% of the original speed.	137.19
All Day Long	Studio performance of this song by Novastar.	The author of this thesis performing a cover of the song using acoustic guitar and vocals.	160.82
Everything	Studio performance of this song by Ben Howard.	The author of this thesis performing a cover of the song using acoustic guitar and vocals.	216.93

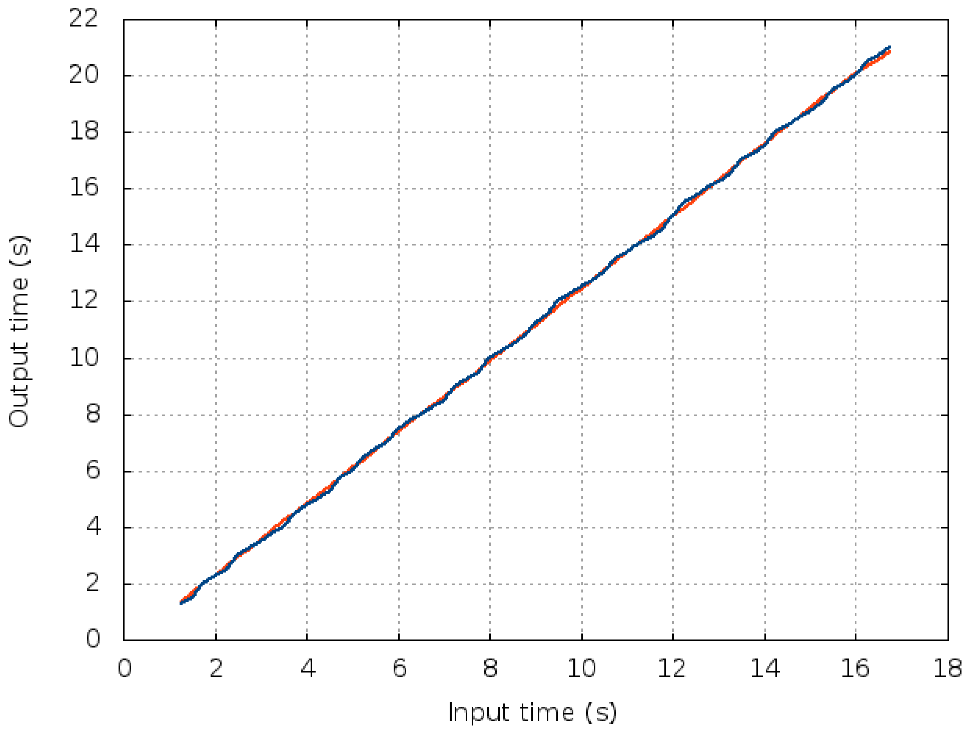
Table 5.1: Overview of the audio files used for the performance matching test.

Name	**% $< 0.5s$**	**Max. error (s)**	**Avg. error (s)***	**σ (s)***
Scale in C Major	100.00	0.23	0.07	0.06
Arpeggio in C Major	100.00	0.20	0.08	0.06
Mary Had A Little Lamb (Piano)	100.00	0.39	0.10	0.08
Mary Had A Little Lamb (Guitar)	98.04	0.63	0.08	0.08
Amsterdam	86.88	1.75	0.15	0.12
Somebody That I Used To Know	95.26	3.22	0.15	0.11
All Day Long	97.05	1.47	0.17	0.12
Everything	95.28	1.50	0.14	0.11

Table 5.2: Results of the performance matching test. The success rate is defined as the percentage of estimations within 0.5 second of the correct value. * The average error and error standard deviation (σ) are calculated over the parts within the success rate only.

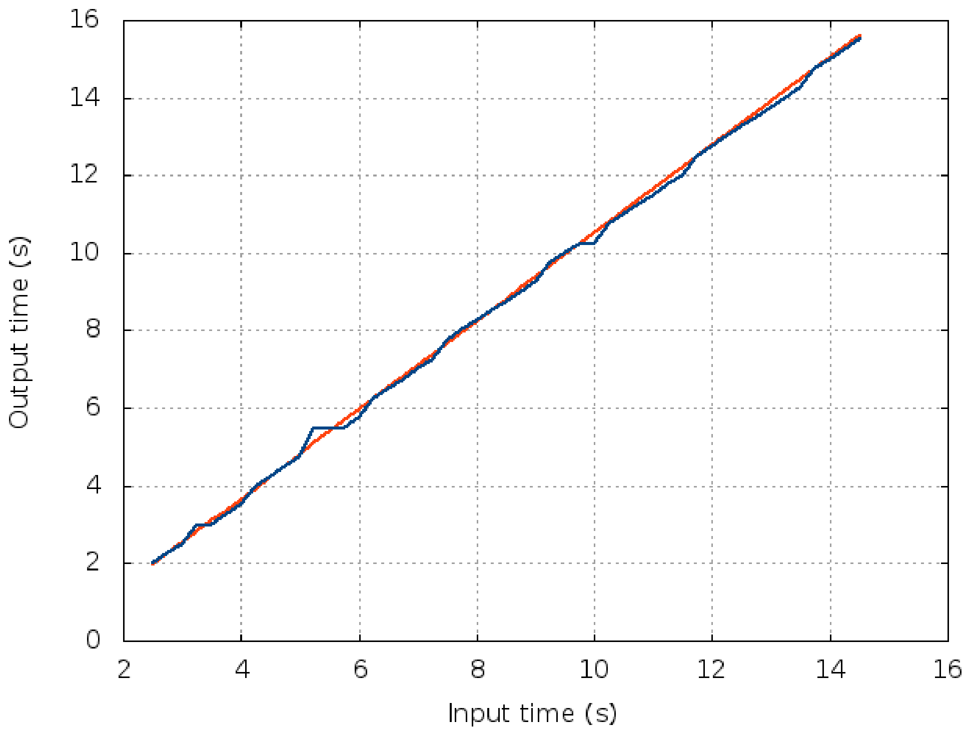


(a) Scale in C Major

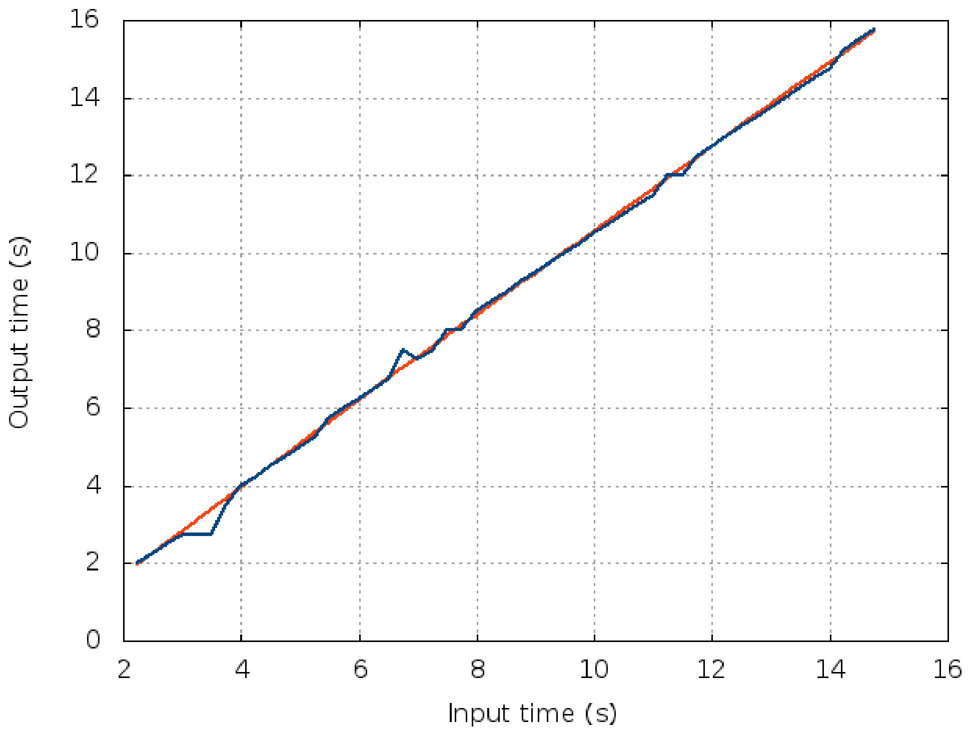


(b) Arpeggio in C Major

Figure 5.3: Following graphs of the tested performances. The orange (light) line represents the correct time, the blue (dark) line the time estimated by the matching algorithm.

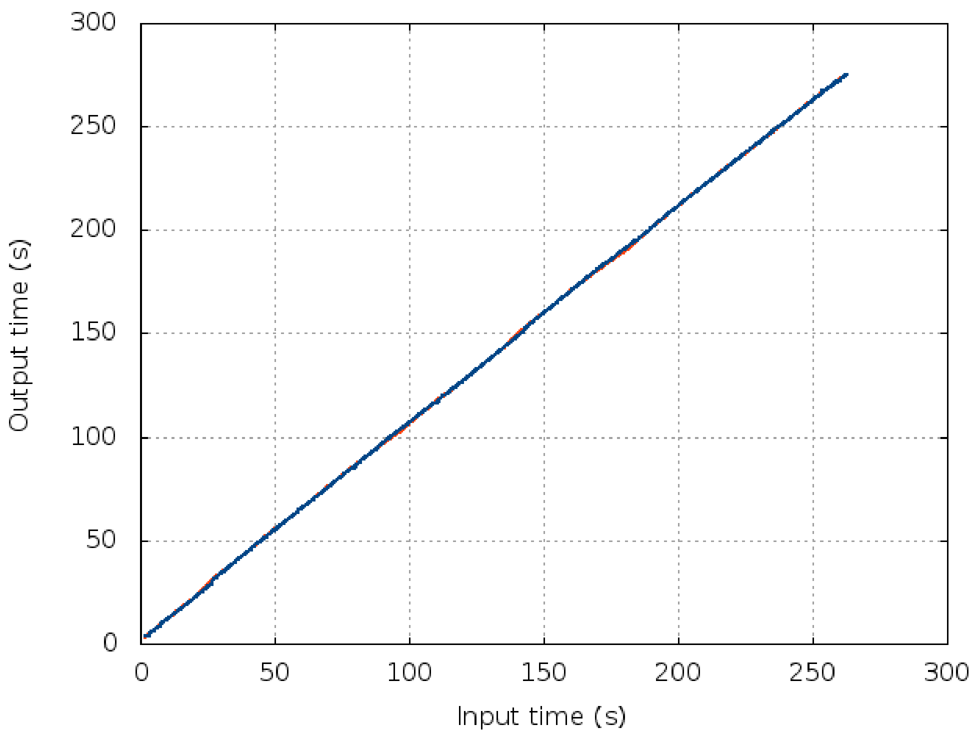


(c) Mary Had A Little Lamb (piano)

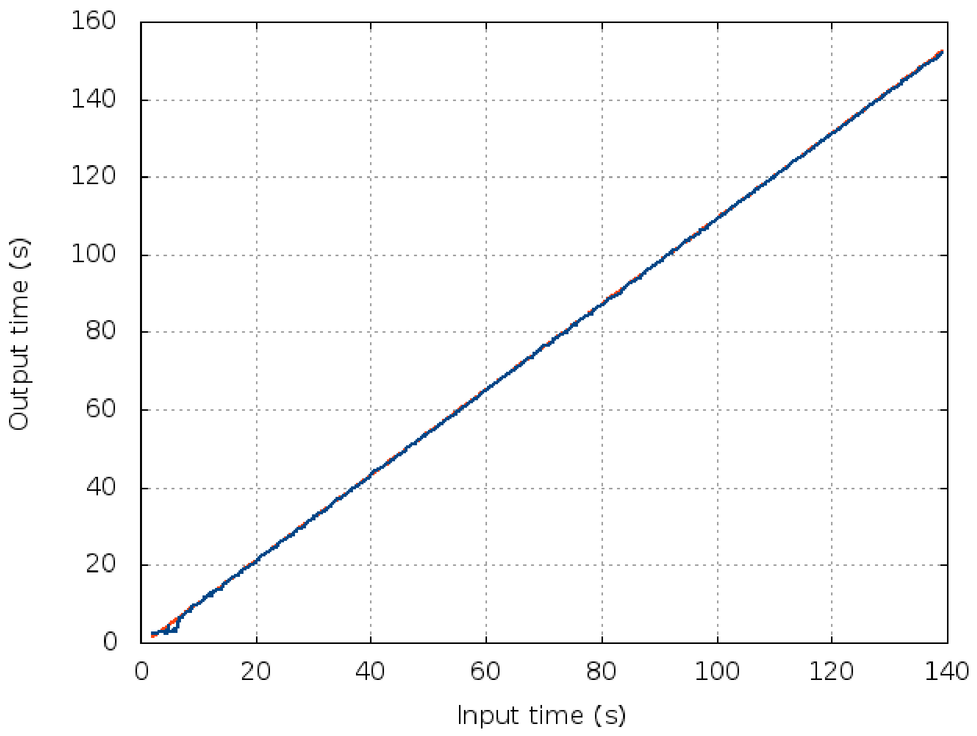


(d) Mary Had A Little Lamb (guitar)

Figure 5.3: Following graphs of the tested performances. The orange (light) line represents the correct time, the blue (dark) line the time estimated by the matching algorithm.

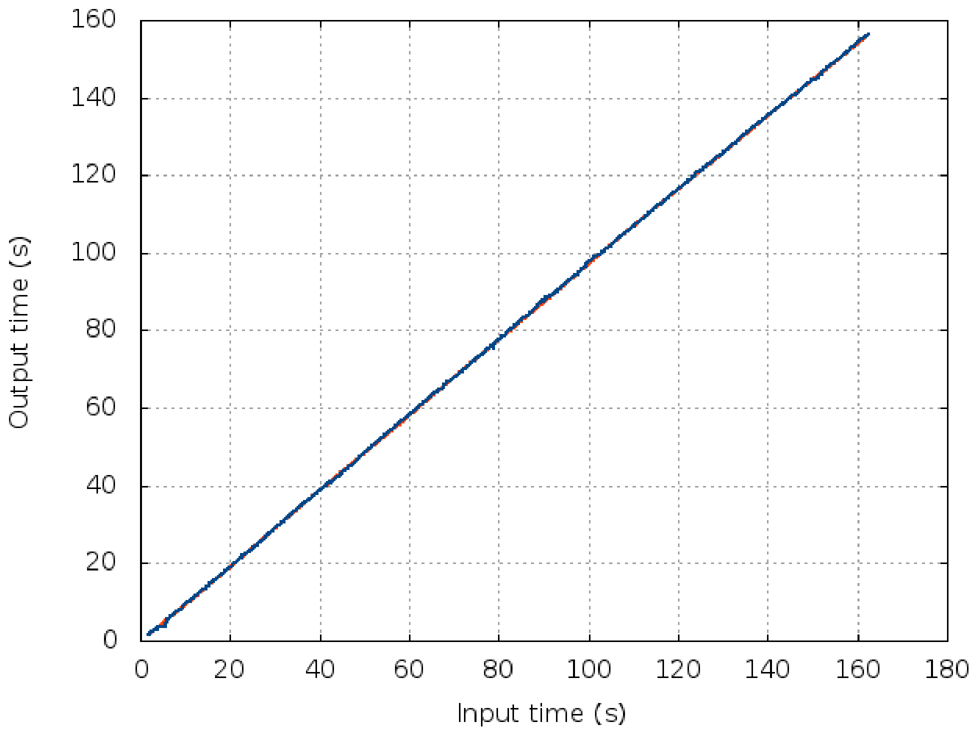


(e) Amsterdam

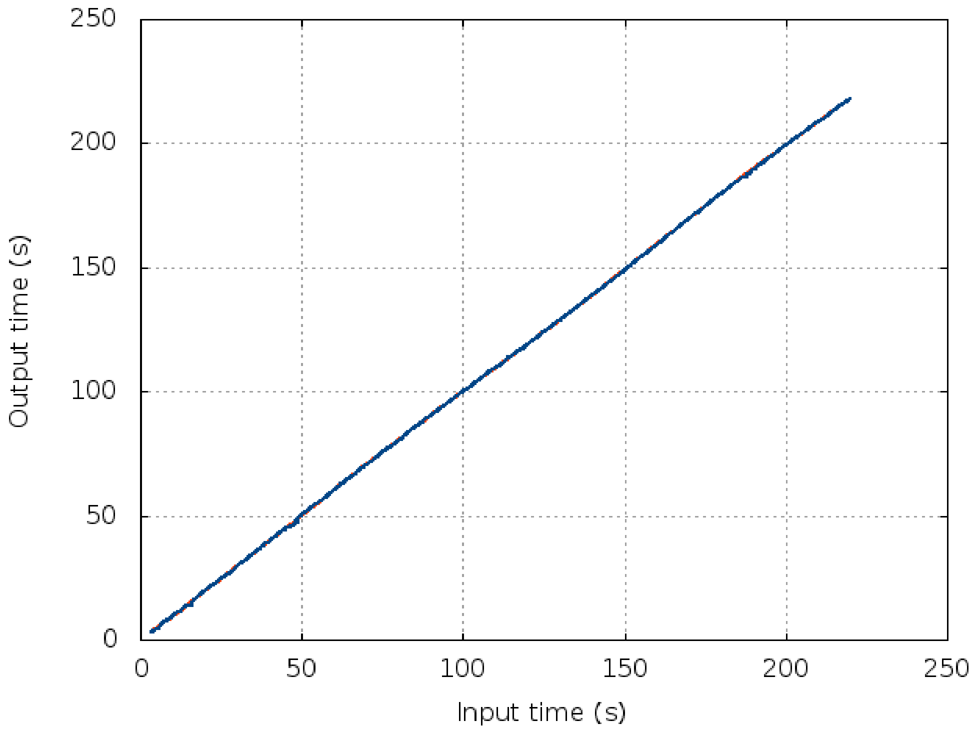


(f) Somebody That I Used To Know

Figure 5.3: Following graphs of the tested performances. The orange (light) line represents the correct time, the blue (dark) line the time estimated by the matching algorithm.



(g) All Day Long



(h) Everything

Figure 5.3: Following graphs of the tested performances. The orange (light) line represents the correct time, the blue (dark) line the time estimated by the matching algorithm.

CHAPTER 6

Conclusions

6.1 Performance

In order for the application to be usable, it cannot "lag" behind the user. Data frames that aren't processed in time for the next arriving frame will result in an increasing delay in the alignment estimation. This puts a constraint on the algorithm to not only be fast enough to be responsive, but to produce its results before new data arrives. In this case this means a state needs to be estimated within the configured hop size of 0.25s. The benchmark in section 5.1 shows no instance of the algorithm surpassing this limit. In a little under 99% (124 out of 9600 samples) of the cases the result is available within 0.1s, a measure often cited for responsiveness [18]. This does however not include updating the user interface (i.e. displaying the correct position on screen), the performance of which is not measured here. The most time is spent extracting the feature vector, approximately 68% of the total time on average. The performance graph shows a stable average line with peaks. These peaks are likely caused by garbage collection inside the Dalvik Virtual Machine [1] used to run Java byte code in Android, but no data has been collected to substantiate this claim.

6.2 Matching quality

Matching results show a high success rate for the tested songs. All short and simple songs show a matching success rate close to 100%, with an average error close to the expected error [2]. Despite using a guitar sound in the reference file, matching is better for the piano version of "Mary Had A Little Lamb". This can be explained by the lower sound levels, and thus relatively higher noise levels present in the guitar version, as depicted in figure 6.1.

Errors become more common for more complicated pieces, however success rates remain high. It should be noted that, due to their duration, the presented following graphs of these pieces have a lower resolution, obscuring the errors. The maximum error found in these tests is 3.22 seconds in "Somebody That I Used To Know". This corresponds to less than two bars of music

[1] `http://code.google.com/p/dalvik/`

[2] While annotated times are continuous, position matches occur at whole window boundaries ($n \times 0.25s$) only. Assuming all correct matches occur at their closest boundary (i.e. $[0.25n, 0.25n + 0.125)$ matches at position n, whereas $[0.25n + 0.125, 0.25(n + 1))$ matches at position $n + 1$), there is an expected error of $0.125/2 \approx 0.06$ seconds.

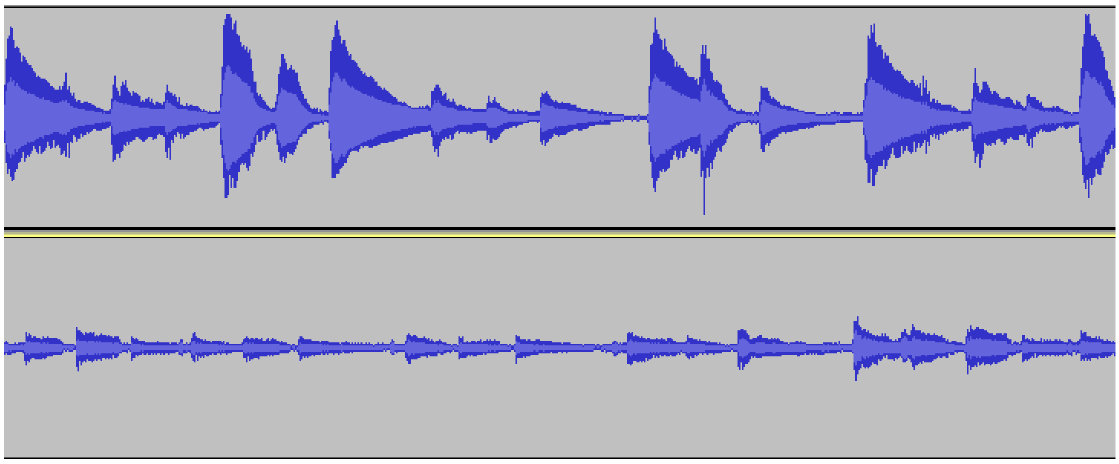


Figure 6.1: Waveforms of piano and guitar reference versions of “Mary Had A Little Lamb”. The piano version has higher amplitudes.

in a typical 120 beats per minute (bpm) score. [3] With a typical length of two or three bars per line in a score or sheet, this keeps the visual error within one line of scroll.

[3]At 120bpm, every quarter note corresponds to 0.5 seconds of data. There are four quarter notes in a bar at 4/4 time. See `http://en.wikipedia.org/wiki/Tempo` a concise explanation of music tempo.

CHAPTER 7

Recommendations for future work

The matching algorithm employed in this thesis has a lot of variables, most notably:

- Feature parameters such as window size, hop size, the used window function, number of pitch classes, intensity type (linear or logarithmic), manner of normalization.
- Matching parameters such as the transition function, the search window, emission probability calculation.

Finding the optimal set of parameters requires substantial testing on a much larger dataset than was used here. Offline training for the HMM might be beneficial, for instance analysing a large annotated data sets of actual performances could reveal a better transition function than the Gaussian curve proposed here. Quite possibly different parameters perform differently on varying music styles, something that needs to be investigated.

The audio samples used for the experiments in chapter 5 are primarily "covers" of existing pieces of various complexity, yet apart from speed variations they do not contain any serious performer mistakes. Analysing and optimizing the robustness of the matcher with respect to mistakes is relevant for the future. This would involve identifying what common mistakes are and creating a dataset suitable for testing them.

There are several places in which the computational performance of the matching can still be improved. The code includes an set of options aimed to simplify experimentation, the overhead of which could be removed. As most of the calculation time is spent doing feature extraction, speeding up the Fast Fourier Transform yields potential benefits. The buffer could be forced to a length that is a power of two, which would allow concurrent execution of the FFT for JTransforms. Another option would be to use a fast C implementation of the transform in *Android Native Code* [1].

The focus of this thesis has been the development of an efficient matching algorithm. Developing a user interface to use with this algorithm is an entirely different field of expertise, but no less important when it comes to these applications actually being used. The interface of the prototype application is sufficient for testing and basic playback, but further work needs to be done in implementing better scrolling logic. For instance, position interpolation can be used that takes the duration of calculating new positions into account. The training mode is problematic as it requires marking active positions during playback. Playing back an existing recording

[1]`http://developer.android.com/sdk/ndk/index.html`

while marking eliminates this problem, but this recording might not be available or show enough similarities to the user performance to be useful. Audio detection provides another solution to this problem, but its implementation is still basic and needs further work. Acquiring user feedback would be useful in improving all of these points.

A last interesting idea for the future is the development of an online platform for sharing training data. Training could be linked to existing libraries of tabs and scores online, allowing users to simply download such a file alongside their tab. This could enrich the existing ecosystem of sharing these notations online, while not putting any constraints on its current existence.

Bibliography

[1] Mark A. Bartsch. "To catch a chorus: Using chroma-based representations for audio thumbnailing". In: *Proceedings of the IEEE Workshop on Applications of Signal Processing to Audio and Acoustics. New Paltz, NY.* 2001, pp. 15–19.

[2] C. Bissell and D. Chapman. *Digital Signal Transmission.* Cambridge University Press, 1992, p. 64. ISBN: 0521425573.

[3] S. Bochner and K. Chandrasekharan. *Fourier transforms.* Princeton, New Jersey: Princeton University Press, 1949.

[4] Pedro Cano, Alex Loscos, and Jordi Bonada. "Score-Performance Matching using HMMs". In: *Proceedings of the ICMC.* 1999, pp. 441–444.

[5] W. Cochran et al. "What is the fast Fourier transform?" In: *Audio and Electroacoustics, IEEE Transactions on* 15.2 (1967), pp. 45–55. URL: `http://ieeexplore.ieee.org/xpls/abs\_all.jsp?arnumber=1161899`.

[6] Markus Cremer et al. "AudioID: Towards Content-Based Identification of Audio Material". In: *Audio Engineering Society Convention 110.* May 2001.

[7] Roger B. Dannenberg and Ning Hu. "Polyphonic Audio Matching for Score Following and Intelligent Audio Editors". In: *Proceedings of the ICMC.* 2003, pp. 27–34.

[8] Roger B. Dannenberg and Christopher Raphael. "Music score alignment and computer accompaniment". In: *Commun. ACM 49(8)* (2006), pp. 39–43.

[9] P. Desain, H. Honing, and H. Heijink. "Robust Score-Performance Matching: Taking Advantage of Structural Information". In: *Proceedings of the ICMC.* 1997, pp. 337–340.

[10] Ellis D. Devaney J. Mandel M. "Improving Midi-Audio Alignment with Acoustic Features". In: *Proceedings of WASPAA.* 2009, pp. 45–48.

[11] Michel M. Deza and Elena Deza. *Encyclopedia of Distances.* 1st ed. Springer, 2009, p. 94. ISBN: 3642002331.

[12] J. Haitsma. "A Highly Robust Audio Fingerprinting System". In: *Proceedings of the ISMIR.* 2002, pp. 107–115.

[13] Hank Heijink et al. "Make Me a Match: An Evaluation of Different Approaches to Score Performance Matching". In: *Comput. Music J.* 24.1 (Apr. 2000), pp. 43–56. ISSN: 0148-9267. DOI: `10.1162/014892600559173`. URL: `http://dx.doi.org/10.1162/014892600559173`.

[14] Ning Hu, Roger B. Dannenberg, and George Tzanetakis. "Polyphonic Audio Matching and Alignment for Music Retrieval". In: *Procceedings of the IEEE WASPAA.* 2003, pp. 185–188.

[15] Adrian Iftene, Andrei Rusu, and Alexandra Leahu. "Music Identification Using Chroma Features". In: *CLEF (Notebook Papers/Labs/Workshop).* Ed. by Vivien Petras, Pamela Forner, and Paul D. Clough. 2011.

[16] National Instruments. *Windowing: Optimizing FFTs using Window Functions.* 2011. URL: `http://www.ni.com/white-paper/4844/en`.

[17] E. Jacobsen and R. Lyons. “The sliding DFT”. In: *Signal Processing Magazine, IEEE* 20.2 (2003), pp. 74–80. ISSN: 1053-5888. URL: `http://dx.doi.org/10.1109/MSP.2003.1184347`.

[18] Christopher Johnson and Charles Hansen. *Visualization Handbook*. Orlando, FL, USA: Academic Press, Inc., 2004, pp. 418–419. ISBN: 012387582X.

[19] B. Logan. “Mel Frequency Cepstral Coefficients for Music Modeling”. In: *Int. Symposium on Music Information Retrieval*. 2000.

[20] B M Oliver, J R Pierce, and C E Shannon. “The Philosophy of PCM”. In: *Proceedings of the IRE* 36.11 (1948), pp. 1324–1331. URL: `http://ieeexplore.ieee.org/stampPDF/getPDF.jsp?tp=&arnumber=1697556&isnumber=35778`.

[21] N. Orio and F. Dechelle. “Score Following Using Spectral Analysis and Hidden Markov Models”. In: *Proceedings of the ICMC*. 2001, pp. 151–154.

[22] J. Rothstein. *Midi: A Comprehensive Introduction*. The Computer Music and Digital Audio Series. A-R Editions, 1995. ISBN: 9780895793096.

[23] C. E. Shannon. “Communication in the Presence of Noise”. In: *Proceedings of the IRE* 37.1 (Jan. 1949), pp. 10–21. ISSN: 0096-8390.

[24] S. W. Smith. *The Scientist and Engineer’s Guide to Digital Signal Processing*. Second ed. San Diego, Calif.: California Technical Publishing, 1999. Chap. 8: The Discrete Fourier Transform. ISBN: 0-9660176-3-3.

[25] S. S. Stevens. “The Measurement of Loudness”. In: *The Journal of the Acoustical Society of America* 27.5 (1955), pp. 815–829. DOI: `10.1121/1.1908048`. URL: `http://link.aip.org/link/?JAS/27/815/1`.

[26] A Viterbi. “Error bounds for convolutional codes and an asymptotically optimum decoding algorithm”. In: *IEEE Transactions on Information Theory* 13.2 (1967), pp. 260–269. URL: `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1054010`.

[27] Avery L. Wang. “An Industrial-Strength Audio Search Algorithm”. In: *ISMIR 2003, 4th Symposium Conference on Music Information Retrieval*. 2003, pp. 7–13.

[28] B Wlodzimierz. *The Normal Distribution: Characterizations with Applications (Lecture Notes in Statistics)*. Springer-Verlag, 1995. ISBN: 0-387-97990-5.