JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis

Automatic Audio and Lyrics Alignment

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

in der Studienrichtung

INFORMATIK

Eingereicht von:
Andreas Kothmeier, 0155676

Angefertigt am:
Institut für Computational Perception

Betreuung:
Univ.-Prof. Dr. Gerhard Widmer

Linz, August 2006

Acknowledgements

I would like to thank the following people

1) **Univ.-Prof. Dr. Gerhard Widmer*** for his support and good suggestions while writing this diploma thesis.
2) **Dipl. Inf. Tim Pohle*** for his introduction into digital audio processing and his support in the early implementation stage.
3) **Dipl.-Ing. Peter Knees*** for providing the topic of this diploma thesis and his support.
4) **Dipl.-Ing. Klaus Seyerlehner*** for his advice on some books about digital audio processing.
5) **Clemens Raab** for his support at the implementation of the band pass filter.

---

* Department of Computational Perception, Johannes Kepler University Linz

Abstract

English Version:
Nowadays computers start to replace the hi-fi system in living rooms all over the world. This opens up more and more possibilities for the user to gain additional information for the song currently played. One kind of information is the lyrics; that is what my diploma thesis deals with. The goal is to provide a program that is able to automatically align the lyrics to the audio signal. This way the annoying scrolling of the lyrics while listening to a song is not necessary anymore.

Deutsche Version:
Heutzutage verdrängen Computer mehr und mehr die HiFi-Anlage im Wohnzimmer. Dadurch ergeben sich immer mehr neue Möglichkeiten für den Benutzer, zusätzliche Informationen zu dem gerade gespielten Lied zu erlangen. Eine dieser Informationen ist der Liedtext, der das Hauptthema in meiner Diplomarbeit darstellt. Der Text eines Songs soll dabei automatisch mit dem Audiosignal synchronisiert werden. Das ermöglicht es dem Benutzer, den Text beim Anhören des Songs mitzulesen, ohne, dass dabei ein manuelles Scrollen notwendig wäre.

Contents

List of Figures

List of Tables

"Music is everything one listens to with the intention of listening to music." – (Luciano Berio, pioneer of electronic music)

1 Introduction

1.1 Development of Intelligent Music Processing

Nowadays the computer is more and more replacing the hi-fi systems. This is a development which can mainly be observed in the last ten years only. During the 1980s nobody even thought about storing music on a computer. One song would have filled the whole hard drive of an average computer system. But in the early 1980s the "Big Bang" of the digital audio revolution was reached with the invention of the Audio CD. Until then audio data could only be stored in an analogue way, which often suffered from quality loss after every play back (e.g. music cassettes). However, audio files in CD quality were still too big to be stored on hard disks. This changed immediately when the Fraunhofer Institute released the first software MP3 encoder in 1994. Because then also the hard disks reached an acceptable size for storing multi media data, many people started to archive their music collection on their PC. Furthermore when Shawn Fanning and Sean Parker released the first version of their peer-to-peer file sharing tool "Napster" in 1999, people started to share their music collections with the whole world [WIK06].

As a consequence music and computers were growing together very fast. This also means that there emerged more and more new application areas for digital music. We can now create our own compilations on recordable CDs, take our music with us wherever we want by using small mp3 players and so on. The computer is currently starting to replace the hi-fi system in our living rooms. Therefore many new research areas like e.g. "Automated Playlist Generation" arose in the last few years. One of these interesting research areas is the topic of my diploma thesis: "Automatic Audio & Lyrics Alignment".

1.2 Motivation

1.2.1 Application Areas for Automatic Lyrics Alignment

Many people all over the world listen to music every day. But listening to music and understanding the lyrics of the song are two different things. Because it is sometimes very difficult to understand every word, people want to have the lyrics of a song in a written form. However, often – especially for long lyrics – the user has to scroll through them manually while a song is played. So it would be more comfortable to only show the lyrics for the currently heard section instead of permanently displaying them for the whole song.

This is one point leading to the topic of my diploma thesis – the automatic alignment between audio and lyrics. But there are two other important reasons for the need of an automatic way to align the lyrics to the audio data. The first – and perhaps most commercial one – is the automatic creation of karaoke songs. Manual annotation of the right lyrics timings is an exhausting work which would become much easier. Especially for large song collections an automatic approach would save much time.

Another interesting application where alignment of lyrics and audio data can be useful is the possibility to search for or even jump to certain words in a song. So if it is known e.g. that the song contains the word “dog” it would be very hard to find it in the raw audio data. But if the lyrics are already aligned to it, it is much easier to find the right location.

1.2.2 Main Reason for Choosing this Topic

Many online lyrics portals have emerged in the last years. However, there are some problems when trying to find the lyrics for a certain song. The portals do not contain lyrics for all songs and if lyrics for the same song are found on several portals, they are mostly different. This can be caused by simple typos, different words, different annotation styles or even different versions of the same song.

Peter Knees, Markus Schedl and Gerhard Widmer from the Department of Computational Perception at the Johannes Kepler University Linz developed an approach to automatically retrieve and extract lyrics of arbitrary songs from the internet [KNE05]. In addition to this basic search feature their method also compares all found lyrics and returns the most probable version of them. This

makes a lot of sense since the lyrics often contain different spellings for the same word. A good example of this is the slang term *cause* which is often written as *'cause*, *'coz*, *cuz*, or even correctly as *because*. Another problem the system has to face is that people who submit lyrics to the online portals often do not understand the same words while listening to a song and writing down its lyrics. This is a well-known issue and the lyrics pages have already reacted to it by providing the possibility to rate the quality of lyrics or to allow the users to submit corrections. Very common problems are different versions and translations of a song, because the artist and the title stay the same but the returned lyrics will often differ significantly. A further barrier when comparing lyrics is that they often contain annotations like for example information about the section (e.g. *[Chorus]*) or the performing artist (e.g. *[Snoop – singing]*). To avoid redundancies and unnecessary typing effort, lyrics mostly are not written completely. Instead of writing the full chorus out three times it is often written only once. Later chorus sections are simply expressed by annotations like *Chorus* or *repeat chorus*. The approach of Knees et al. respects all these issues in order to extract the best possible lyrics. It then estimates the quality of the predicted text so the user is able to decide if he will be satisfied with the results.
This existing system could perhaps be improved by a good alignment between audio and lyrics. It could decide better if certain sections, lines or even words make sense at the estimated position. This is the main reason why I chose this topic for my diploma thesis.
There are enough useful application areas which can be improved by a good alignment between audio data and lyrics. Therefore we will now take a closer look what the actual goal of my work is.

1.3 Goal

Aligning lyrics to the audio signal is anything but a trivial task. Many people cannot believe that it is hard to find out the lyrics the performer is singing. For us it is no problem indeed to hear what a singer is singing, but for a computer it is often nearly impossible to distinguish between human voice and other instruments. This becomes clear very fast when speech recognition software is used in a normal office environment. The ringing phones and talking people can

already be enough to make the software produce unsatisfying results. However, in contrast to the singer's voice in a tune these are almost perfect conditions.

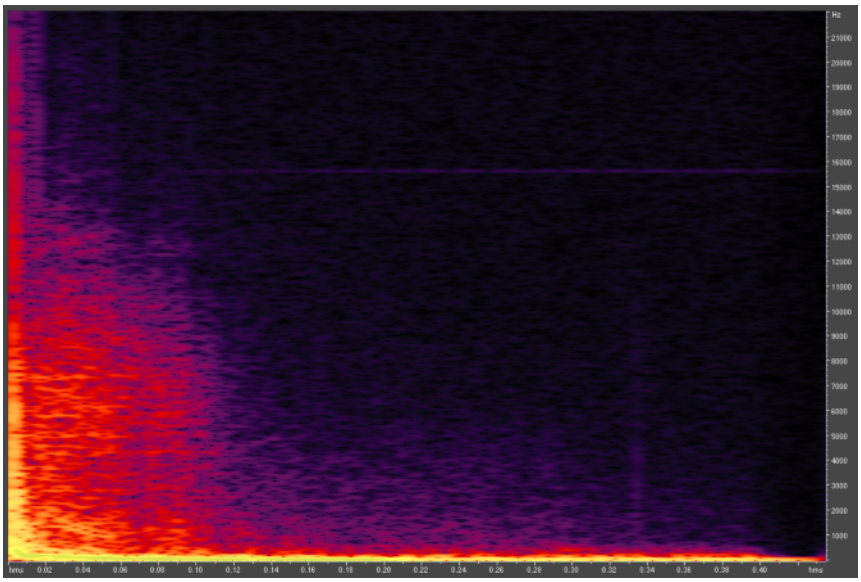


Figure 1-1: Frequency spectrum of a drum.

The aligning system has to face many disturbing factors in a song. Accompanying instruments confuse the speech recognition algorithms. Especially percussion instruments such as drums occupy a broad range of frequencies (see Figure 1-1). Therefore they also overlay the frequencies of the human voice.
But even if an acapella version of a song were used it would be impossible to extract the lyrics with a speech recognition algorithm. The reason for this is that there are big differences between spoken and sung voice. When a closer look is taken at sung voice the phonemes are either stretched or jolted in comparison to normal speech [WAN04]. But because speech recognition software was optimized for recognizing speech, it is clear that it will not work for sung voice.

Another naïve approach is to try it the other way round. This means to synthesize the speech from the lyrics and try to find similar patterns of the resulting audio data in the original song's audio data. Although the algorithms for synthesizing human speech got really powerful in the last years [TTS06], this is almost impossible.
There are several reasons why this approach does not work well. The main problem again is that the synthesized voice is speech and therefore very hard to match against the sung vocals in a song. Although there already exist algo-

rithms that can synthesize sung vocals [YAM06] the results will not be of sufficient quality to use it for the alignment. I already tried out the demo version of this software.[2] After that I came to the conclusion that it would be much more effort to handle this synthesizer and rebuild the vocals of the whole song than to align the lyrics manually. Furthermore the right timings for all the sung words would have to be known in order to synthesize the vocals of the songs. This leads to a vicious circle because to find the right timings is the goal of the aligning method. Another reason why this approach does not work is that it is very hard to find the right locations for the lyrics in the audio signal even if a perfect acapella version would be used. However, the synthesized vocals will never reach the quality of an acapella version.

Consequently a better solution is needed to solve the lyrics alignment exercise. A very promising approach was done by Ye Wang et al. In their paper "LyricAlly: Automatic Synchronization of Acoustic Musical Signals and Textual Lyrics" [WAN04] they present a method that divides the task into two subtasks. First they try to figure out the structure of the song by finding sections like Intro, Verse, Chorus, Bridge and Outro. This leads to a rough alignment of the lyrics and audio sections. After that they complete the alignment process with a per line alignment. This means that in each section every single text line is also aligned to the audio data. Since this would go beyond my time capacities, the main goal of my diploma thesis is to achieve the rough section alignment. More details about this approach are described in chapter 3 .

[2] http://www.zero-g.co.uk/media/mp3/Vocaloid_Demo_Version.zip, 08 2006

2 Basic Knowledge

2.1 Digital Representation of Audio Data

Audio in nature is simply the oscillation of air reaching the human ear. The variations of air pressure are analogue (see Figure 2-1).

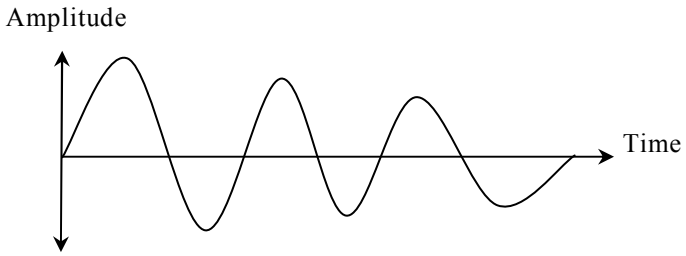


Figure 2-1: Continuous analogue signal.

If this signal should be stored in a digital way an infinite memory would be needed because of its analogue character. To avoid this, an abstraction layer has to be inserted. It reduces memory usage and approximates the original signal as well as possible. The first idea is to only save the amplitude value at certain time positions. Instead of infinite data points now only a finite set of values has to be stored. The frequency at which the sample values are fetched is also called *sampling rate* [STE96] or *sampling frequency* [VAS00]. It is shown as green vertical lines in Figure 2-2.

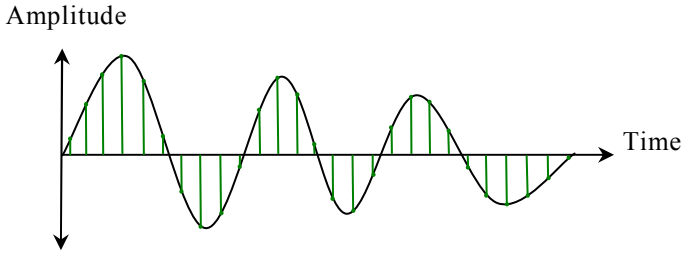


Figure 2-2: Sampling rate.

Hence only 27 values have to be stored in this example to reconstruct an approximation of the original signal. The signal reconstructed by using the sample points (represented by the red line in Figure 2-3) does not completely match the original signal. But nevertheless it is a very good approximation. So the human ear will not notice the difference for a sufficiently high sampling rate.

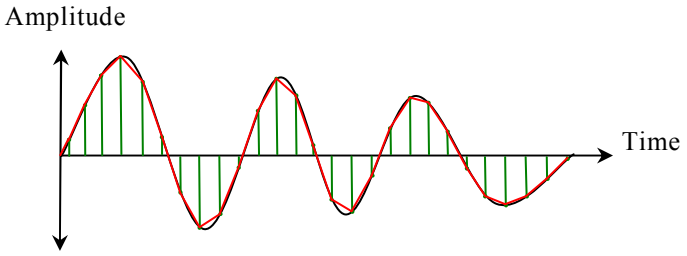


Figure 2-3: Reconstructed signal (red line).

However, the lower the used sampling rate the more the reconstructed signal will differ from the original one. In Figure 2-4 a too low sampling rate is used. It can be observed that the reconstructed signal is totally different from the original one and that at the beginning even a whole valley is left out.

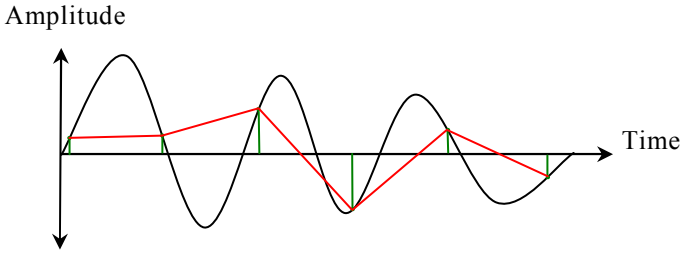


Figure 2-4: Too low sampling rate.

The reason for this behaviour is that the *Nyquist sampling theorem* [VAS00] was violated. This theorem is one of the basics of *Digital Signal Processing*. It says that the sampling rate has to be at least twice as high as the highest observed frequency (=*Nyquist Frequency*) in the original signal. That is a quite logical fact because one oscillation always contains a hill and a valley. As a consequence at least two values (one positive and one negative) have to be sampled for each oscillation in order to not lose the information about one hill or valley (see Figure 2-5).

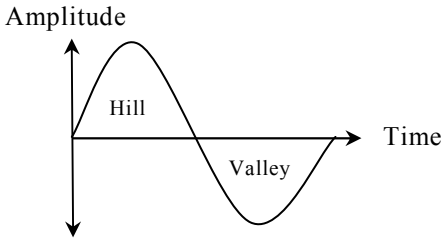


Figure 2-5: One oscillation.

While now memory usage was reduced along the time axis, there are still infinite possible values for the amplitude. Thus another abstraction has to be introduced along the amplitude axis. This abstraction defines the accuracy of the stored amplitude values at the sampled time and is called *bit rate*. It defines how many positions the amplitude values may have. In Figure 2-6 some of the sampled amplitude values do not fit on the horizontal lines. These values will then get rounded off to the nearest integer which is also called *quantising level* [STE96] (blue lines in Figure 2-6).

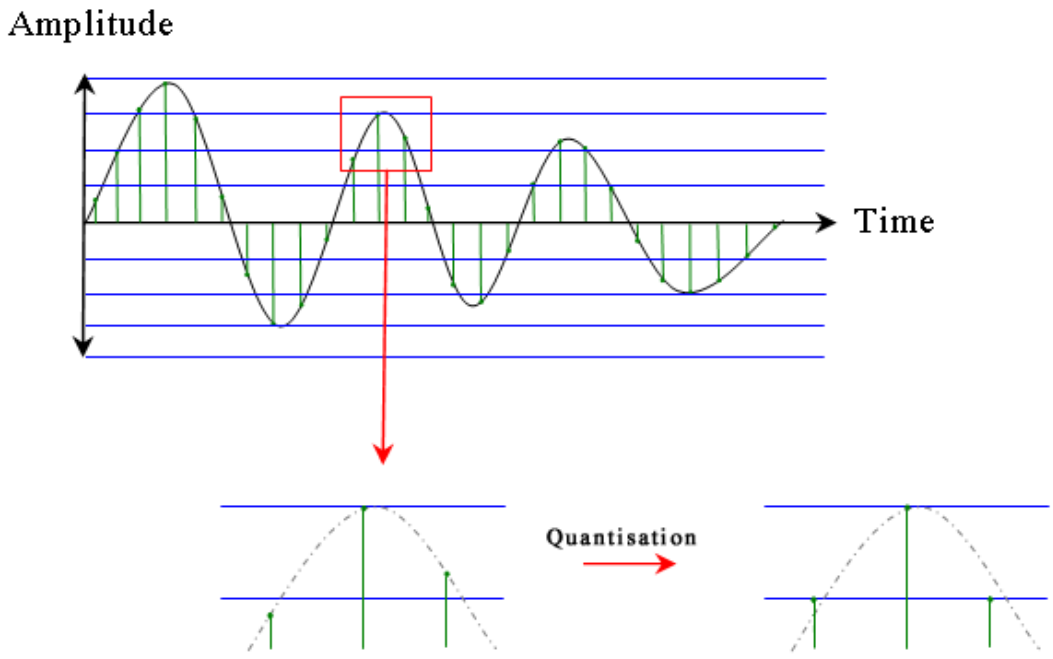


Figure 2-6: Bit rate.

Of course this costs some precision but for good bit rates there is no noticeable quality loss. The lower the bit rate the lower the memory usage and the quality.

An example of a common sampling rate and bit rate configuration is CD quality audio. It is defined by using a sampling rate of 44100 Hz (about twice the highest frequency the human ear is able to hear) and a bit rate of 16 bit. 16 bit means that the amplitude value can have $2^{16} = 65536$ possible values, which is enough for a crystal clear quality. But CD quality of course has the disadvantage that it needs a lot of memory:

- 44100 values in one second with 16 bit (= 2 bytes):
  `44100 * 2 = 88200 bytes = 86.13 KB`
- For stereo we have to use two channels:
  `86.13 * 2 = 172.26 KB/s`
- This means for one minute of audio in CD quality:
  `172.26 * 60 = 10335.60 KB ≈ 10 MB`

As a summary it can be said that digitally stored audio data is a long sequence of amplitude values. So every channel is represented by a vector which can be used later to extract certain features from the audio signal.

2.2 DFT, FFT & Inverse DFT

2.2.1 DFT

Digital audio is represented by a long numeric vector. However, the individual values in this vector only represent the value of the amplitude at a certain time. This information by itself is mostly useless for extracting helpful features from the audio data. But much more important is to know of which frequencies the signal consists. Therefore in digital signal processing the *Discrete-Time Fourier Transformation (DFT)* [STE96] is used to transform the signal from the time domain into the frequency domain. The DFT is given by the formula:

$$X_k = \sum_{t=0}^{N-1} x_t e^{-itk\,2\pi/N} \qquad k = 0,\ldots, N-1$$

In this definition of the DFT e is the base of the natural logarithm, t is the index for the current value in the input vector x and i is the imaginary unit ($i^2 = -1$).

Here the finite sequence of N numbers (representing the signal) $x_0,\ldots,x_{N-1}$ is transformed into a sequence of N complex numbers $X_0,\ldots,X_{N-1}$, representing the frequency spectrum of the signal. The returned values X_k correspond to the energy measured at a certain frequency of the spectrum. The frequency spectrum of the DFT ranges from zero to the sampling rate. Therefore it is important that the measuring frequencies for the returned values must all reside equally spaced in this interval.

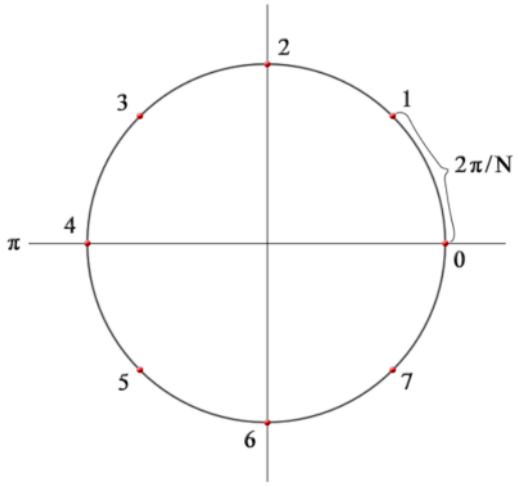


Figure 2-7: Measuring frequencies shown at the unit circle.

Figure 2-7 visualises the placement of the measuring frequencies (red points) at the unit circle with $N = 8$. The frequencies from zero up to the sampling rate are wrapped around the circle counter-clockwise, starting at the measuring point labelled with "0". Because in the unit circle the radius is *1*, the circumference of this circle is 2π. By splitting it into $N = 8$ equal parts, the distance between two parts is $\frac{2\pi}{8}$. Hence in this example the energy values returned by the DFT are measured at the frequencies $0 \cdot \frac{2\pi}{8}$, $1 \cdot \frac{2\pi}{8}$, $2 \cdot \frac{2\pi}{8}$, $3 \cdot \frac{2\pi}{8}$, $4 \cdot \frac{2\pi}{8}$, $5 \cdot \frac{2\pi}{8}$, $6 \cdot \frac{2\pi}{8}$ and $7 \cdot \frac{2\pi}{8}$. Now the rule for placing *N* measuring frequencies equally spaced in this interval is simple:

$$0(2\pi / N), 1(2\pi / N), 2(2\pi / N), ..., (N-1)(2\pi / N).$$

This leads to the factor $k2\pi / N$ in the exponent of *e* (where $0 \le k \le N-1$).

2.2.2 FFT

Since the DFT is only a mathematical concept for transforming the signal into the frequency domain, efficiency is not important. But in reality it would simply get too slow for great amounts of data. This is the reason why in practice only the *Fast Fourier Transformation* (FFT) is used for computing the DFT efficiently. There exist a lot of algorithms for the FFT but the most common one is the *Cooley-Tukey* algorithm [COO65]. It is a divide and conquer algorithm that recursively breaks down a DFT of any composite size $N = N_1N_2$ into many smaller DFTs of sizes N_1 and N_2. This is repeated until the level where only one element is left. Since the calculation of the DFT for one element is trivial ($X_0 = x_0$), the only task is to bring the result values into the right order. Hence the FFT manages to break down the complexity of the DFT from $O(N^2)$ to $O(N \log N)$.

2.2.3 Inverse DFT

The inverse DFT [STE96] is defined as

$$x_t = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{itk\,2\pi / N} \qquad t = 0,..., N-1.$$

This means that in contrast to the DFT the exponent of *e* is negated and the result of the sum is normalized by $\frac{1}{N}$.

In mathematics the *complex conjugate* of a complex number z is defined by changing the sign of the imaginary part. It is commonly denoted by z^*. The result of multiplying the inverse DFT by N and complex conjugating the input (X_k) and output values (x_t) is

$$Nx_t^* = \sum_{k=0}^{N-1} X_k^* e^{-itk\,2\pi / N} \qquad t = 0,..., N-1.$$

It can be seen that the right side of the equation is nothing else than the DFT of X_k^*. Hence it can be said that when calculating the DFT of the complex conju-

gates of X_k the result are nearly the original signal values x_t, that is to say Nx_t^*. Therefore it is very easy to calculate the inverse DFT:

1) Take the complex conjugate of the signal.
2) Calculate the DFT.
3) Take the complex conjugate of the result and divide it by N.

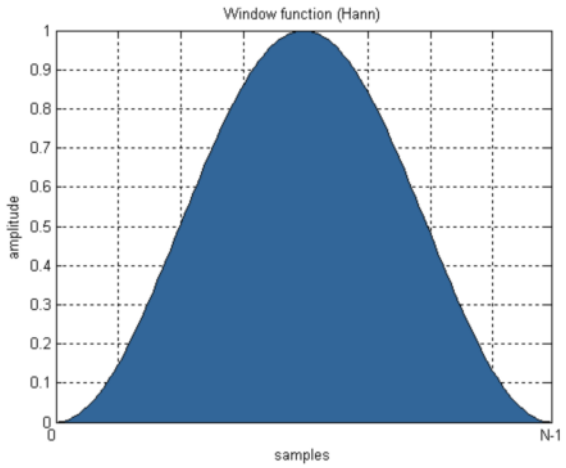
2.3 Window Functions - The Hann Window

Unfortunately it is impossible to use the FFT for transforming the signal into frequency space exactly at a certain location. The reason for this is that oscillations always occur over time. Therefore they cannot be observed by only analysing a certain position. So for frequency analysis always a period of time has to be used. Of course this period can be kept very small. It is defined by a so called *window function*. In digital signal processing, a window function is a function that is zero-valued outside of some chosen interval. The *window size* defines the length of this interval in samples.
Many different window functions are used in digital audio processing. For instance, a function that is constant inside the interval and zero elsewhere is called a *rectangular window*. When multiplying the audio signal by this window function, the product is also zero-valued outside the interval. Everything which is left is the "view" through the window.
In order to smooth the borders of the window, a *Hann window* is used in my implementation. It is given by the following formula:

$$w(n) = \tfrac{1}{2}(1 - \cos(2\pi \tfrac{n}{N})),$$

where $0 \le n \le N$ and the window size $L = N + 1$.

Figure 2-8 shows a Hann window. In Figure 2-9 the schematic "view" through a Hann window is shown. Only the red parts of the signal will be left after the multiplication. The grey parts will be zero afterwards.



[Source: http://en.wikipedia.org/wiki/Hann_window, 06 2006]

Figure 2-8: Hann window.

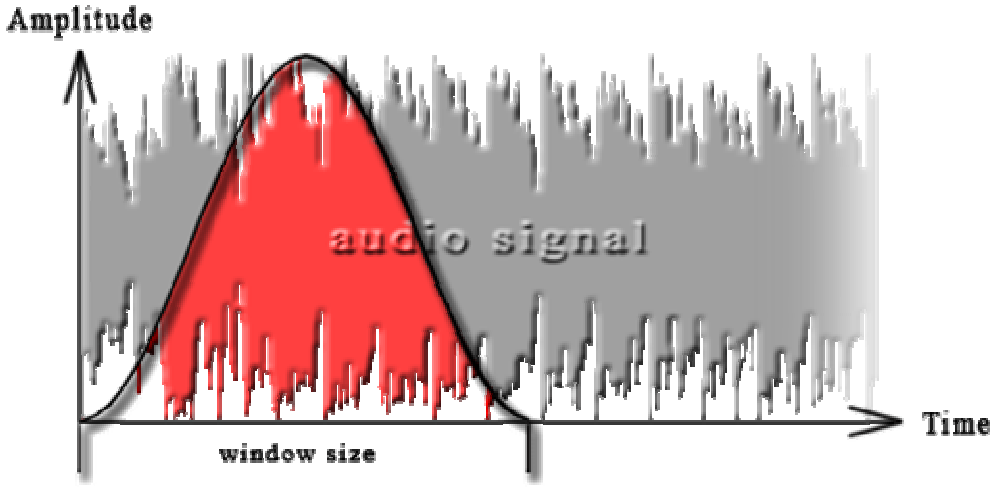
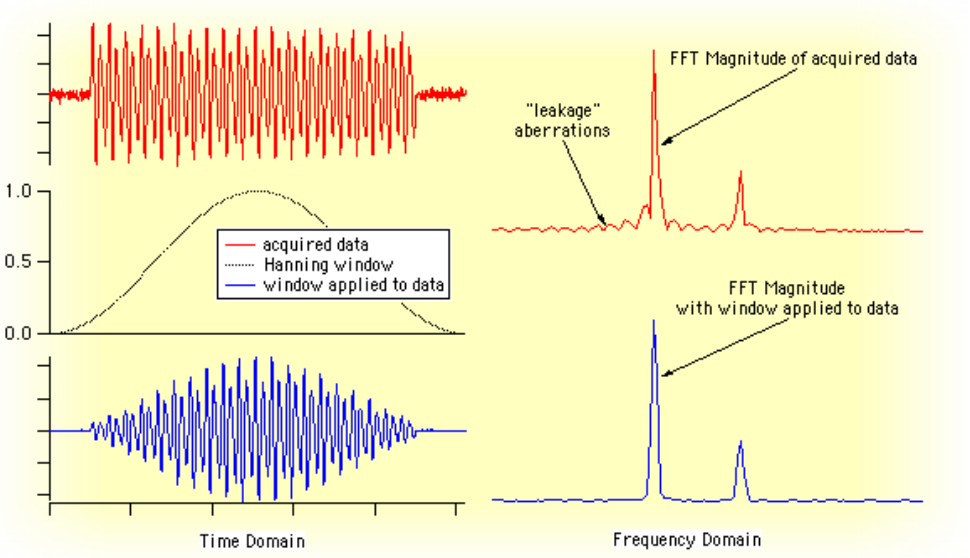


Figure 2-9: "View" through a Hann window.

Figure 2-10 compares the application of a rectangular window function (top) and a Hann window in the time and frequency domain. The time domain is displayed on the left and the frequency domain on the right side. In the first case the frequency spectrum contains small aberrations which are caused by the sudden endings of the signal. A Hann window can smooth these endings and therefore the aberrations in the frequency domain are eliminated.



Source: http://www.wavemetrics.com/products/igorpro/dataanalysis/signalprocessing/spectralwindowingpix/fftwindowingdemo.png, 07 2006

Figure 2-10: Rectangular and Hann window functions applied on signal.

2.4 Calculating the Frequency of a Note

The frequency of a certain note can be calculated very easily because the distance between two notes is logarithmically distributed by the factor $2^{n/12}$. Given the frequency of any note, the frequency of all other notes can be calculated by:

$$Freq = FreqOfStartingNote \cdot 2^{n/12}$$
$$n...number\ of\ notes\ away\ from\ starting\ note$$
$$n > 0\ for\ searchedNote > givenNote$$
$$n < 0\ for\ searchedNote < givenNote$$

Example 1:
Given the frequency of A_2 (=110 Hz), the frequency of E_3 (which is 7 semitones above A_2) should be calculated.

$$Freq_{E_3} = Freq_{A_2} \cdot 2^{n/12}$$
$$n = 7$$
$$Freq_{E_3} = 110 \cdot 2^{7/12} = 154.81\ Hz$$

Example 2:
Given the frequency of A_2 (=110 Hz), the frequency of C_2 (which is 9 semitones below A_2) should be calculated.

$$Freq_{C_2} = Freq_{A_2} \cdot 2^{n/12}$$
$$n = -9$$
$$Freq_{C_2} = 110 \cdot 2^{-9/12} = 110 \cdot 2^{-3/4} = 65.41\,Hz$$

The techniques described here will be necessary for understanding the implementation details in the chapters 3 and 4 .

3 The Process of Lyrics Alignment

In chapter 1.3 was already mentioned how lyrics alignment will not work. Now some existing approaches for the automatic alignment of lyrics are presented.

3.1 Approach 1: Chi Hang Wong et al.

In [WON05] a first method for automatic lyrics alignment is described. It mainly uses the assumption that the recording engineer would make the vocal part of popular music in centre position of the stereo signal. This means that the vocals can be heard in equal loudness on both channels. In contrast to this most musical instruments are recorded in stereo. Therefore it is tried to separate the vocals from the instruments and enhance the signal containing the vocals. Figure 3-1 shows the block diagram of this approach.

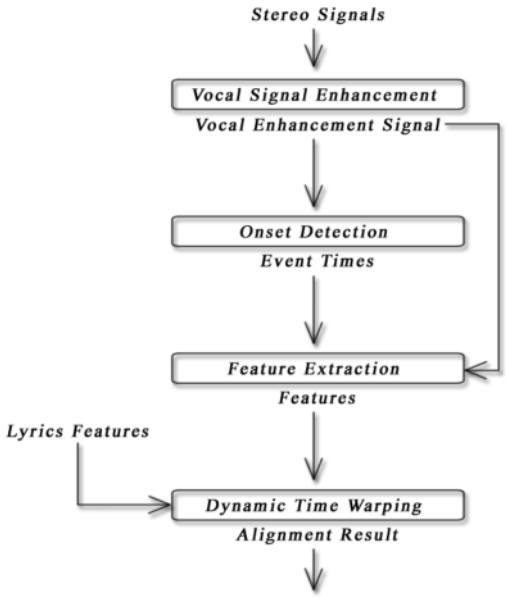


Figure 3-1: Block diagram of the approach presented in [WON05], p. 385.

The approach consists of four main steps:

1) The system tries to enhance the vocal signal. To achieve this, the two channels $s_l(t)$ and $s_r(t)$ of the stereo signal are described by

$$s_l(t) = s_c(t) + s_m^l(t)$$
$$s_r(t) = s_c(t) + s_m^r(t),$$

where $s_c(t)$ is the centre-padded signal which is the same in both the left and right channel. $s_m^l(t)$ and $s_m^r(t)$ are the non-centre-padded parts of the two channels. By subtraction of both channels, the reduced centre-padded signal $s_{\bar{C}}(t)$ is obtained.

$$s_{\bar{C}}(t) = s_l(t) - s_r(t) = s_m^l(t) + (-s_m^r(t)).$$

With it, the centre-padded signal can be extracted by *nonlinear spectral subtraction* [VAS00]. This means that both the original signal (taken from the left channel) and $s_{\bar{C}}(t)$ are first transformed into frequency space using the FFT. Then $\left|\frac{s_{\bar{C}}(\omega)}{2}\right|$ is subtracted from the spectrum of the original signal. Finally the inverse FFT is applied in order to obtain the estimated centre-padded signal $\hat{s}_C(t)$.

However, $\hat{s}_C(t)$ does not only contain vocals, but also includes some other instruments such as drums and bass guitar. It is assumed that these instruments are more or less stationary in parts where no vocals are present. From these instrumental parts, an average spectrum is calculated. It is used for attenuating the instruments in the centre padded signal using the spectral subtraction again. In [WON05] the result is called the "vocal enhancement signal".

2) Now the onset detection is used to find the timings of each sung word in the vocal enhancement signal. In order to find onset timings, the signal is divided into small segments with the window size w_s. For every seg-

ment the energy is calculated. An onset is detected, if the energy difference between two consecutive time segments is above a threshold ε.

3) The feature extraction module extracts pitch and distance features. This paper concentrates on songs with Cantonese lyrics. Cantonese is a tonal language which means that every word in it corresponds to a certain pitch. So the composers of Cantonese songs cannot use arbitrary words in their lyrics. This means that they have to choose certain words for a certain pitch in the melody. Consequently the relative pitch feature (pitch difference between the current and the last word) of a Cantonese word is important for matching the textual lyrics to the right pitch of the melody. Therefore two pitch features are calculated – one for the lyrics and one for the signal.
   The distance features describe the distance between two words. They are again calculated separately for textual lyrics and for the audio signal.

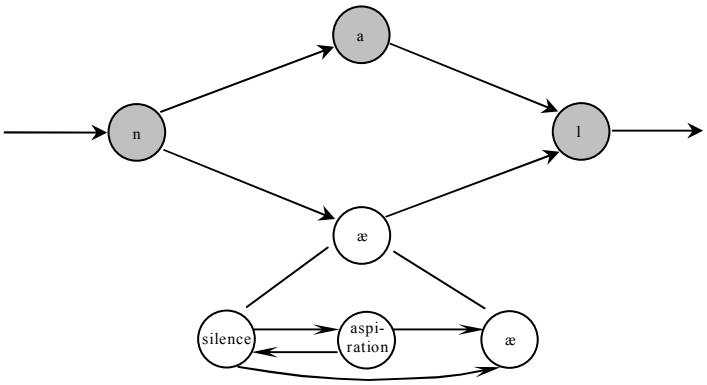
4) The features extracted in step 3) are used for *Dynamic Time Warping (DTW)* [JUA84]. DTW is a robust algorithm in *Automatic Speech Recognition* [RAB85], [MIL95] for matching two sequences with the best time alignment. It generates an error matrix, which is finally backtracked along the path with the minimum accumulated error to find the best alignment.

This approach is not very promising because the experimental results show a very low aligning accuracy. Furthermore it is unusable for the purpose of this diploma thesis since the whole approach is based on the Cantonese language. As already mentioned above all Cantonese pop music composers must write the lyrics to match the melody tones of the song. And this fact is exploited in the approach for finding the right timing for the alignment. Consequently this method would not work for songs in other languages at all.

3.2 Approach 2: Alex Loscos et al.

Another approach for automatic lyrics alignment is presented in [LOS99]. The goal of this work is to solve the lyrics alignment problem in real time. This

could be very helpful, because the alignment could be done while the singer performs. Consequently for example different specific audio effects could be applied depending on which phoneme of the lyrics is currently being sung.
The approach uses successful methods from Automatic Speech Recognition (ASR) for the alignment process. Mainly *Hidden Markov Models* (HMM) [HUA90] are applied for the alignment. But they were modified in order to make the models specific to the case of the singing voice.
Before the actual alignment process can start, a phonetic transcription is made out of the lyrics text. After that a Finite State Network (FSN) is built from the phonetic information. This means that the states in this network represent the individual phonemes of the words in the lyrics. The transitions between them are supplied with a certain probability for reaching the next phoneme. Before reaching the next phoneme state, special states for non-linguistic units (silence and aspiration) are inserted. This is important because they may occur at any time since every singer places them at different positions. Furthermore different singers also pronounce the words differently. This has to be regarded when building up the FSN by adding parallel paths as shown in Figure 3-2.



Source: [LOS99], p. N/A.

Figure 3-2: Parallel paths in Finite State Networks (FSN).

The probability for taking a certain path in the model is delivered by the *Viterbi* algorithm [RAB93]. This is the most common algorithm used in the text to speech alignment process. However, because in this case the delay should be as low as possible, the algorithm is adjusted in order to make it usable for real time decoding. Normally for determining the best path through the phoneme models, backtracking is performed at the end of every utterance. To find the

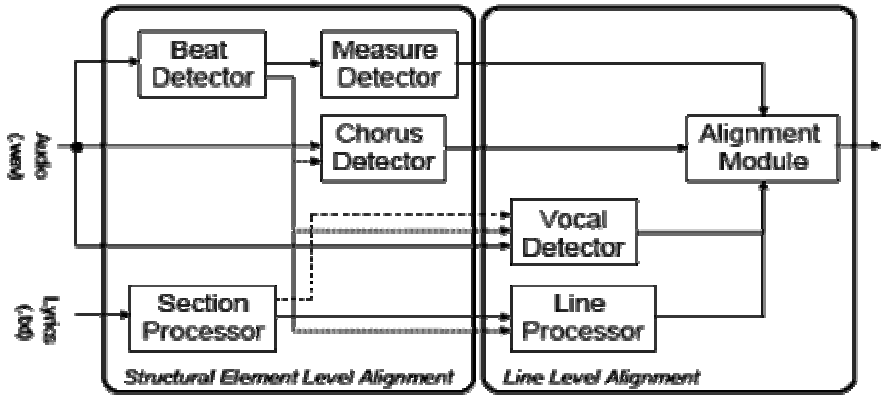
best path in as small intervals as possible, the backtracking has to be executed for every frame.
The advantage of this approach is that for every frame the system can decide the current singer position in the lyrics. But of course the performance vastly decreases. To overcome this problem Loscos et al. have extended their system with some heuristics and strategies. For example additionally the score is used to extract musical information. Hence the system "knows" that the phoneme corresponding to a note in the score is supposed to have certain duration. Furthermore it is supposed that the user follows the tempo of the song, which is again extracted from the score. Therefore the output probabilities of the Viterbi algorithm are adapted according to the tempo of the song.
Loscos et al. state that there is only a delay of 21 ms which has to be added to the hardware latency to get the delay of the whole system. This seems to be quite good, but this approach is also unusable for my purpose because this system only works on pure vocals. So it would be hard to use for pop songs. However, in my opinion it could be an interesting approach to try it out on a "vocal enhancement signal" as described in [WON05]. Another reason why it cannot be used in my system is that it depends on the score. Of course it would generally help at the aligning process to use the score. But since it is not very common to additionally have the score for the songs the goal of my approach is to do the alignment without score.

As I already pointed out in chapter 1.3 the approach used by Wang et al. sounds very promising for me. That is why my whole diploma thesis follows this concept even though with some differences. Because of this in the chapters 3.3, 3.4 and 3.5 an overview over the approach of Wang et al. and mine is given. The details of my implementation will be discussed later in chapter 4 .

3.3 Approach 3: Wang et al.

Wang et al. basically divide the problem into two major tasks. The first stage performs a high-level alignment of the song's structural elements detected in the text and audio streams. It ensures that lyrics lines are only aligned within the section they belong to. After that the second round performs the low-level line alignment. This top-down approach reduces unnecessary errors because the

possible locations of single lyrics lines in the whole song are limited by their section. The whole architecture of LyricAlly is shown in Figure 3-3.



Source: [WAN04], p. N/A.

Figure 3-3: LyricAlly architecture.

The main steps in the high-level alignment are:

1) The ***Beat Detector*** detects the beat times in the audio signal.
2) The ***Measure Detector*** extends the beat information by adding a rhythmical structure. This means that now in addition to beat times also the start times of each bar is known.
3) The ***Chorus Detector*** detects repeated sections in the song which correspond to the refrain.
4) From the textual lyrics the ***Section Processor*** extracts the lyrics for the individual sections. Then it detects repeated sections which contain the lyrics for the chorus. Finally it estimates the approximate duration of each section.

In low-level alignment the main steps are:

1) The ***Vocal Detector*** detects parts in the audio signal which contain vocals.
2) The ***Line Processor*** refines the duration estimation of the *Section Processor* on line level. After that, the estimated durations for each line are rounded off to a multiple of the bar length (which was previously detected by the *Measure Detector*).

3) In the ***Alignment Module*** the information of all other modules is collected in order to start the actual alignment.

Recapitulating, the general strategy in this approach is very different to [WON05] and [LOS99]. The alignment process in these methods is directly based on phoneme and word level alignment. In contrast to this Wang et al. had the idea to minimize the probability for errors in low-level alignment by first exploiting the musical structure of a song in high-level alignment. With their approach they try to leave the naïve way of directly adapting speech recognition methods for the alignment process.

3.3.1 Structural Element Level Alignment

The five structural elements (here also called *sections*) are defined in [WAN04] as follows:

1) Intro (I) is the opening section that leads into the song. It may contain silence and mostly lacks a strong beat (arrhythmic).
2) Verse (V) is a section that roughly corresponds with a poetic stanza and which is the preamble to a chorus section.
3) Chorus (C) is a refrain (lines that are repeated in music) section. It often sharply contrasts the verse melodically, rhythmically and harmonically. Furthermore it also assumes a higher level of dynamics and activity, often with added instrumentation.
4) Bridge (B) is a short section of music played between the parts of a song. It is a form of alternative verse which often modulates to a different key or introduces a new chord progression.
5) Outro (O) is a section which brings the music to a conclusion. Wang et al. assume that it is a section that follows the bridge until the end of the song. It is usually characterized by the chorus section repeating a few times and then fading out.

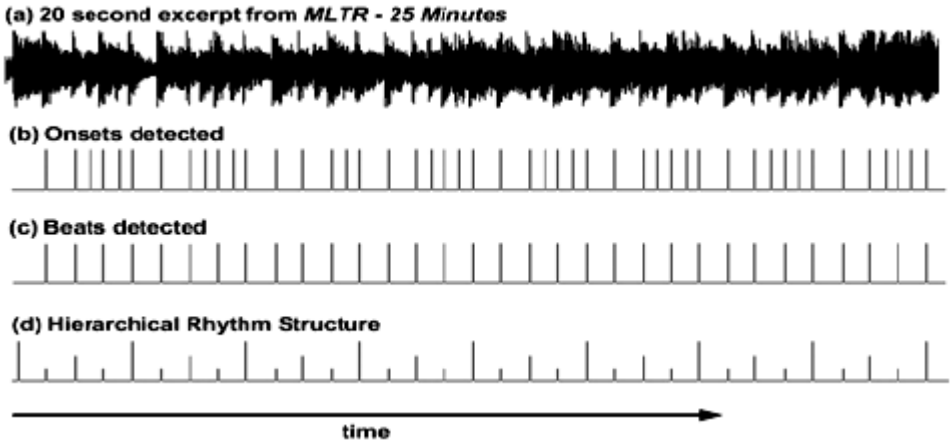
Wang et al. use the following heuristics which are based on an informal survey of popular songs:

1) Instrumental music sections may or may not occur throughout the song.

2) Intro and bridge sections may or may not be present and may or may not contain sung vocals.
3) Popular songs are strophic in form, with a usual arrangement of *verse-chorus-verse-chorus*. Hence the verse and chorus are always present and contain sung vocals.

It is also assumed that the songs have a certain structure of no sung intros, two verses, two choruses, bridge and outro. According to Wang et al. this is the most common structure in popular music.

3.3.1.1 Beat Detector

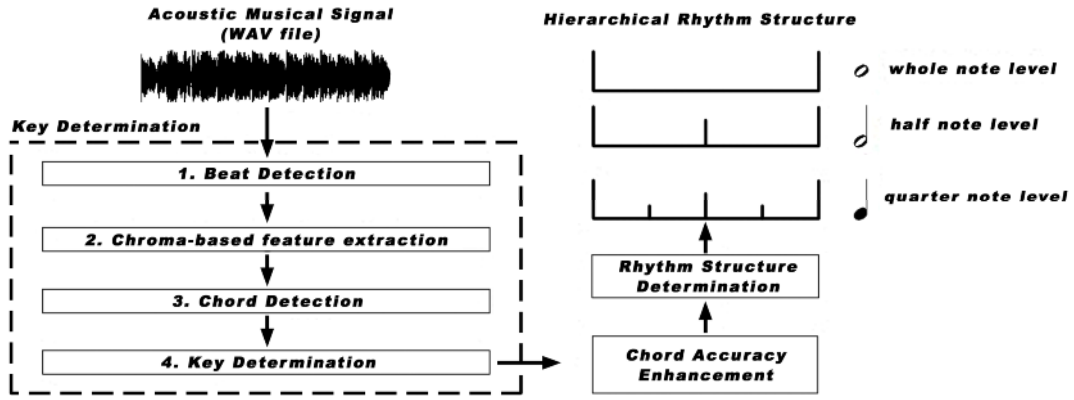


Source: [WAN04], p. N/A.

Figure 3-4: Full hierarchical rhythm structure.

The beat detector extracts rhythm information in real-world musical audio signals at the quarter-note levels. The most common meter for popular songs is 4/4 which means that each measure consists of four beats. Wang et al. assume that the tempo of the input songs is constrained between 40 and 185 beats per minute (BPM) and almost constant.
For quarter note detection the audio signal is framed into beat-length segments. To achieve this, the onsets have to be detected first (see Figure 3-4b). Since additional onsets are often found between the actual beats, a method for eliminating wrong onsets is used. It is based on the assumption that chord changes are more likely to occur on beat times than on other positions and that they are therefore quasi-stationary within the quarter note.

To make the beat detection easier and more accurate a system for determining the key of the song is employed. This is useful because the key defines the diatonic scale that the song uses. In the first post-processing stage (called *Chord Accuracy Enhancement* in Figure 3-5) it is decided if a certain detected chord belongs to the key or not. Chords that do not belong to the detected key are expected to be a result of an error in chord detection. These chords are then eliminated based on a music theoretical analysis of the chord patterns that can be present in the twelve major and twelve minor keys.



Source: [WAN04] , p. N/A.

Figure 3-5: Hierarchical rhythm structure block flow diagram.

3.3.1.2 Measure Detector

The measure detector's aim is to locate the start of measures and therefore extract the hierarchical rhythm information of the song. It is the second post-processing step called *Rhythm Structure Determination* (see Figure 3-5). In addition to the audio signal it also receives the beat positions found by the beat detector as input. Then it detects the starting positions of the measures by assuming that chord changes are more likely to occur at the beginning of a measure than at other positions of beat times. In view of the fact that a measure consists of four quarter notes, the measure detector checks for patterns of four consecutive frames with the same chord to separate all possible measure boundaries. After that the system collects all possible combinations of measures which are separated by four beats. Then it chooses the one which includes the highest number of measures and defines it as the pattern corresponding to the actual measure boundaries. Missing boundary positions are now interpolated across

the rest of the song to arrive at the full hierarchical rhythm structure (see Figure 3-4).

3.3.1.3 Chorus Detector

The chorus detector locates the chorus sections in the audio file and estimates the start and end of each chorus. In [WAN04] its implementation is based on Goto's method [GOT03]. Chorus sections (see Figure 3-6) are identified as the most repeated sections of similar melody and chords using chroma vectors. Since I also use the same approach in my implementation I will explain the details later in chapter 4 .

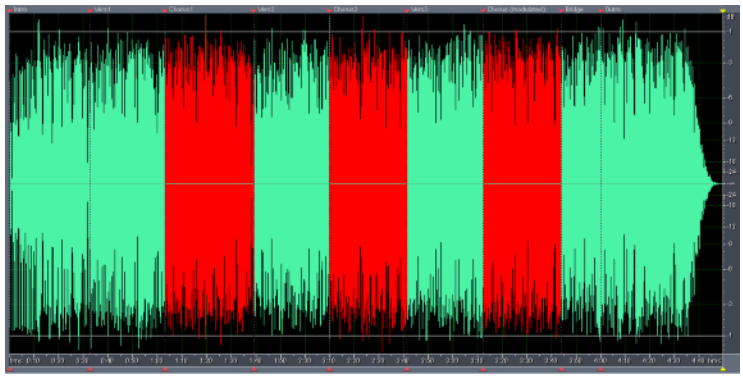


Figure 3-6: Chorus detector has to find the chorus sections (red).

Wang et al. managed to improve the original algorithm by using the rhythmic information gathered by the beat detector. They reduced its complexity by assuming that chords are stable within inter-beat intervals. Therefore instead of using one chroma vector for each 80ms frame they only need to extract one chroma vector from each beat. As a result to the pairwise comparison (see chapter 4.2.3) of the vectors (which is an $O(n^2)$ operation) the improved algorithm only uses about 2% of the time and space required by the original one.

3.3.1.4 Section Processor

The section processor is the last part of the high level section alignment. Its input are the textual lyrics of a song. Assuming that the individual sections are delimited with blank lines the section processor tries to assign the right section type to the lyrics sections. Similar to the audio chorus detector the section processor detects the chorus sections by their high level of repetition. The model

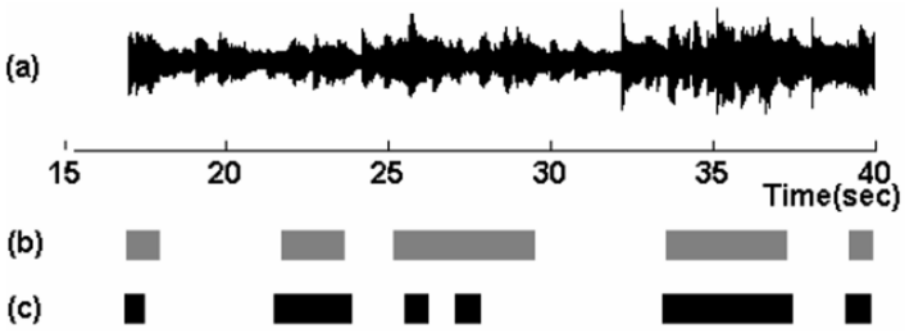
used by Wang et al. accounts for phoneme-, word- and line-level repetition in equal proportions. This should make it insensitive to small errors or variations in the lyrics that may lead to problems for variations of the *longest common subsequence (LCS)* algorithm [LCS06]. For the chorus detection it is defined that chorus sections must be interleaved with one or two other (e.g. verse) intervening sections and to be of approximately the same length in lines.
Another task of the section processor is to calculate an approximate duration of each section. Therefore each word in the lyrics is decomposed into its phonemes. Since phoneme durations in sung lyrics and speech differ, Wang et al. built up a singing phoneme duration database. These durations are learned from annotated sung training data. Finally in order to estimate the length of a section, the durations of all phonemes in this section are summed up.

3.3.2 Line Level Alignment

3.3.2.1 Vocal Detector

A vocal detector is used for finding sections containing human voice in the audio signal. For the implementation of the vocal detector Wang et al. use a *Hidden Markov Model* (HMM) [HUA90] classifier. To improve accuracy, an automatic bootstrapping process which adapts the test song's own models is employed. With it the fact is exploited that the characteristics of vocals show much more differences between different songs than within the same song. For example the vocals in a song by Elton John would sound very different from the vocals in a song by Madonna. But within the song the (intra-song) characteristics of the voice do not change when e.g. comparing the first chorus with the second one.
For the vocal detection process Wang et al. assume that the spectral characteristics of different segments (pure vocals, vocals with instruments and pure instruments) are different. Hence they extract feature parameters based on the distribution of energy in different frequency bands. Again a time resolution of one inter-beat interval is used for the vocal detector. Because the tempo and intensity of vocals are different for every section (intro, verse, chorus, bridge, outro), an intra-song variation is included into the model.
The used training data for the HMMs is manually classified based on section type, tempo and intensity. Since for each class an own model is created, this

results in a total of 2 (vocal or non-vocal) x 5 (section types) x 2 (high or low tempo) x 2 (loud or soft intensity) = 40 distinct HMMs.
Including all these improvements, a classification accuracy of about 84% is achieved. An example of the results of the vocal detector is shown in Figure 3-7.

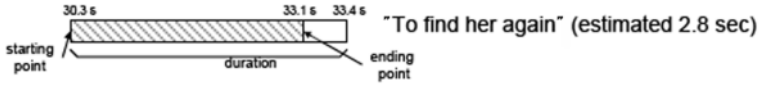


Source: [WAN04] , p. N/A.

Figure 3-7: (a) Signal of a verse segment. (b) Manually annotated and (c) automatically detected vocal segments.

3.3.2.2 Line Processor

The main purpose of the *Line Processor* is to refine the timing estimations for the textual lyrics provided by the *Section Processor*. This gets possible now because the *Line Processor* receives additional input from the beat detector. With that information and the assumption that each text line must start on the beginning of a bar, wrong duration estimates can be corrected. Consequently if the predicted duration for a line ends before a bar end, it is very likely that there is a gap in which the vocals rest. Therefore the estimated line duration will be rounded up to the next bar end (see Figure 3-8).



Source: [WAN04] , p. N/A.

Figure 3-8: Estimated line duration (2.8 sec) will be rounded up to next bar end (3.1 sec).

Furthermore the majority number of bars per line is calculated for each song. All other lines are then forced to be either ½ or 2 times this value.

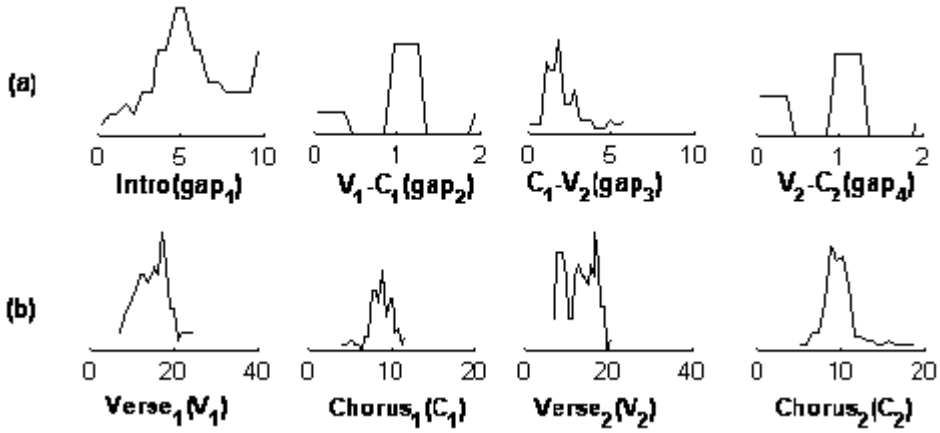
3.3.3 System Integration

Since all needed components were already described, we can now explain how the whole system is assembled. The two alignment levels will get connected to the actual lyrics alignment tool.

3.3.3.1 Section Level Alignment

In the section level alignment detected chorus boundaries are used to determine the boundaries of the verses. Wang et al. observed that the detection of vocal segments is much easier than the detection of non-vocal ones. The reason for this is that both the audio and the text processing can help in the detection process.

For better estimation of the section starts a static gap model is used. It is based on manual annotation of 20 songs. The normalized histogram of all sections in the gap model is shown in Figure 3-9. It can be seen that the duration between verse and chorus (V_1-C_1 and V_2-C_2) is rather stable in comparison to the duration of the sections themselves. This gives the opportunity to determine verse starting points using a combination of gap modelling and positions of the chorus or the song starting point.

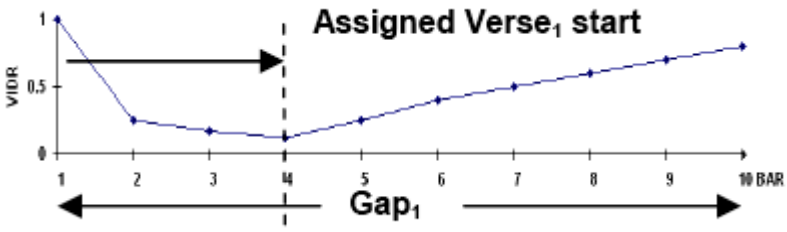


Source: [WAN04] , p. N/A.

Figure 3-9: Duration distributions of (a) non-vocal gaps, (b) different sections of the popular songs with V_1-C_1-V_2-C_2-B-O structure. X-axis represents duration in bars.

LyricAlly uses forward/backward search models, which utilize an anchor point to search for starting and ending points of other sections. For example the be-

ginning of the song is used as an anchor to determine the start of $Verse_1$. In Figure 3-9 it is shown that the intro section is zero to ten bars in length. Over these ten bars of music, the so called *Vocal to Instrumental Duration Ratio (VIDR)* is calculated. It denotes the ratio of vocal to instrument probability within each bar and is calculated by using the results of the vocal detector. In order to determine the beginning of a vocal segment, the global minimum within a window assigned by the gap model is selected (as shown in Figure 3-10).

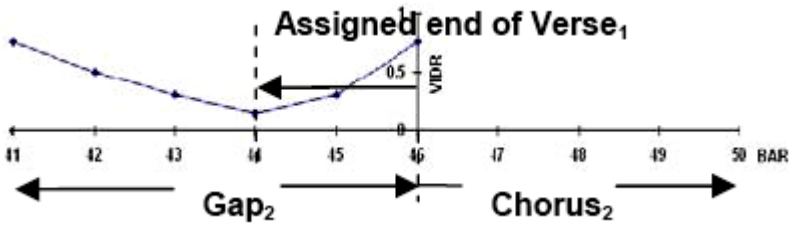


Source: [WAN04] , p. N/A.

Figure 3-10: Forward search in Gap_1 to locate $Verse_1$ start.

This is because usually the beginning of a verse section is characterized by a strong rise of the VIDR. The reason is that in a song the vocals often start at the first verse section and at the beginning of a song the voice detector frequently erroneously detects vocal segments. So the global minimum seems to be a good point for the $Verse_1$ to start at.

In a similar manner the end of $Verse_1$ is detected. The only differences are that now the starting point of $Chorus_1$ is used as anchor and that a backward search model is applied to find the right end location (see Figure 3-11).



Source: [WAN04] , p. N/A.

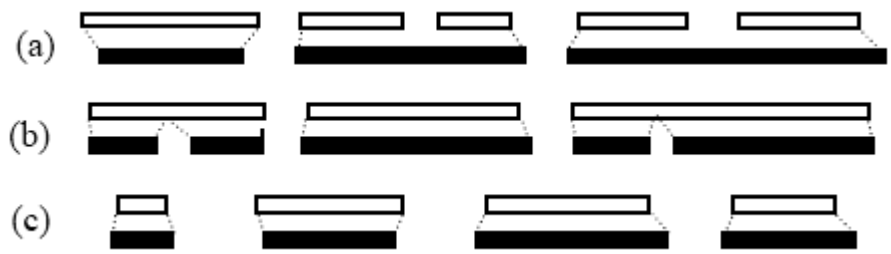
Figure 3-11: Backward search to locate the ending of a verse. [3]

[3] Error in [WAN04]: Here it should be $Chorus_1$ instead of $Chorus_2$.

3.3.3.2 Line Level Alignment

The line level alignment exploits the strengths of both the text processor and the vocal detector. On the one hand the text processor is quite accurate in duration estimation but incapable of providing offsets. This means that the duration of a line can be estimated easily by summing up the average phoneme times. But it is very hard to estimate the start time in the song for each text line. On the other hand the vocal detector is able to detect the presence of vocals but not associate it with the line structure in the song. Therefore the vocal detector can provide the right offsets for the start times of each text line.

Since the segments detected by the vocal detector often do not match the lyrics lines provided by the text processor the main goal is to make them match by grouping or partitioning segments. Wang et al. use the number of text lines as the target number of segments to achieve. This results in three possible scenarios (see Figure 3-12). The number of lyrics lines can either be smaller, equal to or greater than the number of vocal segments. If the number of lyrics lines is smaller or greater than the number of vocal segments, grouping or partitioning needs to be performed. After that a forced alignment is applied. This means that the vocal segments are matched with the right text line, while small time differences are eliminated. If the number of lyrics lines is already equal to the number of vocal segments, no grouping or partitioning is needed. Therefore in this case the forced alignment is executed immediately.



Source: [WAN04] , p. N/A.

Figure 3-12: (a) Grouping, (b) partitioning and (c) forced alignment. White rectangles represent vocal segments and black rectangles represent lyrics lines.

3.3.4 Evaluation

Wang et al. evaluated their approach in two ways. The first one is a holistic evaluation which uses a dataset of manually annotated songs. In order to give a

compact overview of the results, the average and standard deviation of starting point and duration error are shown in Table 3-1.

Alignment Error		Seconds	Bars
Section Level (n = 80)	Starting Point	N(0.80, 9.00)	N(0.30, 1.44)
	Duration	N(-0.50, 10.2)	N(-0.14, 1.44)
Line Level (n = 565)	Starting Point	N(0.58, 3.60)	N(0.22, 0.46)
	Duration	N(-0.48, 0.54)	N(-0.16, 0.06)

Table 3-1: Section- and line-level alignment error over 20 songs. Errors (in seconds) given as normal distributions: $N(\mu,\sigma^2)$.

Seconds are not a good measure for the errors because a one-second error may be perceptually different in songs with different tempo. Therefore measuring the error in bars, as represented in the last column of the table, is more meaningful.

The results show that the calculation of the starting point error is much more difficult than the estimation of the duration of individual lyrics lines. This is because the starting point can only be gained from the audio data. In contrast to this, the duration is estimated by both the audio signal and the textual lyrics.

The second way of evaluation used in this paper is the error analysis of individual modules. Since LyricAlly is only a prototype which integrates separate modules, an individual evaluation of each module can point out bottlenecks in performance and error-proneness. The evaluation results show that the system works best if all components perform well, but performance suffers a lot when certain components fail. If the beat detector fails, all other modules in the system are affected because all estimates are rounded to the nearest bar. However, the error is of course limited to beat length which may be bearable for our purpose. A much more negative effect is observed if the chorus detector fails. The reason for this is that the whole starting point anchors of the chorus sections get lost. Errors in the vocal detector affect both, the starting point and the duration of the sections. Finally since the text processor is only able to calculate durations, its failure leads to less accurate estimations of the duration of sung lyrics lines.

3.4 Overview of my Approach

Since each of the mentioned black boxes shown in Figure 3-3 on its own is a research topic I had to concentrate on the most important ones. These are on the one hand the chorus detector for audio data and on the other hand the text processor for textual data. My approach only concentrates on section level alignment because line level alignment would take too much time and effort to implement. The main reason why only section level alignment is used is that for line level alignment a singing phoneme duration database is needed. This database is not freely available and therefore it would have had to be built on my own by annotating phoneme durations manually for a whole song collection. It is obvious that this would go beyond the time capacities for a diploma thesis.

As I already mentioned above, Wang et al. assume that the songs have a certain structure of no sung intros, two verses, two choruses, bridge and outro. Although they claim that this structure is the most common structure of popular songs it only applies to 40% of their observed songs. In my opinion this is a vast limitation which my implementation approach will try to eliminate. The main difference to Wang et al. is that instead of five possible sections I only use two for the section level alignment. Following Goto's paper *SmartMusicKIOSK: Music Listening Station with Chorus-Search Function* [GOT03] I implemented a chorus detector which detects the start and end points of the refrains in a song. In my approach these sections are called *chorus sections*. Everything else is called *verse section*. Of course this verse section may contain intro, verse, bridge and outro, but they are not distinguished. Consequently songs need not be limited to a certain structure and the system is more flexible. Even if a song started directly with a chorus section, this should be no problem for my approach (see chapter 6.4).

To give a short overview of my implementation I will only explain the seven main steps for the alignment process here. They are shown in the block flow diagram in Figure 3-13. Implementation details are explained later in chapter 4 . At the beginning the user selects a certain song from his hard drive. Then the features extraction can be started. Feature extraction is the process of gaining useful data and information out of the audio signal.

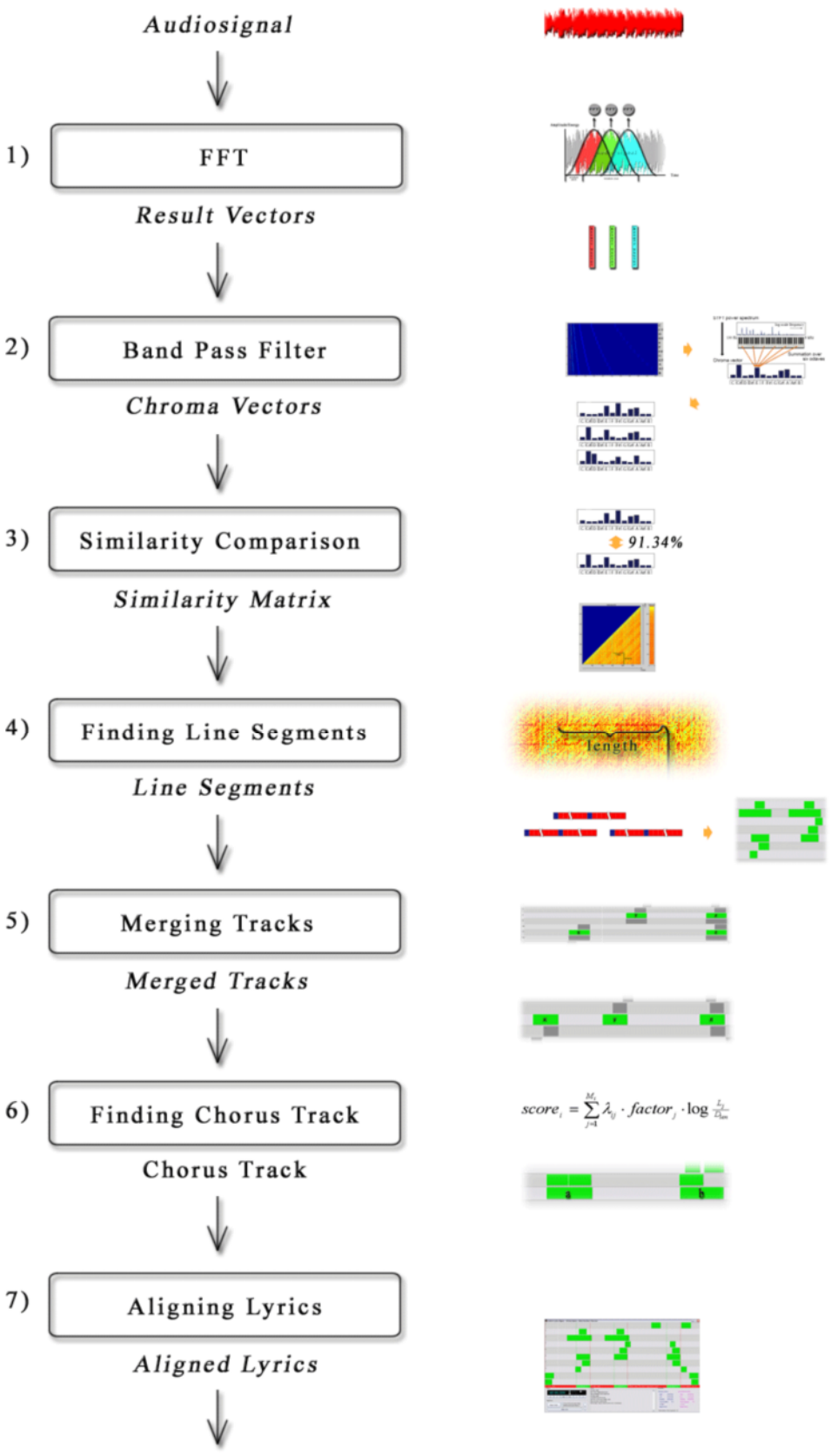


Figure 3-13: The block flow diagram of my approach.

1) First, the audio signal is transformed into frequency space via a FFT. From the FFT so called *result vectors* are obtained. They contain the linear energy distribution of the whole observed frequency spectrum.

2) After that the result vectors are multiplied by a band pass filter matrix in order to set up so called *chroma vectors*. Chroma vectors contain energy values for each of the twelve pitch classes (see Figure 4-1). Hence these values represent the chroma of the notes and chords which are played or sung in the currently observed piece of audio.

3) In the *Similarity Comparison* step the similarity between all the individual chroma vectors is calculated. With the resulting similarity values of this pair wise comparison, the *SimilarityMatrix* is built. For N chroma vectors it contains the similarity values $s_{i,j}$:

$$s_{i,j} = sim(cv_i, cv_j) \qquad 0 \le j < N, \quad 0 \le i < j .$$

Where $sim()$ calculates the similarity between two chroma vectors. The resulting similarity matrix is shown in Table 3-2 (Note: Figure 4-9 shows the vertically flipped similarity matrix).

0	$s_{0,1}$	$s_{0,2}$	$s_{0,3}$	...	$s_{0,n}$
0	0	$s_{1,2}$	$s_{1,3}$	...	$s_{1,n}$
0	0	0	$s_{2,3}$	...	$s_{2,n}$
...	...	...	...	...	...
0	0	0	0	0	0

Table 3-2: Similarity matrix.

4) If there are at least a certain number of consecutive similarity values above a certain threshold in a matrix row then a possible chorus section (a so called *line segment*) was found. These *line segments* are then used for further analysis.

5) Observations show that there seems to be a tendency to redundancy in the found *line segments*. Therefore an algorithm was implemented which

merges such redundant *line segments* in a smart way. For example this means that the two *line segments* shown in Table 3-3 get merged to one single *line segment*.

ID	Start time	End time
1	01:49.314	02:05.598
…	…	…
7	01:49.394	02:05.678

Table 3-3: The *line segments* that would be merged.

This can be done because they have approximately the same starting points in the song and the same length. Consequently they represent the same repeated section in the song and one of them can be eliminated.
Directly linked with the merging process is an algorithm for reconstructing missing *line segments*. This is useful because the chorus detector sometimes does not detect all *line segments*. In this case the missing *line segments* can often be reconstructed by analysing surrounding line segments. The reconstruction modes are not used by default. Therefore the user has to enable them before starting the merge process.

6) All *line segments* only correspond to parts in the song which are possible chorus sections. Therefore the goal of this step is to find out which of these *line segments* actually correspond to the "real" chorus section. Besides the similarity values of the *line segments* additional musical knowledge about chorus sections in popular music is exploited to calculate a score. The higher the score the higher the probability for a *line segment* to correspond to the "real" chorus section in the song.

7) The last step in the alignment process is the actual alignment of the lyrics to the audio data.

3.5 Differences between Existing Approaches and Mine

Although my approach is mainly based on [WAN04] and [GOT03], there are some differences which are discussed in this chapter.

3.5.1 Wang et al. vs. My Approach

The biggest difference to Wang's approach is that my approach only concentrates on section level alignment. Therefore the alignment will never be as accurate as in LyicAlly. However, it makes my implementation much easier because intro, verse, bridge and outro sections need not be distinguished. Furthermore no singing phoneme duration database is needed. Without this database line level alignment would be nearly impossible, since it would only allow very inaccurate duration estimations for text lines [WAN04].

Another important difference is that in my approach neither a beat nor a vocal detector is used. Consequently the system cannot rely on a rhythmical structure and on vocal offsets for the alignment process. But using a beat detector would not only lead to a more accurate alignment. It could also lead to a vast improvement of system performance. The reason for this is that the chords tend to be quasi-stationary during beat times. Therefore one chroma vector could be extracted for each beat instead of for every 80ms. Since the pairwise comparison (see chapter 4.2.3) of the chroma vectors is an $O(n^2)$ operation the algorithm would only use about 2% of the time and space.

For detecting the chorus sections in the lyrics, Wang et al. use a special model which works in a similar way as the audio chorus detector. It accounts for phoneme-, word- and line-level repetition in equal proportions. Therefore Wang et. al. claim that this model also overcomes variations in words and line ordering which could lead to problems for other algorithms like the Longest Commons Sequence (LCS) algorithm. Nevertheless the LCS algorithm is used in my approach which did not cause any problems during test stage. There is another small difference between the two approaches concerning the lyrics. In [WAN04] the section processor estimates the section length using the lyrics. Of course this cannot be done in my approach since the duration of phonemes is used for this calculation.

Furthermore in the approach of Wang et al. it is defined that the chorus sections in the lyrics must be interleaved by one or two other sections (see chapter 3.3.1.4). This means that a repeated chorus would not be accepted while this should be no problem for my aligning method.

3.5.2 Goto vs. My Approach

The chorus detector in my approach was mainly implemented according to [GOT03]. Although Goto's method already seems to work very well, I have developed some improvements.
The first improvement is to introduce a threshold for ignoring outliers while the lines in the *SimilarityMatrix* are analysed for areas with high similarity (see chapter 4.2.4 and Figure 4-12). This makes the method more resistant against sporadic chroma vectors with lower similarity. As the evaluation results in chapter 6 show, found chorus sections tend to be too short. Hence it could perhaps improve the alignment even more if the number of allowed outliers were increased.

My approach eliminates redundant *line segments* by using the merge algorithm described in chapter 4.2.5. Goto's method works a bit different by here. It collects similar *line segments* into groups instead of eliminating them. In my opinion there is no advantage for one of these two approaches. Nevertheless it is easier for the later alignment to only have a minimum number of *line segments*.
In [GOT03] a method for finding *line segments* which were missed by the chorus detector is mentioned. But no further details about the used algorithm are described. Therefore I implemented a method to reconstruct missing *line segments* on my own (see chapter 4.2.5.2).

The last difference to Goto's approach is that I use my own band pass filter to build up the chroma vectors. Goto uses the following formulas for calculating the element $v(t)$ which corresponds to a pitch class c.

$$v_c(t) = \sum_{h=Oct_L}^{Oct_H} \int_{-\infty}^{\infty} BPF_{c,h}(f)\, \psi_p(f,t)\, df\,,$$

where $BPF_{c,h}(f)$ is the band pass filter that passes the signal at the frequency $F_{c,h}$ of pitch class c and octave position h.

$$F_{c,h} = 1200h + 100(c-1)\,.$$

The band pass filter is then defined using a Hann window as follows:

$$BPF_{c,h}(f) = \tfrac{1}{2}\left(1 - \cos\frac{2\pi(f-(F_{c,h}-100))}{200}\right).$$

As can be seen in the formulas, Goto recalculates the values of passing frequencies each time the band pass filter is used. In my approach a band pass filter matrix is generated once before the actual filtering process (a simple matrix multiplication) is started. Therefore my application was optimized by creating this matrix only once at the first program start. Then it is saved to hard disk and can be loaded again for every feature extraction without having to recalculate it every time.

4 Implementation

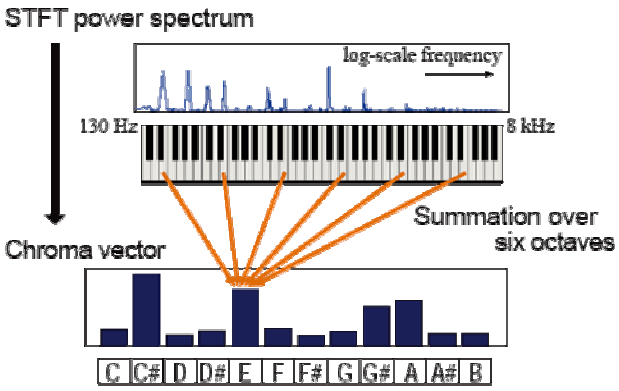
4.1 Advantages & Disadvantages of Java vs. Matlab

For me there were two possible programming languages for implementing the lyrics aligner – Java [JAV06] and Matlab [MAT06].
The main advantage of Java was that I already knew it and therefore could start without having to learn a new programming language. Another point is that Java is one of the standard programming languages today and moreover freely available. Furthermore due to the use of byte code, Java programs can be run on every processor for which a Java Virtual Machine (JVM) exists. The only problem with using Java for audio applications is that it does not provide implemented algorithms for digital signal processing. Therefore the basic algorithms like FFT or multiplication of matrices have to be implemented before the actual implementation of the aligning algorithms can begin.
On the other hand, Matlab already includes these basic algorithms since it is specialised on mathematical programming. So the main advantage of Matlab is that the step of implementing standard algorithms can be omitted. However, a vast limitation is that the Matlab program can only be run in a Matlab environment. Therefore for every computer on which the program should run, a Matlab license has to be purchased.
Nevertheless the final decision was to use Matlab, because in this way it got possible to fully concentrate on the most important algorithms and not to reinvent the wheel.

4.2 Implementation Details

Most pop songs consist of a strophic form. This means that they usually contain an arrangement of alternating verse and chorus sections. Therefore from my point of view it is not necessary to distinguish between five kinds of different sections as done in [WAN04]. For me there exist only two different sections – *chorus sections* and *verse sections* (=all kinds of sections except chorus sec-

tion). Since therefore in my implementation verse sections can contain intro, verse, bridge and outro, it is very difficult to define them. Chorus sections are much easier to find, because they are very similar to each other. So the most important task is to find the locations of the chorus sections. This is done by the *chorus detector* which was implemented based on the approach presented in [GOT03].



Source: [GOT03], p. 35.

Figure 4-1: Chroma vectors contain the energy values corresponding to the twelve pitch classes.

In general the chorus detector tries to locate the chorus sections by using *chroma vectors* for detecting similar sections in the song. Chroma vectors are simple vectors with a length of twelve. Each entry in the vector represents the energy value for one of the twelve pitch classes which can be observed at a certain position of the song (see Figure 4-1). Consequently it is necessary to know which frequencies are involved in the signal. This is the first step of the aligning process (see Figure 4-2).

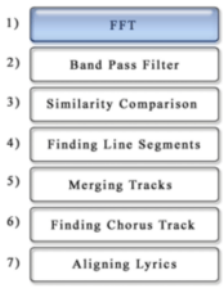


Figure 4-2: Next step in the aligning process: FFT.

4.2.1 Applying the FFT

After smoothing the signal at the current position using a Hann window of the length 4096 it is transformed into frequency space using the FFT. This results in a frequency vector of the length 4096. Since the second half of the vector is the complex conjugate of the first half, it can be ignored for further processing. The so gained FFT result vectors represent the energy distribution of a linear frequency spectrum. This means that the first value in the vector corresponds to the energy measured at the lowest frequency. The lowest frequency can be calculated by

$$LowestFrequency = \frac{SamplingRate}{WindowSize}.$$

For instance a sampling rate of 16000 Hz and a *window size* of 4096 samples would result in:

$$LowestFrequency = \frac{16000}{4096} \approx 4\,\text{Hz}$$

All the other values in the vector are measured at a multiple of this lowest frequency. In other words this means that the second value is measured at $2 \cdot 4 = 8\,\text{Hz}$, the third value at $3 \cdot 4 = 12\,\text{Hz}$ and so on.

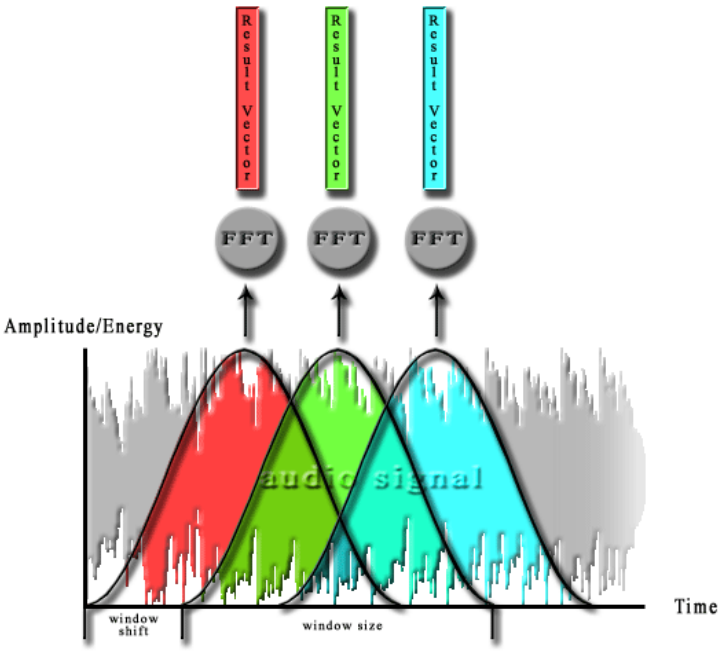


Figure 4-3: Hann window is shifted by the window shift.

To calculate the FFT for the whole audio file, the Hann window is shifted in steps of 1280 samples (=*window shift*). Hence the discrete time step in the current implementation is 80 ms for a sampling rate of 16000 Hz. Consequently a FFT result vector is calculated every 80 ms (see Figure 4-3).
The FFT result vectors get grouped together into the so called *FFTmatrix*. Every column in the *FFTmatrix* contains the energy values for all frequencies in the spectrum. The next task is to isolate only the frequencies which correspond to musical notes. Therefore a band pass filter is needed. So the next step in the aligning process is to construct a suitable band pass filter (see Figure 4-4).

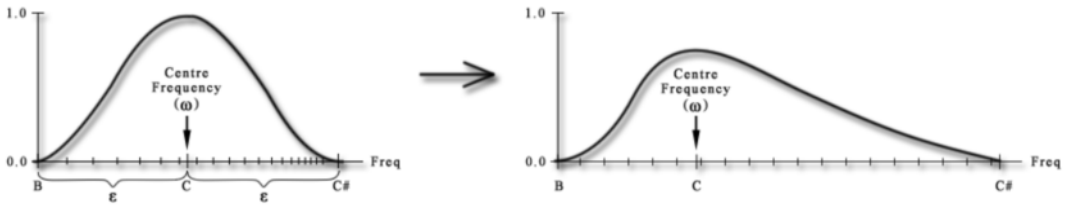
Figure 4-4: Next step in the aligning process: Band Pass Filter.

4.2.2 Band Pass Filter

In my implementation the band pass filter is a matrix which contains values between 0.0 and 1.0. By multiplying this matrix with the *FFTmatrix* the intensities of the individual semitones are computed. The calculation of the semitone's centre frequencies is described in chapter 2.4.
For extracting the semitones Hann windows around the centre frequencies ensure that only as little information as possible is lost. Therefore the Hann window for a certain note N has to meet the following three constraints:

1) It has to start one semitone below N.
2) The peak of the window has to be at the centre frequency of N.
3) The window has to end one semitone above N.

The distance ε between musical notes is distributed in a logarithmic way. But because the measuring frequencies in the FFT result vectors are equidistant in linear scale, the Hann window has to be converted to linear scale too (see Figure 4-5).



$$A_{\log} = \int_{-\varepsilon}^{\varepsilon} i(\omega \cdot e^{t}) \cdot h(\tfrac{t}{\varepsilon})\, dt \qquad A_{lin} = \int_{\omega \cdot e^{-\varepsilon}}^{\omega \cdot e^{\varepsilon}} i(f) \cdot w(f, \omega)\, df$$

Figure 4-5: Left: Normal symmetric Hann window in log-scale. Right: Corresponding result in linear-scale.

ε …is the distance between two semitones in logarithmical scale ($\frac{\ln 2}{12}$)

ω …is the centre frequency of the current semitone (e.g. 261.63 Hz for C_4)

h …is the normalized Hann window function $h(x) = \frac{1+\cos(\pi \cdot x)}{2}, \quad |x| \le 1$

i …is the intensity of the frequency f at the current position of the song

w …is the weighting function resulting from the stretched window

Since the intensity $i(f)$ is only known at the linear distributed measuring frequencies integrals of the form A_{lin} are easy to compute. Nevertheless the goal is to compute the value $A_{\log}$. So the weighting function has to be chosen as $w(f, \omega) = h(\frac{\ln(\frac{f}{\omega})}{\varepsilon}) \cdot \frac{1}{f}$ in order to fulfil $A_{lin} = A_{\log}$.

For the actual computation of the integral A_{lin} only the intensities at the discrete measuring frequencies are available. Therefore the midpoint rule is used.

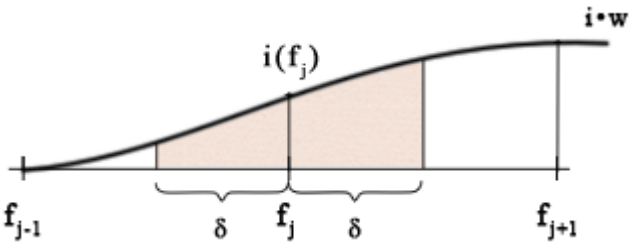


Figure 4-6: Visualisation of the midpoint rule.

δ …half the distance between two linear measuring frequencies

The area below the curve is split up into small segments. Each of them contains exactly one measuring frequency located in its centre. The width of the segments is 2δ (see Figure 4-6). Approximation of each segment's area is done by multiplying the function value at the centre of the interval and the width of the segment. This leads to the following formula:

$$A_{\log} = A_{lin} \approx \sum_j 2\delta \cdot i(f_j) \cdot w(f_j, \omega)$$

However, the intensities $i(f_j)$ are not part of the band pass filter. Instead they come from the current FFT spectrum of the song. Luckily the formula can be represented as matrix-vector multiplication because the intensities are only occurring linearly. Consequently entries of the band pass filter matrix remain to be computed as

$$BPF_{j,\omega} = 2\delta \cdot w(f_j, \omega).$$

To improve the performance of building chroma vectors, the band pass filter is organized as follows:
Every row in the matrix represents a pitch class (e.g. the first row represents the class "C"). In the current implementation the octaves three to eight are analysed. Since each row corresponds to the full linear frequency spectrum, this results in six Hann windows per row. Figure 4-7 shows a graphical view of the completed band pass filter matrix.

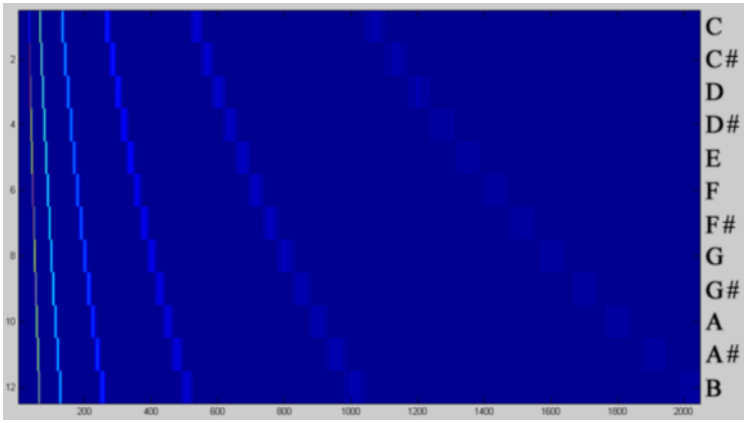


Figure 4-7: Graphical view of the band pass filter matrix. Blue values are close to zero; red values are close to one.

The columns in the *FFTmatrix* (which contain the intensities $i(f_j)$) are multiplied by the twelve rows in the band pass filter matrix. So for each of the twelve rows one value for the chroma vector is calculated. In the end one chroma vector is built for each column in the *FFTmatrix*. As already mentioned before, a chroma vector represents the energy values of the twelve pitch classes. The next chapter will deal with the analysis of similarities between these chroma vectors (see Figure 4-8).

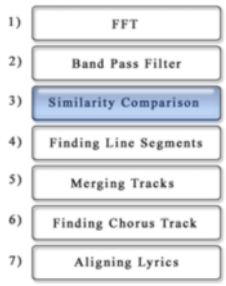


Figure 4-8: Next step in the aligning process: Similarity Comparison.

4.2.3 Similarity Comparison

This chapter explains the implementation details for step 3) in chapter 3.4. The target is still to find the chorus sections in a song. They are characterized by their high degree of similarity. All the other sections do not have this big similarity, because e.g. every verse section is different since the lyrics change. But chorus sections tend to have same instrumentation and same lyrics. In order to measure this degree of similarity a set of chroma vectors was built. Now all these vectors have to be compared with each other.

For calculating the similarity between two chroma vectors $\vec{v}(t)$ and $\vec{v}(t-l)$ Goto uses the following similarity function:

$$r(t,l) = 1 - \left(\left|\frac{\vec{v}(t)}{\max_c v_c(t)} - \frac{\vec{v}(t-l)}{\max_c v_c(t-l)}\right| / \sqrt{12}\right).$$

Where l ($0 \le l \le t$) is the lag between the two vectors. The denominator $\sqrt{12}$ is used for normalizing the similarity value. It represents the length of the diagonal line of the 12-dimensional hypercube with an edge length of 1. Therefore $r(t,l)$ satisfies $0 \le r \le 1$.

Since $r(t,l)$ represents the degree of similarity between two vectors, it only corresponds to 80 ms of the song. Hence the next step is to organize the gained similarity data in order to find longer sections with high similarity. This is done by drawing $r(t,l)$ within the right-angled isosceles triangle in the two-dimensional time-lag space as shown in Figure 4-9. In other words it is a similarity matrix, which shows the similarity between two arbitrary chroma vectors. That is why it will be called *SimilarityMatrix* in the future.

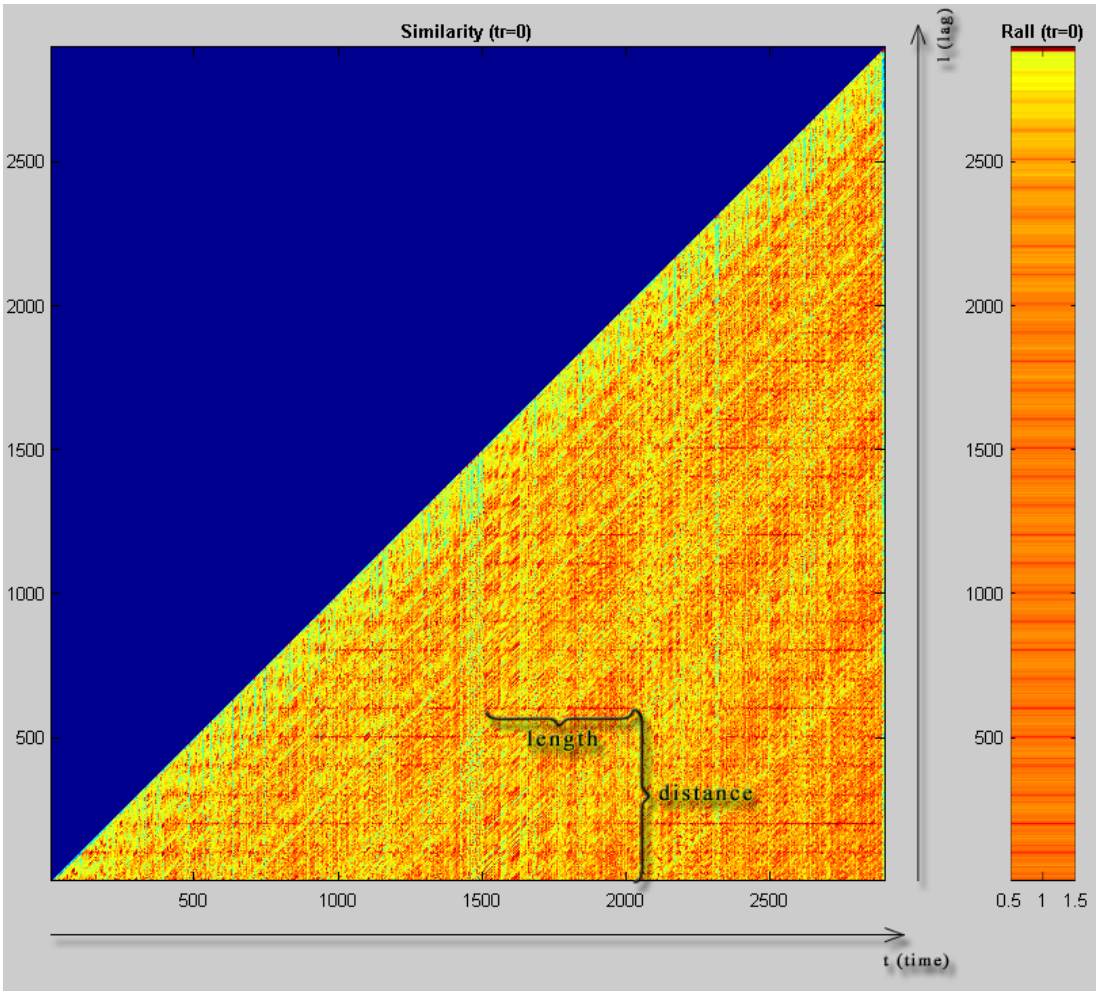


Figure 4-9: The *SimilarityMatrix*.

In the next step of the aligning process (see Figure 4-10) the goal is to find longer segments with high similarity in the horizontal lines of the similarity matrix. These segments are called *line segments*.

Figure 4-10: Next step in the aligning process: Finding Line Segments.

4.2.4 Finding Line Segments

4.2.4.1 The Basic Concept of Finding Line Segments

The human eye is already able to notice horizontal lines with high similarity in the similarity matrix of Figure 4-9. For most songs they are not so clearly visible. But this example shows that the algorithm has to search for those horizontal *line segments*. However, most rows of the *SimilarityMatrix* do not even contain *line segments* with high similarity. In order to speed up the searching algorithm, Goto first calculates the possibility of containing *line segments* at the lag *l* for every line.

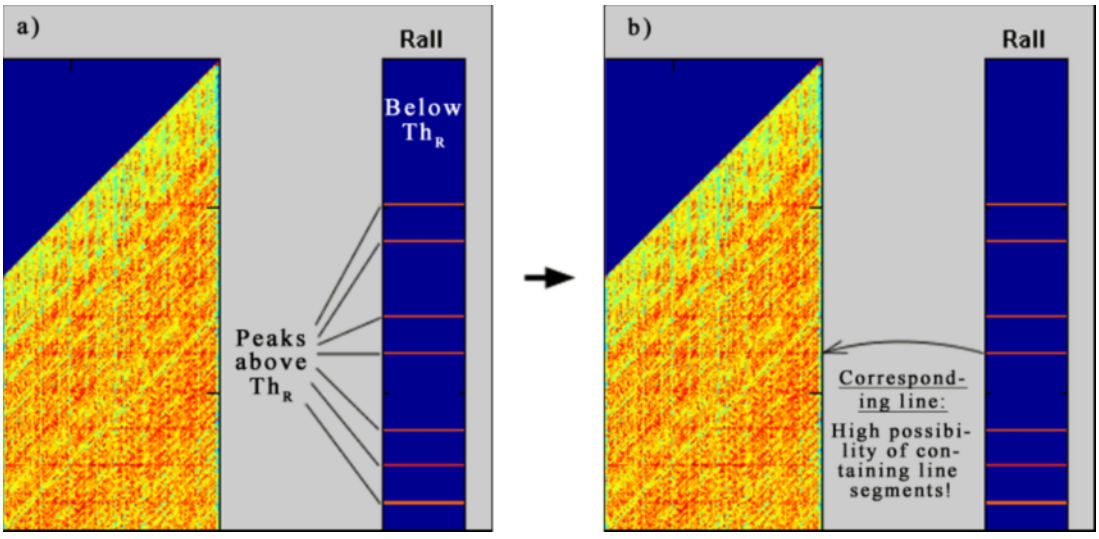


Figure 4-11: Using R_{All} for finding lines with high possibility of containing *line segments*. a) Shows the peak values of R_{All} above Th_R (all other values are displayed in blue colour). b) Every peak value in R_{All} corresponds to a line with high possibility of containing line segments.

This possibility is called $R_{all}(t,l)$ and is defined as follows:

$$R_{all}(t,l) = \int_{l}^{t} \frac{r(\tau,l)}{t-l}\, d\tau.$$

It is evaluated at the time *t* which is the end of the song in the current implementation. Then a threshold Th_R is used in order to divide the values of R_{all} into two classes. The first class contains all values below Th_R and the second one all values above this threshold.
Now the values in the second class are selected from R_{all} (see Figure 4-11a). They indicate that the corresponding horizontal lines in the *SimilarityMatrix* are very likely to contain *line segments* (see Figure 4-11b).
Because the relation between the values in the *SimilarityMatrix* is different for every song, Th_R should be adjusted from case to case. Therefore an automatic threshold selection method is used. It is based on a discriminant criterion. The optimal threshold for dichotomizing the peak heights into two classes is obtained by maximizing the discriminant criterion measure, which is defined by the following between-class variance:

$$\sigma^2{}_B = \omega_1\omega_2(\mu_1 - \mu_2)^2$$

ω_1 and ω_2 are the probabilities of class occurrence ($\frac{number\ of\ peaks\ in\ each\ class}{total\ number\ of\ peaks}$). μ_1 and μ_2 are the mean of peak heights in each class.

With this method, a threshold for R_{all} is obtained. All corresponding lines for which R_{all} is above this threshold are used for further processing. Before continuing, the lines get smoothed in order to remove noise. Therefore for every value v_i in the line, a smoothed value s_i is calculated the following way:

$$s_i = \frac{v_{i-1}+v_i+v_{i+1}}{3}$$

If the index $i-1$ reaches the left edge of the line, instead of v_{i-1} the value of v_{i+1} is taken. The same is done if the right edge of the line is reached.

After this smoothing process a threshold Th_{line} is calculated. This threshold is again adjusted by using the automatic threshold selection method. Now the line is searched for values above Th_{line} from left to right. If such a value was found, the variable *segStart* is set to the current index i. Then the search process is continued until the value at the current index is below Th_{line}. Since according to [GOT03] a potential chorus section must be at least 7.7 seconds long (≈ 97 chroma vectors), the found segment is ignored if it is shorter. Otherwise a region of high similarity was found. Such a region needs not necessarily be one of the "real" chorus sections in the song but it is represents a potential chorus section.

4.2.4.2 Differences to Goto's Approach

In my implementation another threshold Th_{tol} was added. It defines how many values above Th_{line} must follow a single value below Th_{line} in order to ignore this single outlier. The current value of Th_{tol} is 10, which means that at least 10 values above Th_{line} must follow a value below Th_{line} (see Figure 4-12).

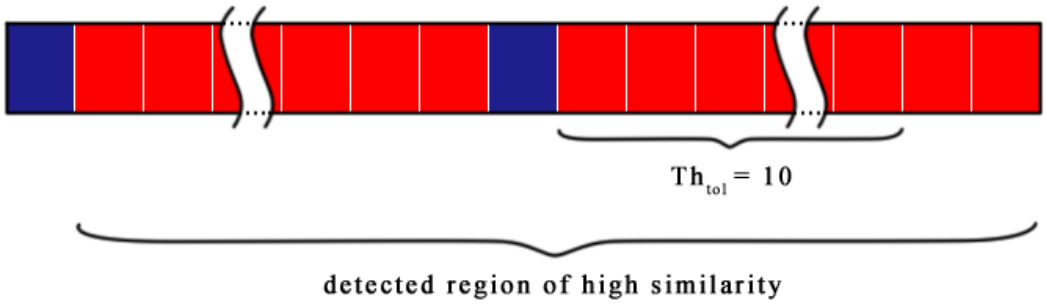


Figure 4-12: The tolerance threshold Th_{tol} within a line. Blue rectangles represent values below and red ones values above Th_{tol}. After the last red rectangle either the end of the line would be reached or at least two blue rectangles would follow.

In my opinion this threshold is very helpful because it is much better to find one long region instead of two shorter ones which are only separated by a single

chroma vector. It could even improve the chorus detection quality if instead of only one value below Th_{line} two or three of them would be tolerated.

Finally if a region with high similarity was found (respecting Th_{tol}), it is added to the line segments matrix (*LSMatrix*). Each of these regions in reality represents two repeated segments in the song which are called *line segments* in [GOT03]. So for every found region two *line segments* are added to the *LSMatrix*. Furthermore each pair of *line segments* gets a unique ID – the so called *TrackID* (see chapter 5.2 for more information). Table 4-1 shows an example of what a typical *LSMatrix* looks like.

LS start [sec]	LS end [sec]	Avg. Similarity (λ)	TrackID
15.374	24.988	0.840	1
83.201	92.815	0.840	1
29.394	47.730	0.753	2
167.498	185.834	0.753	2
61.284	89.332	0.968	3
143.349	171.397	0.968	3
…	…	…	…

Table 4-1: *LSMatrix.*

The first two columns contain the starting and ending time of the *line segment* in seconds. The value λ in the third column is calculated by averaging out all similarity values within the found region in the *SimilarityMatrix*. The last column holds the *TrackID*.

4.2.4.3 Detecting Modulated Chorus Sections

With the presented approach the chorus sections of most songs can be detected. But there may be some problems with a very special class of songs. In some pop songs one of the refrains (mostly the last one) is modulated. This means that the key of the chorus changes for example by a minor second in contrast to the other chorus sections in the song. Such a modulation can be represented by the pitch difference $tr\ (0, 1, \ldots, 11)$ of its key change. So tr denotes the number of semitones a chorus section was transposed. For example, $tr = 5$ means a modulation of five semitones upwards or the modulation of seven semitones downwards.

Consequently the chorus detector has to be adjusted in order to also enable the detection of modulated chorus sections. This can easily be done by making small changes to the existing chroma vector based approach. In a chroma vector $\vec{v}(t)$ the modulation tr corresponds to the amount by which its twelve elements are shifted. Therefore Goto uses a shift matrix for the cyclic shifting of the elements in a chroma vector by tr semitones. The resulting chroma vector is called $\vec{v}(t)'$. Hence it can be said that $\vec{v}(t)$ and $\vec{v}(t)'$ satisfy

$$\vec{v}(t) = S^{tr}\vec{v}(t)'.$$

Where S is a shift matrix defined by

$$S = \begin{pmatrix} 0 & 1 & 0 & \cdots & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ \vdots & & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 & 1 & 0 \\ 0 & \cdots & \cdots & \cdots & 0 & 1 \\ 1 & 0 & \cdots & \cdots & \cdots & 0 \end{pmatrix}.$$

In order to allow the detection of modulated chorus sections, the similarity function $r(t,l)$ has to be changed slightly. The new $r_{tr}(t,l)$ is defined by

$$r_{tr}(t,l) = 1 - \left(\left|\frac{S^{tr}\vec{v}(t)}{\max_c v_c(t)} - \frac{\vec{v}(t-l)}{\max_c v_c(t-l)}\right| / \sqrt{12}\right).$$

This results in twelve different sets of chroma vectors which are all compared to the chroma vectors for $tr = 0$ (the original chroma vectors before modulation). Of course this will take twelve times the processing time and also twelve times the memory of the original $r(t,l)$. Furthermore it will lead to twelve *SimilarityMatrices*.

4.2.4.4 Line Segments vs. Tracks

In the previous chapters a method for extracting regions with high similarity from a song was presented. These regions which are called *line segments* are collected into the *LSMatrix*. For the algorithms of the further aligning process it is no problem to handle this matrix. However, it must be visualized somehow for the user of the lyrics alignment program. Therefore Goto uses so called *tracks* for viewing the *line segments* in an intuitive environment.

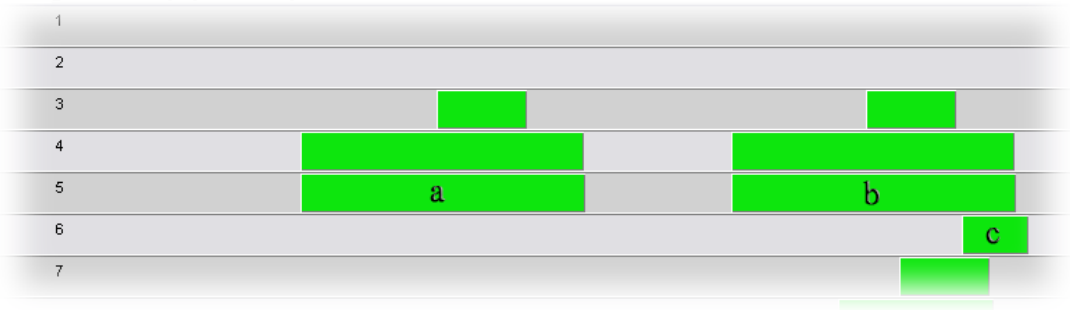


Figure 4-13: *Line segments* (green) placed on tracks.

These tracks are very similar to the tracks used in most studio software. Figure 4-13 shows an example of how tracks look like. In my implementation they are grey horizontal panels which are numbered on the left hand side. While the left end of the tracks represents the beginning of the song, the right end corresponds to the ending.

Each track contains the *line segments* (green panels in Figure 4-13) with a certain *TrackID*. Since every region of high similarity in the *SimilarityMatrix* corresponds to two *line segments*, there exist always two *line segments* with the same *TrackID*. Therefore in this moment each track contains two *line segments*.

The main advantage of the tracks is that they are able to visualize several parts of the song concurrently. For example in Figure 4-13 it is easy to understand the connection between the small green *line segment* on track 3 and the longer one on track 4. The region in the song which corresponds to the small *line segment* is part of the region corresponding to the long one. This is because they occur concurrently on the two tracks and the small *line segment* starts after and ends before the long one.

One problem is that there often exist redundant tracks. An example of this can also be seen in Figure 4-13. The tracks 4 and 5 contain nearly the same *line segments*. To eliminate such redundancies is the main goal of the next step in the aligning process (see Figure 4-14).

Figure 4-14: Next step in the aligning process: Merging Tracks.

4.2.5 Merging Tracks

When searching for the *line segments* (see chapter 4.2.4) there is a tendency for finding redundant repeated segments. The goal of the first part of this chapter is to provide a method for eliminating this redundancy (see chapter 4.2.5.1). This is done by merging two or more tracks to one single track while redundant *line segments* are deleted from the *LSMatrix*. So after applying the merging algorithm a track may contain more than two *line segments*.

Another problem is that sometimes *line segments* are missing. An example of this is shown in Figure 4-13. The *line segment c* occurs at the end of *b*. However, at the end of *a* no such *line segment* in track 6 exists. Since *a* and *b* represent similar parts in the song it can be assumed that there should also exist a *line segment* similar to *c* at the end of *a*. Therefore the second part of the *Merging Tracks* algorithm tries to reconstruct such missing *line segments* by analysing the surrounding line segments on other tracks (see chapter 4.2.5.2).

4.2.5.1 Eliminate Redundant Line Segments

An obvious reason for redundant *line segments* is that the compared chroma vectors represent a very short period of time. Therefore it often happens that after finding a *line segment* in one line of the *SimilarityMatrix*, the next line contains the same *line segment* again. This results in duplicate tracks, which contain the same information. They should be eliminated by the merging algorithm.

A second kind of redundancy is caused by the fact that until now every track may only contain two *line segments*. Consequently when there are for example three chorus sections in a song, at least two tracks would be needed for representing them. Then for example in one track the similarities between the first

and last chorus section are expressed while the other track represents the similarity between the second and last chorus section. Therefore the last chorus section is represented by two *line segments* which is unnecessary.

$x \approx y$ means that the *line segments* x and y correspond to approximately the same region in the song. In other words this means that x and y have approximately (below a certain threshold) the same starting and ending time. $sT(x, y)$ means that x and y are on the same track. Then the following relation between the *line segments* x, y, z_1 and z_2 can be used for merging the tracks (see Figure 4-15 and Figure 4-16)[4]:

$$sT(x, z_2) \wedge sT(y, z_1) \Rightarrow sT(x, y) \Leftrightarrow z_1 \approx z_2$$

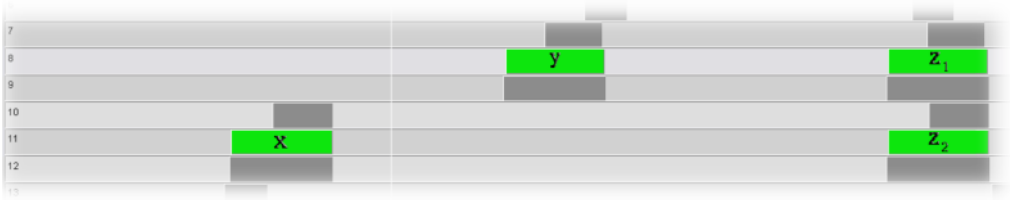


Figure 4-15: Tracks 8, 9, 11 and 12 can be merged to one track.

In Figure 4-15 the *line segment* z_1 and z_2 are redundant. The *line segments* y and z_1 on track 8 correspond to regions in the song with high similarity. The same is true for x and z_2 on track 11. Because $z_1 \approx z_2$ one of these two *line segments* can be eliminated. So for example if z_2 is eliminated, x is moved to track 8 and track 11 is deleted.

Figure 4-16: The resulting track after merging.

[4] See chapters 4.2.4.4 and 5.2.1 for more details about tracks.

Therefore x, y, z_1, z_2 and all their duplicates (grey in Figure 4-15) can be combined to one single track.

The Algorithm (Simplified Pseudo Code):

```
//DEFINITIONS:
//-----------------------------------------------------------------
//Variable names are always printed using italic font style.
//Keywords start with a capital letter and are printed in blue.
//Comments start with "//", end at the end of the line and are
//  printed in green.
//MyList.LEN refers to a list's length (number of elements in it)
//-----------------------------------------------------------------
//These definitions apply for all pseudo code examples!
//-----------------------------------------------------------------

Function MergeTracks(LSMatrix) Returns LSMatrix
Foreach TrackID tID In LSMatrix Do
  Store all line segments with tID In firstList
  Foreach line segment ls1 In firstList Do
    If LSMatrix contains a line segment ls2 which corresponds to
    approximately the same part of the song* Then
      Store line segments with same TrackID as ls2 In secondList
      //calculate score by algorithm described in chapter 4.2.6.2
      Set score1 To CalculateTrackScore(firstList)
      Set score2 To CalculateTrackScore(secondList)
      Remove all line segments in the list with the lower score
        from the LSMatrix (redundant line segments)
      Move remaining line segments from track with lower score to
        track with higher score
    End
  End
End
```

* Defined by a threshold.

The merging algorithm builds up a *firstList* for every *TrackID* in the *LSMatrix*. Then for every *line segment* in this *firstList* another one which corresponds to approximately the same section in the song is searched. If such a *line segment* was found, all *line segments* belonging to the track of the found one are added to the *secondList*. The decision whether two *line segments* occur at the same time is defined by a threshold for the difference of start and end times.
In order to find out which one of both tracks (*firstList* or *secondList*) is more likely to be the chorus track, the algorithm *CalculateTrackScore()* described in chapter 4.2.6.2 is used. After that the reached score for *firstList* and *secondList*

are compared. Consequently the non-redundant *line segments* on the track with the lower score are added to the one with the higher score. This is done by simply changing the *TrackID* in the *LSMatrix*. Finally all *line segments* remaining on the track with the lower score are deleted from the *LSMatrix*.

An important observation was made when testing the algorithm. For some songs a part which does not correspond to a real chorus section is recognized to be similar to a real one. This happens for example if the song starts with a section that (in instrumentation, played chords etc.) sounds very similar to the real chorus section but does not contain any vocals. It also has the same length as the real chorus section. As a result this *line segment* would wrongly get merged with the chorus track. But this would lead to difficulties later at the alignment process because in the audio signal one additional chorus section would be found. Since this "wrong" chorus section does not contain vocals, the similarity is usually significantly lower than between real chorus sections. Therefore another threshold *categoryTh* was introduced. All tracks containing at least one *line segment* with a similarity greater than *categoryTh* belong to the *firstCategory*. All other tracks belong to the *secondCategory*. Now the algorithm enforces that only tracks in the same category can be merged in order to avoid merging real chorus sections with other *line segments*. In the current implementation *categoryTh* is set to 0.8 (=80% of similarity).

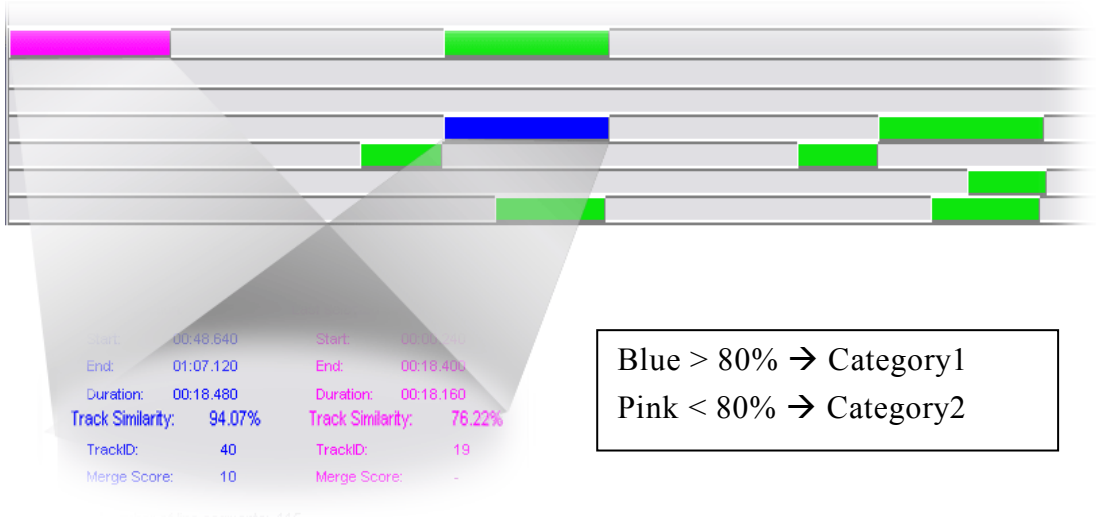


Figure 4-17: Tracks will not be merged because they belong to different categories.

4.2.5.2 Reconstruct Missing Line Segments

Another useful algorithm which raises the probability for a good alignment is directly linked with the merging algorithm. Because the chorus detector is not totally reliable, sometimes important *line segments* are missing. Therefore this algorithm tries to reconstruct missing *line segments* by analysing surrounding *line segments* on other tracks. However, it is disabled by default because it sometimes may cause additional wrong *line segments* too. So the user has to enable it before starting the merging algorithm (see chapter 5.2.5).

The Algorithm (Simplified Pseudo Code):

```
//NOTE:
//-----------------------------------------------------------------
//Some details were omitted for better understanding! However,
//they are described later in the text.
//-----------------------------------------------------------------

Function ReconstructMissingLS(LSMatrix, trackID) Returns LSMatrix
Store all line segments with TrackID equal to trackID In trSegList
Foreach line segment trSeg In trSegList Do
  Store all line segments in the LSMatrix which start or end du-
    ring trSeg In eventSegs
  Foreach line segment E In eventSegs Do
    Set timeDiff To trSeg.StartTime - E.StartTime
    Set LS To line segment in LSMatrix with same TrackID as E
    Create new line segment hypLS
    Set hypLS.StartTime To LS.StartTime + timeDiff
    Set hypLS.EndTime To hypLS.StartTime + Length(trSeg)
    Store all line segments in the LSMatrix which start or end du-
      ring hypLS In newEventSegs
    Set counter To 0
    Foreach line segment b In newEventSegs
      If eventSegs contains a line segment corresponding to b Then
        Set counter To counter + 1
      End
    End
    If counter is above a certain threshold Then
      Store hypLS In newLineSegments
    End
  End
  If newLineSegments is not empty Then
    Choose the most frequent line segments in newLineSegments and
      add it to the LSMatrix
  End
End
```

As inputs the algorithm receives the whole *LSMatrix* and the *TrackID* of the track for which the missing *line segments* should be reconstructed.

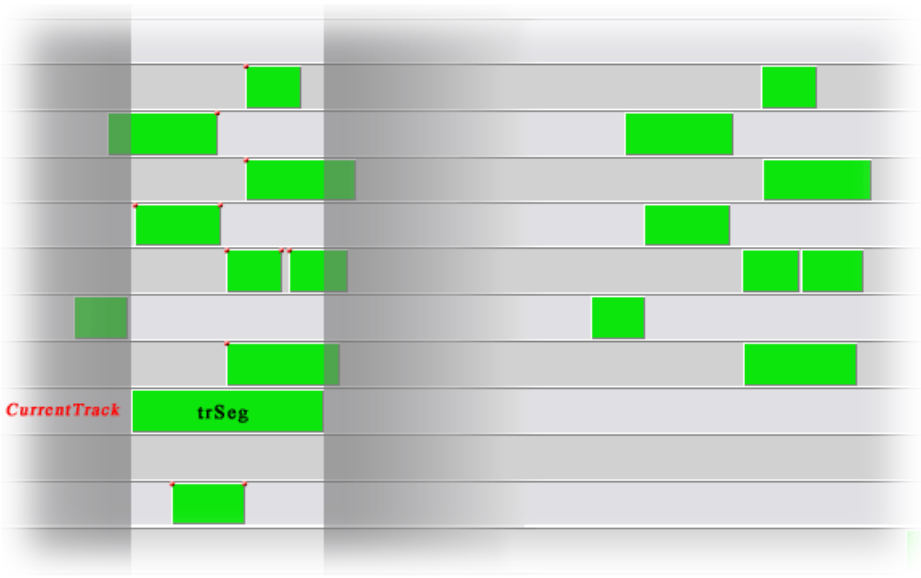


Figure 4-18: Building the list *eventSegs*. Events are shown as small red points.

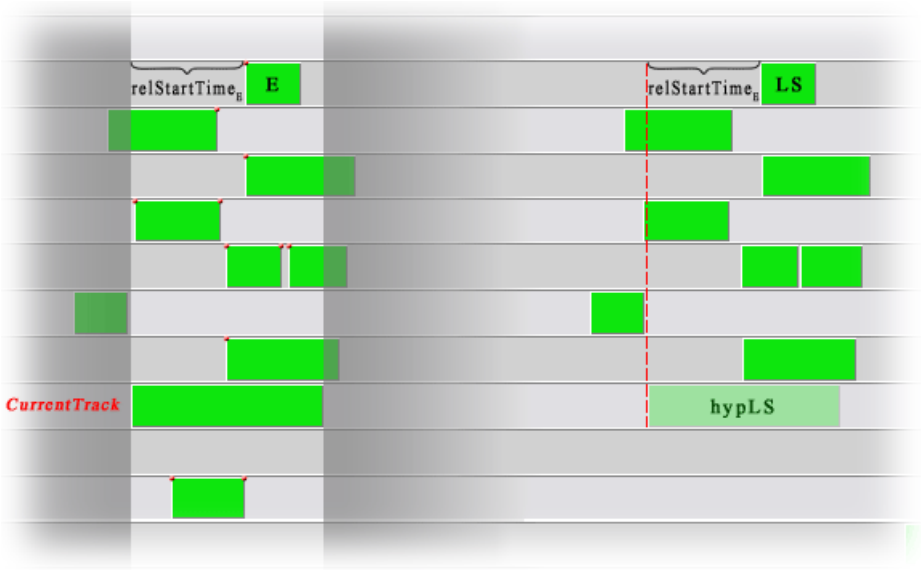


Figure 4-19: The hypothetical line segment *hypLS* is created.

At the beginning a list of *line segments* is built up. It contains all *line segments* that have an "event" during the time when a *line segment trSeg* exists on the "CurrentTrack". Therefore this list is called *eventSegs*. An event is simply either the start or the end of a *line segment* on another track than the "Current-

Track". In Figure 4-18 the events are shown as small red points in the highlighted area. This area indicates the time in the song corresponding to *trSeg*. For easier handling the start and end times of the segments in the list are converted into relative times. This means that in *eventSegs* the start and end times of all *line segments* are stored as the time difference to the start time of *trSeg*.
Now the algorithm searches for every element *E* in *eventSegs* a *line segment LS* with the same *TrackID* in the *LSMatrix*. After finding one, a hypothetical *line segment hypLS* is created (see Figure 4-19). Its start and end times are calculated as follows:

$$startTime_{hypLS} = startTime_{LS} - relStartTime_{E}$$
$$endTime_{hypLS} = startTime_{hypLS} + length(trSeg)$$

Of course it can happen that there is not enough space for *hypLS* on the current track. Then the algorithm searches other fitting *line segments* for the current element *E* and all other *line segments* in the list *eventSegs*.

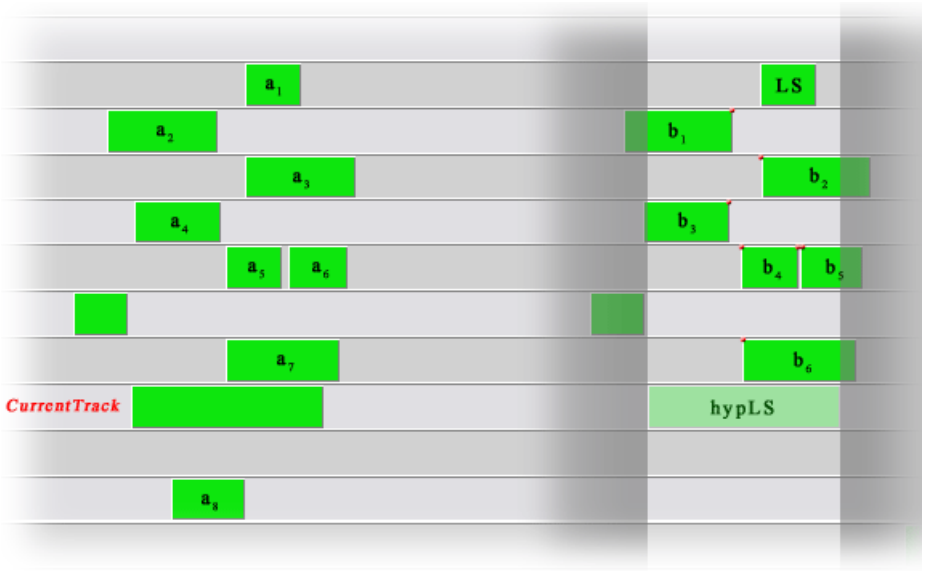


Figure 4-20: Building the second list *newEventSegs*.

Otherwise if there was enough space for inserting *hypLS*, a second event list called *newEventSegs* is built up. This list contains all *line segments* which have an event during *hypLS*. Then for every element in *newEventSegs* (b_1-b_6 in Figure 4-20) a corresponding one in *eventSegs* (a_1-a_8 in Figure 4-20) is

searched. Depending on a threshold *confirmHypLsThresh* a certain number of corresponding *line segments* have to be found in order to confirm the hypothetical *line segment*. For deciding if a corresponding *line segment* was found, the differences between the start and end times are calculated. If both are below the so called *diffThresh*, the counter *corrLSfound* is increased. Is *corrLSfound* greater or equal *confirmHypLsThresh* then *hypLS* is added to a list called *newLineSegments*.

Since there are often lots of possibilities for positioning the reconstructed *line segments*, the list *newLineSegments* can quickly get large. In order to decide, which position should be taken, the *line segments* are separated into classes. For the classification the start time of the *line segments* is taken as the deciding value. If there does not already exist a class for this value, a new one is created. To check if a *line segment* belongs into a certain class, its start time is compared to the one of the first element in this class. If the difference of the start times is below the threshold *classThresh*, the *line segment* is added to this class. In the current implementation *classThresh* is set to one second. The class containing the most *line segments* is used for inserting the real reconstructed *line segment* into the *LSMatrix*. For the start and end time an average value is calculated from all the segments in the class. The similarity value and the *TrackID* are inherited from the segment *trSeg*.

There are several possibilities of how this algorithm is used. Mainly there are two classes. One only uses this algorithm on the track which is supposed to be the chorus track. The other one applies it to all tracks. See chapter 5.2.5 for more details.

Figure 4-21: Next step in the aligning process: Finding Chorus Track.

In the next step of the aligning process the main goal is to select one single track called the *chorus track* out of all available tracks. The chorus track is the

track with the highest probability of containing *line segments* that correspond to the "real" chorus section in the song (see Figure 4-21).

4.2.6 Finding Chorus Track

To pick the correct chorus track out of all available tracks a score is calculated for every track. Only the track with the highest score is finally used by the aligning algorithm. The calculation method of the score exploits some special features of chorus sections in popular music. A very important one is based on so called "half-length sub-segments" described in chapter 4.2.6.1.

4.2.6.1 Half-Length Sub-Segments

The Algorithm (Simplified Pseudo Code):

```
Function HalfLengthSubSegments(lineSeg, LSMatrix) Returns F
F: vector with a length of 2, initialised with [0 0]
Foreach line segment LS In LSMatrix Do
  If length of LS is approximately half the length of lineSeg Then
    If LS starts approximately at the same time as lineSeg Then
      Set F(1) To F(1) + 1
    End
    If LS ends approximately at the same time as lineSeg Then
      Set F(2) To F(2) + 1
    End
  End
End
```

In popular music chorus sections tend to consist of two half-length repeated sub-sections. Consequently a section having such sub-sections is likely to be the chorus section. The target of this algorithm is to find *line segments* in the other tracks which occupy the first or second half of a given *line segment*. It counts the number of sub-segments found for the first and second half (allowed time differences are defined by a threshold). Finally a vector F containing these two values is returned.

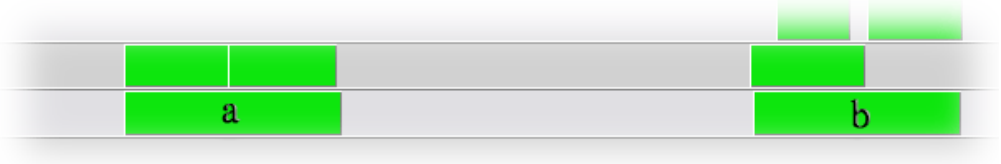


Figure 4-22: Finding half-length sub segments.

Figure 4-22 shows two *line segments* called *a* and *b*. For *a* two half-length sub-segments are found on the track above *a*. One occupies the first and one the second half of *a*. However, only one sub-segment corresponding to the first half is found for *b*. Therefore the algorithm returns the vector $F_a = [1 \quad 1]$ for *a* and $F_b = [1 \quad 0]$ for *b*.

4.2.6.2 Calculating the Track Score

This algorithm is used by both the merge algorithm and the lyrics alignment algorithm. It calculates a score for a certain track. As its input it takes an arbitrary list of *line segments* (such as e.g. the whole *LSMatrix*) and the *TrackID* of the track for which the score should be calculated. By using the score of every single track in the *LSMatrix* it can be decided which track has the highest probability for containing the "real" chorus sections.

The Algorithm (Simplified Pseudo Code):

```
Function CalculateTrackScore(curTrack, LSMatrix) Returns score
Dlen: constant value (=1.4) used for weighting the score

Set score To 0
For all line segments ls on curTrack Do
  Set len To the duration of ls
  Set lamda To the similarity value of ls
  If len less than 7.7 Or len greater than 40 Then
    Set lamda To 0
  End
  Set F To HalfLengthSubSegments(ls,LSMatrix)//see chapter 4.2.6.1
  Set factor To 1.0
  If F(1) greater than 0 And F(2) greater than 0 Then
    Set factor To 2.0
    Set factor To factor + (F(1)-1 + F(2)-1)/8
  End
  Set score To score + lambda*factor*log(len/Dlen)
End
Set score To score/number of line segments on current track
```

The starting point for calculating the score is the similarity value λ of each *line segment LS* in the current track. This similarity value was already calculated by the algorithm for *Finding Line Segments* (presented in chapter 4.2.4) and was stored in the *LSMatrix*.

In [GOT03] the following duration constraint for a chorus section CS is defined:

$$7.7 < length(CS) < 40$$

Hence in my implementation λ is simply set to zero if this constraint is not met.

Another important characteristic of chorus sections is the tendency to consist of two sub-sections with approximately half the chorus length. The algorithm described in chapter 4.2.6.1 searches for *line segments* that are half the length of a given one and have approximately the same starting or ending time. It returns a vector F containing two integer numbers. These numbers correspond to the number of found sub-segments on other tracks corresponding to the first and second half of the given *line segment*.

Now a new variable called *factor* is introduced. It is used later to weight λ by a certain factor depending on how many half-length sub-segments are found for the current *line segment*. At the beginning *factor* is initialized by 1.0. In order to boost the score for *line segments* with at least one sub-segment in each half *factor* is set to 2.0 if $F(1) > 0$ and $F(2) > 0$. After that *factor* is increased by $(F(1) + F(2) - 2)/8$. The denominator of 8 was gained through experiments.

Calculation of the score for a whole track is now very similar to the one presented in [GOT03] except that the individual *line segments* get weighted by $factor_j$:

$$score = \sum_{j=1}^{M_i} \lambda_j \cdot factor_j \cdot \log \tfrac{L_j}{D_{len}},$$

where i represents the *TrackID* of the current track. This track contains a number of M_i *line segments* j. For each of them a $factor_j$ is calculated as explained above. Finally the resulting score for each *line segment* is weighted by $\log \frac{L_j}{D_{len}}$, where L_j is the length of the *line segment* in seconds and the constant D_{len} is $1.4\,\text{sec}$. This ensures that longer *line segments* reach a higher score than short ones. The idea behind this is that tracks with longer *line segments* are

more likely to be the chorus track than e.g. a track that contains small *line segments* with half the length of the chorus section.
After finishing the calculation of the score for the current track, the score value is divided by the number of *line segments* in the track. This is not done in [GOT03] but in my opinion it prevents tracks with lots of *line segments* in it to get a higher score more easily. For example the tracks that only contain half-length sub segments would also result in high scores because there are approximately twice as many segments than on the real chorus track. In the end the algorithm returns the score value for the current track.

After calculating the score for every track in the *LSMatrix* the track with the highest score is selected as the *chorus track*. Now the actual aligning process can start (see Figure 4-23).

Figure 4-23: Next step in the aligning process: Aligning Lyrics.

4.2.7 Aligning Lyrics

The goal of this algorithm is to align the individual lyrics sections to the right audio sections. Before this can be done, the merging algorithm must be executed (see chapter 4.2.5).

4.2.7.1 Preparations of the Lyrics

At the beginning the lyrics are split up in several sections. It is assumed that sections are delimited by one or more blank lines and that the lyrics accurately reflect the words sung in the song. Hence no further pre-processing of the text is necessary. These assumptions were made because there already exist approaches which are able to optimise the textual lyrics (e.g. [KNE05], see chap-

ter 1.2.2). The optimisation of lyrics would go beyond the scope of this diploma thesis since it concentrates on the aligning process of audio and lyrics.
Now all extracted sections are compared with each other by using the Longest Common Substring (LCS) algorithm. A java implementation of this algorithm can be found in [LCS06]. By using the length *len* of the longest common substring, a score is calculated for every pair of lyrics sections *str1* and *str2*:

$$score = len / \max(length(str1), length(str2))$$

From this a similarity matrix called *lcsResultMatrix* is built. For every pair of sections for which the *score* was greater than 0.8 the corresponding place in the matrix is set to 1. A typical *lcsResultMatrix* is shown in Table 4-2.

Section	1	2	3	4	5	6	7
1	1	0	0	0	0	0	0
2	0	1	0	0	0	0	0
3	0	0	1	0	0	0	0
4	0	1	0	1	0	0	0
5	0	0	0	0	1	0	0
6	0	1	0	1	0	1	0
7	0	0	0	0	0	0	1
Sum	1	**3**	1	2	1	1	1

Table 4-2: Lyrics similarity matrix.

To decide which sections are the most similar ones, the values in the columns are summed up. The column which reaches the maximum sum (red in Table 4-2) contains the value 1 for every row which represents a chorus section (green). In this example the sections 2, 4 and 6 are chorus sections and 1, 3, 5 and 7 are assumed to be verse sections.

Now that every preparation step is completed, all necessary information for starting the actual alignment of audio and lyrics was gained. This is the goal of the very last chapter dealing with the implementation details.

4.2.7.2 Audio and Lyrics Alignment

The Algorithm (Simplified Pseudo Code):

```
//DEFINITIONS:
//-----------------------------------------------------------------
//It is already known which of the lyrics and audio sections are
//chorus sections or verse sections. However, here only a minimum
//of variables and lists are used to make the background of the
//algorithm clearer.
//-----------------------------------------------------------------
//Since the whole aligning process is running from right (song
//end) to left (song begin) the word "next" in variable names
//always refers to the previous chorus section in time.
//-----------------------------------------------------------------
Function AlignLyrics()
aChrSec: list containing the line segments of the audio chorus
  track
nextaChrSec: index variable pointing to the next audio chorus in
  aChrSec
nextEndPos: stores the end time of the next audio chorus section
allowVerses: if 1 then a verse section may be inserted
allowChorus: if 1 then a chorus section may be inserted
verseSec: list used for temporarily storing lyrics of verse
  sections
label: the graphical control on which the lyrics are displayed
  (red colour for verse sections and green for chorus sections)

If song ends with a verse section Then
  Set nextEndPos To end time of last audio chorus section
Else
  Set nextEndPos To end time of the last but one chorus section
End

Set nextaChrSec To aChrSec.LEN
Set allowVerses To 1
Set allowChorus To 1


For i Is number of lyrics sections To 1 Do
  If the current lyrics section is a verse section
  And allowVerses is equal to 1 Then
    CODE_FOR_THE_VERSE_SECTION (see page 67)
  End //verse section

  If the current lyrics section is a chorus section
  And allowChorus is equal to 1 Then
    CODE_FOR_THE_CHORUS_SECTION (see page 68)
  End //chorus section
End
```

The whole aligning process is running from the end of the song to the beginning. The reason for this is that I noticed that chorus sections are more likely to occur at the end of a song than at the beginning. This helps if for example one more chorus section was detected in the audio data than in the lyrics. In this case the first audio chorus section would be ignored instead of the last one. Because the aligning process is running from the end to the beginning of the song the meaning of some words in this chapter is special:

- "next" always means the previous section in the time of the song
- "previous" always means the next section in the time of the song
- "last" means actually the last section at the end of the song

The variable *nextEndPos* indicates the end time of the next chorus section in order to know where the next verse section should start. Normally it is set to the end time of the last chorus section. But if the song ends with a chorus section, *nextEndPos* is initialized with the end time of the previous chorus in time.
Another variable called *nextaChrSec* is used for saving the index of the next audio chorus section in the chorus track. For forcing alternating chorus and verse sections, two flags *allowVerses* and *allowChorus* are used. They are both set to 1 at the beginning because songs may end with either a chorus or a verse section. Now the algorithm steps through the lyrics sections from the end to the beginning.

I. Verse Section

```
Set allowVerses To 0 //after the verse(s) a chorus must follow
Set allowChorus To 1 //now a chorus is allowed again
Store next lyrics sections into verseSec until next chorus
  section is reached
If sections in verseSec do not have the same number of lines
Or no more chorus section is left before this verse Then
concatenate all sections in verseSec to 1 big verse section
  Store this verse section again in verseSec
End
Set verseSpace To time difference between beginning of pre-
  vious chorus and nextEndPos
For j Is 1 To verseSec.LEN Do //now display the lyrics
  Set width of label To verseSpace/versSec.LEN
  Set x position of label To nextEndPos + (j-1) * label.WIDTH
  Set text of label To verseSec(j)
End
```

If the current lyrics section is a verse section and *allowVerses* is 1 the aligning of the verse section can start. The flag *allowVerses* is set to 0 and *allowChorus* is set to 1. Then the lyrics of all possibly existing verse sections until reaching the next chorus are added to the list *verseSec*. If the verse sections in *verseSec* do not have the same number of lines, all verse sections in this list are concatenated to one big verse section. If the number of lines in the individual verse sections is the same, it can be assumed that the sections are equally distributed over time. So in this case the available space between the two chorus sections (previous and next one) is divided by the number of verse sections in *verseSec* (see Figure 4-24).

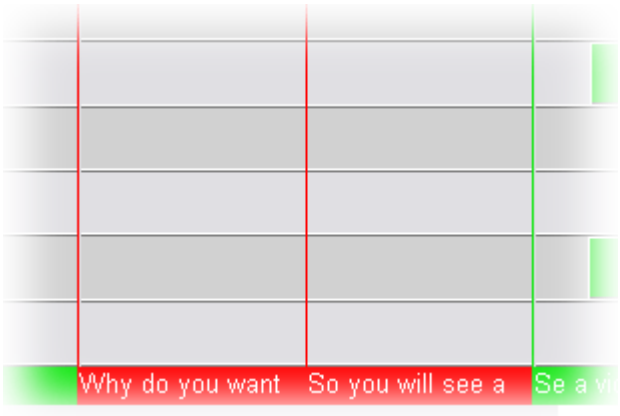


Figure 4-24: Space is divided by two because of two consecutive lyrics sections with the same number of lines.

Consequently there are now several verse sections to be placed in the gap. However, this is not done if the very first lyrics section is involved. The reason for this is that many songs start with an instrumental intro part and the vocals tend to start later. If for example the song in Figure 4-24 directly started with the lyrics "Why do you want…" it would be possible that in the first half of the verse gap (red area) no sung vocals occur. So in this case it is better to concatenate the verse sections to a big one again. This procedure may not apply for all songs, but it worked quite well with most of the songs in the test set.

II. Chorus Section

```
Set allowVerses To 1
Set x position of label To the beginning of current audio
  chorus section
Set width of label To length of current audio chorus section
```

```
If there are more chorus sections in lyrics than in audio
And the next lyrics section is also a chorus Then
  concatenate the current and next chorus sections in the lyrics
  Set text of label To these concatenated lyrics
  If a third chorus section follows in the lyrics Then
    Set allowChorus To 1
  Else
    Set allowChorus To 0
  End
Else
  Set text of label To the current lyrics section
End

If there is at least one more chorus left in the lyrics Then
  Set nextEndPos To ending time of next chorus section
Else
  Set nextEndPos To 0
End

If nextaChrSec greater than 0 Then
  Set nextaChrSec To nextaChrSec - 1
End
```

If the current lyrics section is a chorus section and *allowChorus* is 1 the aligning of the chorus section can begin. The flag *allowChorus* is set to 0 and *allowVerses* is set to 1, because the next section should be a verse again.
The label on which the lyrics will be displayed is moved so that it starts at the position where the audio chorus section starts. Its width is set to the length of the audio chorus section.
A special case occurs if not all audio chorus sections were detected. This means that there were more chorus sections found in the lyrics than in the audio data. As a consequence the algorithm tries to concatenate two consecutive chorus sections found in the lyrics. This is done because it is common that audio chorus sections sometimes contain variations (for example at the end of the song) and therefore will not be detected well by the chorus detector.
At the end of the chorus alignment *nextEndPos* is set to the end time of the next chorus section if there is one left ($nextaChrSec > 1$). Otherwise it is set to 0. Finally *nextaChrSec* is decreased by one and the algorithm steps along to the next lyrics section.
If the first lyrics section is reached the aligning process terminates. Figure 4-25 shows an example of an aligned song.

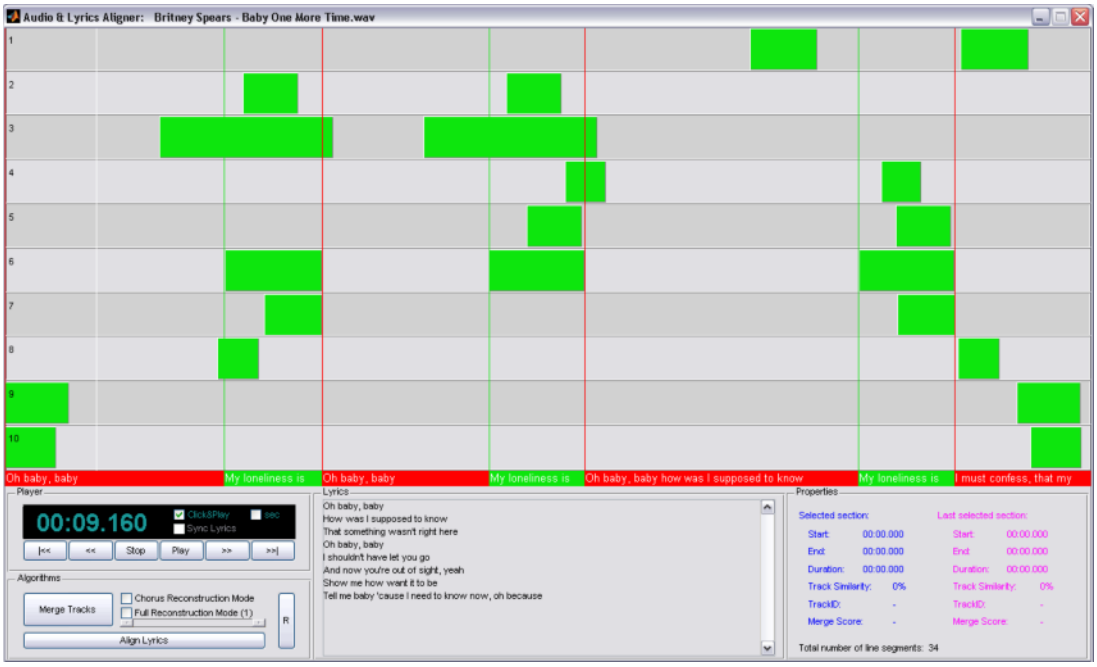


Figure 4-25: The aligned song.

5 User Manual

As it is very uncomfortable to work via the Matlab *Command Window*, a graphical user interface (GUI) was created. So the user e.g. does not always have to enter the whole path for the audio file to be loaded. It can simply be chosen by a common file selection dialog. This chapter explains the graphical user interface of my application, which is called *Lyrics Aligner 1.0*.
Mainly the GUI is divided into two parts. The feature extraction for finding the *line segments* is done in the main window. It is the most time consuming task and uses most of the algorithms introduced in the previous chapters. After finishing this process, the user can click the "Align Audio & Lyrics" button to open the *Audio & Lyrics Alignment Window*. Here the text processing, merging and aligning algorithms are used for aligning the lyrics to the audio signal.

5.1 The Main Window

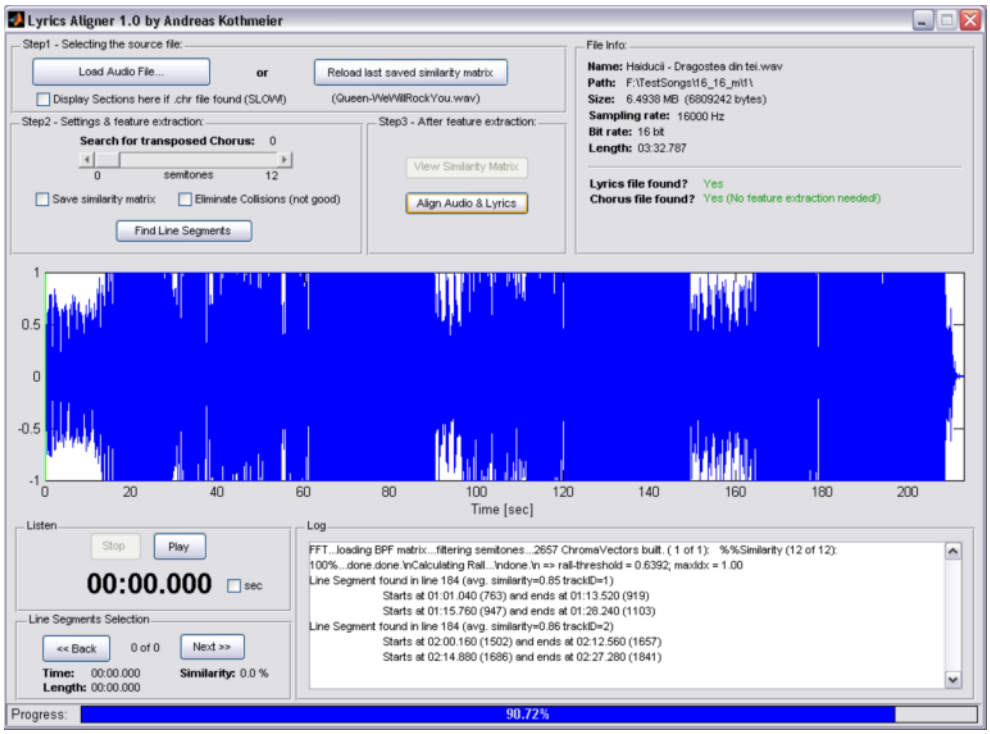


Figure 5-1: The Lyrics Aligner main window.

In Figure 5-1 an overview of the main window is shown. It is divided into three parts which represent different importance levels for the user. The most important items are placed in the upper part. In principle the user even does not need the other two thirds for the whole feature extraction process. Nevertheless they are sometimes very helpful because they provide further information about the audio data and the current process.
In the middle part the audio signal is shown graphically. While the x-axis represents the time in seconds, the y-axis shows the amplitude at a certain time. This way the user is able to get an idea of what the song "looks" like. For calm ballades the amplitude will not be as high as for loud techno tracks with strong beats. Furthermore often e.g. loud chorus sections and more silent areas such as breaks can be distinguished.
The last third contains a player with which the user can listen to the current song. During the feature extraction the user is able to watch the progress by observing the "Log" and the "Progress" bar. After finishing the feature extraction he may directly jump to the positions of the found *line segments* by using the "Line Segments Selection".
This was only a brief overview of the main window. In the following chapters all parts of the window and the used algorithms behind them will be explained in detail.

5.1.1 Step1 – Selecting the source file

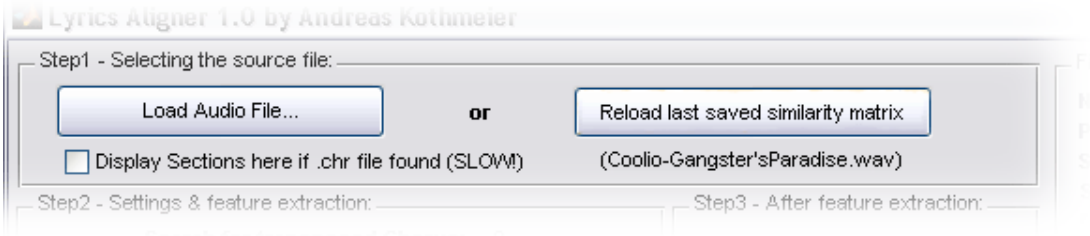


Figure 5-2: Step1 – Selecting the source file.

As Figure 5-2 shows, there are two available buttons in this part of the window. The button "Load Audio File…" opens a file selection dialog. Here the user can choose an audio file from his hard disk. In the current version of *Lyrics Aligner* the file format is limited to *WAVE* files with a sampling rate of 16000 Hz and

bit rate of 16 bit. This can be changed easily later, so the user will also be able to load other file formats including *mp3* files in the future.
Alternatively the button "Reload last saved similarity matrix" can be used. However, at least one feature extraction must have been completed before this button can be pressed. Furthermore the check box "Save similarity matrix" must have been checked before the feature extraction was started. This causes the program to store the *SimilarityMatrix* on the hard disk. When the user presses the button "Reload last saved similarity matrix", the saved *SimilarityMatrix* is loaded into the memory. Then the feature extraction can be continued just at the step after which the similarity matrix has been build. As a consequence the time consuming process of comparing all chroma vectors can be omitted. Of course this speeds up the feature extraction enormously. This feature was mainly built in for testing purpose. It sped up the testing of the algorithms for finding the *line segments* in the *SimilarityMatrix*. For the user it will not be a very useful feature because it only loads the *SimilarityMatrix* of the last processed song.
Much more useful is that the *Lyrics Aligner* automatically stores the *LSMatrix* for the current song after the feature extraction. In order to find this file again when the audio file is loaded the next time, the following simple file name pattern is used:

Audio file: `<path>\<filename>.wav`
LSMatrix file: `<path>\<filename>.chr`

This means that after the feature extraction (when all *line segments* were found) a *.chr* file is generated for the current audio file. The file contains all necessary information for the lyrics alignment process. So the next time the user loads the audio file, the alignment process can be started directly without any feature extraction.

If the user selects the check box "Display Sections here if .chr file found", the starting points of the *line segments* are shown as vertical red lines in the plot below. Since *Matlab* always redraws the whole plot after inserting a line, this vastly slows down the loading process. One reason for selecting this check box is that the user is able to easily recognise interesting parts of the song (accumulation of red lines in the plot). Another advantage is that it enables the possibility to jump between the *line segment* starts using the "Line Segments Selection"

(see chapter 5.1.7). This approach of displaying the starting points of *line segments* by red lines was only used before the implementation of the *Audio & Lyrics Alignment Window*. It is recommended to use this window instead of the red lines because it is much faster, more intuitive and also offers much more features.

5.1.2 File Info

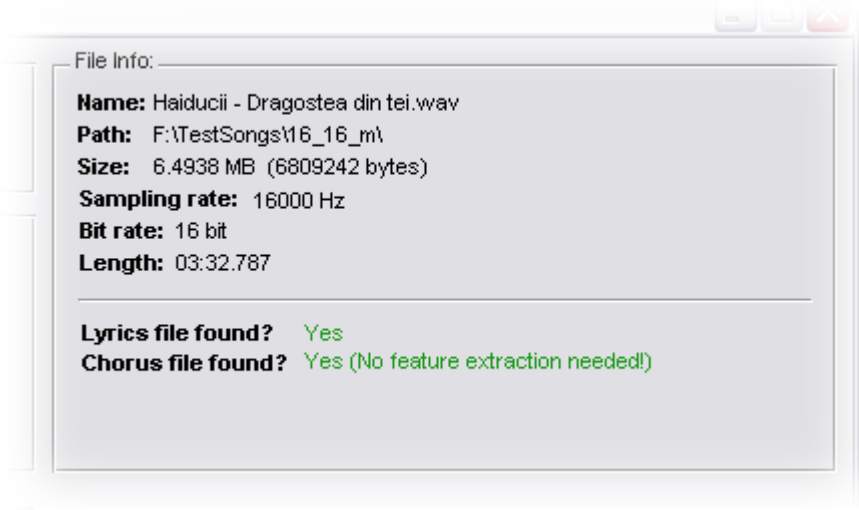


Figure 5-3: File Info.

After loading an audio file, its properties are summarized in the "File Info" section (see Figure 5-3). Above the horizontal line the user can see which file is currently loaded. Furthermore path, size, sampling rate, bit rate and playing length of the file are shown. But the most interesting information for the user is displayed below the horizontal separation line. Here can be checked whether a lyrics file for the current song was found. *Lyrics Aligner* searches for the lyrics corresponding to the current loaded song using the following file naming pattern:

Audio file: `<path>\<filename>.wav`
Lyrics file: `<path>\<filename>.txt`

The *.txt* file is a simple text file containing the lyrics for a certain song (also see chapter 4.2.7). If such a lyrics file was found, a green "Yes" is displayed next to

"Lyrics file found?". Otherwise the green "Yes" is replaced by a red "NO (Lyrics file must be called '<filename>.txt')". In this case the user is able to start the feature extraction and to open the *Audio & Lyrics Alignment Window*. But of course no lyrics are shown and no alignment can be done.

The information next to "Chorus file found?" refers to the *.chr* file. It contains the *LSMatrix* for the current song if the feature extraction for it has already been finished at least once. If such a *.chr* file was found, a green "Yes (No feature extraction needed!)" is shown. Then the user is able to open the *Audio & Lyrics Alignment Window* immediately and start with the lyrics alignment. Otherwise if no *.chr* file was found for the current song, a red "NO (Feature extraction needed!)" is shown. Then the button "Align Audio & Lyrics" is disabled. So the user is forced to click the button "Find Line Segments" in order to start the feature extraction first.

5.1.3 Step2 - Settings & feature extraction

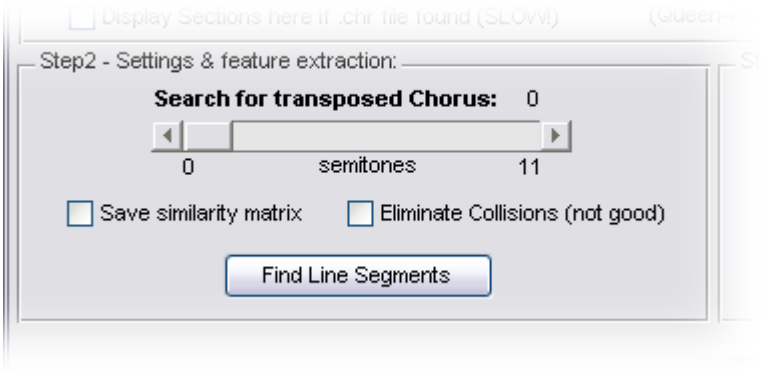


Figure 5-4: Step2 – Settings & feature extraction.

In this part of the main window three settings can be adjusted before starting the actual feature extraction process (see Figure 5-4). With the slider a value TR can be adjusted. TR defines how many semitones a transposed chorus may be away from the normal key of the chorus sections in the song. This means the following for the value tr of chapter 4.2.4:

$$0 \leq tr \leq TR$$

Therefore TR is a measure for detection quality and time complexity at the same time. A high TR value means that the feature extraction gets significantly slower because instead of one, now $TR+1$ similarity matrices have to be calculated. Hence it is only recommended to use a $TR>0$ for songs where not all chorus sections are detected with $TR=0$ or the user is able to hear a modulated chorus.
If the check box "Save similarity matrix" is selected, the similarity matrix is saved to the hard disk after finishing the feature extraction. This enables the user to later reload the similarity matrix by using the button "Reload last saved similarity matrix" instead of "Load Audio File…" (see chapter 5.1.1).
The check box "Eliminate Collisions" was only used for testing purposes and therefore it normally should not be used. If it is selected then no new *line segment*, whose starting point is less than 7.7 seconds away from an already detected *line segment*, is added to the *LSMatrix*.

5.1.4 Step3 – After feature extraction

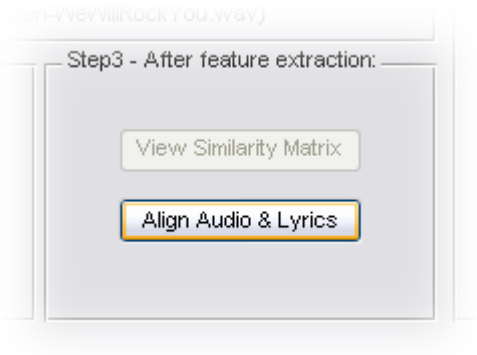


Figure 5-5: Step3 – After feature extraction.

When the feature extraction is finished, the user may want to take a look at the *SimilarityMatrix*. This can be done by clicking the "View Similarity Matrix" button. A window containing a graphical representation of the *SimilarityMatrix* (similar to Figure 4-9) is opened.
However, the more interesting button in the "Step3" panel is the button "Align Audio & Lyrics" (see Figure 5-5). After pressing it, the *Audio & Lyrics Alignment Window* is opened where the actual lyrics alignment process takes place (see chapter 5.2).

5.1.5 Plotted Audio Signal

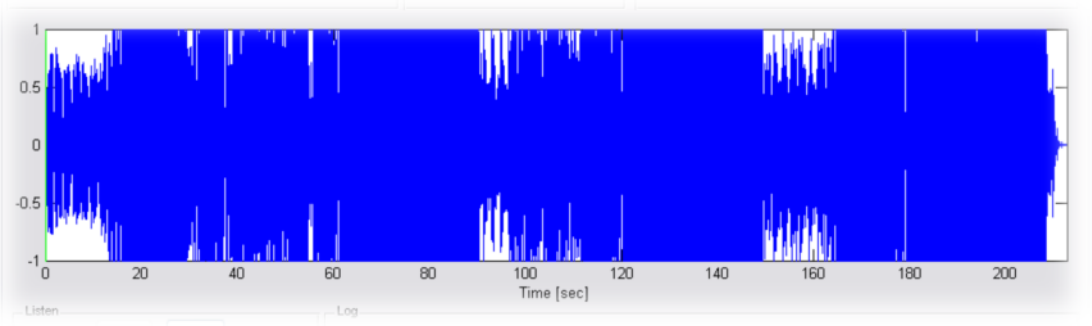


Figure 5-6: Plotted audio signal.

In the middle of the main window the audio signal of the loaded song is plotted (see Figure 5-6). The x-axis represents the time in seconds while the y-axis corresponds to the amplitude. With the help of this plot, the user can easily gain an overview of the loaded song. By clicking the time axis the current playing position can be set. This is used by the small player in the "Listen" panel (see the following chapter).

5.1.6 Listen

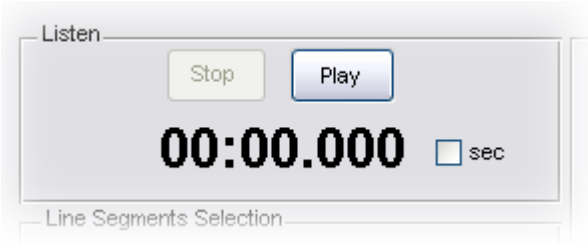


Figure 5-7: Listen panel.

The panel shown in Figure 5-7 mainly acts as a small player for previewing the loaded song. By clicking the "Play" button the play back will be started at the green vertical line in the plotted audio signal. The position of this line can be set by clicking on the time axis in the plot (see chapter 5.1.5). While playing the audio file, the current position is shown in the label below the "Stop" and "Play" buttons. By default the time is shown in the format `<min:sec.ms>`.

But by selecting the check box “sec” it can be changed to <sec.ms>. The “Stop” button is pressed in order to stop the running play back.

5.1.7 Line Segments Selection

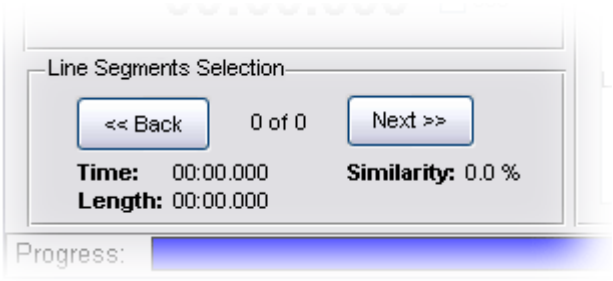


Figure 5-8: Line segments selection.

Here the green cursor (which indicates the position where the player starts the playback) can be directly set to the starting position of the found *line segments*. Using the “Back” and “Next” buttons the user can step through all found *line segments* (see Figure 5-8). Between these two buttons the number of the current selected *line segment* is shown. Below the buttons some important information (start time, length and similarity) about the currently selected *line segment* is displayed. As this panel was mainly used during the implementation when the *Audio & Lyrics Alignment Window* had not existed, it is not recommended to be used by the user. This is because the *Audio & Lyrics Alignment Window* is much easier to understand and additionally offers more features and possibilities.

5.1.8 The Log

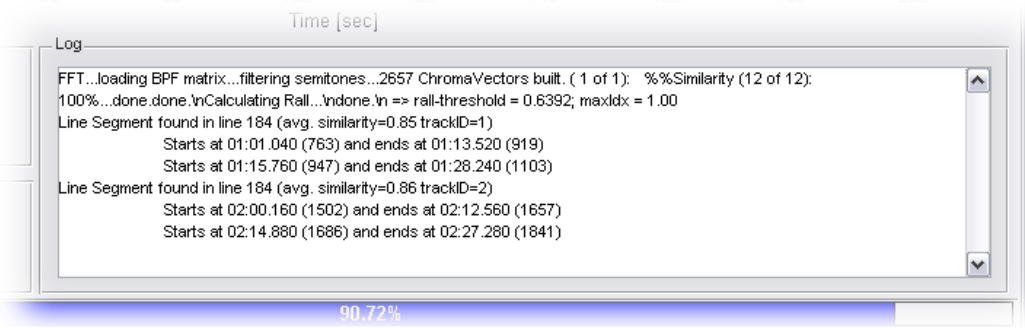


Figure 5-9: The log.

The log is used for displaying important status information during the feature extraction process (see Figure 5-9). Here the user is able to see some of the output which is normally only shown in the MATLAB Command Window. It was mainly intended for debugging purposes.

5.1.9 Progress Bar

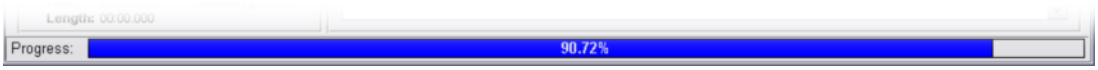


Figure 5-10: Progress bar.

The progress bar shown in Figure 5-10 indicates the progress of the time consuming feature extraction. This way the user can estimate the approximate time it will take to finish.

5.2 The Audio & Lyrics Alignment Window

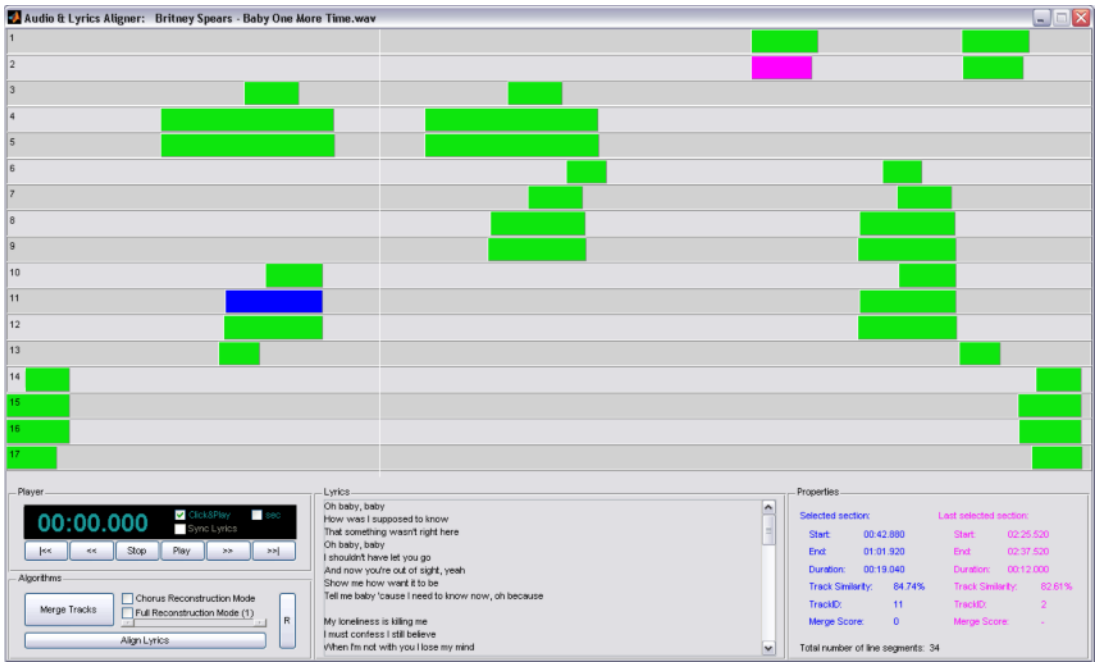


Figure 5-11: Audio & Lyrics Alignment Window.

In this window the actual aligning process is done. Furthermore, here the user can read along the current lyrics while listening to the aligned song (see Figure

5-11). It opens up when the user clicks onto the “Align Audio & Lyrics” button in the main window.

5.2.1 The Tracks

Figure 5-12: The tracks.

The tracks (horizontal grey panels) shown in Figure 5-12 are a graphical representation of the *LSMatrix*. Every *TrackID* in the *LSMatrix* corresponds to exactly one track. The left and right ends of a track represent begin and end of the current song. Each track contains the two *line segments* which were found by analysing the *SimilarityMatrix* as described in chapter 4.2.4. So the two *line segments* where *TrackID=1* are placed in the first track, those with *TrackID=2* in the second one, and so on. *Line segments* are displayed as green rectangles. The start and end of these rectangles correspond to the start and end of the *line segment* it represents. Hence the width of the rectangle is indicating the length of the *line segment* in relation to the length of the whole song. This enables the user to see where repeated sections are located in the song.
After applying the merge algorithm (see chapter 4.2.5) one track can also contain more than only two line segments. This happens because this algorithm combines *line segments* of several tracks to one single track.

5.2.2 The Cursor

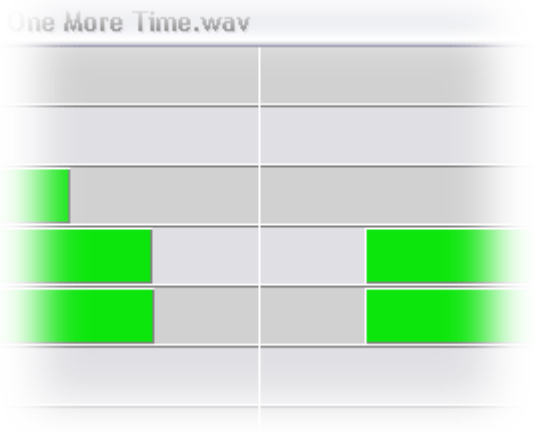


Figure 5-13: The cursor.

The cursor shows the current position within the song (see Figure 5-13). During play back it updates its position automatically to always represent the current position. The user can set its position manually by clicking into the empty space of a track. If the song is currently played back, the player will jump to the new position immediately and continue play back from there. Otherwise only the current position in the song is updated and the user must click the “Play” button to start play back at this position. In both cases the player updates the displayed time in order to show the current position to the user (see chapter 5.2.4).

5.2.3 Properties

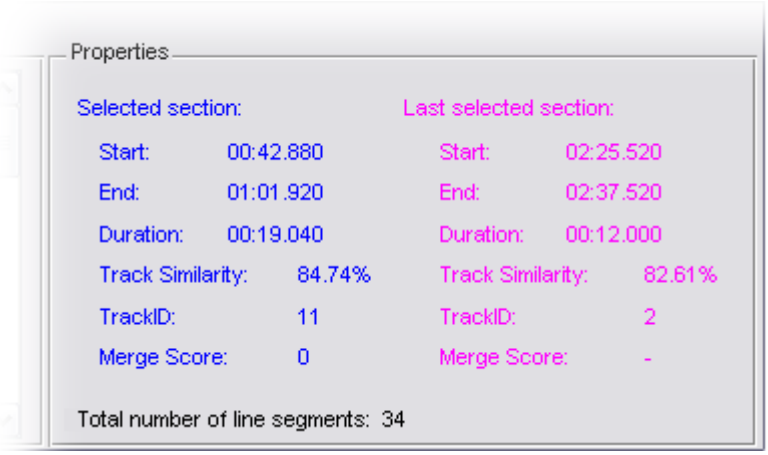


Figure 5-14: The properties.

When a *line segment* is clicked, it gets selected by changing the rectangle's colour from green to blue. At the same time important information (such as *TrackID*, start and end time etc.) for this *line segment* is shown in the "Properties" panel (see Figure 5-14). In order to allow the user to compare two *line segments*, two columns are available in this panel. The blue column shows the information for the currently selected *line segment*, while the pink one shows it for the last selected one. Therefore also the blue colour of the selected rectangle changes to pink if another rectangle is selected. This way it is easy to compare two tracks i.e. for their similarity.

5.2.4 The Player

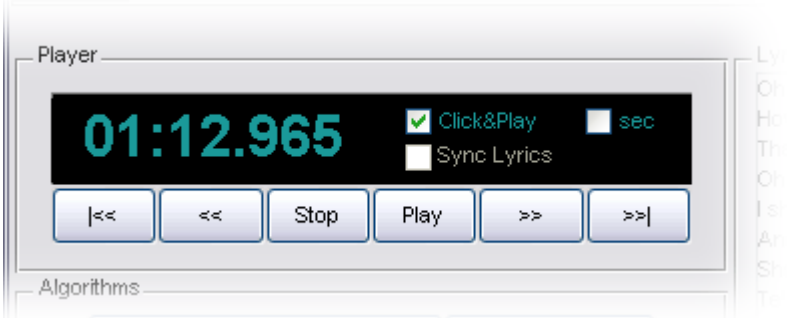


Figure 5-15: The player.

The player shown in Figure 5-15 works very similarly to the one in the main window although it offers some advanced features. The time shown in the left half of the black panel indicates the current time in the song. It can be changed by placing the cursor on a certain position in the song (see chapter 5.2.2). During play back it is automatically updated in order to always display the time of the current position in the song.

The check boxes in the right half of the black panel are used to change the behaviour of the player. If the "Click&Play" check box is selected then every time the user clicks onto a *line segment*, the player starts playing at the start of this *line segment*. This is very useful for listening to certain *line segments* fast and easily. It therefore replaces the feature "Line Segments Selection" of the main window (see chapter 5.1.7). The check box "Sync Lyrics" is only enabled after finishing the alignment process. If it is selected, the lyrics are automatically updated during playback or if the cursor position is changed manually. By selecting the last check box "sec" the user can decide if the time should be dis-

played in the format <sec.ms> instead of the default format <min:sec.ms>.
By clicking onto the "Play" button the song is played back starting at the current cursor position. It can be stopped again by pressing the "Stop" button. With << and >> the user is able to jump five seconds back or forwards. The |<< and >>| buttons enable the user to jump to the last or next section (verse, chorus) start. Of course this only makes sense after the full alignment process was finished.

5.2.5 Algorithms

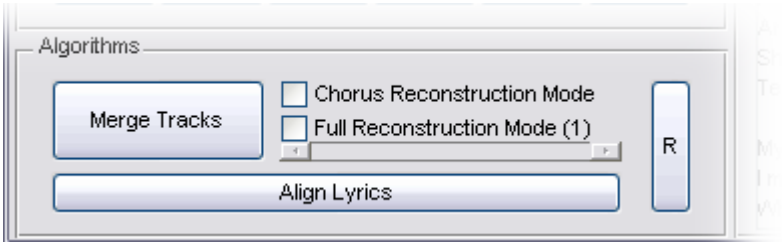


Figure 5-16: Algorithms.

Basically these algorithms are used for the final lyrics aligning process. The buttons on this panel are arranged in two rows (see Figure 5-16). These two rows correspond to the two main steps in the aligning process. In the first step ("Merge Tracks"-button) redundant *line segments* are removed and missing *line segments* are reconstructed. After that the lyrics can be aligned by using the button "Align Lyrics". The "R"-Button can be used in order to reset all changes. The target of the merging algorithm is to eliminate unnecessary *line segments*. It depends on the song how many *line segments* can be eliminated. For some songs hundreds or even thousands of *line segments* are found. Hence they often include a high ratio of redundancy which can often be diminished vastly. Another reason why the merging of tracks is absolutely necessary before the aligning process can start is that after the detection only two *line segments* can be placed on each track (see chapter 4.2.4). Therefore before merging two tracks are needed for a song with three refrains in order to represent the three chorus sections (see chapter 4.2.5 for more details about the merging algorithm).
Until now only the algorithms behind the merge button were discussed. The options to the right of this buttons can be used to activate the reconstruction of

missing *line segments*. This is useful because sometimes not all *line segments* are detected by the chorus detector. By selecting “Chorus Reconstruction Mode” (CRM), the algorithm described in chapter 4.2.5.2 is executed only for the track which is supposed to be the chorus track. If “Full Reconstruction Mode” is selected, this algorithm is applied to all tracks. Furthermore the user can define the number of iterations by using the slider below. Such an iteration always consists of first merging the tracks and then applying the “Reconstruct Missing Line Segments” algorithm. The maximum number of iterations is six. However, normally three or four iterations should be enough since the algorithm may take a long time if many *line segments* were found for the current song. Another disadvantage of too many iterations is that the probability for the occurrence of wrong *line segments* rises (see chapter 4.2.5.2).

5.2.6 The Lyrics

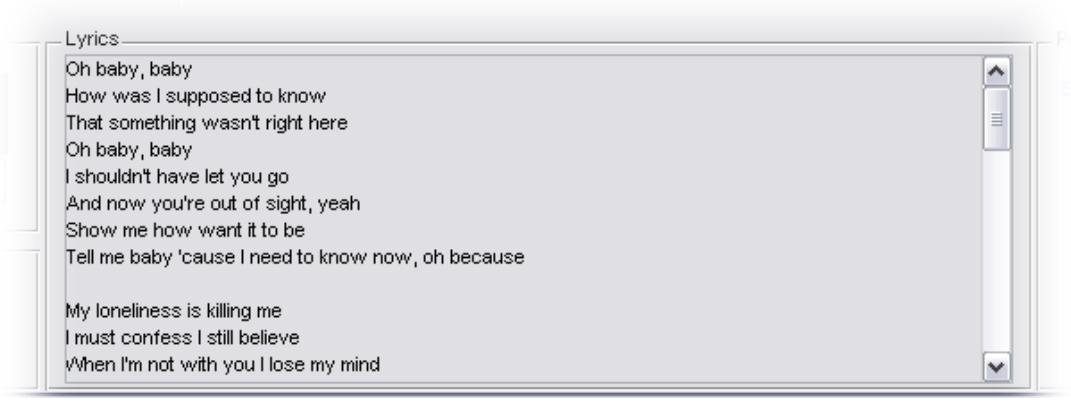


Figure 5-17: The lyrics.

In this text field the lyrics for the current song are displayed (see Figure 5-17). If they have not been aligned yet, the lyrics for the whole song are displayed at once. After aligning the lyrics to the audio signal only the lyrics section for the current part of the song is displayed. The current part is indicated by the cursor (see chapter 5.2.2). So wherever the cursor is moved the lyrics get updated automatically (either while the song is played back or when the user changes the cursor position manually). To use this feature the check box “Sync Lyrics” in the player must be selected (see chapter 5.2.4). Otherwise the lyrics for the whole song are displayed all the time.

6 Evaluation

For evaluating the *Lyrics Aligner 1.0* a test set of 50 pop songs is used. They were selected carefully to cover different music styles like Rock, Dance, Ballads and Hip Hop. In order to make the evaluation faster and more uniform the feature for searching modulated chorus sections was turned off for all songs (see chapter 5.1.3). After the alignment process the results were analysed by the following criteria:

1) Was the right chorus track selected?
   Yes or no.
2) How many percent of the chorus sections were detected correctly?
   Percentage of the correctly detected chorus sections.[5]
3) What was the average error at the start times of the chorus sections?
   Average error in seconds.
4) What was the average error at the end times of the chorus sections?
   Average error in seconds.
5) Was a repeated chorus section detected correctly?
   Yes or no (only if the song contains a repeated chorus).
6) Which merge mode was used?
   Normal, CRM, FRM (<number of iterations>).

Since the merge mode often plays a decisive role for the quality of the alignment, the evaluation is basically split up into two experiments. In the first one only the mode *Normal* is allowed. This means that no missing *line segments* are reconstructed. The tracks are only merged in order to eliminate redundant *line segments*.
In the second experiment the "Normal" merge mode was also used wherever possible. However, if the alignment was not good enough[6], first *Chorus Reconstruction Mode* (CRM) and then *Full Reconstruction Mode* (FRM) was tried to

[5] It is possible that additional wrong chorus sections are detected. Then only chorus sections located at the correct position are used for the calculation.

[6] For example if missing chorus sections or large timing errors occur.

get better results. In CRM mode missing *line segments* are reconstructed only once and exclusively for the track with the highest probability to be the chorus track. In FRM mode the reconstruction process is executed for all tracks. There exist six versions of the FRM mode. The only difference between these six FRM modes is the number of iterations. See chapters 4.2.5 and 5.2.5 for more details.

Of course the first experiment will lead to worse results than the second one. But it simulates the conditions for real automatic alignment, where the user does not need to interact with the program and its settings (e.g. when using the algorithms for batch processing a whole song collection). The second experiment shows the achievable results with fine tuning done by the user.

6.1 Experiment 1 - Using Normal Merge Mode

In this first experiment (see Table 6-1), the correct chorus track was selected in 36 songs. It can be seen that in these cases, at least half of the chorus sections were detected. Furthermore the correct chorus track was selected for 21 of 25 songs in which 100% of the chorus sections were detected correctly. This shows the importance of finding the right chorus track. In spite of this it is also possible that 100% of the chorus sections are detected although the chorus track was not selected correctly. The reason for this is that there sometimes exist two possible chorus tracks which cover all “real” chorus sections in a song. Nevertheless the length of the *line segments* on them is not the same. Consequently the timing errors could be minimized by selecting the other possible chorus track. That is why in some cases the table contains a “No” for “Right Chorus Track Selected” although 100% of the chorus sections were detected (also see chapter 6.4 example song 1).

Serious problems emerged in those 10 cases in which the wrong chorus track was selected. In three of them not even one of the chorus sections was covered by the chorus detector. Although in the remaining seven cases often 66% or even 100% were reached, the average timing errors are mostly very high (as already mentioned above).

	Song	Used Merge Mode	Correct Chorus Track Selected	#Chorus	#Chorus detected	Chorus sect. match (%)	Ø Error start (sec)*	Ø Error end (sec)*	Repeated Chorus Detected
1	A-ha - Take on Me	Normal	Yes	3	2	66,67%	+0.1	-3.5	-
2	Akon - Lonely	Normal	Yes	4	2	50,00%	+0.3	-2.2	-
3	Atomic Kitten - Whole Again	Normal	Yes	4	2	50,00%	+2.4	+1.0	-
4	Backstreet Boys - Everybody	Normal	Yes	4	4	100,00%	+1.25	-1.5	Yes
5	Bell Book & Candle - Rescue Me	Normal	No	3	3	100,00%	0.0	-6.5	-
6	Black - Wonderful Life	Normal	Yes/No?	3	3	100,00%	+3.9	-18.8	-
7	Bloodhound Gang - The Ballad Of Chasey Lane	Normal	Yes	3	3	100,00%	-5.0	0.0	-
8	Bon Jovi - Always	Normal	No	3	0	0,00%	-	-	-
9	Bon Jovi - It's My Life	Normal	Yes/No?	4	3	75,00%	+6.6	0.0	No
10	Brian Adams - Summer of '69	Normal	Yes	2	2	100,00%	+3.0	0.0	-
11	Britney Spears - Baby One More Time	Normal	Yes	3	3	100,00%	+0.1	-1.33	-
12	Britney Spears - Everytime	Normal	Yes	3	2	66,67%	+0.1	0.0	-
13	Captain Jack - Soldier, Soldier	Normal	Yes/No?	4	4	100,00%	+2.7	-2.2	Yes
14	Carl douglas - Kung Fu Fighting	Normal	Yes	4	2	50,00%	+0.1	-2.7	-
15	Christina Aquilera - Genie In The Bottle	Normal	Yes	4	3	75,00%	0.0	-10.9	-
16	Coolio - Gangster's Paradise	Normal	Yes	3	3	100,00%	0.0	+0.66	-
17	Die Firma - Die Eine 2005	Normal	Yes	3	3	100,00%	0.0	0.0	-
18	Eminem - Lose Yourself	Normal	No	3	3	100,00%	-1.0	-5.3	-
19	Eminem - Without Me	Normal	Yes	3	3	100,00%	0.0	+0.2	-
20	Haiducii - Dragostea din tei	Normal	Yes	3	3	100,00%	0.0	-2.0	-
21	IN-MOOD feat. Juliette - The Last Unicorn	Normal	Yes	4	4	100,00%	-1.5	0.0	Yes
22	Jennifer Lopez - Let's Get Loud	Normal	Yes	4	3	75,00%	+0.1	-1.65	-
23	Jennifer Lopez - Waiting For Tonight	Normal	No	3	1	33,33%	+0.2	0.0	-
24	Juli - Perfekte Welle	Normal	Yes	3	2	66,67%	+2.0	+0.25	-
25	Kate Ryan - Desenchantee	Normal	No	3	2	66,67%	+6.6	-5.5	No
26	Kelly Clarkson - Behind These Hazel Eyes	Normal	Yes	4	4	100,00%	0.0	-0.5	Yes
27	Madonna - Sorry	Normal	Yes	3	3	100,00%	-1.2	0.0	-
28	Masterboy - Mister Feeling	Normal	No	3	2	66,67%	+2.0	-8.5	-
29	Melanie C - First day of my life	Normal	Yes	3	3	100,00%	-0.3	-6.5	-
30	Mike Oldfield - Moonlight Shadow	Normal	Yes	2	2	100,00%	0.0	+0.75	-
31	Morrissey - You Have Killed Me	Normal	Yes	3	3	100,00%	0.0	+0.05	-
32	Mr. President - Coco Jambo	Normal	Yes	4	3	75,00%	+0.2	-0.5	-
33	Nena - Irgendwie, Irgendwo, Irgendwann	Normal	Yes	3	4**	100,00%	0.0	-2.0	-
34	Nickelback - This Is How You Remind Me	Normal	Yes/No?	3	2	66,67%	+23.9	-4.0	-
35	Pet Shop Boys - It's a sin	Normal	No	3	2	66,67%	-16.4	+1.74	-
36	Pet Shop Boys - Se a vida e	Normal	Yes	3	2	66,67%	-2.79	+0.1	-
37	Petula Clark - Downtown	Normal	No	3	0	0,00%	-	-	-
38	Pink - Family Portrait	Normal	Yes	3	2	66,67%	+0.4	-1.4	Lyrics merged
39	Pink - Get The Party Started	Normal	Yes	5	5	100,00%	-1.6	-0.3	Yes
40	Queen - Radio Ga Ga	Normal	Yes	3	3	100,00%	0.0	+0.46	-
41	Robbie Williams - Angels	Normal	Yes	3	3	100,00%	+0.5	-17.0	-
42	Roxette - Sleeping In My Car	Normal	No	4	0	0,00%	-	-	-
43	Sarah Connor - From Zero To Hero	Normal	Yes	3	2	66,67%	+7.2	+0.9	-
44	Scorpions - Wind Of Change	Normal	Yes	3	3	100,00%	0.0	0.0	Yes
45	Sean Paul - Get Busy	Normal	Yes	5	4	80,00%	+0.1	-1.8	Yes
46	Soft Cell - Tainted Love	Normal	Yes	2	2	100,00%	+6.0	-4.9	-
47	T.a.t.u - All the Things She Said	Normal	Yes	4	3	75,00%	+0.1	+4.5	-
48	Tina Turner - Simply The Best	Normal	No	2	1	50,00%	+1.16	-7.64	No
49	Wes - Alane	Normal	Yes	4	3	75,00%	0.0	+5.8	-
50	Will Smith - Miami	Normal	Yes	4	4	100,00%	+0.3	-0.4	-

* In the columns “Error begin” and “Error end” the + and – signs in front of the values indicate the most common shift direction of the chorus start and end. If there is for example a + sign in front of the value this means that the detected chorus sections in the song start too late. However, the average value is calculated without taking care on the direction!

** In this song one additional chorus section was detected. Since the value in the column “#Chorus detected” only indicates how many percent of the chorus sections in the song were found, “100%” is shown because all chorus sections were found. This column can also for example show 0% although some chorus sections were found if the found chorus sections do not match the “real” ones.

Table 6-1: Result table of Experiment 1.

Special cases are the four “Yes/No” entries. They correspond to songs for which it would lead to advantages and disadvantages at the same time, if another track would have been selected to be the chorus track. For example it could happen that by selecting another track, one more chorus section would be detected. But at the same time the average timing errors of all chorus sections would increase. The reason for this is that the *line segments* on this new chorus

track are often shorter than on the other one. Consequently it is not clear which case would be better. Since in these cases it is already hard to decide for a human, in my opinion the algorithm for detecting the right chorus track did not fail. Therefore I would count these cases rather to “Yes” than to “No”.

There are nine tracks in which all chorus sections were detected and the average errors for start and end times are below one second. This means that 18% of all songs were aligned perfectly. The average start time error over all songs is 2.02 seconds while the average end time error is 2.72 seconds. Generally the chorus sections tend to start too late and end too early. The main reason for this is that there often occur variations at the start and end of chorus sections which are caused by the different verse sections between them. So for example the vocals (and especially their echo) at the end of a verse section may linger into the beginning of the chorus. Also the transition from a chorus to a verse is often very fluent and therefore may vary from one chorus section to another. This leads to problems at finding the *line segments* in the *SimilarityMatrix* because the start and end parts of such a chorus section are below the threshold Th_{line} (see chapter 4.2.4).

6.2 Experiment 2 - Using Best Merge Mode

Here in 47 songs the right chorus track was chosen. In 43 of the test songs all chorus sections were found and in the remaining seven songs only one chorus section was missed.
An interesting point is that although in three cases the wrong chorus track was selected, in all of them 100% of the chorus sections were found. Nevertheless in these three songs the timing errors are above-average. The reason for this behaviour was already explained in chapter 6.1.

There are thirteen tracks in which all chorus sections were detected and the average errors for start and end times are below one second. In other words this means that 26% of all songs were aligned perfectly. The average start time error over all songs is 1.6 seconds while the average end time error is 2.32 seconds. As in *Experiment 1* the chorus sections tend to start too late and end too early (see Table 6-2 for detailed results).

	Song	Used Merge Mode	Correct Chorus Track Selected	#Chorus	#Chorus detected	Chorus sect. match (%)	Ø Error start (sec)*	Ø Error end (sec)*	Repeated Chorus Detected
1	A-ha - Take on Me	CRM	Yes	3	3	100,00%	+0.1	-3.5	-
2	Akon - Lonely	CRM	Yes	4	3	75,00%	+0.3	-1.7	-
3	Atomic Kitten - Whole Again	FRM (2)	Yes	4	4	100,00%	+2.3	+1.0	-
4	Backstreet Boys - Everybody	Normal	Yes	4	4	100,00%	+1.25	-1.5	Yes
5	Bell Book & Candle - Rescue Me	Normal	No	3	3	100,00%	0.0	-6.5	-
6	Black - Wonderful Life	FRM (2)	Yes	3	3	100,00%	+3.5	-3.4	-
7	Bloodhound Gang - The Ballad Of Chasey Lane	Normal	Yes	3	3	100,00%	-5.0	0.0	-
8	Bon Jovi - Always	FRM (2)	Yes	3	3	100,00%	+8.3	0.0	-
9	Bon Jovi - It's My Life	FRM (2)	Yes	4	4	100,00%	-0.95	0.0	Yes
10	Brian Adams - Summer of '69	Normal	Yes	2	2	100,00%	+3.0	0.0	-
11	Britney Spears - Baby One More Time	Normal	Yes	3	3	100,00%	+0.1	-1.33	-
12	Britney Spears - Everytime	CRM	Yes	3	3	100,00%	+0.1	0.0	-
13	Captain Jack - Soldier, Soldier	CRM	Yes	4	3	75,00%	0.0	0.0	No
14	Carl douglas - Kung Fu Fighting	FRM (2)	Yes	4	4	100,00%	+2.1	-3.5	-
15	Christina Aquilera - Genie In The Bottle	FRM (2)	Yes	4	4	100,00%	0.0	-6.0	-
16	Coolio - Gangster's Paradise	Normal	Yes	3	3	100,00%	0.0	+0.66	-
17	Die Firma - Die Eine 2005	Normal	Yes	3	3	100,00%	0.0	0.0	-
18	Eminem - Lose Yourself	Normal	No	3	3	100,00%	-1.0	-5.3	-
19	Eminem - Without Me	Normal	Yes	3	3	100,00%	0.0	+0.2	-
20	Haiducii - Dragostea din tei	Normal	Yes	3	3	100,00%	0.0	-2.0	-
21	IN-MOOD feat. Juliette - The Last Unicorn	FRM (2)	Yes	4	4	100,00%	+1.25	+0.75	Yes
22	Jennifer Lopez - Let's Get Loud	FRM (2)	Yes	4	3	75,00%	0.0	-2.0	-
23	Jennifer Lopez - Waiting For Tonight	FRM (1)	Yes	3	3	100,00%	+0.7	+4.38	-
24	Juli - Perfekte Welle	Normal	Yes	3	2	66,67%	+2.0	+0.25	-
25	Kate Ryan - Desenchantee	FRM (1)	Yes	3	3	100,00%	+1.0	-3.6	Yes
26	Kelly Clarkson - Behind These Hazel Eyes	Normal	Yes	4	4	100,00%	0.0	-0.5	Yes
27	Madonna - Sorry	Normal	Yes	3	3	100,00%	-1.2	0.0	-
28	Masterboy - Mister Feeling	CRM	Yes	3	3	100,00%	0.0	0.0	-
29	Melanie C - First day of my life	Normal	Yes	3	3	100,00%	-0.3	-6.5	-
30	Mike Oldfield - Moonlight Shadow	Normal	Yes	2	2	100,00%	0.0	+0.75	-
31	Morrissey - You Have Killed Me	Normal	Yes	3	3	100,00%	0.0	+0.05	-
32	Mr. President - Coco Jambo	CRM	Yes	4	4	100,00%	+0.3	0.0	-
33	Nena - Irgendwie, Irgendwo, Irgendwann	Normal	Yes	3	4**	100,00%	0.0	-2.0	-
34	Nickelback - This Is How You Remind Me	FRM (1)	Yes	3	3	100,00%	+3.4	-13.6	-
35	Pet Shop Boys - It's a sin	CRM	No	3	3	100,00%	-16.5	+0.3	-
36	Pet Shop Boys - Se a vida e	CRM	Yes	3	3	100,00%	-2.8	0.0	-
37	Petula Clark - Downtown	FRM (5)	Yes	3	2	66,67%	+3.0	+3.0	-
38	Pink - Family Portrait	Normal	Yes	3	2	66,67%	+0.4	-1.4	Lyrics merged
39	Pink - Get The Party Started	Normal	Yes	5	5	100,00%	-1.6	-0.3	Yes
40	Queen - Radio Ga Ga	Normal	Yes	3	3	100,00%	0.0	+0.46	-
41	Robbie Williams - Angels	Normal	Yes	3	3	100,00%	+0.5	-17	-
42	Roxette - Sleeping In My Car	FRM (1)	Yes	4	3	75,00%	+0.2	0.0	-
43	Sarah Connor - From Zero To Hero	FRM (5)	Yes	3	3	100,00%	+5.0	+4.0	-
44	Scorpions - Wind Of Change	Normal	Yes	3	3	100,00%	0.0	0.0	Yes
45	Sean Paul - Get Busy	FRM (3)	Yes	5	5	100,00%	0.0	+1.5	No,but 1 long
46	Soft Cell - Tainted Love	Normal	Yes	2	2	100,00%	+6.0	-4.9	-
47	T.a.t.u - All the Things She Said	FRM (1)	Yes	4	4	100,00%	+1.25	-4.05	-
48	Tina Turner - Simply The Best	FRM (4)	Yes	2	1,5	75,00%	-2.1	+3.5	Yes (1 of 2)
49	Wes - Alane	CRM	Yes	4	4	100,00%	-2.0	+4.1	-
50	Will Smith - Miami	Normal	Yes	4	4	100,00%	+0.3	-0.4	-

* In the columns "Error begin" and "Error end" the + and – signs in front of the values indicate the most common shift direction of the chorus start and end. If there is for example a + sign in front of the value this means that the detected chorus sections in the song start too late. However, the average value is calculated without taking care on the direction!

** In this song one additional chorus section was detected. Since the value in the column "#Chorus detected" only indicates how many percent of the chorus sections in the song were found, "100%" is shown because all chorus sections were found. This column can also for example show 0% although some chorus sections were found if the found chorus sections do not match the "real" ones.

Table 6-2: Result table of Experiment 2.

Figure 6-1 shows how often the different merge modes were used for the test set. Merge modes with no or only few iterations ("Normal", "CRM" and "FRM (1,2)") are mostly used (92%).

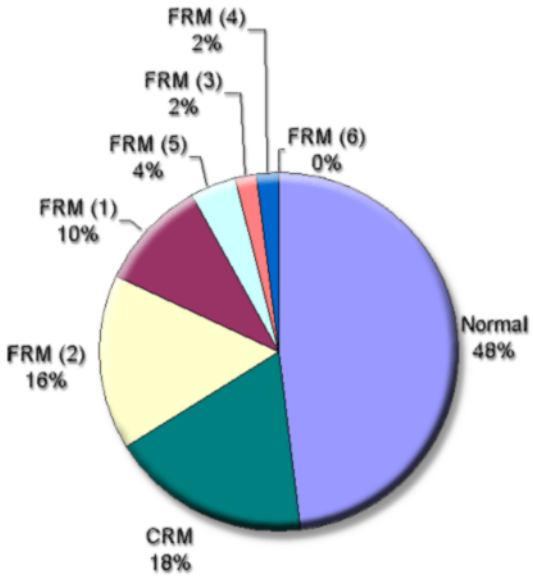


Figure 6-1: Frequency of used merge modes.

6.3 Experiment 1 vs. Experiment 2

Even when counting the four "Yes/No" cases to "Yes" in *Experiment 1*, the selection of the right chorus track could be raised by 14.9% in *Experiment 2*. The average number of chorus sections for the songs in the test set is 3.3. In *Experiment 1* an average number of 2.58 chorus sections per song (which are 78.18%) were detected. In *Experiment 2* this value could be raised to 3.15, which means that 95.45% of all chorus sections were detected.

The average timing error of the chorus section starts could be reduced from 2.02 to 1.60 seconds. This is an improvement of 20.8%. Also the average timing error of the chorus section endings could be reduced from 2.72 to 2.32 seconds which is an improvement of 14.7%.

Both experiments show that detected chorus sections tend to start too late and end too early. This means that the *line segments* found by the chorus detector seem to be too short.

6.4 Qualitative Evaluation

In order to illustrate the strengths and flaccidities of my approach this chapter describes some interesting cases which were picked out of the test set.

1) Song: Bell Book & Candle - Rescue Me
Merge Mode: Normal
Description: Wrong chorus track selected.

Figure 6-2: Wrong chorus track selected.

In this song the average start error of the chorus sections is 0.0, while the end error is 6.5 seconds. In Figure 6-2 these 6.5 seconds correspond to the small length difference between the *line segments* on the first and second highlighted track. If track *b)* were selected as the chorus track, this would lead to a more accurate alignment. However, there are more small segments in the part of the song where a *line segment* exists on track *a)*. Therefore the probability to reach a higher score because of half-length sub-segments is greater for this track (see chapter 4.2.6.1).

2) Song: Bryan Adams - Summer of 69
Merge Mode: Normal
Description: Verse sections with same number of lines.

This song is a good example in which the lyrics of two consecutive verse sections are not simply merged for displaying them at the same time. Here both sections have the same number of lines. Therefore the space between the two chorus sections is split up into two equally long sec-

tions. In Figure 6-3 the parts where the lyrics of the two sections are performed is marked by a small red waveform. Since both of them are completely placed within their section, it can be seen that this is a good approach to reach a finer alignment.

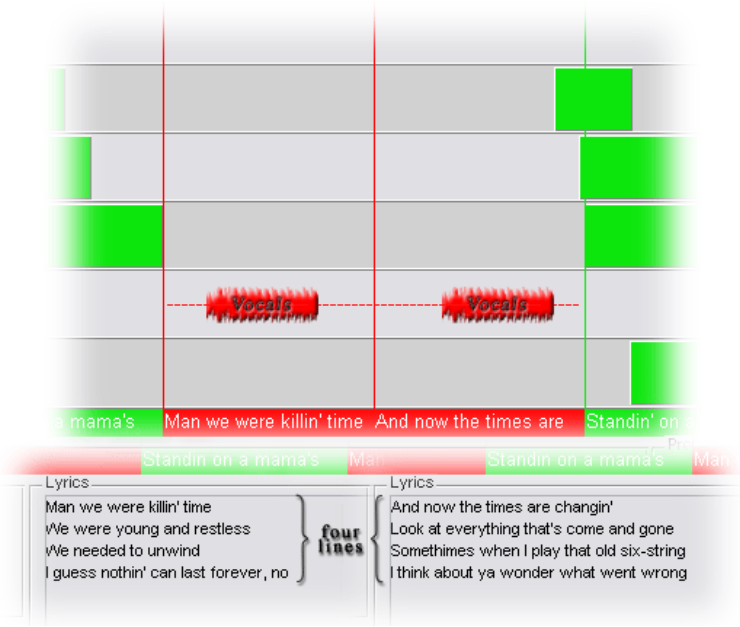


Figure 6-3: Verse sections with same number of lines.

3) Song: Die Firma - Die Eine 2005
Merge Mode: Normal
Description: Vast reduction of *line segments*.

For this song, the chorus detecting algorithm finds 1758 *line segments* (see Figure 6-4). After merging there are only 220 *line segments* left which is a reduction of about 87.5%. This helps the aligning algorithm a lot since many redundant *line segments* are eliminated. Therefore the decision for finding the chorus track is much easier. However, the most interesting fact about the alignment in this song is that although only the "Normal" merge mode is used, a perfect alignment can be achieved. This means that 100% of the chorus sections were detected and 0.0 seconds average start and end error was measured during evaluation.

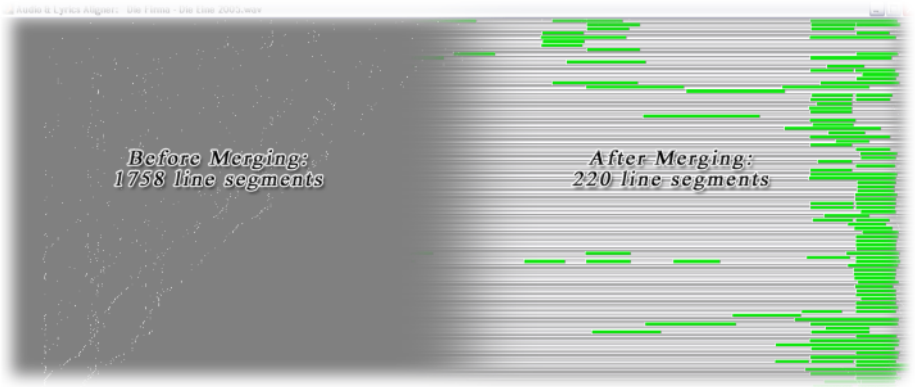


Figure 6-4: Vast reduction of *line segments*.

4) Song: Haiducii - Dragostea din tei
Merge Mode: Normal
Description: Chorus section ends too early.

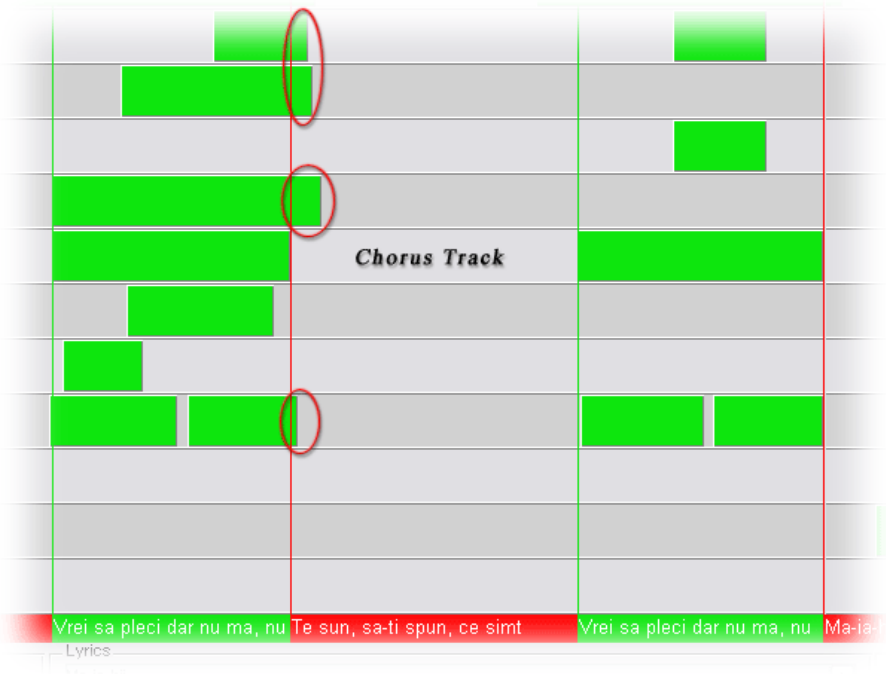


Figure 6-5: Chorus section ends too early.

In this song the *line segments* on the chorus track end too early while on other tracks many *line segments* end a bit later (see Figure 6-5). The *line segment* above the chorus track for example ends exactly at the end of the chorus. In future versions the existence of several *line segments* that are a bit longer than the one on the chorus track could perhaps be ex-

ploited for improvements. As it was already noticed in the evaluation chapter, the chorus sections tend to be too short. Therefore it could perhaps improve the aligning accuracy if such *line segments* that are shorter than the majority in this region would get extended.

5) Song: Kelly Clarkson - Behind These Hazel Eyes
Merge Mode: Normal
Description: Gap between two consecutive chorus sections.

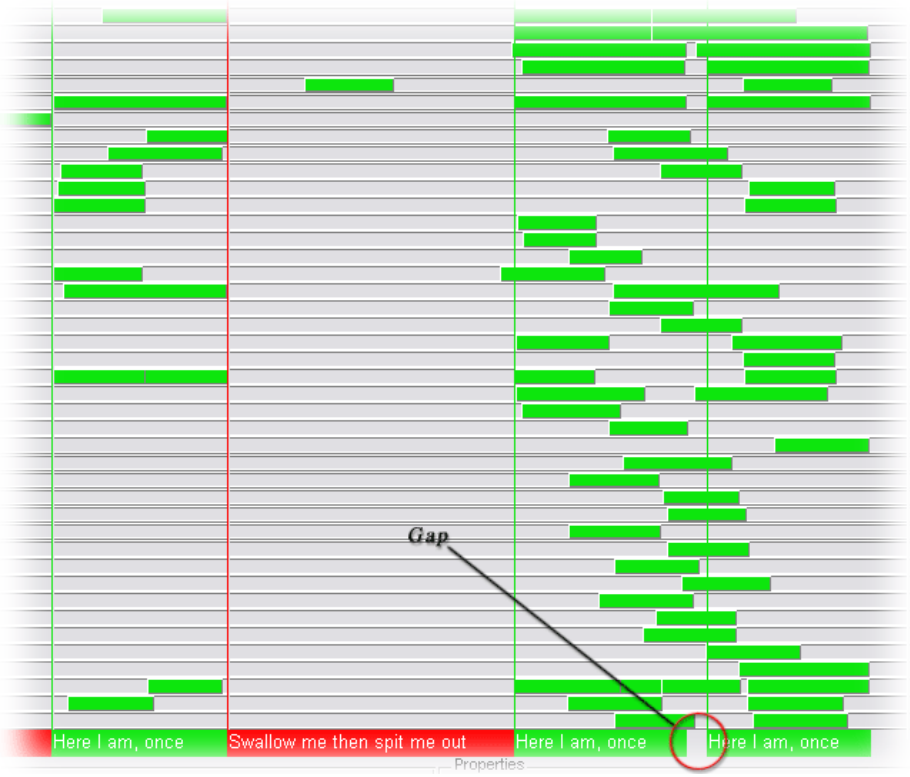


Figure 6-6: Gap between two consecutive chorus sections.

If a song contains two consecutive chorus sections, there often appears a small gap between the two *line segments* on the chorus track (see Figure 6-6). Observations show that this gap mostly corresponds to exactly the time which is missing at the end of all chorus sections in this song. As already mentioned, chorus sections are likely to be too short. Therefore this fact could be exploited in order to enhance the accuracy of the duration for all sections in the song.

6) Song: Masterboy - Mister Feeling
Merge Mode: CRM
Description: Song starts with a chorus section.

Figure 6-7: Song starting with chorus section.

This example shows that my approach also works even if the song directly starts with a chorus section. So this is one of the advantages of only distinguishing two kinds of sections. With the approach presented by [WAN04] this would not be possible because here the song has to start with an intro followed by a verse section.

7) Song: Nena - Irgendwie, Irgendwo, Irgendwann
Merge Mode: Normal
Description: Additional instrumental chorus section detected.

In this song an additional chorus section was found. The reason for this is that at the end the chorus is repeated once again but without vocals. However, since the accompanying instruments are the same, the algorithm detects it as an additional chorus section. Therefore of course all sections before this additional chorus section get assigned wrong lyrics.

Figure 6-8: Additional instrumental chorus section detected.

8) Song: Nickelback - This is how you remind me
Merge Mode: FRM (1)
Description: Duration of chorus section is 51 seconds.

The duration of this song's chorus sections is 51 seconds. In the algorithm for finding the chorus track a *line segment* which has a length greater than 40 seconds automatically gets zero points (see chapter 4.2.6.2). Therefore in this case the right chorus track is not selected.

Figure 6-9: Duration of chorus section is 51 seconds.

7 Conclusion

7.1 Results

An approach for aligning audio and lyrics on section level was presented. As experiments show (see chapter 6) the automatic alignment by only choosing the "Normal" merge mode is quite good. However, improvements could be made by automatic choice of the right merge mode. The performance of the feature extraction process could also get better by adding a beat detector to the system.

7.2 Limits of my Implementation

Since no beat and vocal detector as well as a singing phoneme duration database could be used, my approach is limited to section level alignment. One problem is also that the system can only rely on the chorus detector for placing the sections. A second module providing alignment information like for example a beat detector could improve the quality of the alignment algorithm.
Furthermore in the current implementation there is no automatic selection method for the best merge mode. However, as the evaluation results (see chapter 6) show, the quality of the alignment can be improved a lot by using the right merge mode.
Another limitation of my approach is that the sung vocals or their phonemes are not used for better alignment. By including for example *Hidden Markov Models (HMM)* [HUA90] the alignment could perhaps be verified by detecting certain words in a section.

7.3 Possible Improvements

In the current implementation the feature extraction for a song takes a long time. Of course this process could be optimised. For example as described in [WAN04] a vast improvement could be reached by using a beat detector. Since

then for every beat only one chroma vector needs to be extracted, an average speed-up of 98% could be reached.

As *Experiment 2* shows, an automatic detection of the most suitable merge mode could improve the aligning quality enormously. This could be done by comparing the number of found chorus sections in lyrics and audio. If there are more chorus sections in the lyrics, another merge mode could be tried to get better results. Because observations show (see Figure 6-1) that merge modes with no or few iterations are most frequent, the system should start with "Normal" mode and then go upwards to "CRM", "FRM (1)" and "FRM (2)". The first mode for which the number of chorus sections matches the amount of found refrains in the lyrics should finally be chosen for the alignment.

Adding a vocal detector to the system could also improve the alignment process because then instrumental sections could be detected. So for example the lyrics for the first verse section would be placed after the instrumental intro. Furthermore the current approach is sometimes fragile for songs with an instrumental part between two chorus sections. This problem could also be solved by using a vocal detector.

In the current approach the lyrics have to accurately reflect the words sung in the song. For example often a chorus is repeated several times in the end of a song. Then the corresponding lyrics also have to occur several times in the lyrics file. This could be improved in future versions of the software by simply copying the lyrics of the chorus section several times.

8 Bibliography

[COO65] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comput.* 19, 297–301, 1965.

[GOT03] Masataka Goto. SmartMusicKIOSK: Music Listening Station with Chorus-Search Function. In *Proceedings of the 16th Annual ACM Symposium on User Interface Software and Technology (UIST 2003)*, pp. 31-40, November 6-7, 2003, Vancouver, British Columbia, Canada.

[HUA90] X. D. Huang, Y. Ariki and M. A. Jack. *Hidden Markov Models for Speech Recognition*. Edinburgh University Press, Edinburgh, 1990.

[JAV06] Java, Sun Microsystems. http://java.sun.com, 07 2006.

[JUA84] B. H. Juang. On the Hidden Markov Model and Dynamic Time Warping for Speech Recognition - A unified Overview. *AT&T Technical J.*, 63, pp. 1213–1243, 1984.

[KNE05] Peter Knees, Markus Schedl, Gerhard Widmer. Multiple Lyrics Alignment: Automatic Retrieval of Song Lyrics. In *Proceedings of the 6th International Conference on Music Information Retrieval (ISMIR'05)*, pp. 564-569, September 11-15, 2005, London, UK.

[LCS06] Java implementation of the Longest Common Subsequence/Substring algorithm (LCS). http://www.bioalgorithms.info/downloads/code/LCS.java, 05 2006.

[LOS99] Alex Loscos, Pedro Cano, Jordi Bonada. Low-Delay Singing Voice Alignment to Text. In *Proceedings of the International Computer Music Conference 1999 (ICMC99)*, Beijing, China, 1999.

[MAT06] Matlab, MathWorks. http://www.mathworks.com, 07 2006.

[MIL95] B. P. Milner. *Speech Recognition in Adverse Environments (PhD. Thesis)*. University of East Anglia, England, 1995.

[RAB85] L. R. Rabiner, B. H. Juang, S. E. Levinsong and M. M. Sondhi. Recognition of Isolated Digits using Hidden Markov Models with Continuous Mixture Densities. *AT&T Technical Journal*, 64, pp. 1211-1235, 1985.

[RAB93] L. R. Rabiner and B. H. Juang. *Fundamentals of Speech Recognition*. Prentice Hall, 1993.

[STE96] Ken Steiglitz. *Digital Signal Processing Primer - With Applications to Digital Audio and Computer Music*. Addison Wesley, 1996.

[TTS06] Online Text-to-Speech. AT&T Labs – Research. http://public.research.att.com/~ttsweb/tts/demo.php, 05 2006.

[VAS00] Saeed V. Vaseghi. *Advanced Digital Signal Processing and Noise Reduction, Second Edition*. John Wiley & Sons, 2000.

[WAN04] Ye Wang, Min-Yen Kan, Tin Lay New, Arun Shenoy and JunYin. LyricAlly: Automatic Synchronization of Acoustic Musical Signals and Textual Lyrics. In *Proceedings of ACM Multimedia'04*, October 10-15, 2004, New York, NY, USA.

[WIK06] Wikipedia, the free encyclopedia. http://en.wikipedia.org, 06 2006

[WON05] Chi Hang Wong, Kin Hong Wong and Wai Man Szeto. Automatic Lyrics Alignment on Popular Music. In *Proceedings of the ISCA (The International Society for Computers and Their Applications) 20th International Conference*, pp. 385-390, March 16-18, 2005, New Orleans, Louisiana, USA.

[YAM06] Yamaha Vocaloid – New Singing Synthesis Technology. http://www.vocaloid.com, 05 2006.

Curriculum Vitae

Andreas Kothmeier

Personal

Date of Birth	December 9th, 1982
Place of Birth	Linz, Austria
Home Address	Seeweg 3/12, 4040 Linz
Parents	Robert Kothmeier and Brigitte Kothmeier
Nationality	Austrian

Education

1989-1993	Volksschule Steyregg
1993-2001	Europagymnasium BRG Auhof
June 2001	Matura passed with distinction
Autumn 2001	Enrolment at the Johannes Kepler University Linz, Subject Informatik (Computer Sciences)

Languages

English	Fluent in spoken and written
Spanish	Fluent in spoken and written

Employment History

Summer 2004	Employment during holidays at KEBA AG
Since October 2004	"Freier Mitarbeiter" at KEBA AG

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Plesching, August 2006