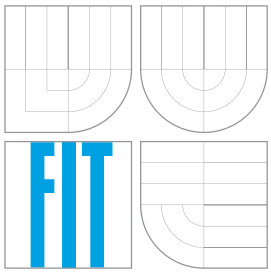
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

ČTEČKA NOT PRO ANDROID

SMART SHEET MUSIC READER FOR ANDROID

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. VOJTĚCH SMEJKAL

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. IGOR SZŐKE, Ph.D.

BRNO 2014

Abstrakt

Oblasti jako automatické otáčení stránek nebo automatický hudební doprovod jsou studovány již několik desetiletí. Tato práce shrnuje současné metody pro počítačové sledování not v reálném čase. Zabývá se také hudebními příznaky jako jsou chroma třídy a syntetizované spektrální šablony. Dále popisuje klíčové části systému jako krátkodobou Fourierovu transformaci a Dynamické borcení času. V rámci projektu byl navrhnut a vyvinut vlastní systém pro sledování pozice hráče v notách, který byl následně implementován jako mobilní aplikace. Výsledný systém dokáže sledovat i skladby s výrazně odlišným tempem, pauzami během hry nebo drobnými odchylkami od předepsaných not.

Abstract

Automatic page turning and automatic music accompaniment have been studied for several decades. This work summarizes the state of art approaches to real-time score following. It studies various audio features such as chroma classes and synthesized spectral templates. It also describes short-time Fourier transform and online Dynamic time warping as key components of the system. This project analyzes in detail developed solution for tracking the player position in score, which was then implemented as mobile application. Final system is able to follow pieces even with changing tempo, pauses during performance, and minor deviations from the original score.

Klíčová slova

sledování not, automatické otáčení stran, zarovnání zvuku k notovému zápisu, zpracování zvuku, zpracování hudby, hudební příznaky, chroma třídy, spektrální šablony, Dynamické borcení času, Fourierova transformace

Keywords

score following, automatic page turning, audio to score alignment, audio processing, music processing, music features, chroma classes, spectral templates, Dynamic time warping, Fourier transform

Citace

Vojtěch Smejkal: Smart Sheet Music Reader for Android, diplomová práce, Brno, FIT VUT v Brně, 2014

Smart Sheet Music Reader for Android

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Igora Szökeho, Ph.D.

.....
Vojtěch Smejkal
May 21, 2014

Poděkování

Děkuji vedoucímu práce panu Ing. Igoru Szökemu, Ph.D., za cenné rady v průběhu řešení projektu a také své rodině za podporu a motivaci.

Contents

Chapter 1

Introduction

The aim of this thesis is to describe a system and accompanying algorithms capable of following the musician's position in score in real-time. Resulting application for Android operating system should be able to read sheet music from MIDI files, render it to the display of smartphone, and listen to and react on incoming sound signal by scrolling the score. For simplicity, focus will be primarily given to **piano** as the most widespread musical instrument.

Score following task consists of two main challenges [1]:

- **Interpretation problem:** Audio stream coming from live performance has to be correctly interpreted, in order to build a common representation of the score and performance, which is used to create corresponding pairs later.
- **Alignment problem:** The matching has to be found in real time with adequate flexibility and error tolerance.

Chapter 1 will further introduce history and background conditions which initiated work on this topic. Also some typical use cases of this kind of application will be studied. Chapter 2 describes in detail problem of score following, familiarizes readers with related tasks, score following system architecture and typical algorithms. Chapter 3 focuses on features extracted from audio as the most significant part which affects the total performance. Each feature set is analyzed and compared to others in terms of its performance and computational complexity. Chapter 4 reveals the implementation details and various internal parameters settings. Chapter 5 discusses specifics of implemented mobile application such as sheet music extraction and rendering. Chapter 6 finally reports on performed experiments and obtained results.

1.1 Background

Score following isn't any artificial task and same or similar problems have been studied for several decades. For example, French institute for science about music and sound, IRCAM, was founded in 1977 [14]. There are also various conferences and contests about music information processing and retrieval held worldwide, like ISMIR (International Conference on Music Information Retrieval) or SMC (Sound and Music Computing Conference).

The research on score following is motivated by the situation most of the professional musicians have ever faced. While playing some piece of music which is longer, circa four pages and more, it is not possible to unfold pages in one row, and thus it is necessary to

turn the pages manually. This usually brings unwanted pauses to performance and distracts the player. Common solution is to have some other person nearby, which follows the track along with the musician and turns the pages instead of him. However, this arrangement is still far from optimal. Even though the page-turner is highly trained person, it can turn the page too early or too late and player can get lost. Also it takes a significant period of time to turn the page which introduce another cognitive load on the performer.

With rapid development in smartphone and tablet market, a new possibilities have emerged and first applications for reading sheet music have started to appear. They are able to open PDF document or set of images and allow the musician to browse it page by page. While it's more comfortable, it is still needed to swipe or touch the screen to turn the pages of document.

As a result, a new hands-free page turners in form of pedal were produced, which allow musicians to turn the pages on the screen using their feet. One example of these products is popular AirTurn[1], which uses Bluetooth technology to communicate with iPads, Android tablets and computers.

Increasing power of mobile architectures allows to build more complex application than ever before. Now it's possible to process live audio stream in real time, most often through Fast Fourier Transform. This first led to simple applications like guitar tuners. But there is no reason why it shouldn't be possible to come up with sophisticated apps like automatic score followers. Section 1.2 describes several cases where this tool could be useful.

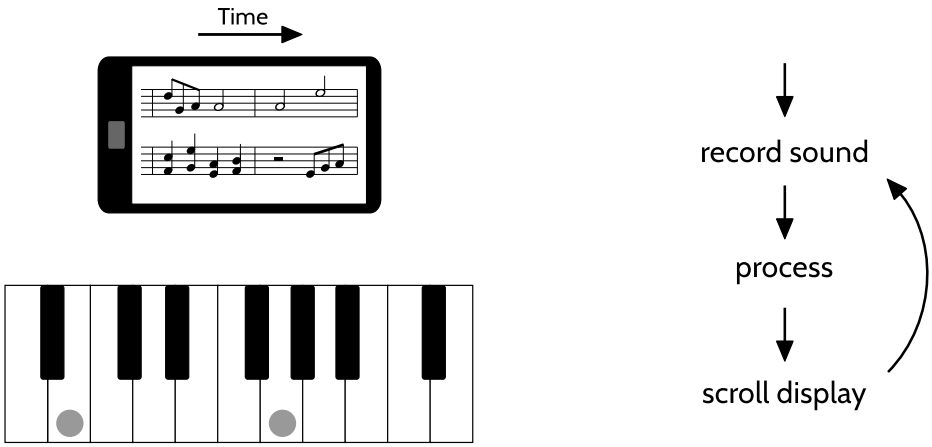


Figure 1.1: Intended arrangement of piano and mobile application (on the left) and score follower inner loop (on the right).

Figure 1.1 on the left shows intended placement of mobile device above the piano keyboard. Device is put on the same stand as regular papery sheet music. While the musician is playing, application is continuously recording the sound, processing it with score following system, and updating scroll position in virtual score on the device display (right side of the picture). Since this loop is executed several times per second, the score is smoothly moving, and musician can keep up with it even on the small screen of smartphone.

[1] http://www.airturn.com/

1.2 Use Cases

Software which can synchronize live piano playing with position in the music score would bring many advantages to the field. For better illustration, there are several typical use cases ranging from personal to professional use where people would benefit from the score-following application:

1. Casual home players who play the instrument for fun in their free time. They has just seen a YouTube video with song they like and want to play it by themselves. They don't want to print the pages or some may don't even have a printer. So they take a smartphone or tablet, run the app, and download the song. Then they put the device on the piano music stand and play the song while reading the sheet of music from the screen. Their playing tempo varies a lot during the play as they are learning new piece and going through more difficult parts. When they make a mistake and repeat the measure, app notices it and returns back to the correct position.

2. Professional pianists performing on the concerts. They utilize the full scale of dynamics and tempo from ***ppp*** (very soft) to ***fff*** (very loud) with accelerations followed by slowing down. They generally play by heart on concerts, but the app can be useful for them during training, because as previously mentioned, they don't need to turn pages by hand. They can also record the whole performance and compare the recording to the reference score for fine tuning.

3. People who come across the piano somewhere. They want to play it, but they neither remember the music nor have any sheet music with them. So they run the app on their smartphone, put it on the piano, pick their favorite song, and play it.

According to these use cases it is obvious that final application has to support a variety of styles of playing the instrument. It should tolerate errors of beginner playing slow as well as it needs to stay responsive in parts with high density of notes played in fast tempo by professional. It means that the system should adapt to the given conditions.

Chapter 2

Score Following

The task of following a performance of musician by deciding about his position in music score is called Score Following. In other definition by Cont [10] it is *a real-time mapping from Audio abstractions towards Music symbols and from performer live performance to the score in question.* Basically, given the score representation and live audio stream, system returns the current time in the score.

The difficulties of effective score following are caused by several facts. The perfect music performance where played pitches absolutely match the reference score rarely occurs. The most situations contain random note additions, misses, and changes. Also high level of polyphony (several notes played at once) makes the recognition of pitches harder. Last but not least, fast tempo changes in some expressive pieces increase the problem complexity.

In case of score following, we need to distinguish between two fundamental types:

- **Online alignment:** System doesn't know the whole audio track in advance and returns the actual position after every sound fragment. In general, it is less precise and trickier than the latter type. This type is nowadays called *Score following.*
- **Offline alignment:** Is easier, because the system knows the whole audio track and can see to the future. Algorithms designed for online alignment can be used for offline, but usually not the other way around. For this type was established the name *Audio to score alignment.*

Also depending on the input data, we can divide it into two groups:

- **MIDI input:** Many score following systems in the past worked with live MIDI data coming from the digital instrument. While this approach is easier, it is not practical and desirable in this project, since we are building app for smartphone without any MIDI interface.
- **Audio input:** Using solely audio signal, the task becomes much more difficult, because of noise, out of tune instrument, and many other factors. On the other hand, the final solution is more versatile and fits requirements of this mobile application.

2.1 System Architecture

Figure 2.1 depicts a structure of general score follower based on scheme published by Orio [16]. There are two inputs: *Sound*, which stands for live piano audio stream, and *Score representation*, which system has to follow.

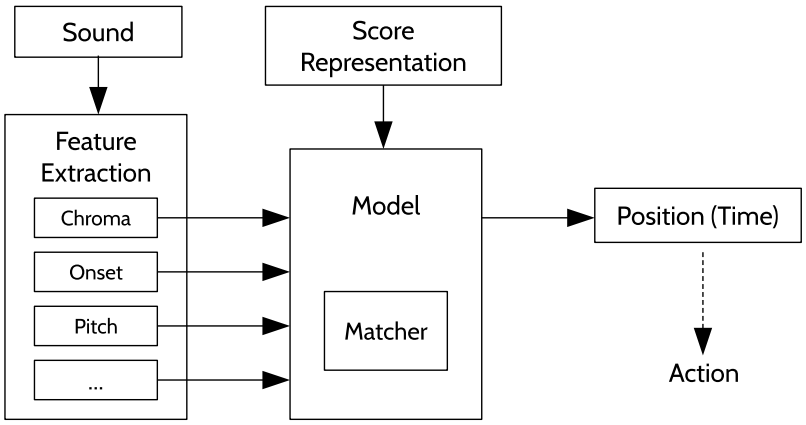


Figure 2.1: Structure of general score follower.

Score representation is most often MIDI file, but it can be also other formats like MusicXML as it will be mentioned later. It encodes the pianist gestures, which can be simple (single note or rest) or more complex (trill, glissando, ...). Therefore the choice of its format is important for ergonomics and performance of the target system.

Incoming sound is processed by *Feature extraction*, where the sound signal is analyzed using, for example, *Pitch* or *Chroma* detection. In each score following system the set of features is used as descriptors of musician's performance and usually has the biggest impact on the precision of the system [16]. Various methods for feature extraction are described in Chapter 3.

Obtained features are provided to *Model*, which using *Matcher* finds best matches with the reference score. Model is usually implemented using Dynamic time warping, Hidden Markov models or Neural networks [2]. Since it is generally difficult to extract from the score precise features comparable to extracted audio features, the score is first synthesized to audio and then it is possible to use the same feature extraction methods which have been already developed for audio.

Given current tempo, previous position, and matchings, model returns the expected *Position* in form of time in the reference score. The new position serves as action for scrolling the display or turning the page.

Requirements

The running environment for this application are smartphones and tablets, first of all Android operating system. In order to be able to make proper design decisions later, we first need to set several criteria for score follower running under this specific environment. Key features which optimal design should satisfy are:

- **High performance**: The system should achieve high recognition rate with minimal delay between the actual and reference position in the score.
- **Robustness**: The algorithm must be durable to small perturbations in *melody* (when performer ignores some musical ornament), *tempo* (when it suddenly changes) or *position* (when some notes are repeated after mistake). It should also ignore various

background noises – like clapping, coughing and creaking – which are typical during real performances.

- **Responsiveness**: It should react as fast as possible, because the music score position on small device display has to be updated frequently.
- **Lightweight**: Smartphones have limited CPU performance and memory usage. Heavy computations can drain the battery in a very short time. Also if application reaches the memory limit, it is killed by operating system. Therefore the application has to utilize available resources wisely.

2.2 Current Approaches

The first score following approaches utilized bottom-up dynamic programming techniques for error-tolerant alignment. They just paired notes of score to notes of performance and didn't pay any attention to any rhythm. The algorithms at that time would limit the calculations around a window in order to comply with real-time constraints [1].

Later, some enhancement were proposed like detecting and grouping incoming notes to model complex events – chords, trills and similar music ornaments. More advanced strategies were developed, which were able to handle ambiguous situations like when performer suddenly stops.

All the approaches which have been taken till today can be divided into two groups: the ones using some sort of on-line **time warping**, and the ones using **stochastic models** like Neural networks, Hidden Markov models, Conditional random fields, etc. The both of them are being actively used nowadays.

Among many people who are researching this field, there is a few of them which have been active in the longer term. Namely, *Andreas Arzt* from Johannes Kepler University in Linz and *Arshia Cont* from French institute IRCAM in Paris.

Short summary of the most successful, innovative, or interesting solutions will follow.

Chroma and Octave Representation

This methods is presented in paper called *Score Following and Retrieval Based on Chroma and Octave Representation* by Chu and Li [9]. It is based on two feature vectors, chroma and octave, which are more deeply described in section 3.2. The feature design is based on human perception of musical pitch, which is like helix where vertical dimension represents *tone height* and angular dimension represents *chroma*.

To extract features from MIDI file, the authors first divide the time scale into non-overlapping 0.2-second score frames. In each score frame, they accumulate the energy which belongs to the corresponding chroma bins. This process is illustrated in figure 2.2, where the note lengths are separated to frames and mapped to the chroma bins.

MIDI notes are identified by their numbers. How to compute chroma and octave index from MIDI number is described in formulas 2.1 and 2.2, where n_k is MIDI note number.

$$c = (n_k \bmod 12) + 1 \tag{2.1}$$

$$o = [n_k \ / \ 12] + 1 \tag{2.2}$$

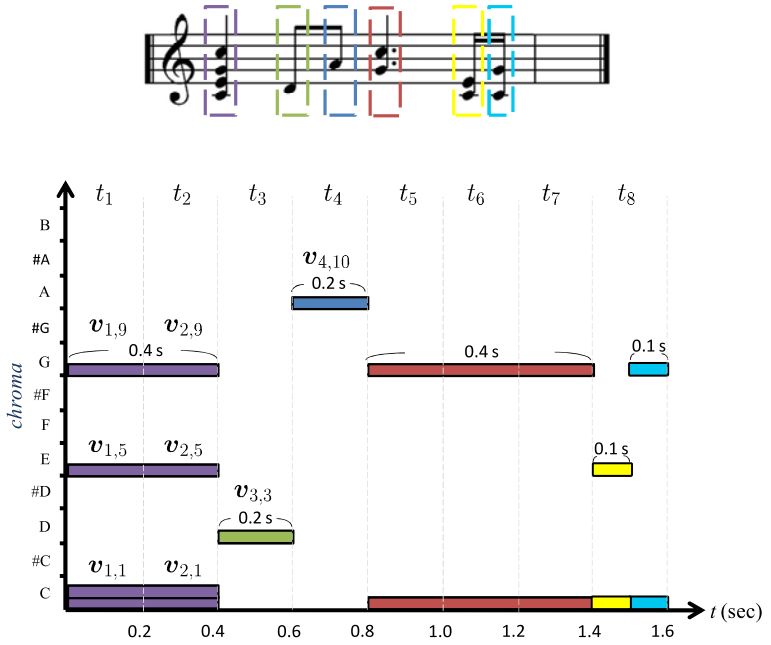


Figure 2.2: Calculating chroma energies from music score. [9]

Normalized value of the of the k-th chroma bin at the score frame f can be calculated by dividing it by the sum of the vector (formula 2.3). The same normalization is done also for octave vector in formula 2.4.

$$\tilde{\mathbf{p}}^c_{f,k} = \frac{\mathbf{p}^c_{f,k}}{\sum_{j=1}^{12} \mathbf{p}^c_{f,j}} \tag{2.3}$$

$$\tilde{\mathbf{p}}^o_{f,k} = \frac{\mathbf{p}^o_{f,k}}{\sum_{j=1}^{8} \mathbf{p}^o_{f,j}} \tag{2.4}$$

Extracting audio features is similar and is described in detail in section 3.2 in chapter about audio feature extraction. Authors also proposed pre-emphasis process for audio chroma features, which closely approximates energy distribution to the MIDI features and further improves performance of the score follower.

After both music signal and score are transformed into sequences of feature vector, we can perceive the whole problem of aligning as a sequence matching task. The authors utilize Dynamic Time Warping (DTW) algorithm (section 2.4). In order to find the alignment with lowest cost between two feature sequences, we need to construct a cost matrix.

The authors are using *Euclidean norm* as a distance measure. For chroma, the cost matrix between audio feature vector q_i and score vector p_j is calculated as

$$M^c_{i,j} = d(\tilde{\mathbf{q}}^c_i, \tilde{\mathbf{p}}^c_j) = \sqrt{\sum_{l=1}^{12} (\tilde{\mathbf{q}}^c_{i,l} - \tilde{\mathbf{p}}^c_{j,l})^2} \tag{2.5}$$

Similarly, the octave cost matrix is calculated as

$$M^o_{i,j} = d(\tilde{\mathbf{q}}^o_i, \tilde{\mathbf{p}}^o_j) = \sqrt{\sum_{l=1}^{8} (\tilde{\mathbf{q}}^o_{i,l} - \tilde{\mathbf{p}}^o_{j,l})^2} \tag{2.6}$$

The the both matrices are then combined by weighted sum

$$M^b_{i,j} = w_c M^c_{i,j} + (1 - w_c) M^o_{i,j} \tag{2.7}$$

where w_c is the relative weight of chroma feature. Experiments show that the optimal value of w_c is around 0.7.

Accuracy of music-score matching was defined as number of points correctly aligned on the DTW path divided by number of all points in the path. However, this evaluation scheme tolerates slight skews which human can hardly perceive. For real piano recordings the optimal settings of $w_c = 0.7$ and pre-emphasis weight $w_e = 10$ reached almost **95%** accuracy.

Adaptive Distance Normalization

This approach was being researched in paper called *Adaptive Distance Normalization for Real-time Music Tracking* and was presented at EUSIPCO conference in 2012 by Arzt [6]. It utilizes both harmonic (general energy distribution) and onset emphasized (increases in energy) features and proposes effective distance normalization strategy.

This audio tracking system is also based on on-line dynamic time warping. It takes two sequences of feature vectors as input. One is known beforehand (score) and the other is coming real-time during the performance. The DTW algorithm is in detail explained in section 2.4, although several improvements to this basic technique were proposed.

The first is called **backward-forward strategy**. It repeatedly reconsiders past decisions and uses revised hypotheses to improve the precision. As a results, this strategy increases robustness against tempo changes and performance errors.

Algorithm 1 shows pseudo-code of the backward-forward method. By following the backward path, the system gets a new point which should lie nearer to the optimal alignment than the corresponding point of the forward path. It is because the backward computation takes into account information from the *future* that is not available to the real-time forward path [5]. Author's implementation uses two different backtracking length: after each 4 short backtrackings with length $b = 10$, one of length $b = 50$ is performed.

The second improvement are **tempo models**, which hold the information about current tempo, and stretch or compress corresponding score representation. In this way it reduces differences in absolute tempo between original score and live performance.

Tempo computation is derived from DTW backward path. More precisely, the $n = 20$ most recent note onsets which lie at least one second in the past are selected, and local tempo for each onset is calculated from slope of the path in a 3 seconds-long window centered on the onset [3]. To emphasize more recent tempo development (which are more up-to-date, but less precise) while not throwing out older tempo information, the resulting tempo t is computed using weighted average of previous tempos t_i as

$$t = \frac{\sum_{i=1}^{n} (t_i \cdot i)}{\sum_{i=1}^{n} i} \tag{2.8}$$

Algorithm 1: The Backward-Forward Strategy

```
if frame is even then
    let current_position = (i, j);
    follow backward path from current_position;
    get backward_position;
    follow forward path from backward_position;
    get forward_position;
    if forward_position ends in row l < j then
        calculation of new rows is stopped until the current row j is reached
    end
    else if forward_position ends in column k < i then
        new rows are calculated until the current column i is reached
    end
    else
        confirmation of the current position
    end
end
```

Feeding the tempo information t to the score follower is done by alternation step. If $t > 1$, feature vector is removed from the score representation by replacing vectors $p_s + 1$ and $p_s + 2$ by their mean vector. If $0 < t < 1$, new feature is computed as mean of p_s and $p_s + 1$ and inserted between them. To avoid that the system gets stuck, maximum of 3 alternation in a row may take place.

Obtaining score features, which are comparable to live audio stream features, from individual notes in MIDI file is not an easy task. Rather than that, author suggests to convert given score into a sound file using some software synthesizer – the system solves *audio-to-audio alignment*. Both audio streams are then analyzed via short-time Fourier transform (STFT) with hamming window of size 92 ms and a hop size of 23 ms.

The final stage before DTW alignment is extraction of chroma features (NC) described in section 3.2 together with special onset features (LNSO), which are closely presented in section 3.4. Total distance d_{tot} between two feature vectors is computed as a sum of normalized and weighted LNSO and normalized NC:

$$|I|_1 = \sum_{k=1}^{n} |I_k| \tag{2.9}$$

$$d(I, J) = \sum_{k=1}^{n} |I_k - J_k| \tag{2.10}$$

$$d_n(I, J) = \frac{d(I, J)}{|I|_1 + |J|_1} \tag{2.11}$$

$$d_{nw}(I, J) = d_n(I, J) + \sqrt[4]{\frac{|I|_1 + |J|_1}{2}} \tag{2.12}$$

$$d_{tot}(I, J) = d_{nw}^{LNSO}(I, J) + d_n^{NC}(I, J) \tag{2.13}$$

Evaluation of this approach was performed on two pieces by Chopin, several sonatas by Mozart and on prelude by Rachmaninoff. The note is accepted as correctly aligned if its computed time differs from actual onset time not more than 250 ms. Single fea-

ture NC reached in average 87% successful recognitions as well as single LNSO reached 93 %. However, their combination performance achieved almost **97 %**. Author states that with this result, the possibilities of signal processing are exhausted and further significant improvements are only possible by introducing musical knowledge to the system [6].

2.3 Short-Time Fourier Transform

Signal coming out of musical instrument is composed from isolated tones, whose amplitudes were summed together. Each tone corresponds to one *fundamental frequency*, plus several lower volume *harmonic overtones* at multiples of fundamental frequency.

These overtones determine the *timbre* (also know as tone color), which allows a human to distinguish among different musical instruments, such as string instruments, wind instruments, or voice. Musical instrument without any overtones is a plain sine-wave generator.

Since the tones are defined by their frequencies, we first need to map recorded time data to the frequency domain. Most common method for this task is **Discrete Fourier transform** (DFT) described by formula 2.14. It reveals all frequency components present in the signal for further processing. The problem with this analysis is that it cannot provide simultaneous time and frequency localization [8]. It was designed to work only with stationary signal (the one which doesn't change in time).

$$F(u) = \frac{1}{N} \sum_{x=0}^{N-1} f(x) e^{\frac{-j2\pi ux}{N}}, u = 0, 1, \ldots, N-1 \tag{2.14}$$

In order to convert 1-D time data to 2-D time-frequency domain, we need to use **Short-time Fourier transform** (STFT) explained in formula 2.15 from Bebis [8], where t' is time parameter, u is frequency parameter, $f(t)$ is signal to be analyzed and W is windowing function centered at $t = t'$.

$$STFT_f^u(t', u) = \int_t [f(t) \cdot W(t - t')] \cdot e^{-j2\pi ut} dt \tag{2.15}$$

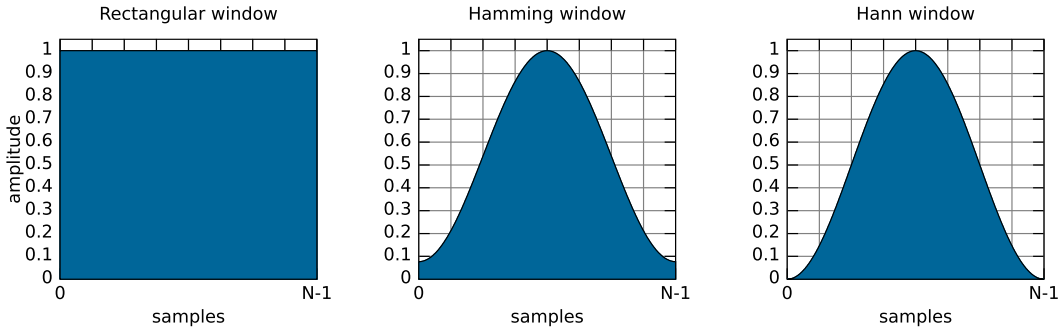
The result of STFT is 2-D plane called **spectrogram**, where time scale is on the horizontal axis, frequency is on the vertical axis, and values in bins are differentiated using colors (usually heat map palette).

Window Shape

Using short-time sliding window we can obtain a different Fourier transform for each time segment where the window is centered. However, as the *rectangular window* trims the signal amplitude at different phases, it causes the spectral leakage in the frequency domain. Therefore to avoid side lobes around dominant frequencies, it is required to use some more sophisticated window function, typically *Hamming* or *Hann window*. In figure 2.3 are shapes and corresponding equations of commonly used windows.

Window Size

When considering window size, there is a **trade-off** between time versus frequency precision. Narrow windows offer precise temporal localization and ensure that the portion of the signal falling within the window is stationary. On the other hand, frequency resolution is poor, because wide range of frequencies maps into one discrete spectral bin.



RECTANGULAR $w(n) = 1$

HAMMING $w(n) = 0.54 - 0.46 \cos\left(\frac{2\pi n}{N-1}\right)$

HANN $w(n) = 0.5 \cdot \left(1 - \cos\left(\frac{2\pi n}{N-1}\right)\right)$

Figure 2.3: Most commonly used window functions.

The resolution problem gets even worse when working with music. FFT spectrum is linearly scaled, while human ear perceives sounds logarithmically. It can happen that several low tones are mapped into the same frequency bin making them unable to distinguish.

- **Narrow window** → good time resolution, poor frequency resolution
- **Wide window** → good frequency resolution, poor time resolution

This fact can be explained by **Heisenberg's Uncertainty principle** expressed by formula 2.16. Time resolution Δt is the minimal distance between two spikes in time separated from each other in the transform domain. Frequency resolution Δf is the minimal distance between two spectral components separated from each other in the transform domain. Main consequence of this principle is that Δt and Δf cannot be arbitrarily small [8].

$$\Delta t \cdot \Delta f \geq \frac{1}{4\pi} \tag{2.16}$$

In specific analysis of piano music, we need to think about experienced players who are able to play series of tones very fast. Hence, the upper limit of window length lies at around 150 ms. At the same time, there's not any big room for improvement - window cannot be much shorter. With the same length, spectral resolution is around 7 Hz, and two neighboring tones are already starting mapping to the same frequency bins just at frequency around 110 Hz (A2 tone).

A common technique for increasing time resolution is window **overlapping**. If we set the overlap rate to 50 %, we shift window only by half of its size after each DFT. Time resolution gets higher by factor of two, while window length stays the same.

2.4 Dynamic Time Warping

Dynamic time warping (DTW) is an algorithm for aligning two time series, usually of various lengths, as it is depicted in figure 2.4. It has been successfully used in areas like speech recognition, gesture recognition, shape matching, and many others. The main advantage of this technique is no need for training.

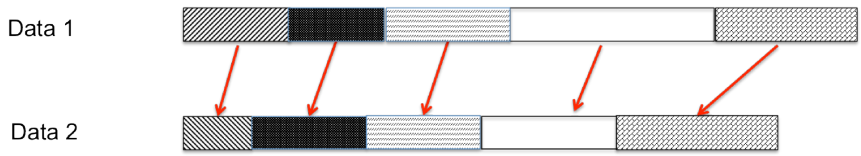


Figure 2.4: Alignment of two time-dependent series. [17]

In score following, DTW is used for matching music score to audio, and obtaining the actual position in the score. The process consists of the following steps [2]:

1. Conversion of score and audio to common representation using feature extraction (see Chapter 3).
2. Calculation of similarity between score and audio feature vectors using suitable distance function.
3. Computation of the optimal path with respect to the global distance (cost).

Offline DTW

Original offline DTW algorithm is well known for a long time. It is suitable for off-line alignment, when the both time series are known in advance. The series are represented by sequences of feature vectors $U = u_1, \ldots, u_m$ and $V = v_1, \ldots, v_n$.

To find the most optimal alignment we need to compute the $m \times n$ matrix $d_{U,V}(i,j)$ of local distances. This matrix contains Euclidean distance (or any other similarity measure) for every pair of feature vectors (u_i, v_j).

DTW seeks minimum cost path $W = W_1, \ldots, W_i$ such that W_k is an ordered pair (i_k,j_k). Each element in this pair corresponds to positions in one of the time series. Global alignment then can be perceived as series of these matches.

Path W is constrained by several criteria [2]:

- is **bounded** by the both sequences (cannot get out of the matrix).
- is **monotonic** – it can only increase or stay on the same level.
- is **continuous** – the alignment is defined in every time.

To compute alignment path W, the cost matrix D needs to be constructed, which contains the sums of local minimum costs from beginning to the current position. It is usually defined by recursive formula

$$D(i,j) = d(i,j) + \min \left\{ \begin{array}{l} w_a \cdot D(i,j-1) \\ w_a \cdot D(i-1,j) \\ w_b \cdot D(i-1,j-1) \end{array} \right\}, \tag{2.17}$$

where $w_a = 1$ is a weight for going to the horizontal or vertical direction and $w_b = 2$ for diagonal direction, $D(i, j)$ is the cost of the minimum cost path from $(1, 1)$ to (i, j). Initial element D$(1, 1)$ is set to $d(1, 1)$. After the cost matrix is filled, the path W is extracted by backtracking from $D(m, n)$.

Online DTW

The previous technique is unfeasible in case of online score following, because one time series is just partially known. Therefore, a few modifications were proposed by Dixon [11] to adapt DTW for real-time tasks.

To run in real-time, only the constant part c of the cost matrix is computed. At each time t, we seek the best alignment $u_1, \ldots, u_t$ of partially unknown sequence U to some initial subsequence of V. Algorithm starts at position $(t, j) = (1, 1)$ and gradually expands the cost matrix. First, the minimum path cost is found for the cells in the current row and column. If it is found at the current position (t, j), both the next row and column are calculated. If it occurs in row j, the next row is calculated. If it occurs in column t, the next column is calculated.

Example of this expansion procedure is visible in figure 2.5. Exact definition including pseudo-code can be found in original Dixon's paper [11].

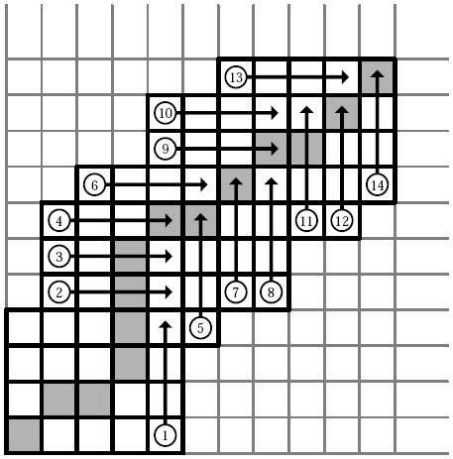


Figure 2.5: Example of online DTW with search window $c = 4$ shows the order of evaluation. Numbers denote the computation order of individual rows/columns. Optimal path is colored gray. [2]

Alternatives

Dynamic time warping and *Hidden Markov models* (HMM) are two most common approaches for score following. Each of them has its pros and cons.

HMM has proven to offer increased accuracy in performances with large amount of wrong or missing notes. On the contrary, it requires a exhaustive pre-modeling of the score and training of the system, which leads to higher complexity and time consumption.

DTW is simpler and more flexible, because it doesn't need any training and discretization of problem to the sequence of states. On the other hand, fine tuning of internal parameters is needed for good performance.

Chapter 3

Music Features

In ideal case, mapping of time data into frequency domain would be sufficient to reconstruct back played notes. However, the situation is complicated by various phenomena like noise in signal, harmonic overtones appearing together with the fundamental frequency, or full spectrum distortions caused by note onsets.

To suppress these undesirable effects and better approximate the score representation, several types of feature extraction methods have been developed, which will be described in this chapter.

3.1 Note Pitches

Note pitches are very simple to extract, but provides comparable performance to chroma representation, which is described further. It originates from MIDI note numbers, but in this case it is limited only to notes on piano keyboard. For each note we compute the range bounded by minimum and maximum frequency, and map all the FFT spectrum bins in this range to corresponding pitch.

Main advantage of this approach is high information density and direct mapping of sound to MIDI note numbers, which simplifies consequential matching stage. On the other hand, there is a problem with spectral bins that are overlapping on lower pitches.

3.2 Chroma and Octave

This feature set was described in paper by Chu and Li [9], which is introduced in section 2.2. It is based on mapping frequency ranges to musical tones.

Musical scale contains 12 semitones (C, C#, D, D#, ...) with increasing pitch. Set of these 12 semitones forms an octave, each containing tones with twice as higher frequency than the previous octave. Chroma class then contains all pitches with squares of frequency of a fundamental pitch.

It can be demonstrated on the helix model, where as the pitch moving along the helix, it passes through individual semitones. After the turn it reaches the initial position, just one octave higher. Therefore, two different music notes can be grouped into the same chroma class if their corresponding frequencies have some relationship (one is multiply of the other).

In figure 3.1 you can see three different music parts with the same sequence of chroma vectors (chromagram). It means that some sequences of notes may have the same chroma representation – the feature vector with 12 chroma values is ambiguous. Hence it can be

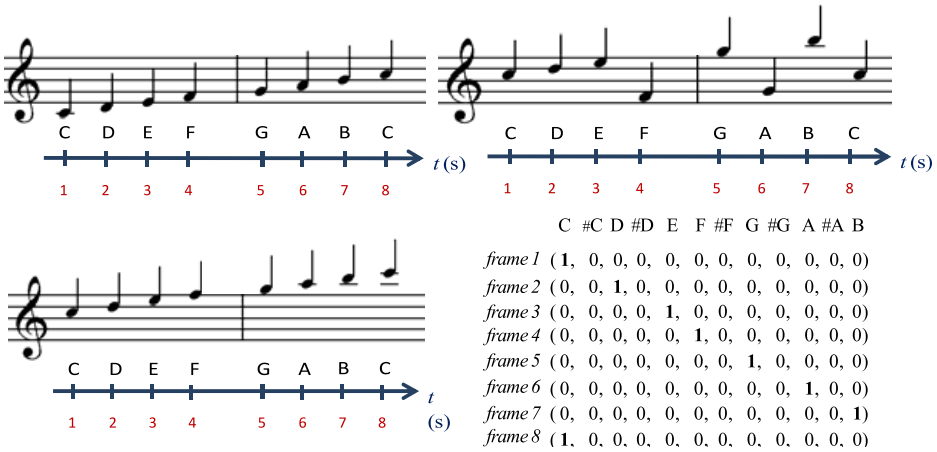


Figure 3.1: Three music segments with the same chromagram. [9]

helpful to supply another 8-dimensional octave vector holding information about absolute pitch.

To extract the features from audio, the data is first transformed into frequency domain using FFT. Then this domain is divided into eight regions, each region contains twelve chroma bins. The algorithm go through all the regions and accumulate energy to the corresponding bins in chroma and octave feature vectors.

Let $E_{f,k}$ be the energy of the k-th frequency bin at frame f, and $E_{f,r}$ be the energy of the r-th region in frame f. Then average energy of the region is calculated as $AvgE_{f,r} = E_{f,r}/12$. Finally, we compute the emphasized energy of the frequency bin k as

$$\hat{E}_{f,k} = \begin{cases} w_e \times E_{f,k}, & \text{for } E_{f,k \in S_r} \geq AvgE_{f,r} \\ E_{f,k}, & \text{otherwise} \end{cases} \tag{3.1}$$

where $w_e \geq 1$ is emphasis weight, and S_r is a set of all frequency bins in the region r.

After pre-emphasis, we can compute the chromagram for each pitch family h and octave vector for each octave g as

$$\tilde{E}^c_{f,h} = \sum_{k \in S_h} \hat{E}_{f,k} \tag{3.2}$$

$$\tilde{E}^o_{f,g} = \sum_{k \in S_g} \hat{E}_{f,k}, \tag{3.3}$$

where S_h is a set of frequency bins of the chroma family h, and S_g is set of frequency bins which belongs to octave g. Finally, in order to get normalized chroma vector $\tilde{\mathbf{q}}^c_f$ and octave vector $\tilde{\mathbf{q}}^o_f$ at music frame f, we calculate each its k-th bin:

$$\tilde{\mathbf{q}}^c_{f,k} = \frac{\tilde{E}^c_{f,k}}{\sum_{j=1}^{12} \tilde{E}^c_{f,j}} \tag{3.4}$$

$$\tilde{\mathbf{q}}^o_{f,k} = \frac{\tilde{E}^o_{f,k}}{\sum_{j=1}^{8} \tilde{E}^o_{f,j}} \tag{3.5}$$

3.3 Locally Normalized Chroma Onsets

These features were introduced by Ewert [12] for off-line audio synchronization and were also used later by Arzt [6] for on-line score following. They are motivated by the observation that for musical instruments such as the piano, playing a note is manifested by rapid energy increase.

The audio signal is transformed by FFT to frequency domain and decomposed into 88 subbands, which correspond to the notes A0 to C8. Only increases in energy are stored in each bin relative to the previous frame (plot **(c)** in figure 3.2).

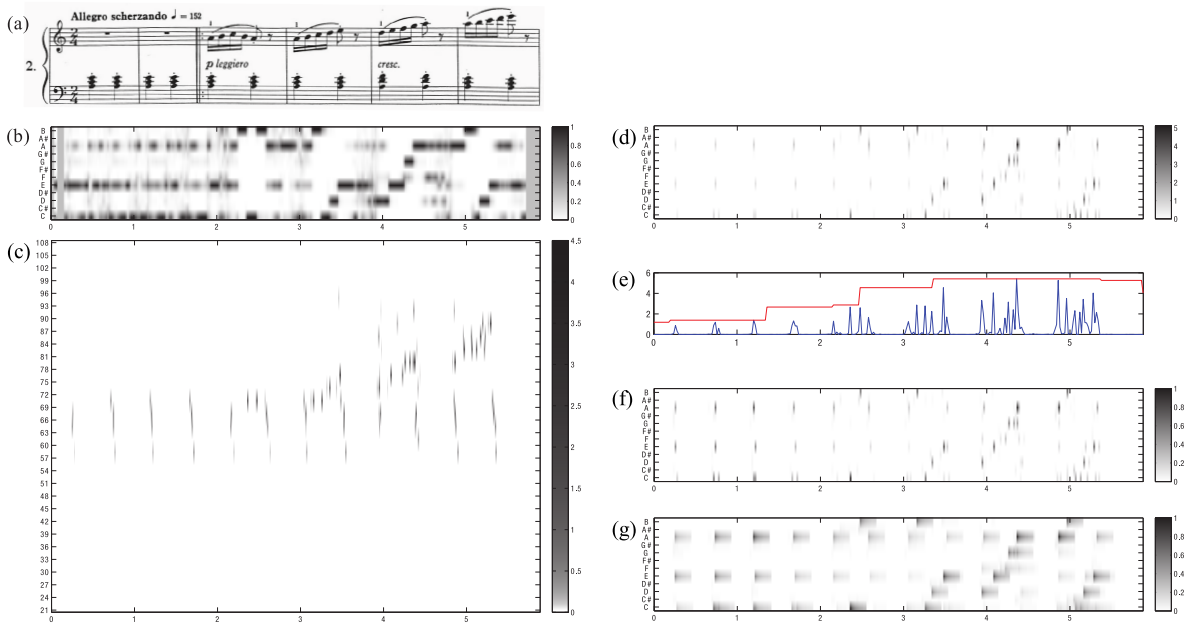


Figure 3.2: **(a)** Original music score. **(b)** Normalized chroma representation. **(c)** – **(g)** Gradual transformation to LNCO features. [12]

Features belonging to the same chroma class are added up to improve robustness of onset features. Before it, the logarithm of each pitch value is computed which simulates the logarithmic sensation of sound intensity. We will call the resulting features as CO (chroma onset) features – figure 3.2 d.

To make these features invariant to variations in dynamic, we employ normalization strategy which adapts to a local maximum intensity. First, we compute norm of each CO vector, which lies in window of suitable length ranging from current position to the left. Then we select for each position maximum norm in the window (red curve in figure 3.2 e, and divide the sequence of CO features in the window by this norm (figure 3.2 f). The resulting features are called *LNCO (locally adaptive normalized chroma onset) features.*

Authors state that this approach, when they first compute onset for all the pitches and merge them to chroma classes later, is more successful than opposite strategy of first getting chroma features and then computing their onsets. The reason for that is, when we merge one sharp onset with another more blurry onset, the valuable sharp one may get blurry as well.

Onset features express *attack phase* of played note. Authors also try to model *decay phase* by copying onsets multiplied by decreasing weight in sequence n times (they chose $n = 10$ and weights $1, \sqrt{0.9}, \sqrt{0.8}, \ldots, \sqrt{0.1}$). They refer to this representation as *DLNCO (decaying LNCO) features.*

3.4 Locally Normalized Semitone Onsets

These features were presented in paper by Ewert [12]. Later, they were used in score following system by Arzt [6] described in section 2.2. They are officially called *LNSO (locally adaptive normalized semitone onset) features.*

Procedure of obtaining these features is very similar to the previous Locally Normalized Chroma Onset feature extraction (section 3.3). The only difference is that instead of merging pitch onsets to chroma classes (figure 3.2 c,d), we leave them as they are and we work all the time with the whole set of 88 pitches.

Arzt in his research paper [6] states, that these LNSO features reach better performance in on-line score following than LNCO features.

3.5 Synthesized Spectral Templates

Spectral model is a prominent tool for estimating the similarity between score and audio. Several systems have used it as a generic templates for modeling the expected tonal content according to the score. To improve the alignment quality, it is needed to incorporate instrument-specific properties. A novel proposal was given in paper by Korzeniowski [13], which will be introduced in this section.

The expected tonal content is shaped here as a magnitude spectrum produced by short-time Fourier transform (STFT) on audio stream. Using frame window of length N_w, the resulting feature vector has $N_w/2$ bins.

There are two levels of spectral templates: *note templates*, which are simple spectral images of individual notes, and *score templates*, which include all sounding notes at a specific score position. Process of generating note templates and their composition into score templates will be now explained.

Note Templates

Note spectral templates are basic building blocks in most state-of-art score followers [13]. They can be either generated using Gaussian mixture models (GMM) or synthesized from audio samples of specific musical instrument.

The GMMs use Gaussians to model fundamental frequency plus several harmonics of a tone. They work well and are able to generalize on one instrument specifics to some degree. However, instrument adjusted synthesized templates are assumed to improve the alignment even more. Especially, if they would have been adapted on the fly using latent harmonic allocation, which is still impossible because of high computation cost. Figure 3.3 shows the harmonic structure of GMM and synthesized templates for two different notes. In the upcoming text, only synthesized templates will be studied since they proved to perform better.

To extract template for each note, a software synthesizer is utilized to generate sound from MIDI file. These sounds are analyzed using STFT and resulting spectrum is averaged

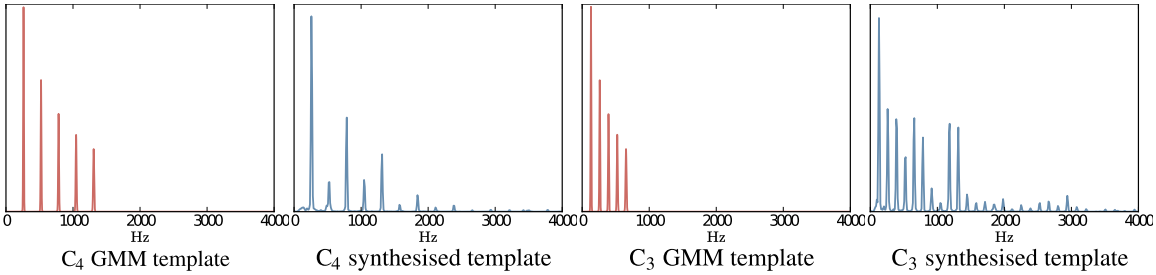


Figure 3.3: Differences between GMM generated and piano synthesized spectral templates for a single note. [13]

over duration of each note. Resulting vector has the form:

$$\phi_S^g = (z_1, \ldots, z_{N_b}), \tag{3.6}$$

where z_i is mean of the i-th frequency bin in the magnitude spectrum.

Score Templates

Sound synthesizers are using ADSR envelope (see Figure 3.4) to model volume of generated tones. Different instruments are characterized by different ADSR envelopes. This sequence of four phases includes:

1. **Attack:** Time between the note is activated and when it reaches the initial maximal volume.
2. **Decay:** How fast the volume decreases until it gets to the tone sustain volume.
3. **Sustain:** Time during the musician holds the tone and volume is at constant level.
4. **Release:** How fast the volume falls down to zero after musician stopped playing the note.

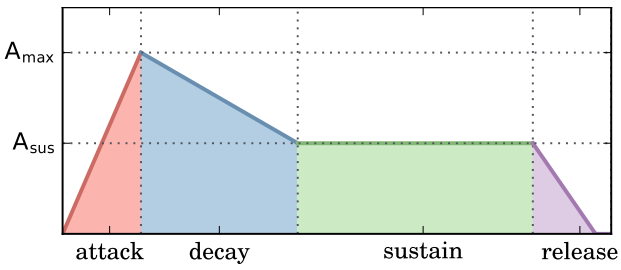


Figure 3.4: Generic ADSR envelope typical for instruments with percussive onsets. [13]

Since we are focusing on piano, we try to simulate envelope, which the most resembles piano sound. The attack phase will be ignored, because it is usually very short. The rest is simulated by two combined weighting function as

$$\psi(x, v, g) = \psi_{ds}(x, v, g) \cdot \psi_r(x, v, g) \tag{3.7}$$

where x is score position in beats, v is tempo in beats per second, and g is note. The decay and sustain phase are defined by ψ_{ds} and release by ψ_r. Both depend on the time of note start and note end, and presently played score position. Therefore, we define time between note start and score position Δ_s and note end and score position Δ_e as

$$\Delta_s(x, v, g) = \Delta_s = \frac{x - s_g}{v} \tag{3.8}$$

$$\Delta_e(x, v, g) = \Delta_e = \frac{x - e_g}{v} \tag{3.9}$$

where s_g is note start position, and e_g is note end position in beats. Then, decay-sustain weight and release weight are defined as

$$\psi_{ds}(x, v, g) = \begin{cases} 0 & \text{if } \Delta_s < 0 \\ \max(\lambda^{\Delta_s}, \eta) & \text{else} \end{cases} \tag{3.10}$$

$$\psi_r(x, v, g) = \begin{cases} 1 & \text{if } \Delta_e < 0 \\ \max(1 - \beta \cdot \Delta_e, 0) & \text{else} \end{cases} \tag{3.11}$$

where $\lambda = 0.1$ is decay parameter, $\eta = 0.1$ is sustain weight, and $\beta = 20$ is the release rate. Figure 3.5 shows the plots of both weighting functions. The decay-sustain weight ψ_{ds} models the exponential loss of energy after initial note attack together with volume preservation when the note is on. The release weight ψ_r expresses sudden cutoff when the note is released. Combination of these two function quite accurately approximates ADSR envelope.

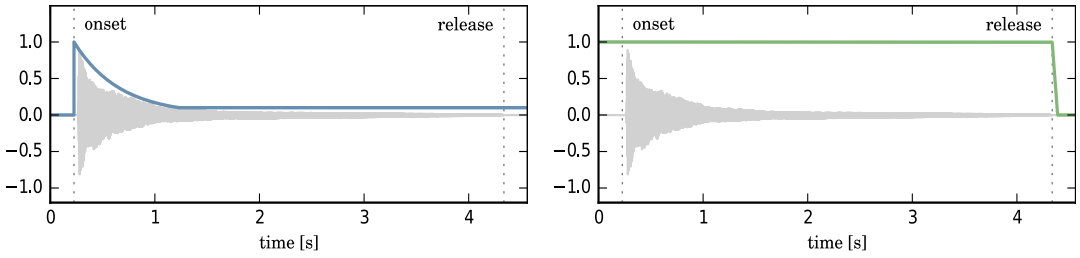


Figure 3.5: **Left**: Decay-sustain function ψ_{ds}. **Right**: Release function ψ_r. [13]

Finally, to compute spectral template Φ for all the combined notes at specific score position x with tempo v, we add up all individual note templates with weighted sum as

$$\Phi(x, v) = \frac{1}{Z(x, v)} \sum_{g \in G} \psi(x, v, g) \cdot \phi(g) \tag{3.12}$$

$$Z(x, v) = \sum_{g \in G} \psi(x, v, g) \tag{3.13}$$

where G is set of all possible notes and $\phi(g)$ is unweighted spectral template for individual note g.

As stated before, this approach works best for instruments with percussive onsets like piano. For other instruments, especially the ones which allow the musician to continuously control the volume, it is difficult to define specific envelope.

Chapter 4

Score Follower Implementation

Before creating the real Android application as is written in thesis specification, it is worthwhile to create a simple demo app to verify the system functionality like feature extraction and score following algorithm. Otherwise it would be tough to debug these key components on mobile device. This section explains internal parts of the solution and problem decomposition into program blocks. In the end it mentions tools and libraries, which were used during development.

4.1 Early prototypes

The first prototypes were written in C++ and are based on implementation of Score follower using chroma and octave representation by Simon Zaaijer[1]. Even though I rewrote many of its core parts, it serves well as score following framework and provides important insight into the issue. I have created several versions, each using different feature set, which will be now described.

Original demo used microphone to record live music. This wasn't much handy for testing purposes, so I modified it to read the music data directly from WAV files instead. After the application loaded the WAV data, it starts to play it. At the same time, for each time frame it takes the corresponding part of audio buffer and executes FFT on data scaled by Hann window. The resulting spectrum magnitude is sent to feature extraction method.

This early implementation used very simple DTW algorithm. In order to perform real-time dynamic time warping, it employed only the latest audio vector available instead of the whole matrix. Because of this simplification, the performance was fluctuating under different audio tempos and the overall system was sensitive to internal parameters. Assuming that DTW cost matrix has columns as score position and rows as audio position, these parameters are:

- **Advance enforcement** is used to prevent DTW from getting stuck in one place. This parameter decreases cost of the column next to the current position in the DTW cost matrix, and therefore supports the position to move forward.
- **Seek width** specifies the range of a parabolic proximity enforcement (next variable).
- **Proximity enforcement** is a variable that controls how aggressively should system stay close to the current position. It decreases the value of columns nearby in the cost matrix, and prevents the position from jumping arbitrarily all over the score.

[1] http://www.liacs.nl/~szaaijer/api/

- **Repeats** states how many times is one step of DTW repeated after feature extraction. It helps the system to follow the faster passages easily, because it allows to go over several columns in the same row of DTW cost matrix.

As you can see in figure 4.1, the Advance enforcement is the most problematic parameter, because it needs to be customized for each audio tempo. Left part plots the mean alignment error for different advance enforcement values when using Chroma and Octave as features (Approach 1), right part uses Note Pitches as features (Approach 2). The optimal value of the parameter for reference tempo is around `0.05`.

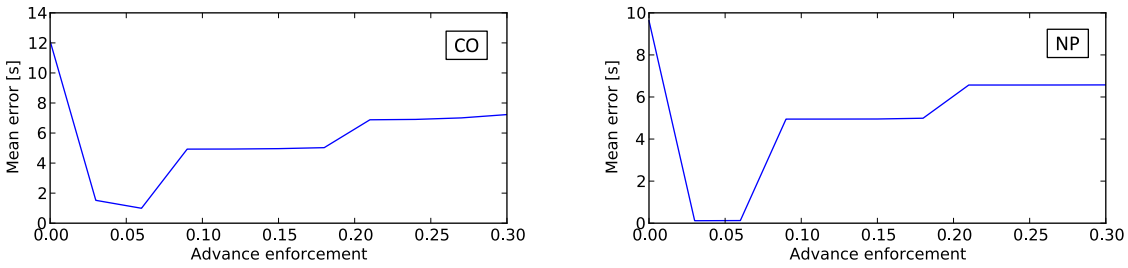


Figure 4.1: Precision of the alignment depending on Advance enforcement.

Approach 1: Chroma and Octave

This version is based on 12 music chroma and 8 octave features, as described in original paper by Chu and Li [9] and in section 3.2. First, the MIDI file is loaded and parsed, and chroma and octave vectors are directly extracted to discretized time frames. If a note is contained in bin just partially, only the proportional value is added to the current bin. Each note is also multiplied by its MIDI velocity to better mimic the audio features, which are later gradually extracted from WAV file.

Before each DTW alignment, the chroma and octave features are separately normalized using Euclidean norm. Then the squared Euclidean distance is computed between actual audio feature vector and each of the MIDI vectors. After the DTW execution, the position in score is returned.

Approach 2: Note Pitches

This version is variation of the previous one and results from features described in section 3.1. Instead of using two feature vectors, there is only one with 88 absolute note pitches resembling piano keyboard.

Note pitches features are more computationally expensive, because before each DTW we need to compute distances between two vectors with 88 elements instead of 20 as in previous method. It also requires much more memory to store these vectors.

Approach 3: Chroma and Semitone Onsets

Adaptive locally-normalized semitone onsets together with chroma were proposed in paper by Arzt [6] and are described in section 3.4. For maximum similarity between reference and audio features, the MIDI file is first synthesized to audio and then the same feature extraction method is used for both audio and score.

Although this approach had looked promising, it didn't bring any interesting results. Recognition performance was slightly better or almost the same as in previous prototypes. It could be because of some bug in code, but I am inclined to believe that it was due to the simplicity of the matching algorithm. Moreover, this method requires MIDI to audio synthesizer, which adds another layer of complexity.

4.2 Application Architecture

The final solution was completely rewritten from the group up. Especially the online Dynamic time warping algorithm needed radical change since it significantly influences the matching behavior. I also decided to switch from C++ to Python as a target language. It allows fast prototyping without need to compile source code, it contains useful libraries such as math library NumPy with FFT, and it also makes possible to directly visualize data using interactive plotting library Matplotlib.

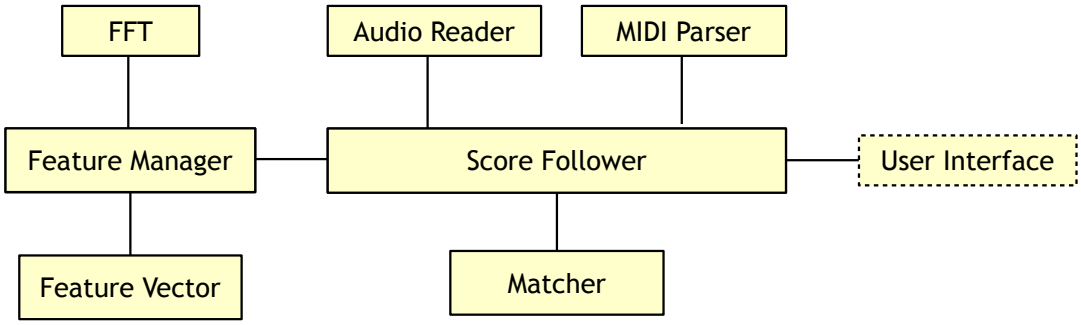


Figure 4.2: Architecture overview of final score follower

Architecture schema with modules and their dependencies is shown in figure 4.2. The main difference between this and previous approaches is that Matcher is now individual unit, which contains sophisticated DTW algorithm and communicates with *Score Follower*. Application also no longer displays sheet music nor *Audio Reader* plays music in real time. Instead, it process the audio file as fast as possible and plots the DTW alignment path together with cumulative alignment error. It serves much better for developing purposes than previous interface.

Below follows an overview of all modules used in the demo application. Less important units are shortly described while key components will be studied in detail in following separate sections. Some modules such as *Locator* or *Decision Maker* are mentioned even though they haven't been deployed with application yet. These parts are not critical for functionality and they are supposed to be plugged-in in the future.

- **Score Follower** – This is the control part of application, which drives data flows among all the subcomponents. First, it receives a two file URIs: MIDI file location and corresponding WAV file location. Then it provides the MIDI file to *MIDI Parser* and audio file to *Audio Reader*. MIDI file is parsed, list of notes is returned back, and through *Feature Manager* is converted to a list of *Feature Vectors*. Finally, it starts reading audio frames one by one from *Audio Reader* and converting them to *Feature*

Vectors. Each vector is passed to *Matcher*, which computes the current position in music score. After that, *Score Follower* notifies *User Interface* about the new position.

- **Audio Reader** – It reads the audio files and is further described in section 4.3.
- **Feature Vector** – It stores features needed to find matching between MIDI and audio. Detailed description follows in section 4.4.
- **Feature Manager** – It cares primarily about audio and MIDI features extraction and is further described in section 4.5.
- **Matcher** – It executes the matching algorithm and is discussed in section 4.6.
- **MIDI Parser** – As the name implies, it processes the MIDI files and return all the occurring notes. MIDI is binary format, which consists of usually several tracks. Each track contains messages which inform about events that happened in the lifetime of the particular track. It can be volume adjustment, tempo adjustment, pedal press or release (see MIDI specification [7]). We are interested in events `NOTE_ON` and `NOTE_OFF`, which delimit the duration of one or group of notes. Each note is then stored to a list as structure of: note number (pitch), start time and end time in seconds, and velocity (note volume).
- **FFT** – Fast Fourier transform is used just before feature extraction. This demo application utilizes discrete Fourier Transform for real input from library NumPy. Resulting spectrum is complex, hence it is needed to take magnitude of each complex sample to transform it to real numbers. Android application uses its own DFT algorithm, which is described in detail in section 5.1.
- **User Interface** – UI is more domain of mobile application. Only graphical element of the demo app is a plot of alignment path displaying error between score follower alignment and ideal alignment. However, there is another interactive plot in *Matcher* module, where by turning on the debug flag, it is possible to see current position and backward path in the DTW cost matrix.

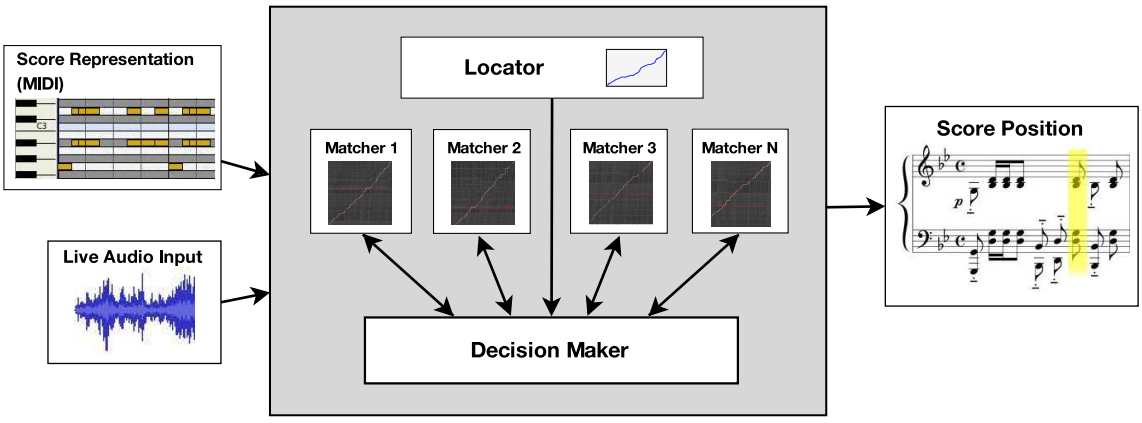


Figure 4.3: Architecture with multiple Matcher instances inspired by Arzt [4].

Below are mentioned yet two another modules, which hasn't been used in final application even if they were implemented, because they didn't bring satisfactory results. However, this couple isn't critical part of the system and can be temporarily left out. They will require more fine tuning to be deployed into the system in future.

- **Locator** – This module estimates the rough position in score. It cooperates with *Decision Maker* and gives it a clue where the player location currently is. It performs the standard DTW algorithm in time window centered around the actual position. Each time when the position is changed, window has to be recomputed. Since it would be expensive to do this on original feature set, *Locator* first downsample the features by the factor of six. It means that it averages six consecutive feature vectors into one. In this way it go approximately through two iteration per second.
- **Decision Maker** – Its task is to manage a set of *Matcher* instances (four in original paper), which tracks the score from different positions. At any time only one instance is marked as trusted and represents the system's current belief about the score position. *Decision Maker* can reassign inactive instances to a new position close to the *Locator* estimation. This usually happens when cumulative alignment cost is higher than some threshold. These two modules were inspired by score follower architecture presented in paper by Arzt [4]. A possible arrangement of these components in the system is depicted in figure 4.3.

4.3 Audio Reader

Audio Reader is component which opens WAV files, decodes the PCM audio data, splits them into chunks and continuously streams it to the Score Follower. In case of Android application, this module is replaced by Audio Recorder, which handles the audio data from device microphone. More information about audio processing on mobile app can be found in section 5.1.

Audio Reader works with a window sliding over the audio buffer. Window has constant length specified by **window size** parameter and starts at the beginning of the buffer. Audio Reader copies data from audio buffer to the window and returns it to the output. After the window is processed by some other module, it shifts the window to the right by some fixed offset and repeats the procedure. How far is shifted depends on **window overlap** parameter - if the window overlap is 25 %, window is shifted by 75 % of its size. That is often used to improve temporal resolution of STFT.

Important parameter which affects the overall follower recognition and computational performance is **sampling rate**. It is the number of discretized audio samples per second. It must be at least twice as high than the maximum expected music frequency. Common sampling frequency for CD music is 44 100 Hz. It was designed to cover the human hearing range from 20 Hz to 20 kHz. However, it is unnecessarily high for purposes of score following. Piano with 88 keys produces frequencies ranging from 27.5 Hz (A0) to 4 186 Hz (C8). Hence given Shannon theorem, the lowest sampling rate which is both at least double of the highest frequency and is supported by hardware of mobile devices is 11 025 Hz. All the audio parameters values are listed in table 4.1.

Parameter	Value
Audio encoding	PCM 16b
Audio channels	1
Sampling rate	11 025 Hz
Window size	1024 samples
Window overlap	50 %

Table 4.1: Audio Reader parameters

4.4 Feature Vector

Feature vector serves as storage for extracted audio features. It also contains related methods such as normalization and vector distance calculation. It is one of the crucial aspects that contribute to the total score follower performance.

Structure

After many experiments with different types of audio features it turned out that plain **chroma classes** work the best. Twelve elements long chroma vector has a good ability to generalize and outperforms even the individual music pitches. Moreover, since the chroma vector is quite short (about seven times shorter than the pitches vector), it is also much faster to compute distance between two vectors of this type. And the distance function is heavily used during the matching process.

Chroma vector is a good foundation for feature vector. But itself it only represents distribution of harmonics in every music frame. Equally important is also **note onset**, which expresses a time when the note is activated. When the player pushes the piano key, energy of the particular chroma class immediately rises. While the note is still pressed, this energy is continually decreasing until the key is released. Given these information, the note onset is possible to compute as half-wave rectified first-order difference. The precise procedure is commented in section 4.5 and is inspired by similar approach for local normalization described in section 3.3. The overall composition of features is depicted in figure 4.4.

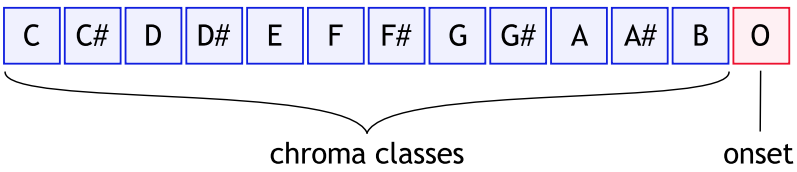


Figure 4.4: Internal structure of the feature vector.

Distance function

To evaluate similarity between audio-extracted and midi-extracted feature vectors, it is necessary to introduce a distance function. It takes two feature vectors and returns a scalar value which indicates how much these two vectors differ from each other. Similar vectors will produce a value close to the zero while completely different vectors should produce a high value.

Even though it is possible to compute distance of the whole feature vector at once, it is desirable to do it for each vector segment separately - here namely chroma classes and onset. Then the function can return a weighted sum of all the individual components. This allows us precise fine-tuning in the end by changing the appropriate weights.

I have implemented three different distance metrics: L_1 (or Manhattan), L_2 (or Euclidean) , and normalized L_1. Best results for chroma vector were achieved by normalized L_1 distance, which slightly surpassed L_2 distance. Since this function is frequently used, we can reduce execution time of normalized L_1 distance by precomputing vector norms used in denominator.

$$\mathrm{d}_1(\mathbf{p},\mathbf{q}) = \sum_{i=1}^{n} |p_i - q_i| \qquad \mathrm{d}_2(\mathbf{p},\mathbf{q}) = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2} \qquad \mathrm{d}_1^{\mathrm{n}}(\mathbf{p},\mathbf{q}) = \frac{\sum_{i=1}^{n} |p_i - q_i|}{\|\mathbf{p}\|_1 + \|\mathbf{q}\|_1}$$

L_1 distance | L_2 distance | Normalized L_1 distance

Onset distance is computed as absolute value of difference of two onset scalars. The total distance returned by distance function is calculated as

$$d = \omega_c \cdot d_c + \omega_o \cdot d_o \tag{4.1}$$

where d_c is chroma distance, d_o is onset distance, and ω_c and ω_o are chroma and onset weights. These weights were experimentally set to $\omega_c = 3$ and $\omega_o = 2$.

Normalization

Proper normalization is critical for any task success. In this case it reduces differences between various volume levels. If some note is played once quietly and hen loudly, the distance between relevant feature vectors will be without normalization very high despite the same frequency distribution. It is because chroma vector is accumulated in the latter case from a signal with higher energy and thus total values are higher than in the former case.

What truly holds information are not values but the distribution of chroma classes. In order to emphasize this distribution and minimize energy bias, every value in the chroma vector is divided by vector **norm**. Resulting normalized vectors are thereafter comparable among each other. Onset values are normalized separately as was mentioned before in section about feature vector structure.

I have tested two vector norms: L_1 (Manhattan) and L_2 (Euclidean). For the final application was picked L_1 norm, which performed better. Respective equations are written below.

$$\|\mathbf{p}\|_1 = \sum_{i=1}^{n} |p_i| \qquad \|\mathbf{p}\|_2 = \sqrt{\sum_{i=1}^{n} p_i^2}$$

L_1 norm L_2 norm

Quantization

Although chroma energy distribution after normalization is quite stable under various volume levels, there is still a room for improvement. For example, the chroma vector is little sensitive to different variations in the music articulation. To address this problem we further introduce quantization function $Q : \langle 0, 1\rangle \to \{0, 1, 2, 3, 4\}$ defined by equation

$$Q(x) = \begin{cases} 0 & \text{if } x < 0.05 \\ 1 & \text{if } x < 0.1 \\ 2 & \text{if } x < 0.2 \\ 3 & \text{if } x < 0.4 \\ 4 & \text{else} \end{cases} \tag{4.2}$$

Normalized chroma vector is quantized by applying Q to each of its component. This means that if there is a dominant chroma component with relative energy greater than or equal to 40 %, it gets assigned the value 4. Components with values under 0.05 are suppressed. Thresholds are chosen in the logarithmic fashion which mimics the natural characteristics of sound. This quantization process was adopted from work by Muller [15].

Except better matching performance, quantization also speeds up computation, because it transforms floating point numbers to integers, which are on typical CPUs faster to operate with. Furthermore, it also reduces memory space required to store feature vectors, since quantized values can be converted from 32-bit `float` type to 8-bit `byte` type.

Noise detection

Audio frames are recorded at constant rate and continuously coming into the score follower. These frames can be divided into two groups: music frames and noise frames. Noise frames mostly occur during silent periods when there is no music playing. Microphone has not enough input data to mask the background noise, and thus records useless values. Another much less frequent source of noise are various random sounds like clapping, cracking, or coughing. The task is to filter out these unwanted effects. For this reason every feature vector instance contains method that tells whether the vector comes from the music frame or noise frame.

First attempt I made was **energy-based detector**. It is a simple detector which adds up all the components of unnormalized chroma vector (which is the same as sum of magnitude of FFT spectrum bands) and takes a logarithm of the sum. Then it compares this value to threshold T and decides if the value is greater than T (music) or not (noise).

More sophisticated approach was achieved later by **harmonics-based detector**. It targets drawbacks of previous detector, which was volume-dependent and hence it doesn't have to behave correctly under different devices. Moreover, it also ignore various loud noises like clapping, which would have been falsely classified by the previous one. This detector examines the distribution of chroma components. It is typical for music frames that this

distribution is usually not uniform and one or more chroma classes are dominant. To find out how uniform the vector is, we take standard deviation of normalized chroma vector and compare it to threshold T. Then if the value is greater than T, it is the music frame, otherwise it is a noise.

4.5 Feature Manager

Feature Manager takes care of two types of transformation:

1. Extraction of feature vectors from incoming audio frames (audio features)
2. Conversion of MIDI file into the list of feature vectors (MIDI features)

Each of these tasks follows its own procedure which will be further in details explained. The main aim is to minimize differences between corresponding audio and MIDI features, which allows to achieve a better matching.

Audio features

Audio feature vector is generated for each audio frame coming from the Audio Reader. Frame is first multiplied by Hanning window and then passed to Fourier transform. Output complex spectrum is transformed to real values by taking magnitude of each complex sample.

First we initialize the pitch vector to very small numbers. This guarantees correct normalization even if the input is zero. Then we go through the all piano pitches and for each one add up the corresponding spectral bands. If the band is covered just partially, we take the respective part of it. To increase robustness each pitch ranges from lower frequency located exactly halfway between previous and current pitch to upper frequency located exactly halfway between current and next pitch. Python implementation of pitches extraction is described in algorithm 4.1.

To get chroma components, we add up all the pitches which belong to the same chroma class as it is explained in section 3.2. Resulting vector is normalized and quantized.

Onset is calculated on normalized chroma vector just before quantization. We first subtract component-wise previous chroma from the current chroma and add up all the positive differences. From this energy difference we subtract the same first-order energy difference calculated in the previous frame. Resulting second-order difference is final onset, which is yet normalized by maximum onset in a time window. Length of the time window can be set from one to a few seconds.

```
def get_pitches(self, bands):
    pitches = [1e-16] * (MIDI_HIGH + 1)

    # Go through all piano pitches
    for p in range(MIDI_LOW, MIDI_HIGH + 1):
        flow = 2 ** ((p - 0.5 - 69) / 12) * 440 * WINSIZE / SAMPLERATE
        fhigh = 2 ** ((p + 0.5 - 69) / 12) * 440 * WINSIZE / SAMPLERATE
        iflow = int(flow)
        ifhigh = int(fhigh)
```

```
            # Pitch lies within one band
            if iflow == ifhigh:
                pitches[p] += bands[iflow] * (fhigh - flow)
            # Pitch spans over several bands
            else:
                pitches[p] += bands[iflow] * (1.0 - (flow - iflow))
                pitches[p] += bands[ifhigh] * (fhigh - ifhigh)
                for f in range(iflow + 1, ifhigh):
                    pitches[p] += bands[f]

    return pitches
```

Algorithm 4.1: Extraction of pitches from FFT bands in Python

MIDI features

MIDI features are extracted all at once from the list of notes parsed from a MIDI file. Before extraction, index of notes is built for higher efficiency and easier manipulation. First we split the score timeline to the time frames similar to ones produced by Audio Reader. Then we map each note to its corresponding time frames and store in each frame a pointer to the note. Now it is possible to directly access all the notes occurring at the specific time.

After the index was established, algorithm walks through all the time frames and for each one composes chroma vector using synthesized spectral templates (section 3.5). Chroma vector is normalized and onset is obtained the same way as in case of audio features.

Although the original paper proposes extracting full spectral templates (including all FFT bands), I discovered that storing only extracted chroma classes gives almost equal results while the note reconstruction is much faster. Templates are generated using software synthesizer on prepared MIDI file with all the piano pitches arranged one after another. The resulting WAV file is converted the same way as audio features to the chroma vectors. All the vectors belonging to the one note are summed, normalized and stored into a binary file.

The main advantage of this technique is no need to manually synthesize each score we want to follow. Templates can be used generally for arbitrary music compositions. Furthermore, the templates file in binary form takes just a few kilobytes, and so the application can even contain templates for several music instruments.

Comparison of audio features and score features extracted from the same piece are displayed in figure 4.5.

4.6 Matcher

Matcher is the main component responsible for aligning audio data to their score representation. It receives fresh audio frames from Audio Reader and continually updates current position in the score. Using improved online dynamic time warping algorithm (described further) it can process arbitrary long musical performance with linear time complexity. This section also discusses several approaches that has been taken to make the alignment path smooth and robust to perturbations. Despite the fact that the internal parts of this module have been rapidly evolving over the whole development time, the current solution is quite mature and stable. However, there is still enough room for additional enhancements.

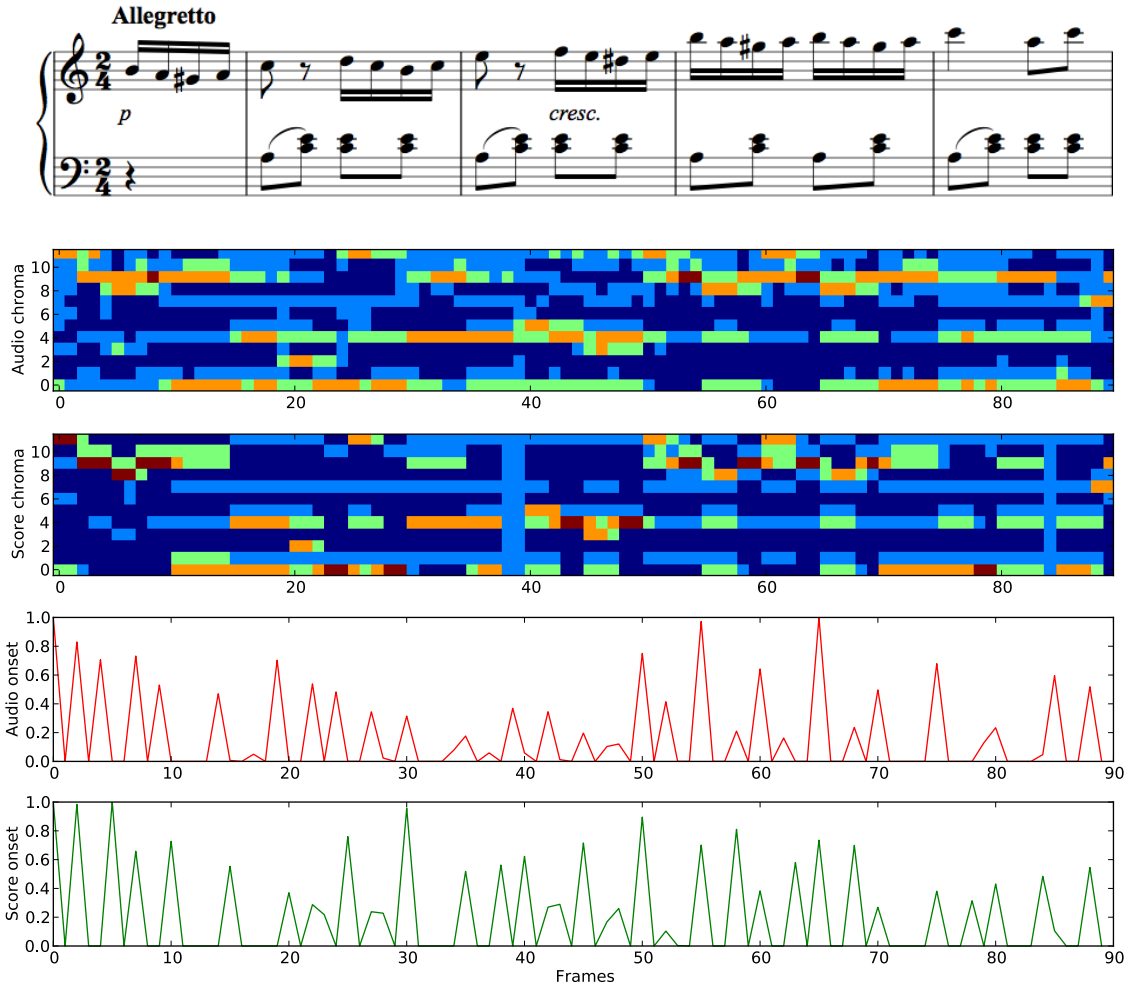


Figure 4.5: First five bars of Rondo Alla Turca with extracted audio and score chroma vectors, audio onsets (red curve) and score onsets (green curve).

Windowed Time Warping

The well-known Dynamic time warping works great for aligning two time series which are known beforehand and which are not excessively long. But when it comes to online audio-to-score alignment like in this case, when we know the score series beforehand but not the audio, we need to use different algorithm. The online DTW proposed in section 2.4 was implemented and tested. While it has worked quite good for some recordings, it gets easily lost in others or in the recordings played at faster or slower tempo.

To bring a more robust solution, I took ideas from the both offline and online DTWs and created concept which I called Windowed time warping. It uses only forward-path estimation similar to Dixon's online DTW, but computes the full cost matrix for a local area. Cost matrix window has fixed size and it shifts after alignment path reached its borders. The whole procedure consists of these steps:

1. Allocate cost matrix window M of size `WIDTH`.

2. Set audio index i, score index j and score offset f to zero.

3. Wait for incoming audio frame.
4. Compute accumulated distance for each cell in column i of matrix M.
5. Increase i and calculate appropriate j using tempo estimation (described in the next section).
6. If the i or j reaches window size, move the window along the path by `HOP`, update i, j and f and recompute the overlapping columns.
7. Go to step 3 until score finishes.

Good trade-off between precision and performance was found with window size `WIDTH` set to 100. In the worst case it takes 10 000 feature vector distance computations to fill the window or 2 000 computations for one second of audio. When the window is shifted, it moves along the alignment path just by its proportional part `HOP`. This is to suppress inaccurate values occurring with the first few frames and also to utilize already known audio data. The `HOP` variable has been experimentally set to 70 %.

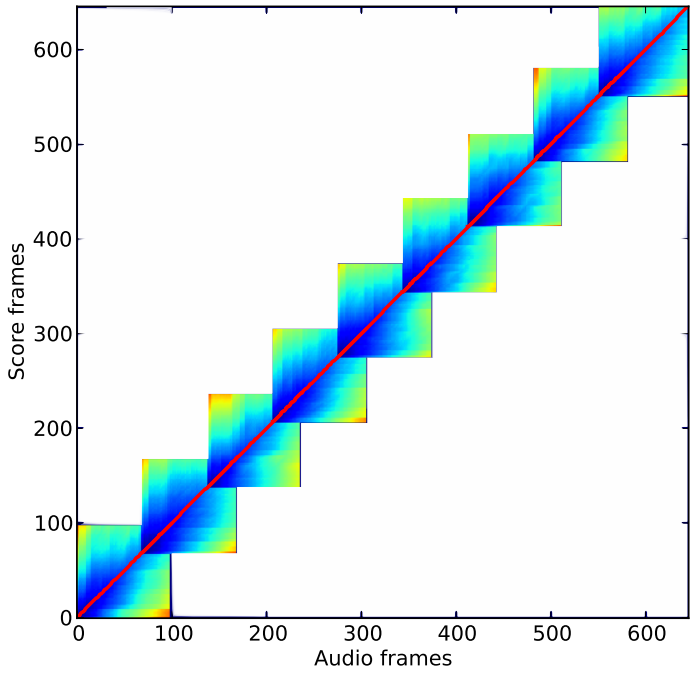


Figure 4.6: Examined segments in the global cost matrix using Windowed time warping.

Visual demonstration of how this method works is depicted in figure 4.6. Colorful overlapping squares are local cost matrices, in which blue color stands for lower cost while green to red colors represent higher alignment cost. Global alignment path is drawn by the red curve.

Tempo Estimation

In original offline DTW algorithm the alignment path is reconstructed by tracing neighboring cells with minimum cost backwards from end to start. Since this procedure cannot be used in online DTW, we need to obtain the path in different way. Suitable approximation can by computed by estimation of current tempo, which is the same as path slope in the cost matrix. As soon as the actual tempo and starting point are known, it is easy to reconstruct the alignment path.

For each incoming audio frame a new column in cost matrix window is computed. Then a cell with minimum value in this column is found. Vertical position of this cell divided by its horizontal position represents the current most probable path slope. In order to get the current **tempo**, which should be a robust variable, we take a weighted average of m slopes in a row (specifically $m = 25$). Slopes are weighted by horizontal position of appropriate column in the cost matrix window as explained by example 4.2. It is because the higher position involves information from more audio frames and thus it is more reliable.

Column	1	2	3	4	5
Slope	1.66	1.33	1.25	1.18	1.16
Tempo	$\frac{1\times1.66+2\times1.33+...+5\times1.16}{1+2+...+5} = \mathbf{1.24}$				

Table 4.2: Example of tempo computation using weighted average of slopes.

Path Stabilization

Besides tempo estimation, another mechanism to increase robustness and smoothness is path stabilization. It isn't integral component and can be omitted, but when employed it improves total precision of Matcher. It works as follows: take n consecutive tempos and compute standard deviation, if the deviation is smaller than threshold t_s, mark the path as stable. In my implementation $n = 50$ and $t_s = 0.015$.

As soon as the path gets stable, computation of a new slope changes. It doesn't look anymore for minimum value in all the cells of a new column in DTW cost matrix, but just in close neighborhood r around current position ($r = 5$). In this way a new slope value can't vary a lot from the previous one and therefore the total alignment path is smoother, especially with slow tempo.

4.7 Tools and Technologies

Without various third-party tools it wouldn't we possible to develop the demo and mobile applications in dedicated time. Here follows a short summary of all important tools and libraries which were used:

- **Python** scripting language allowed rapid prototyping of demo application without need to compile any source code. It performed reasonably fast thanks to native integrated libraries.

- **NumPy** is extension library for Python, which includes optimized scientific and mathematical modules for Fast Fourier transform, matrix creation and manipulation, or statistical functions.
- **Matplotlib** is a plotting library used for data and error visualizations. It proved to be very helpful, since a lot of time of this thesis was spent by measuring, analyzing, and evaluating the data.
- **TiMidity++** was used for MIDI to audio synthesis. This flexible command-line tool was useful to obtain a WAV file which exactly matches given MIDI file.
- **SoX** (Sound eXchange) is another command-line application. It was used for conversion among different audio formats and for change of sample rate and audio channels.
- **Audacity** is audio editor which was used for precise audio analysis, normalization, and trimming off silent parts at the beginning or at the end of audio recordings.
- **MIDIUtil** is Python library which offers simple MIDI files generation. It was useful during spectral templates extraction to create MIDI file with all individual piano pitches.
- **Midi Sheet Music** is Android application for displaying and playing MIDI files. It provided algorithm for converting MIDI to notation described in section 5.2.
- **Android SDK** together with Android Studio was used to develop and debug mobile application in Java programing language. More about this in chapter 5.
- **Texture Atlas Generator**[2] is command line tool that generated texture atlas with all the music symbols used in application. Texture atlas is then used for effective drawing of music notation using OpenGL.

[2] https://github.com/pjohalloran/texture-atlas-generator

Chapter 5

Mobile Application

After the demo prototype had been done, it was time to develop mobile application for Android, which is the main aim of this project. All the Python classes were rewritten to Java language and missing modules were programmed from scratch. Main effort was put to performance optimizations and low memory footprint, since mobile devices have limited resources as CPU, memory, and battery capacity.

5.1 Audio Processing

Audio Reader from the demo application was replaced by Audio Recorder module, because mobile app uses microphone instead of reading data from WAV files. Standard sampling rate for sound is 44 100 Hz. According to official Android documentation[1], every Android device should support this frequency. However, this value is unnecessarily high for needs of our music follower. For performance reasons it is good to examine other available values. Each mobile device supports besides 44 100 Hz a different set of sampling rates, but most of them is able to record sound also at 22 050 Hz and 11 025 Hz. The last mentioned value 11 025 Hz was chosen as default sampling rate for this application, because it is the closest frequency higher than double of the highest piano pitch C8 (4 186 Hz). It could happen that this default sampling rate is not available on a device. In that case application falls back to 44 100 Hz and dynamically downsamples each audio chunk by factor of four. This task is very fast and doesn't influence much total performance.

When the audio chunk is prepared, it is passed to Fast Fourier transform for frequency analysis. In original Python implementation, there was FFT module included in the NumPy library. Android platform doesn't have such a standard library and developers are forced to code their own or used external library. I used implementation of **Decimation-in-time Radix-2 FFT** by Douglas L. Jones[2] written in Java. It is highly optimized in-place algorithm for complex input. Real audio data are put to real array and imaginary array is set to zero. After computation, complex results are in these two arrays. Finally, magnitude of each complex number is taken to get FFT bands.

Even though the FFT is written in Java, it performs surprisingly fast. On smartphone Samsung i9000 Galaxy S from 2010 with 1 GHz single-core ARM Cortex A8 the running application takes just around 40–50 % of CPU resources. While performance is not a problem, energy consumption could be. Considering the user running application for several

[1] http://developer.android.com/reference/android/media/AudioRecord.html
[2] http://cnx.org/content/m12016/latest/

hours, battery level can drop rapidly. To target this issue I plan in future versions to employ some native FFT library such as FFTS[3], which can utilize ARM NEON instructions.

5.2 Music Notation Extraction

The current source of music data for this application are MIDI files. MIDI is a very simple format based on messages signalizing which event happened at what time. Messages are organized into tracks, where usually one audio source corresponds to one MIDI track. We are interested only in messages `NOTE ON` and `NOTE OFF`, which define lifetime of one specific note.

First step of MIDI parsing is to convert MIDI file to a list of notes. Each record in the list contains information about note start time, duration, velocity and MIDI note number. Duration can be calculated as difference between `NOTE OFF` and `NOTE ON` time. Velocity expresses how fast the piano key was pressed and corresponds to the note volume.

After these informations were gathered, the **second step** is to generate music notation. Each duration is quantized to the closest note length with respect to the MIDI time signature: whole note, half note, quarter note, eight note, etc. Notes are thens split up to the bars (music segments of the same length). It is also needed to assign accidentals (sharps, flats, naturals) according to given MIDI key signature.

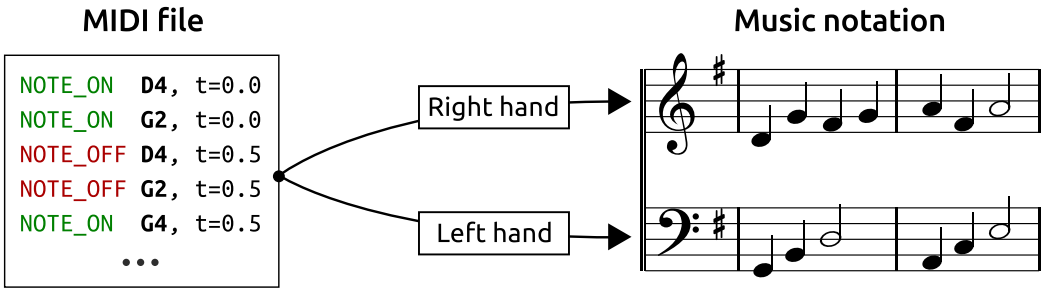


Figure 5.1: Generation of music notation from MIDI format.

The final **third step** consists of splitting one series of notes to two staffs for left and right hand as it is illustrated in figure 5.1. There are many algorithm trying to solve this problem of splitting, but non of them works perfectly, since there is not enough information for an ideal split. I used algorithm from the original Midi Sheet Music[4] notation module, which works as follows:

- If the note is more than octave from the highest or lowest note that starts exactly at this start time, choose the closest note.
- If this note is more than octave from the highest or lowest note in this note's time duration, choose the closest note.
- If the highest and lowest notes that start exactly at this start time are more than octave apart, choose the closest note.

[3] http://anthonix.com/ffts/
[4] http://midisheetmusic.sourceforge.net/

- If the highest and lowest notes that overlap this start time are more than octave apart, choose the closest note.
- Otherwise, look at the previous highest and lowest notes that were more than octave apart, and choose the closest note.

MIDI is established and ubiquitous music format. Its advantage is simplicity and easy manipulation. On the other hand, it is not well suited for sheet music rendering, because it doesn't specify how the notation should exactly look. Much better option that was designed from the beginning for music notation interchange is **MusicXML**. Is is XML-based file format, so it is easily readable by both computer and human. It splits by design note pitches into measures and staffs, provides information about music ornaments, beaming, and even stem directions. A lot of current music software supports it and also the number of sheet music in this format is increasing. For these reasons I plan to replace MIDI format by MusicXML in some future release.

5.3 Music Notation Rendering

When we have the logical representation of music representation ready, the next phase is to draw it to the display. First, all the measures are calculated such as note sizes, margins and stem directions. Then the music is split into as many staffs as needed to fit it to the size of screen with specified zoom level. Depending on the type of visual representation, it can be one long staff or several vertically aligned staffs. In the end every staff is justified to fill all the available space. This physical representation is stored in a hierarchical data structure, which offers easy access for renderer.

For the sheet music rendering I decided to use OpenGL for several reasons:

- It shifts the load from CPU to GPU, while CPU can handle more important computations like Fourier transform and score follower.
- Android 2D rendering pipeline supports hardware acceleration as from Android 3.0, but I wanted to support even older devices, which still had around 20 % of market share at the time of beginning of this thesis (autumn 2013).
- Application should allow fast scrolling and zooming, which requires high-performance renderer.

I use OpenGL ES 2.0, which is available on majority of Android devices. It is set to orthographic projection with depth test disabled. It requires to write vertex and fragment shaders, which are in this case simple and doesn't perform any transformations. Information about objects in scene is stored in vertex, drawlist and texture-coordinate buffers, which are used to bridge the gap between OpenGL native system library and Java virtual machine. Sheet music objects are during rendering accumulated in these preallocated buffers. After that, application calls the function `glDrawElements`, which draws everything on the screen.

I tried to utilize all the possibilities that OpenGL offers. Staff lines as well as note stems are represented as `GL_LINES`. All the other music symbols are stored in one big texture. This texture atlas is then loaded into graphics memory during initialization. To draw some music symbol it is just needed to map a particular area from the texture to a quad mesh. This technique is displayed in figure 5.2.

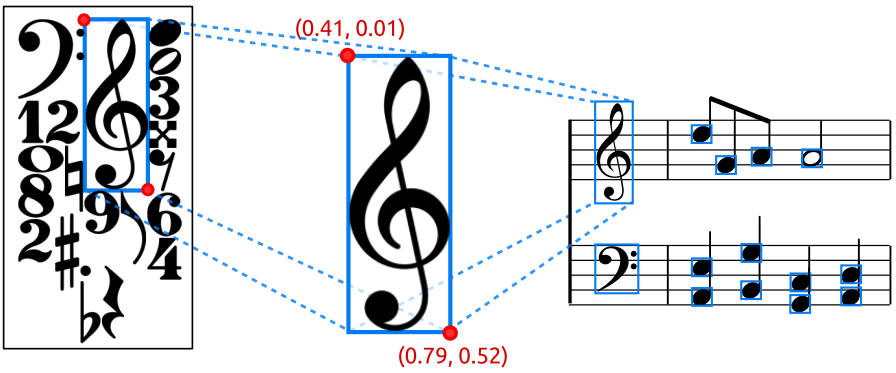


Figure 5.2: Use of texture atlas for music symbols rendering.

Another important issue to solve is style of presenting a sheet music on the display. Nowadays there are many different Android devices with various screen sizes ranging from small smartphones, through phablets and 7″ tablets, to full-size 10″ tablets. Users usually expect that application adapts to their hardware. Android SDK provides a sophisticated method for accessing resources such as images, strings and variables transparently according to the screen size or density. Customized resources are put to a folders with given name and system automatically uses on each device the proper one. In this way application stores, for instance, page margin and zoom level different for tablets and for smartphones.

The last thing to discuss is the type of scrolling, which has a high impact on user experience. As musician plays the instrument, position in the sheet music moves forward. The key responsibility of application is to show current position and upcoming measures inside the display viewport in a predictable manner. It is unacceptable to suddenly scroll display for no reason, because a musician could get lost.

There are basically two ways how to follow the current position in music:

- **Page turning** is a method which needs sheet music to be divided into pages and is suitable for larger screen in tablets. It is the most natural way of displaying music, because it mimics the sheet of paper. Since there is no scrolling, it is pleasant to eyes. On the other hand there is a risky transition between changing one page to another, which can distract the musician.

- **Gradual scrolling** is the other way, which requires one long canvas and works well also for devices with small screen. It displays just a fraction of the whole sheet music around the actual position. As the musician plays, it continually scrolls the canvas by small pieces. The scrolling must be slow and smooth, because otherwise the music would have been blurred and the user could have got the motion sickness.

I decided to use gradual scrolling, because it's easier to implement and can be used universally across screen sizes. I have tried several layouts with different scrolling directions, which are shown in figure 5.3:

1. First example illustrates horizontal layout with one long music staff. Sheet music is scrolled in such a way that actual position is always in center of the screen. This type is suitable mainly for small screens. Little disadvantage of this solution is fast motion of the canvas under high tempo.

2. Second example shows vertical layout on landscape-oriented device with music split into staffs long as screen width. As musician progresses through each staff from left to right, the viewport is slightly scrolling vertically from top to bottom until the current staff is at upper edge of the screen. This method is very unobtrusive and is used as default in the application.

3. Third example is portrait-oriented variation of the second one. It doesn't make much sense on smartphones, but when used on devices with large screen such as tablets, it feels like natural sheet music page. Moreover, the shorter lines are also easier to read.

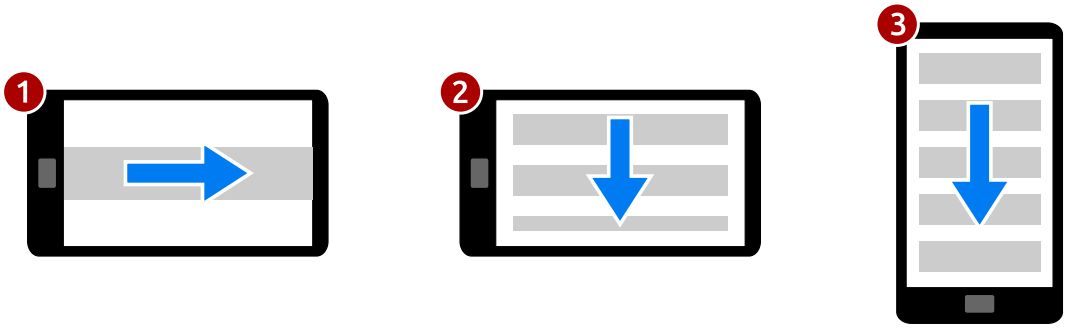


Figure 5.3: Different types of sheet music layout with respective scrolling directions.

5.4 User Interface

Graphical user interface of the application is very simple. It consists of two Android activities (views):

- **SongChooserActivity** is displayed on application startup and its task is to render a list of available music scores (figure 5.4). Each item displays title of the score and artist, and contains URI of the corresponding MIDI file. The list is so far loaded from static JSON file with some preselected classical and popular songs, but in future versions I plan to offer possibility to search and download new songs from online music library. When user taps a list item, activity gets the MIDI file path and pass it as a parameter to the other activity.

- **SheetMusicActivity** receives the path, parses the MIDI file and renders a sheet music on the screen (figure 5.5). At the same time it runs the score follower, which starts listening to incoming sound. It also keeps the screen on in order to prevent it from automatic dimming. User can scroll the canvas by scrolling gesture. For performance reasons, renderer draws only notation lying inside virtual buffer, which is circa 2 times bigger than the screen. This buffer is redrawn only when the current viewport reaches its borders. Otherwise it is just moved by translation. User can also change the position of cursor. When user taps some specific place in the sheet music, system finds out position in score corresponding to the tapped position, cursor is moved to the that place and score follower is restarted.

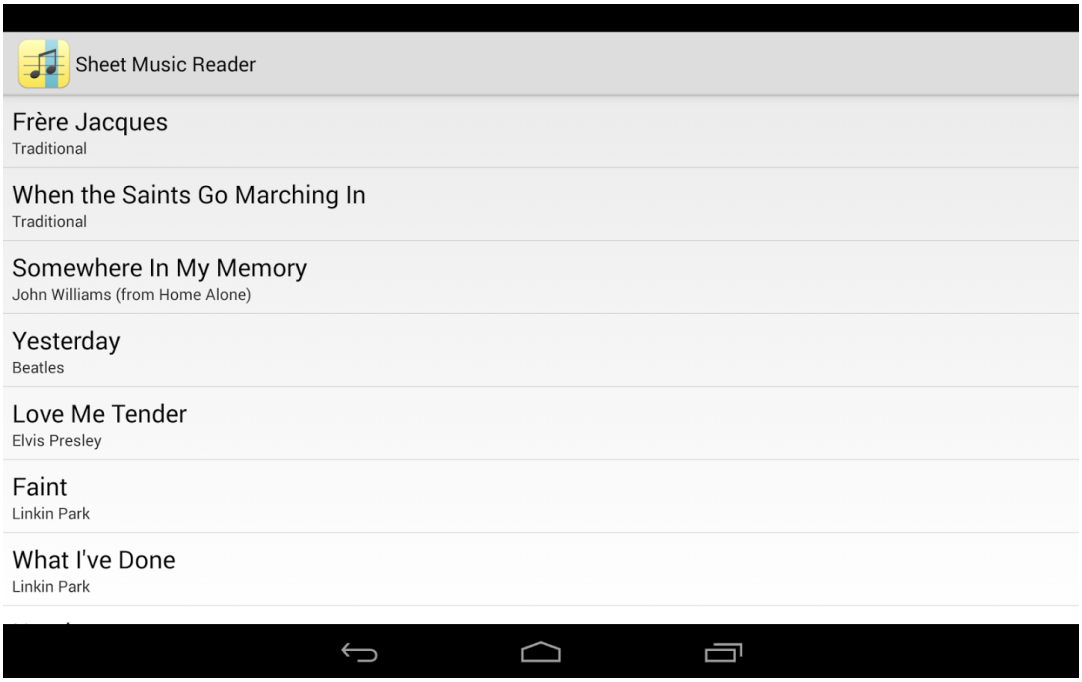


Figure 5.4: Mobile application – list of available music scores.

Figure 5.5: Mobile application – sheet music canvas with cursor at actual position.

Chapter 6

Experiments

Experiments and measurements are important part of software testing. Using them, one can decide if a newly introduced feature improves or degrades the system performance. In this project, the testing scripts were continually evolving during the development.

All the test scripts operate on demo application and are written in Python. They work as follows: for every tested parameter, script runs demo application on each file of dataset and averages total alignment error. Besides standard test on alignment error, test cases examine also the system behavior on modified internal parameters (windows size, overlap) or on modified input audio files (tempo, noise, delay).

The main tested criterion is *mean error* – average time deviation from reference position. This error is calculated by application as mean of piecewise difference between position computed by Matcher and corresponding estimated position on tempo curve. A typical threshold for considering the alignment to be successful is 250 ms. It is the maximum time in which the player usually doesn't distinguish if the position is correct or slightly out.

6.1 Dataset

As a dataset, 21 musical pieces were selected. It contains hand-picked MIDI files of classical compositions by Beethoven, Chopin, or Mozart, as well as traditional songs and piano arrangements of music from movies. Total length of this dataset is 32 minutes.

To generate audio from MIDI files, these steps were taken:

1. MIDI files were software synthesized using TiMidity++ to corresponding WAV files.
2. Audio was resampled to 11 025 Hz and converted to mono.
3. Audio samples were normalized by maximum amplitude.
4. Silent part in the end of some audio files was trimmed out.

There were some problems with software synthesizer TiMidity++, which produced strange crackling noises at the beginning of files. Solution was to run it with parameters `-A120` and `--output-24bit`, which force the synthesizer to produce 24-bit samples and turns on amplification. Audio samples are then converted back to 16-bit width to be readable for application.

The main disadvantage of this dataset is a lack of labeling – the reference score position is estimated from the current audio position. It assumes that tempo of recording is constant

over time. Fortunately, it turned out that most recordings have tempo curve very close to the straight line. Therefore this criterion is sufficient for the needs of this project. Poor performing system usually have alignment error in seconds while estimated position distortion is in milliseconds.

6.2 Parameters Adjustment

This section demonstrates the effect of parameters on system behavior. Proper choice of parameter values is crucial for good performance. In each figure, the plot expresses mean alignment error with respect to the examined variable. Some figures contains also the second plot, which shows the error on noised audio. Amount of added white noise is 10 % of the maximum amplitude.

Matcher window size

Size of the WTW (Windowed time warping) window used in Matcher module influences the robustness of the system. The bigger window, the better stability, since Matcher can process longer chunks of audio. On the other hand, total processing time increases by square of the window size, so there is a trade-off. From plots in figure 6.1 is evident that the optimal window size is around 100×100.

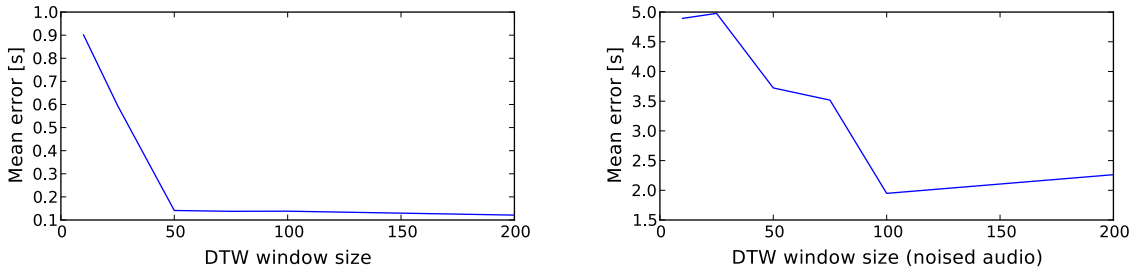


Figure 6.1: Performance of the system depending on Matcher window size.

Matcher window overlap

WTW window overlap helps to stabilize matching process in a way that it bypasses inaccurate slope estimation at the start of each iteration, where there is not enough previous data in cost matrix. On the other hand, it slightly increases computation time at the end of each iteration, where overlapped part needs to be recomputed. Figure 6.2 shows that reasonable overlap percentage starts around 30 %.

Slopes history size

Matcher stores history of previously computed slopes to estimate current tempo. Short history causes low fault-tolerance, while long history needs longer series of slopes to change estimated tempo. According to plots in figure 6.3, the good trade-off value is between 30 and 40. The mobile application has this parameter set to 25, because it needs to react faster on rapid tempo changes.

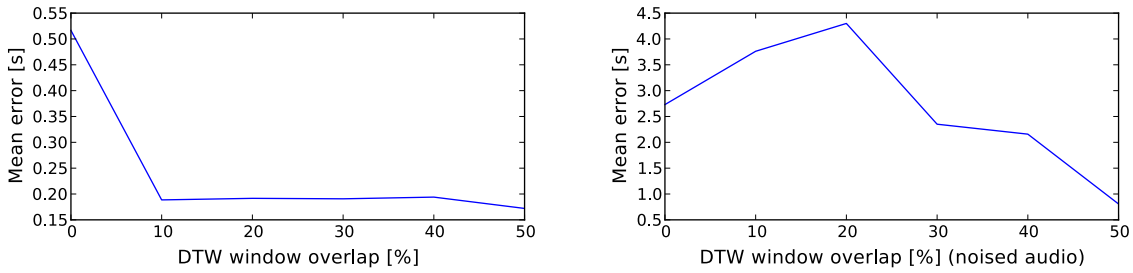


Figure 6.2: Performance of the system depending on Matcher window overlap.

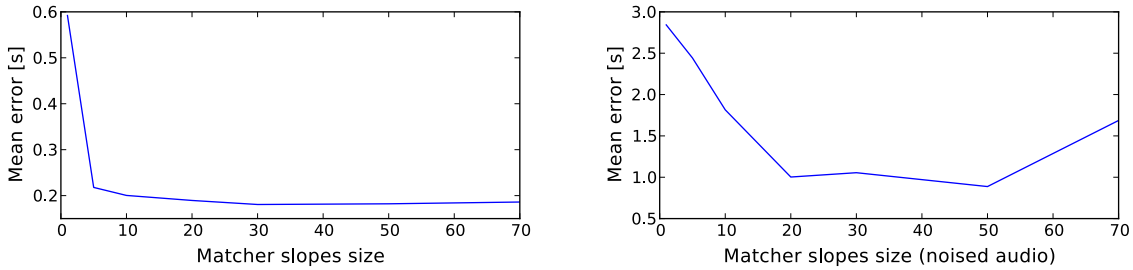


Figure 6.3: Performance of the system depending on slopes history size.

Audio speed

To inspect the system behavior in real conditions, valuable measure is how the system reacts to different speeds (tempos) of music. It mimics the situation when musician is practicing a new piece, which he usually plays a little slower. Experienced pianists in contrast might play even faster than 100 %. Plot in figure 6.4 illustrates that application can handle the both cases. Higher error rates under lower tempos might be caused by sound distortions (blurred onsets) and by the fact that error is influenced by tempo scale.

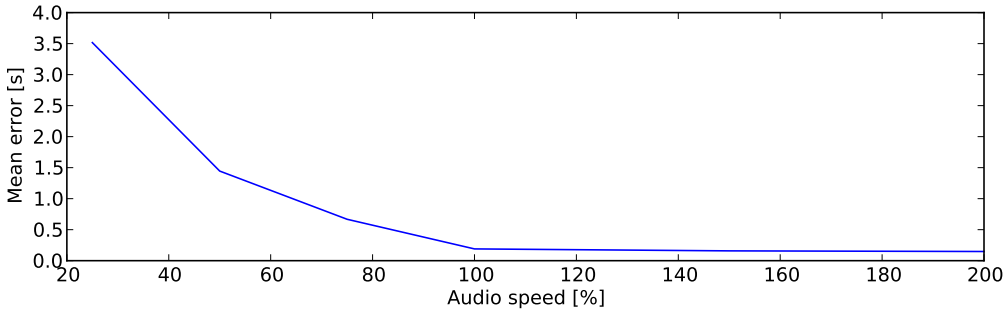


Figure 6.4: Performance of the system depending on relative audio speed.

Audio delay

Last tested measure is audio delay, which gives how shifted in time is audio to score. It indicates how fast and if at all the system is able to recover. It also corresponds to real situation, when user start playing on slightly different position than where cursor currently

is. Figure 6.5 shows that system handles well starts ahead the current score position, but has problems to recover with starts more than one second in the past.

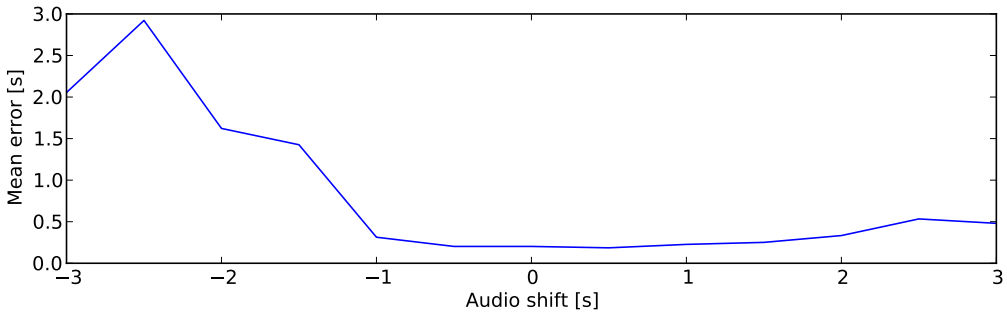


Figure 6.5: Performance of the system depending on audio delay.

6.3 Evaluation

This section will summarize performance of the score following system. After all the internal parameters were properly set to optimal values, each file from the dataset was tested as is without any modifications of audio speed and without added noise. Results are displayed in bar chart of figure 6.6. Individual bars are labeled by the name of composer or by the name of movie where the music comes from. You can see that with default tempo almost all the audio files were aligned with mean error less than 300 ms. Average alignment error for the whole dataset is marked with red dashed line. Its value is 155 ms, which is error almost indistinguishable by human player.

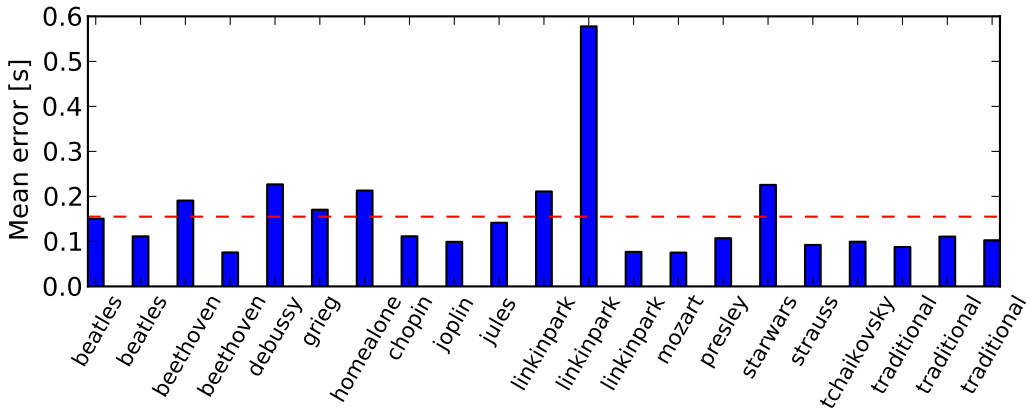


Figure 6.6: Alignment error for all individual files in dataset. Total average error is marked by red dashed line.

More detailed piecewise precision is described by table 6.1. Hit rate in each line represents a percentage of audio frames that were aligned with error less than time given by the first column. Second column in the table contains error measured on original synthesized audio, third column was measured on audio with 5 % of white noise added, and fourth column on files slowed down to 50 %.

Indistinguishable delay lower than 250 ms was achieved during **85.5 %** of time on original dataset. Altered noised files were so successful in 81.5 % of cases, respectively 62.5 % on slowed down files. Using twice as long metrics, which is still acceptable for this task of score following, the system correctly aligned **93.3 %** samples on original dataset, 88.2 % and 73.4 % on altered ones.

Error	**Hit**	**Hit (5% noise)**	**Hit (50% speed)**
$<$ **0.05 s**	20.1 %	21.0 %	7.1 %
$<$ **0.10 s**	48.6 %	49.0 %	21.2 %
$<$ **0.15 s**	68.7 %	66.1 %	38.3 %
$<$ **0.20 s**	79.6 %	75.9 %	53.4 %
$<$ **0.25 s**	85.5 %	81.5 %	62.5 %
$<$ **0.30 s**	88.5 %	84.5 %	67.6 %
$<$ **0.35 s**	90.4 %	86.3 %	70.2 %
$<$ **0.40 s**	91.6 %	87.4 %	71.6 %
$<$ **0.45 s**	92.6 %	87.9 %	72.7 %
$<$ **0.50 s**	93.3 %	88.2 %	73.4 %
$<$ **0.60 s**	94.3 %	88.7 %	74.2 %
$<$ **0.70 s**	95.1 %	89.1 %	74.7 %
$<$ **0.80 s**	95.7 %	89.3 %	75.1 %
$<$ **0.90 s**	96.5 %	89.4 %	75.3 %
$<$ **1.00 s**	97.5 %	89.6 %	75.5 %

Table 6.1: Piecewise alignment error on original audio, audio with 5 % of white noise, and audio slowed down to half tempo.

Chapter 7

Conclusion

This project implements a score following system that is able to track arbitrary music piece given a reference MIDI file and corresponding audio recording or real time audio stream. The other state of the art score following system were studied and described as well as innovative audio features extraction methods.

To align audio to music score, an online variant of Dynamic Time Warping (DTW) called Windowed Time Warping was developed. This algorithm estimates path in DTW cost matrix using only forward pass without need to perform additional computationally expensive backward path. Audio features are obtained by mapping FFT spectral bands to music chroma classes. Score features are extracted directly from MIDI file using sophisticated techniques called Synthesized Spectral Templates and Locally Normalized Chroma Onsets. Increased robustness and smoothness were achieved by using stability mode and DTW path slope history to compute current tempo.

The final system was rewritten for Android platform and integrated into mobile application. It contains preselected music compositions and hardware-accelerated music rendering engine. It can adapt to various screen sizes, densities and display orientations of present smartphones and tablets. Focus was also given to gradual display scrolling, which doesn't distract the musician. Application was released on Google Play Store[1].

Performed experiments showed that the system aligns **85.5 %** of time with error less than 250 ms, or **93.3 %** of time with error less than 500 ms. Even though the experiments with noised and slowed down audio produced lower hit rates, the mobile application performs in real conditions quite well and is able to follow pieces even with changing tempo, pauses during performance, and minor deviations from the original score.

In further development I would like to make the score follower even more robust. It should be able to better match events in the past and to recover when it gets stuck. I also plan to add possibility to search and download new compositions from an online archive. Long term goal is to switch from MIDI to MusicXML format, which contains much more musical information and is suitable for music rendering.

[1] https://play.google.com/store/apps/details?id=cz.vsmejkal.sheem

Bibliography

[1] Jonathan Aceituno. Real-time score following techniques. http://p.oin.name/projetsfac/ir.pdf, 2011. Accessed: 2014-04-10.

[2] Andreas Arzt. Score following with dynamic time warping: An automatic page-turner. Master's thesis, Johannes Kepler Universität Linz, 2007.

[3] Andreas Arzt and Gerhard Widmer. Simple tempo models for real-time music tracking. In *Proc. of the Sound and Music Computing Conference (SMC), Barcelona, Spain*, 2010.

[4] Andreas Arzt and Gerhard Widmer. Towards effective 'any-time' music tracking. In *Proceedings of the 2010 conference on STAIRS 2010: Proceedings of the Fifth Starting AI Researchers' Symposium*, pages 24–36. IOS Press, 2010.

[5] Andreas Arzt, Gerhard Widmer, and Simon Dixon. Automatic page turning for musicians via real-time machine listening. In *ECAI*, pages 241–245, 2008.

[6] Andreas Arzt, Gerhard Widmer, and Simon Dixon. Adaptive distance normalization for real-time music tracking. In *Signal Processing Conference (EUSIPCO), 2012 Proceedings of the 20th European*, pages 2689–2693. IEEE, 2012.

[7] MIDI Manufacturers Association. MIDI messages. http://www.midi.org/techspecs/midimessages.php. Accessed: 2014-04-10.

[8] George Bebis. Short time fourier transform. http://cse.unr.edu/~bebis/CS474/Lectures/ShortTimeFourierTransform.ppt. Accessed: 2013-12-22.

[9] Wei-Ta Chu and Meng-Luen Li. Score following and retrieval based on chroma and octave representation. In *Advances in Multimedia Modeling*, pages 229–239. Springer, 2011.

[10] Arshia Cont. Improvement of observation modeling for score following. *Dea atiam, University of Paris*, 6, 2004.

[11] Simon Dixon. An on-line time warping algorithm for tracking musical performances. In *IJCAI*, pages 1727–1728, 2005.

[12] Sebastian Ewert, Meinard Muller, and Peter Grosche. High resolution audio synchronization using chroma onset features. In *Acoustics, Speech and Signal Processing, 2009. ICASSP 2009. IEEE International Conference on*, pages 1869–1872. IEEE, 2009.

[13] Filip Korzeniowski and Gerhard Widmer. Refined spectral template models for score following. In *Proceedings of the Sound and Music Computing Conference (SMC)*, Stockholm, Sweden, 2013.

[14] Tod Machover. A view of music at IRCAM. *Contemporary Music Review*, 1(1):1–10, 1984.

[15] Meinard Muller, Frank Kurth, and Michael Clausen. Chroma-based statistical audio features for audio matching. In *Applications of Signal Processing to Audio and Acoustics, 2005. IEEE Workshop on*, pages 275–278. IEEE, 2005.

[16] Nicola Orio, Serge Lemouton, and Diemo Schwarz. Score following: state of the art and new developments. In *Proceedings of the 2003 conference on New interfaces for musical expression*, pages 36–41. National University of Singapore, 2003.

[17] Zhengshan Shi. An automatic music score alignment system for music recordings appreciation. Master's thesis, New York University, 2012.