MUSIC INFORMATION RETRIEVAL
A QUERY-BY-HUMMING (QBH) SYSTEM
哼唱检索
SEGMENTATION OF THE SONGS
AND
APPROXIMATIVE MELODY MATCHING
BASED ON
THE DTW ALGORITHM

Jean-Louis DURRIEU
陈福新 2005400106
jean-louis.durrieu@m4tp.org

July 11, 2006

Abstract

The Information Technologies (IT) "explosion" of these days has led to new needs in data search abilities. Thanks to a lot of research on Natural Language Processing (NLP), for instance, we can imagine and almost achieve content based search and text classification, or exploring a database on behalf of the semantic level and not only with keywords matching as it used to be.

Music Information Retrieval (MIR) is a topic that is less popular than NLP, but its use might grow more and more in the future. New devices such as mp3 players, having a huge memory storage even if they are small, allow the user to keep his whole music collection in his pocket. However, the smaller the device the smaller the screen that goes with it. A visual exploration of this portable database might not be the most adapted solution. That is why new ways are to be found that allow the user to select the song he wants to listen to without a minimum effort.

We introduce in this paper the technique known as **"Query By Humming"**, or **QBH**. We implemented a system based on the *SoundCompass* system [6] and tested some features and their influence on the result. The songs in our database and the query sequence are divided into segments. An approximative pattern matching algorithm then finds the best match for each segment of the query. Up to now, the tests we did have shown us that the length and overlap chosen for the segments do not have a big influence in the computation time. The size of the database itself is more critical for the speed problem. At last, the representation matter such as choosing between global or relative pitch, or the problem of rhythm detection can in theory improve the success rate but the solutions we bring in this paper still need to be adjusted.

Contents

1 Introduction

The development of the Internet and the Information Technologies has led to new needs in a lot of domains. In fact, in fields such as Information Retrieval, Data Mining or web search, we have to adapt our way of thinking to the new means. Finding new ways of processing the huge amount of data available is actually necessary to obtain better and more efficient results. Thus we can see that new features are being developed such as the semantic web, or text information retrieval, which helps to classify documents, generate sum-ups or extract information for "Question-Answer" systems. These automatic Natural Language Processing (NLP) tasks, if accomplished by mere keyword matching or other superficial means, cannot lead to reasonably correct results. That is why, more than just using the words, these techniques have to explore the relations between words in order to be more accurate in their results.

Here, we are interested in another domain, not as popular as NLP but certainly as important (if not more!): Music Information Retrieval (MIR), or, more precisely, Music Recognition. One cannot be unaware of nowadays' explosion of the mp3-players. Even mobile phones are now equipped with features so that they can read music files. However, even if these devices are getting smaller and smaller, the way one have to explore their whole content is often not really convenient. Except from the ones with big screens, they often only allow one line to be displayed and the most common way of exploring is merely to check one by one the songs in the order they were memorized until the user finds the song he wants to listen to.

Our goal is to accomplish the task of **Query-By-Humming**, a system that tries to return the name of the song the user sings. Our database is composed of several MIDI files of Beatles songs. In order to compare the files in the database and the audio input query of the user, we need to transform it into a "symbolic representation", understandable by the computer.

Actually at first it seems quite mandatory that, to describe a piece of music, we need the **sequence of notes** that it is formed of. Thus we need a **Pitch Recognition Module** that would take the raw audio data as input and turn it into the wanted sequence of pitches. Of course, for the sake of efficiency, the final representation for the comparison between the database and the query can evolve. For example, we could try to compare not the sequences of pitches, but the sequences of intervals. However, for any of the other representations, we still need the sequence of pitches, thus the importance of this part. Another key-issue is the way how we represent the sequences. We will have to compare some audio data to songs stored in a symbolic representation: we have to find the **Common Representation** that will lead to the best results. At last, with these sequences, we can run a **Pattern Matching** algorithm that will return the song in the database that is the most similar to the query.

This paper is organized as follows. We first describe in detail what a Query-By-Humming system consists of, and give a review of some related works and previous systems. Then we discuss about the Pitch Detection algorithms and the way we implemented the ACF method in our system. Next come the Representation issues and our choices for this part. At last we describe the Song Matching strategy, the dynamic programming algorithm we used and the improvements that could be done. The following part gives some of our results with some discussions about what went right and what went wrong. In the end, you will find in the annexe some tools we had to research, such as the MIDI file format and the WAV file format, in order to achieve our program.

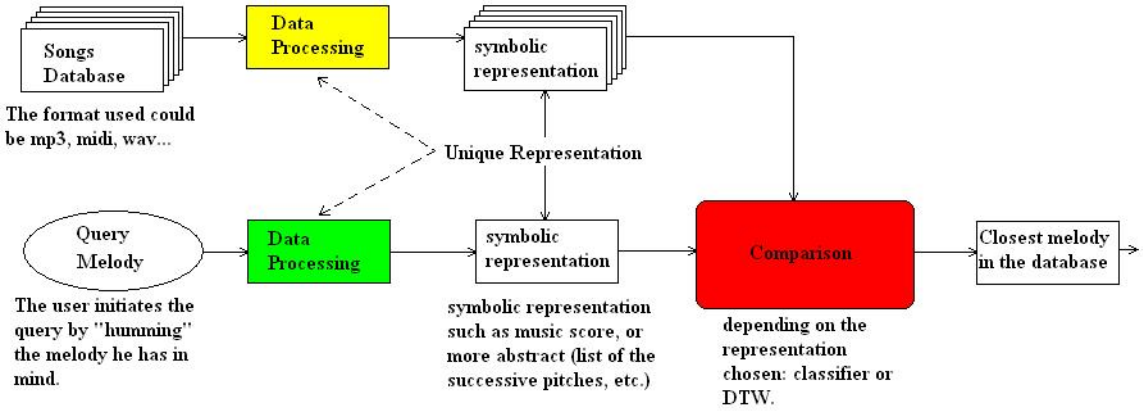

Figure 1: Organization of a generic QbH system.

2 Query-By-Humming, Principles and Related Works

Query-By-Humming (QbH) is a branch of the Music Information Retrieval study. The goal is simple enough: the system has to find the song the user intended to sing (either by singing or "humming", act of singing without opening one's mouth, singing with onomatopoeia).

2.1 Principle

The Query-By-Humming process has several steps:

1. **Building a Database (DB):** based on songs in any format, we need to have a symbolic representation of these songs and especially of the melodies. The QbH system uses a melody matching algorithm.
2. **Processing the Input Query:** the input query is usually in raw audio format (WAV file format, for instance). We need to find the symbolic representation that is the closest to the intended melody.
3. **Comparing the Query with the Melodies in the DB:** This section is the most critical in terms of time consuming. We basically have to compare the obtained representation of the query with every single entry in the DB.
4. **Returning the Result:** the result can either be the best match, but it is more reliable to return a whole list with the ten or twenty best matches.

In the following sections, we give a more detailed description of these different parts.

2.1.1 Building a Database

Since our task is to find the name of the song of the query, we need a Database which defines the domain of application of our system. For example, the system we tested for this study is based on

some Beatles songs (at most 150 of them). The more songs we have, the wider the system can be applied. However, for performance reasons, it is hard to use a big DB. We will see why later.

To build our DB, we need songs. We could for example take the recordings of the original songs and try to extract the features out of them and find a representation vector that characterize each song. However, for the task at hand, it seems more interesting to find a symbolic representation close to the musical notation. We focused our study on the melody representation and melody matching. Nevertheless one should know that there are other ways of addressing the QbH problem. One can imagine a classification that uses a feature vector extracted from the sound. However, this classification seemed too rough to be used here. Some examples of applications for this kind of problem can be found in [1] (in French). We can note for instance applications such as sound classification, instrument identification or style identification ("what kind of songs is it?").

We will see later what choices were made for several systems, as concerns the Database. The difficult part about building it is to rewrite the original songs into the wanted form. Some take music scores while others take MIDI files as original song material. Depending on the original form, the effort put in this task can potentially be very big. When the Database is meant to be large, we hope we can find an automatic solution to extract the melodies from the songs. We will come back on this task later in the section 4. The songs are stored in the Database as **sequences**.

2.1.2 Processing the Input Query

The input query is in raw audio data format. For example, it can be a recording on a computer in the well-known WAV format. The small devices such as mobile phones or mp3 players can record in such format as well.

Once the representation form has been decided for the Database, we need to use the same form so we can compare the query to the songs in the Database. One thing that seem mandatory in this part is to extract the series of pitches form the audio data. Since our choice is to represent at least in one way or the other the melody, we need, in the beginning, the series of notes that form the melody.

The rest of the processing is essentially about quantifying the result, in terms of frequency, of time and rhythms. This part is further developed in the section 3, where we explain our pitch detection algorithm and some of the "tricks" to quantize the obtained series. After this part, when referring to the transformation form the audio format to the symbolic representation, we will talk about the **query sequence**.

2.1.3 Comparing the Query with the Melodies in the DB

This part is the core of the system, because this is where we will find out which song fits the best the query sequence. It is also the part where the system will spend most of the computing time. Since the purpose of our study was not to do a system viable commercially speaking, we did not focus on the speed aspect of the program. However, since this can be a key-issue in the future of this technique, we tried several algorithm of melody matching. More specifically, we implemented the DTW algorithm, that allows us to compare two sequences that potentially do not have the same length, nor the same "timing", case which is quite frequent when considering songs and the queries. We explain the DTW algorithm in section 5.

Since this algorithm is known to be very slow, we need some enhancement to it. Still in section 5, we search for better performances thanks to algorithms adapted to the DTW. The overall speed

of the matching part is also dependent on the number of segments we have, that is to say dependent to the number of songs in the database.

2.1.4 Results

QbH systems, up to now, do not have outstanding results, such that it is necessary to return a list of the ten first best results. This is done by simply keeping an updated list of results during the comparisons.

We would also like to highlight an interesting result provided in [7]. It is a test where three singers where compared to three QbH systems. What we are interested in here are the results obtained by the singers. They were asked to sing and then had to give the title of the songs they (themselves and the two other ones) had sung. They had a limited time to answer.

Even if we can consider that such a small survey does probably not very trustfully reflect the situation, we can however note that the singers did not have very high results. We would have expected them to be higher, especially since they seem to be almost professional. The average recognition rate observed was 66%, which can be considered as quite low! The singers were only allowed one answer, so it makes it even harder. The test songs were some beatles songs. What we could say as explanation could be that the songs, all being authored by the same group, must be quite similar, so that confusion is possible either on the titles or even on the melodies themselves (especially if you do not have the lyrics with it).

Does that mean that a QbH system will not be able to over-perform this 66% threshold? Does that mean that our systems are doomed to fail? Well, this probably means that giving the right answer as first choice is very difficult even for a human, not to say for a machine! That is why giving a list of ten possible answers instead of just one might seem more realistic. What is more, if we consider the above study again, we could say that human is subject to confusion, introducing some more errors in the results. We hope a program would not be influenced by loss of memory for example. We can only hope a program to be faster and more accurate than a human. However, as any recognition problem, the machine is still far from what a human is capable of. This paper still tries to find ways to perform even better recognition, find the guidelines at least for further studies to avoid misinterpretations.

We present some of the related works we based our system on: “SoundCompass”, “CubyHum” or “RePReD”, and so on. We especially took a lot of the elements of the first system, for it seemed the most state-of-art one.

2.2 SoundCompass

Developed by a Japanese laboratory, the **“SoundCompass”** system is meant to run in Karaoke Houses, where the customers can ask for their songs by singing them directly.

Since we are using most of their ideas in this project, we will stay general about it in this introduction. In Kosugi et al. [6], the authors explain that the database they used contained $21,804$ songs. The original songs are in the MIDI file format. the songs are segmented and cut into segments according to a given algorithm (given in section 4). To sum up, the algorithm takes every melody track and according to a time unit (computed thanks to the rhythms in the track), they define the length and the overlap of these segments, the “subsequences” - or CSTPV (for Constant Slide-length Tone-In Phonetic Value). After that first segmentation, they cut again the track, but with a variable overlap, taking in account “distinguishable” notes giving the “subsubsequences” - or VSTPV (for Variable Slide-length TPV).

The queries are recorded with the following properties: sampling rate of 11kHz, bit resolution of 8 bits, monoral. The pitch detection algorithm is done on 512-points data, every 256 points. This corresponds to analyzing the data almost every 0.025 seconds, analyzing 0.05 seconds of data each time. They also track the timing of the utterances but they did not explain which pitch detection algorithm nor which utterance detection algorithm they used.

As matching strategy, they chose as similarity distance between one song of the database and the query the minimal distance amongst the query sequences and the sub/subsubsequences of the song in store. Let m be the number of query sequences, h_i for $i \in [1, m]$ these sequences and n the number of sub/subsubsequences for the song to be tested, s_j, for $j \in [1, n]$ these sub/subsubsequences. We obtain that the measure D between h the humming and s the song is:

$$D(h, s) = \min_{i \in [1,m], j \in [1,n]} d(h_i, s_j)$$

where d is the similarity distance between two sequences (not explained in [6]). The authors also included as evaluation test the distribution of tone difference, and they separated the tests for the CSTPV and the VSTPV, only taking the best of the two results.

The representation they adopted is, as we said, a set of segments, called "sub/subsubsequences". The notes are represented by their MIDI code and their duration is mapped thanks to the time unit that was computed. For example, the score on figure 17 (see later in section 4) can be described as `(0, 2, 4, 5, 7, 7, 4, 4, 0, 0)` or, depending on the time unit chosen:
`(0, 0, 2, 2, 4, 4, 5, 5, 7, 7, 7, 7, 4, 4, 4, 4, 0, 0, 0, 0)`. On that example, the first vector is based on a time unit of one quarter note, while the second vector is based on a time unit of one eighth note.

As for the performance of the system, the paper reports, for the best version of their algorithm, a 60% of right answers, which goes up to around 75% of right answer in the first 25 given answers. The authors insist on the fact that the time normalization (that is to say using a time unit to normalize the representation) is critical for the algorithm to work.

2.3 CubyHum

In [8], Stephen Pauwns details how the **CubyHum** system works, from the pitch detection algorithm to the pattern matching, through the musical representation of the hummed song and of the songs in the database.

As for the melody transcription, he used the "sub-harmonic summation" method (SHS) to detect the pitches in the hummed audio. At the same time, an event detector essentially based on "short-time energy" cuts the sequence into notes (thanks to two threshold, a "note onset" threshold and a "note offset" one).

The representation the author chose for each melody is the sequence of tone differences. The range of differences allowed goes from -6 to $+6$ in terms of semi-tones. This also means that this representation does not allow to show intervals greater than fifths. This can be a judicious choice, since most of popular songs do rarely use intervals that are hard to sing (such as fifths and greater). The system quantizes the extracted intervals and time durations.

The matching algorithm is also a Dynamic Programming algorithm, which the paper reports that, for the 510 songs database, one query needs a few seconds to get the answers. The author also proposes a means to improve the speed by filtering (pruning) the elements before running the approximative pattern matching.

Unfortunately, the given paper did not give much information about the performances of the algorithm. The only comparison we have here are the results found in [3], by Roger B. Dannenberg et al. The CubyHum algorithm for representing and matching melodies is compared to a very simple algorithm based on relative pitch representation. We come back later on these results, they were really low, even compared to this simple algorithm. We can however wonder if the implementation of the Dannenberg team was the most optimal for the CubyHum system.

2.4 RePReD

In [5], Kline et al., the authors did some experiments to measure how far the errors in the queries would induce errors in the results of query-by-humming systems. Their solution, called **RePReD**, "Relative Pitch, Relative Duration", is based on a representation that takes into accounts a relative pitch representation as well as a relative duration representation. This is, in a certain extend, similar to what we saw for the SoundCompass system.

They used three versions of their algorithms to run their tests. The first version was based only on the pitch intervals. It gave an average of 50% of correct answers, and 61% of the correct answers were present in the first ten best result (the correct answer was rank between the first and the tenth place). The database was a collection of 3600 folk songs (the DIgital Tradition Folksong Database). The second system introduces weighting the durations as an improvement. Actually, the authors report a decrease in the success rate, probably due to erroneous notes from the database or the user's query.

At last, the third model, based on relative pitch and relative durations representation gave better results. The similarity distance is computed out of the pitch and duration information, and not only with the pitch information. The correct target was given the first rank for 68% of the queries and ranked in the first ten for 78%.

As concerns the speed performance, the system seems to be able to return the result in less than one second for the database of 3600 melodies.

2.5 and the others...

Some other interesting studies have been held. We could talk more about the QbH system provided by Yunyue Zhu et al. in [13], where the authors describe a technique to improve the search performance, or the system imagined by Roger B. Dannenberg et al. in [3], for the **MUSART** project, which uses n-grams to represent the melodies.

The first study, as exposed in [14], first uses a simple representation form, based on some symbols (letters) to describe the melody. For example a 'U' would mean that the melody goes "Up" while 'D' and 'S' would respectively mean that the melody goes "Down" and that the melody does not change (two successive notes are therefore the "Same"). Then they compare this representation against a time series representation. In Zhu [13], the authors also introduce a speed enhancement for the DTW algorithm thanks to a Lower Bounding technique based on melodic contour approximations. Their results show that the time series concept beats the "traditional" contour string approach. What is more, their DTW lower bounding technique seems to give optimistic results. However, the tests for the query by humming system are very few, we can just know there that they used 50 beatles songs, manually extracted some 1000 melodies to construct their database. Over 20 queries, their time series approach ranked the right answer in first rank for 16 of them, while the contour string approach classified 14 of them over the tenth rank. This would tend to validate the time

series approach.

The **MUSART** project compares some QbH techniques, implementing the CuByHum system, simple systems with basic techniques of representing the melodies such as absolute pitch, relative pitch, inter-onset intervals and so on, and at last their system, based on n-grams. Their first result is that even simple algorithm (that is to say not enhanced ones) can beat the CuByHum approach. The authors defined the *mean reciprocal rank* to evaluate the different algorithms: if r_i is the rank of the correct answer for the i^{th} query, then, for n queries, the "MRR" is given by the following formula:

$$MRR = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{r_i}$$

According to this study, the CubyHum, tested on two databases, one with 2844 themes from the beatles songs and the other one with 8926 themes form popular and traditional songs, obtained only an MRR of 0.02, while the simple algorithm they programmed, "NOTE-SIMPLE", based on dynamic programming matching, with relative pitches and "log Inter-Onset-Interval" ratios (to represent the durations of the notes) obtained better results, respectively 0.12 and 0.23. At last, the authors tested an n-grams based algorithm, but the results (inferior to the "NOTE-SIMPLE" algorithm) seem to show that this method is not suited for this application, probably because of the errors in the queries.

At last, we would like to draw the attention to the **"MIR systems"** website, http://mirsystems.info/index.php, by Rainer Typke, where one can find almost every existing QbH system (and more), with a little description of the way they work.

3 Sound Processing: From Raw Audio Format to a Sequence of Notes

The transformation from the audio recording to the symbolic representation is a key part of the system. Getting the pitch time series corresponding to an audio song is far from being an easy task.

The specification for the pitch recognition are as follows: given a query melody sequence (in the form of a `WAV` file, for instance), we want as output the sequence of notes that are more likely to correspond to the initial sequence. This means that apart from the pitch recognition task in itself, we still need some quantization of the signal: in time, in order to restrain the values of the notes to quarter or eighth notes, since the common singer will supposedly not be able to reproduce very accurately faster rhythms; and in frequency, to stay within a given set of pitches (for example, the classical music notes).

we first introduce the techniques that we considered, namely the cepstral method and the autocorrelation method. We used *MATLAB* to compute those algorithms and empirically compared the results. Next, we explain how we intend to quantize the sequence of mere pitches obtained, essentially using heuristics, in order to obtain a code that we could use to make a MIDI file, for example. Actually, this step is not mandatory in the optic of our study, and perhaps will lead to more errors in the final implementation. However, if we are to try other representations such as inter-note pitch difference, this quantization is necessary. And of course, if we wanted to make a "WAV-to-MIDI" system, it would be compulsory.

3.1 Pitch Recognition

The first task to address is the pitch recognition one. We tried different configurations and algorithms. The task at hand can be summed up as follows: given a melodic sequence, how do we get the sequence of fundamental frequencies? We limit our study to monophonic melodies. What is more, we do not exactly need a high resolution for the frequency. In fact, since we intend to compare it to sequences of pre-defined pitches, with discrete values (non-continuous) of fundamental frequencies, we just need some rough estimation of the pitch (at least in the first step of the algorithm).

In the following parts, we first introduce the pitch recognition in the case of one note, and then we explain how to apply it to a sequence of notes.

3.1.1 Pitch evaluation for one note

There are several methods that allow to find the fundamental frequency of a sound. We can classify them in temporal and spectral method, for the former uses the information contained in the signal in the time scale and the latter uses the information out of the spectral content (after a Fourier Transform).

The most popular algorithms include the use of the Autocorrelation Function (ACF methods), the Harmonic Product Spectrum (HPS), the maximum likelihood or the cepstral method. We chose to experiment the ACF method and the cepstral method for those two methods seemed to appear more often in the papers concerning ptich recognition. We base our theory on the **ENST Paris** (Ecole Nationale Superieure des Telecommunications Paris) lessons: **PAMU** ("parole, acoustique, musique", "speech, acoustics, music"), **MSA** ("Musique et signaux audiofrequence", "music and audiofrequency signal"), *"Pitch Detection"* chapter, 2003-2004 (see [10]).

1. The need for a pitch detection algorithm

For people who are used to signal processing and musical notions such as fundamental frequencies and harmonics, a first idea to detect the pitch (ie the fundamental frequency) of a note is to use the Fourier transform of the signal, the maximum of which should be the main frequency present in the signal and, hopefully, the wanted fundamental frequency. We can see a spectrogram of the signal for the beatles' song "Let It Be" (only the melody, sung for the study) on figure 2.

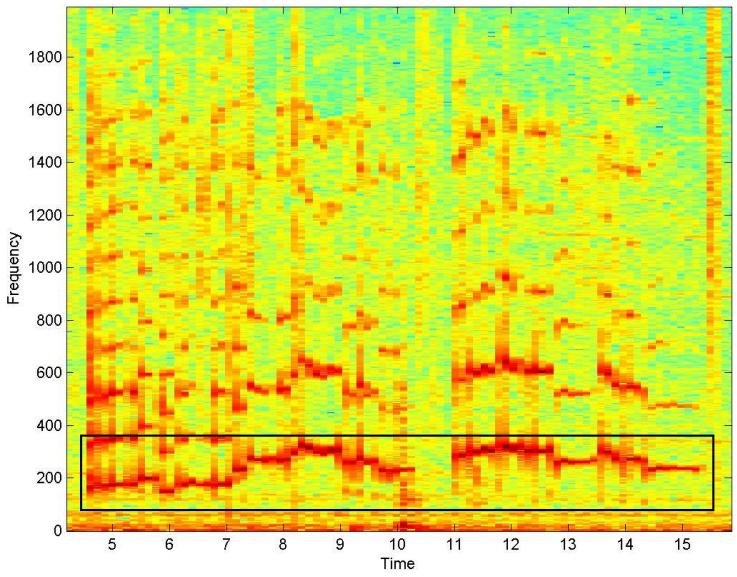

Figure 2: Spectrogram of "Let It Be", first verse. The sampling rate is $Fs = 16000$ and the FFT were computed on $nfft = 4096$ points. The section in the black frame shows the fundamental frequencies of the signal (thus showing the "melody" of the song).

The spectrogram represents the Fourier transform of the signal and its variation through time, also called "STFT", Short Time Fourier Transform. We can identify on the figure what we can consider as the sequence of fundamental frequencies as being the first maxima in the low frequencies. However, one can not prove that if, for a given section, you choose the maximum of the Fourier Transform of the signal, then you get the wanted pitch. It is actually highly possible that the result is wrong. The second harmonic in the sound can be as strong as the fundamental, if not stronger. Such case would lead to an error of one octave. What is more, in certain situations, the fundamental can also simply not exist, and only some algorithms (such as those seen below) can retrieve this kind of signal.

We notice however that for a human recording, a simple algorithm looking for the maximum in a restrained window, from 80Hz to 1200Hz, for example, can be quite reliable. But since this method is not robust enough, especially as concerns noisy recordings, we preferred to use more specific algorithms.

2. Autocorrelation Function: ACF method

   The ACF method is a temporal based method. It consists, as many of these methods, in comparing the signal to a shifted version of itself. Shifting the signal and comparing it to the original permits to identify the values of the time shift that gives the highest similarity score.

   - *Signal model*

     We assume the signal is the sum of a centered signal, which period is exactly $P \in \mathbb{N}$, with a white noise:

     $$x[n] = \sum_{k=1}^{H} 2A_k cos(2\pi k f_0 n + \varphi_k) + w[n]$$

     where

     - $f_0 = 1/P$ is the reduced fundamental frequency (in "number of samples per second")
     - H is the number of harmonics of the signal ($H \leq \frac{P-1}{2}$ in order to comply with the Nyquist-Shannon rule)
     - A_k such that $A_k \in \mathbb{R}^{+*}$
     - φ_k are independent random variables, with uniform distribution in $[0, 2\pi[$
     - w is a white noise, centered, of variance σ^2 and independent from the φ_k

     We can demonstrate that:

     - $E[x[n]] = 0$ since w is centered and $E[cos(\varphi_k)] = E[sin(\varphi_k)] = 0$.
     - $E[x[n]x[n+m]] = \sum_{k=1}^{H} 2A_k^2 cos(2\pi k f_0 m) + \sigma^2\delta[m]$ is independent of n

     Since the definition of the autocovariance function r_x is:

     $$r_x[m] = E[x[n]x[n+m]] \quad (1)$$

     then:

     $$r_x[m] = \sum_{k=1}^{H} 2A_k^2 cos(2\pi k f_0 m) + \sigma^2\delta[m] \quad (2)$$

   - *Estimation of the autocovariance function*

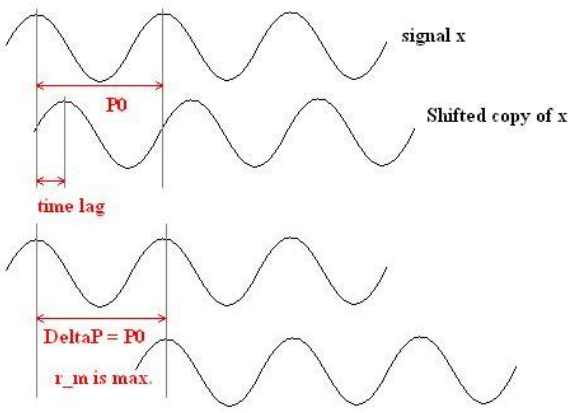

Figure 3: Principle to determine the fundamental period

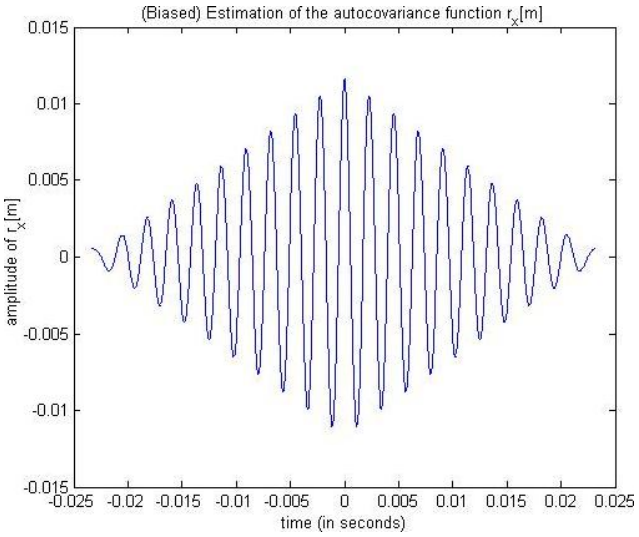

Figure 4: autocovariance function $r_x[m]$

The reason why we need the autocovariance function is, as explained before, that this function reflects the level of similarity between the signal and a shifted version of itself, with regards to the time lag (as shown figure 3). The more this function is high, the more the two versions are similar. The figure 4 shows the estimation of the autocovariance of a cosine function of period 0.0023s, that is to say a frequence of 440 Hz.

Each peak stands where the time lag is a multiple of the period P. This is normal, since a periodic signal of period P is also periodic with period $n * P$, $n \in \mathbb{N}$.

As seen in the equation 1, above, if we want to compute the exact autocovariance function, we would compute an infinite sum of terms. Practically, it is impossible, so we need to find an estimator for this function.

Let the observed signal $x[n]$, $n \in [0, N-1]$. We can define the estimator $\hat{r}_x[m]$ for $m \in [-N+1, N-1]$:

$$\begin{aligned} \hat{r}_x[m] &= \frac{1}{N} \sum_{n=0}^{N-1-m} x(n)x(n+m) \quad \text{if } m \geq 0 \\ &= \hat{r}_x[-m] \quad \text{otherwise} \end{aligned} \tag{3}$$

This estimator is biased. However, for N big enough, this bias can be not so significant and we will use this estimator in the rest of the paper. The autocorrelation function is the normalization of the autocovariance function.

We are looking for the maxima of this function, especially the first maximum to appear, since it will correspond to the smallest possible period of the signal, thus corresponding to the fundamental frequency. However, one should notice that the maximum of the estimator is actually the value at $m = 0$. It is easy to demonstrate and to understand: the similarity is the highest when we compare the signal to itself (with no time lag).

The figure 3 shows how the algorithm works. For those reason, when we look for the maximum, we have to define a maximum and a minimum allowed period.

For our purpose, application to voice, we can restraint the interval of possible frequencies to $[80, 1200]$, which correspond to periods between 0.000833s and 0.0125s.

- *Application*

  As shown on figure 4, we can analyze the autocovariance function and find the first maximum (excluding the one at $P = 0$).

  For convenience, we are not doing a straight-forward computing of the estimator, but instead compute the periodogram $\hat{R}_x(e^{j2\pi f}) = \sum_{m=-(N-1)}^{N-1} \hat{r}_x[m]e^{-j2\pi fm}$. We can demonstrate that $\hat{R}_x(e^{j2\pi f}) = \frac{1}{N}|\sum_{n=0}^{N-1} x[n]e^{j2\pi fn}|^2 = \frac{1}{N}|X(e^{j2\pi f})|^2$, where X is the Fourier transform of x. A convenient way of computing the autocovariance (especially when using Matlab) is then:

  (a) Compute the Fourier transform X of the signal x, using for example the Matlab function `fft`

  (b) Periodogram $R = \frac{1}{N}X. * conj(X)$

  (c) Autocovariance Coefficient with the inverse Fourier transform of R: `r = ifft(R)`

  (d) Determine the maximum of r in the interval $[P_{min}, P_{Max}]$

  Some remarks about the application of the algorithm: using Matlab, we can apply the `fft` and `ifft` functions. Since we are using discrete values, digital data, it is interesting to think a while about **the units we are using**.

  Let us say that we are computing the Fourier transform on `Nfft` points. Then, assuming the sampling rate is `Fs`, each unit in the spectrum corresponds to $\frac{Fs}{Nfft}$.

  Let us assume that we are doing the inverse transform in the algorithm with the same `Nfft`. Each sample then corresponds to $\frac{1}{Fs}$ second. This comes from the fact that the Fourier transform has a "sampling rate" of $\frac{Nfft}{Fs}$, which means that in the spectrum, for each 1 Hertz, we have $\frac{Nfft}{Fs}$ values. Therefore, its inverse transform is also periodic with period $\frac{Nfft}{Fs}$. The inverse Fourier transform is computed for `Nfft` points, thus the result: $\frac{\frac{Nfft}{Fs}}{Nfft} = \frac{1}{Fs}$

  We can at last use the following formula so as to convert the period in samples into a period in seconds:

$$P_{\text{in seconds}} = \frac{P_{\text{in samples}}}{Fs}$$

3. Cepstral Method

- *Signal Model: a source-filter model*

  We assume that the signal is determinist and periodic with period P. Its spectrum X is then constituted of harmonics of the fundamental frequency $f_0 = \frac{1}{P}$.

  Let the harmonics envelop H be continuous, X can be considered as the product of H by a Dirac train S with frequency f_0: $X = H.S$. This means that in time scale, the signal is the convolution of the filter h and the source s, which is also a Dirac train, where two impulses are spaced by a period P: $x = h \otimes s$.

  Physically speaking, this corresponds to a speech production model, where s is the signal emitted by the vocal cords and h represents the filter standing for the vocal tract.

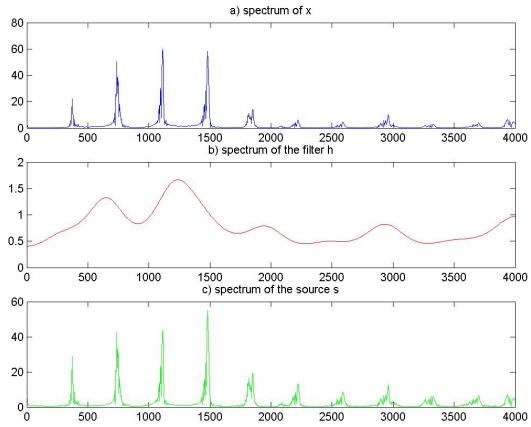

Figure 5: spectral decomposition for a 'a' sound

The figures 5 show the decomposition of the spectrum of x for the pronounced vowel `a`. The spectrum of the filter (`b`)) shows clearly the formants that are used for each sound, while the spectrum of the source (`c`)) shows the fundamental frequency and its harmonics.

- *Cepstrum of the signal, definition and properties*
  The cepstral coefficient of a signal is defined as follows:

$$C_x[n] = \int_{f=-\frac{1}{2}}^{\frac{1}{2}} \ln\left(|X(f)|\right) e^{+j2\pi nf} df$$

The cepstrum of a signal is actually the inverse Fourier transform of the logarithm of the amplitude of its spectrum. Therefore, one of the first properties of the cepstrum is that the cepstrum of the product of two spectra is the sum of the cepstra of these spectra. The proof is straight-forward, since we are dealing with ln (logarithmic) functions.

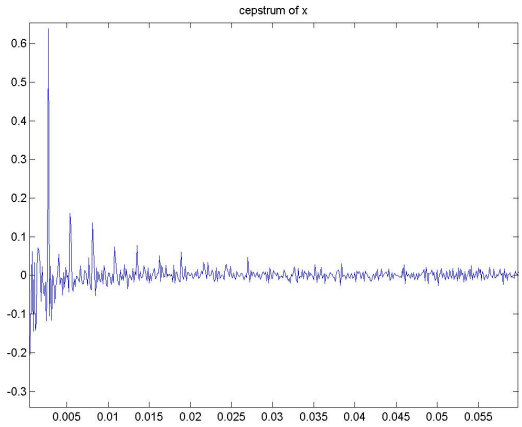

Figure 6: cepstrum of a 'a' sound

What is more, we can prove that for a voiced signal, the cepstra of the filter H and the source S have two different supports.
The first one is essentially localized near the zero value, and one can assume that above a value of P_{min} seconds, where P_min is the minimum audible period of a sound the human ear can hear, the cepstrum of the filter is almost zero.
If we assume that the source is an impulse train, which impulses are spaced by $f_0 = \frac{1}{P_0}$, then the cepstrum of this source is an impulse train as well, whose impulses are spaced by a P_0 space.
The figure 6 allows to see quite clearly the different support and the two distinct cepstra.

- *Application*

  To compute the several Fourier transform, we use once again Matlab and the `fft` and `ifft` functions. To find the fundamental frequency, the objective is to find the maximum of the cepstrum in a certain window $[P_{min}, P_{Max}]$, assuming that above P_{min}, the cepstrum of the filter H is almost null. As concerns the implementation, we have the same calculus of the units as for the above algorithm (ACF method), since we have also one `fft` and one `ifft` to do. The algorithm is sum up below:

  (a) $X = fft(x)$;
  (b) $C_x = real(ifft(log(X)))$;
  (c) Take the maximum of C_x in $[P_{min}, P_{Max}]$.

3.1.2 pitch evaluation for a sequence of notes

The evaluation of the pitches for a sequence of notes is done by evaluating the pitch for overlapping short-time sequences. The user defines a window length L_W for the analysis window and the overlap $N_{overlap}$. This process is actually similar to the process we can see in a phase vocoder.

1. Parameters issue

The **window length** has to do with the duration one note is supposed to last. For the preceding algorithms to work properly, the length should be as wide as possible, the more the algorithm has samples, the more accurate the result, since the estimation of the autocorrelation coefficient is strongly affected by the length of the original signal.

However, in practice, a singer or player will either play different notes or add ornaments to the original notes. This fact will lead to miscalculations in the pitch recognition algorithm. The smaller the window, the more the musician can play notes in a given time. That is why, it is necessary to make a trade-off between a wide window, allowing better precision of the algorithm and a narrow one, which allows the musician to play more notes. The figures 7 and 8 show how the trade-off has to be done, the former being very precise in time but quite fuzzy in frequency while the latter has a high resolution in frequency with a bad time resolution.

We also assume that, within a window, the pitch does not change, so that the sound in one window is "pure". This is another (good) reason why the length of the analysis windows should not be too large.

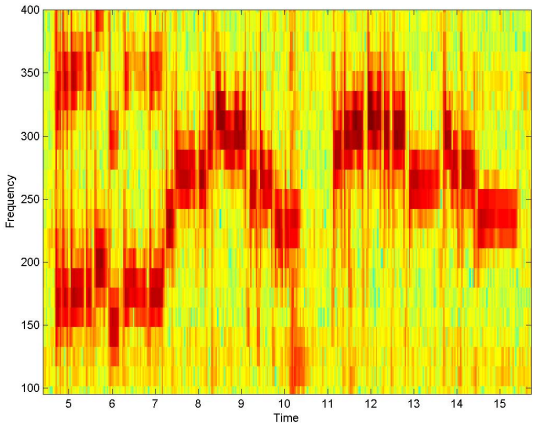

Figure 7: Spectrogram of "Let It Be", FFT on 1024 points.

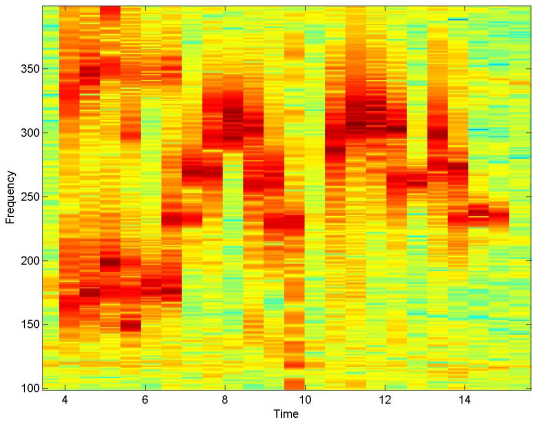

Figure 8: Same song, but with a 16384 points FFT.

Another issue is the **overlap**. In a "wav to midi" application, we have to define the resolution of the allowed rhythms for the transcription. For example, if you consider that the tempo is 120 bpm (beats per minute), and that we want as minimal unit of rhythm eighth notes for a time signature of $\frac{4}{4}$, then we should allow an overlap of $\frac{60}{120} * \frac{1}{2} = 0.25$s where $\frac{60}{120}$ represents the duration of one quarter note and the number of eighth notes in one quarter note being 2.

We can assume that in most popular songs, the minimal rhythm value is the sixteenth note. This corresponds to an overlap of $\frac{60}{120} * \frac{1}{4} = 0.125$s. However, since we cannot be sure that our segmentation will fall exactly where the singer places its rhythm changes (especially if it is not at a very precise tempo), we chose an even smaller overlap, so that we can adapt afterwards. We will see in the Time Quantization section how we can generate the final sequence of notes with the wanted duration, in accordance with the actual performance.

The formula we used above in order to find the fitted overlap is:

$$T_{\text{Overlap}} = \frac{60}{\text{Tempo}} * \frac{1}{n}$$

where n represents the number of time the wanted rhythm appears in one time unit. n thus depends on the time signature. Let us give some examples. If the time signature is $\frac{4}{4}$, then the time unit is the quarter note. If the wanted minimal rhythm is a sixteenth note, since we have 4 sixteenth notes in one quarter note, we have $n = 4$. If the time signature is $\frac{6}{8}$, then the time unit is (usually) a dotted quarter note, which means that, in the same case, n would be 6.

Because a lot of popular songs are actually based on the former time signature, we will more likely use the value decided above, ie 0.125s as overlap for the analysis window.

We have to point out that, regarding the task at hand, we do not actually need a very powerful pitch detection algorithm. If we wanted to compute a tune synchronizer, which has to give a

really precise estimation of the pitch, then we should talk more about the resolution of our pitch detection algorithm. There are techniques that allow to increase the pitch resolution, but this also means losses in computing time. Since we just need a rough estimation of the pitch sequence, we can limit our study to what was expressed before.

2. The final output form

   We need an output that we could use later in our program. We wanted to write a program that could handle both the query transcription into a time series and the melody matching. But the C/C++ program we wrote for the pitch detection was not as accurate as the one in MATLAB. The results were obviously not as close to reality as the MATLAB results. We chose to use MATLAB to compute the time series, until we found a solution to improve the C/C++ program results.

   To be able to use the time series, we chose to write the output into a formatted file, which we could use both to reproduce the time series and write a "feedback" MIDI file. Thus we had to express, for each note, its pitch and its duration. That is why the output file for our MATLAB program will look like what follows:

   ```
   f1          t1
   f2          t2
   f3          t3
   ...         ...
   ```

   `fi` and `ti`, for $i = 1, 2, \ldots$, respectively are the frequency (in Hertz or in MIDI code, we keep both) and the duration of the note (in seconds).

3. Application to a Chinese Pop Song

   The figures 9, 10, 15 and 16 show the steps that lead to the pitch sequence that we are interested in. The first step dealt in this section is the frequency tracking, shown on figure 9.

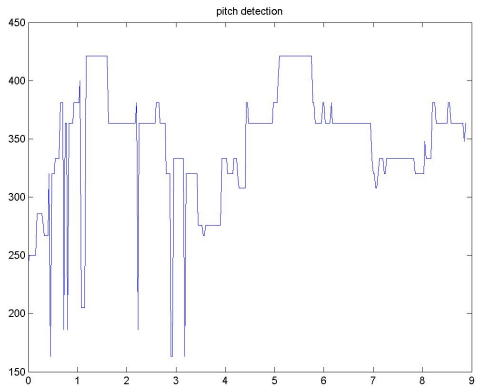

Figure 9: Step 1. Pitch sequence for the Chinese song.

Since there was no other processing apart from the pitch tracking, this figure shows a rather complicated curve, a melody that would actually be hard to reproduce. The reason why it is not perfect are multiple. For example the singer could, from one "analysis window" to another, have shifted the tone, even if it was meant to be the same note, or the recording was too bad at moments so that the Signal/Noise Ratio was not strong enough. Many reasons we have to deal with. In the next sections, we try to find the intended melody out of this chaotic pitch series, especially thanks to heuristics based on common sense.

3.2 Pitch Quantization

We want to compare the recorded songs with our database, which was made out of MIDI files. One step that has to be clarified is the pitch quantization of the recording, from the output of the preceding algorithms, expressed in Hertz, to a midi scale, expressed in integers from 0 to 127, thus covering a range from C0 ("Do 0") to G10 ("Sol 10").

We first review some basics about occidental music notes, then we introduce the midi representation for the notes and at last we explain an attempt to stay as close of the original performance as possible. Even if the performer makes mistakes, especially when he sings slightly "out of key", our program should compensate and find the note that is most likely to be the intended one.

3.2.1 Some Music basics on "classical" notes

In music, the "classical" notes are related thanks to a $\log_2$ formula, because the sensation of tones is not linear but (almost) log2-linear. Concretely, we know that a leap of one octave corresponds to the multiplication of the first frequency by two. Therefore, if we assume the 'A3' note frequency is 440 (nowadays usual standard), then the octave 'A4' is 880. The formula for f_2, octave for f_1 is:

$$f_2 \quad = \quad 2 * f_1 \tag{4}$$

In one octave, in occidental music, we have 12 tones, which we can consider equally distributed. Actually, this is not completely true. Historically, the way of tempering a keyboard (piano or clavichord) works as follows: you first give the reference, say 'A3', at f_0, the first octave, 'A4' at $2f_0$ and then the fifth note, 'E4' at $3f_0$. Then to fix 'E3', you use 'E4' as a reference, that is to say you fix it at $\frac{3}{2} * f_0$. If you go on like that, when you come back to 'A3', thanks to the '×C' note ("double ♯ 'C' "), you have a fraction $\frac{3^{12}}{2^{19}} f_0$, which is almost f_0, but not exactly. Actually, this is because this note is not exactly what we can call 'A3', but more precisely, it is a '×G 3' . Thus we can explain the difference between them. However, since in occidental music, we usually do not make this difference when performing, we make the approximation that each note is equally distributed in the interval of one octave.

the formula for two notes separated by one half tone is:

$$f_2 \quad = \quad 2^{\frac{1}{12}} * f_1 \tag{5}$$

The reason why we said it was a "log2-linear" relation is that, taking the $\log_2$ of equation 5 gives:

$$\log_2 f_2 \quad = \quad \frac{1}{12} + \log_2 f_1 \tag{6}$$

Which leads that, given any notes f_1 and f_2, we can find an integer $i \in \mathbb{Z}$ such that:

$$\log_2 \frac{f_2}{f_1} \quad = \quad \frac{i}{12} \tag{7}$$

Note that i can be negative.

We have introduced in the above paragraphs a way of obtaining discrete music notes. One should bear in mind that these are conventions. When someone is singing, one can not guarantee that the pitches the singer uses are exactly the ones defined before. If it is a professional singer, then you can assume that, given the reference, a majority of the tunes have the above relation. But if we take a non-professional, there can be several errors. The singer can sing "out of key", which means that his reference is moving , and that the relations between two notes are not constant. For professional singers, we can say that they are using an absolute reference to sing notes, whereas non-professional ones use a relative reference. If they lose the reference somehow during the performance, they will most probably use the last notes they sang as references. We exploit this fact to adapt our algorithm, explained a bit later below.

3.2.2 Midi Quantization

Under Matlab, we used the Midi Toolbox, especially the function `hz2midi`, which according to the specifications converts a note from its value in Hertz into the corresponding Midi number. The formula is:

$$F_{\text{midi}} = 69 + 12 * \log_2\left(\frac{F_{\text{Hz}}}{440}\right) \tag{8}$$

In midi encoding, 69 represents the note A3, "la 3". The other notes are scaled after this reference. We can see in the equation 8 some of the elements for the preceding section, that in one octave, we have 12 notes, distributed thanks to the $\log_2$ formula. It can be used to give the user a feedback of what the program recorded and understood from his performance.

Using this equation we can also get the frequencies corresponding to the midi number by:

$$F_{\text{Hz}} = 2^{\frac{F_{\text{midi}}-69}{12}} * 440 \tag{9}$$

The less we use quantizationS on the recorded sample, the closer we can get to the intended song, limiting as much as possible the errors due to the computations. Since we can compute exactly the frequencies from the MIDI files, it would be more interesting to transform the midi numbers into frequencies and then compare the obtained sequence with the recorded one, in a higher dimension space (since the frequencies are real numbers).

3.2.3 Absolute (or Global) and Relative Pitch Quantization

There are several ways of quantizing the pitch we get from the pitch detection module. This quantization can be skipped, if one thinks to directly apply the DTW algorithm. However, in order to be complete, we think that having a quantization can provide some enhancement, especially with better fitted representation ways, such as inter-note pitch differences.

An **absolute quantization** would be to decide before the performance what the reference is. It is similar to the synchronization of the tune for an orchestra. The reference we can choose is, for example, a 'A3' note at $f_0 = 440$Hz. Afterwards, we assign to the analyzed frequency f the value $f_{\text{quant}} = 2^{\frac{i}{12}} * f_0$, where $i \in \mathbb{Z}$ is the integer number that is the closest to $12 * log_2(\frac{f}{f_0})$. The outline of the algorithm is then:

1. Initialization: set f_0 to 440 Hz (or whatever fundamental frequency or note).

2. For each pitch f_n in the sequence to analyze:

   - Compute $I = round(12 * \log_2\left(\frac{f_n}{f_0}\right))$
   - The new frequency is therefore: $f_n^* = f_0 * 2^{\frac{I}{12}}$

Note that the function round returns the nearest integer of the evaluated quantity. I can be negative.

As we said earlier, if in general we can consider that a professional singer always refers himself to a global tone that is defined before the performance (in our example, f_0), this can hardly be the case for an amateur performer. In order to be able to deal with "poor performances" in the sense that the notes sung are not related to a unique reference, we introduce another algorithm.

The main difference here is that the singer is supposed to be singing partly with a global reference and partly with the reference of the notes he just sang. The latter can be called a **"relative reference"**. In order to model this type of reference, when assigning a frequency to a new note, besides the global reference, we consider the K notes sung before and we compare them to the new one. If the new note is too "far" from the global reference, we then compare it to these K notes and choose the one that is closer in the sense of equation 7, meaning that we are to compare the logarithms of the frequencies.

1. Initialization: set f_0 to 440 (or whatever fundamental frequency or note).
2. for each pitch f_n in the sequence to analyze:

   - compute $I_{\text{real}} = 12 * \log_2\left(\frac{f_n}{f_0}\right)$
   - Evaluate the quantity $Diff = I_{\text{real}} - \text{round}(I_{\text{real}})$.
   - If Diff $<$ threshold, then $I_{\text{real}} = \text{round}(I_{\text{real}})$,
     else
     - for $k = 1 \dots K$, $I_k = 12 * \log_2\left(\frac{f_n}{f_{n-k}}\right)$
     - $k^* = \text{argmin}_{k \in [1,K] | k = \text{real}} \left(I_k - \text{round}(I_k)\right)$
     - $f_n^* = f_{n-k^*}^* * 2^{\frac{\text{round}(I_{k^*})}{12}}$

Some remarks on the preceding algorithm: again, the round function returns the nearest integer, so that the threshold only has a meaning if `threshold` < 0.5. The choice of K is arbitrary, however, it seems more close to a certain "reality" if we take it not too big. The model is about relative tones singing, such that the singer is more likely to refer to two or three notes before. This model should be able to compensate errors such as a change of tone during the piece - that is to say the reference at the beginning and the one at the end is not the same, as well as errors such as accidentally false notes - which means that the reference did not change, except for occasional notes out of key.
The last formula, to compute f_n^*, uses $f_{n-k^*}^*$. We use as reference the formerly computed value of the frequency, that is to say the already quantized one. This algorithm, in the end, finds the previous of the K notes of the original sequence (plus the global reference f_0) that is closer reference for the current note, let f_n be the current note and f_m the closest reference. Then, in order to compute the new quantized note, we chose the corresponding value for f_m in the quantized frequency sequence, f_m^*. A last, f_m^* is taken as the reference to compute f_n^*.

We also tried an algorithm based only on the relative tones, but it led to some undesirable results. With relative reference, say the previous note as reference, the algorithm was going too easily out of bound and in that way was modifying too deeply the intended melody.

3.2.4 Application to the Chinese Pop Song

For our example of the previous section, we obtained the result shown on figure 10.

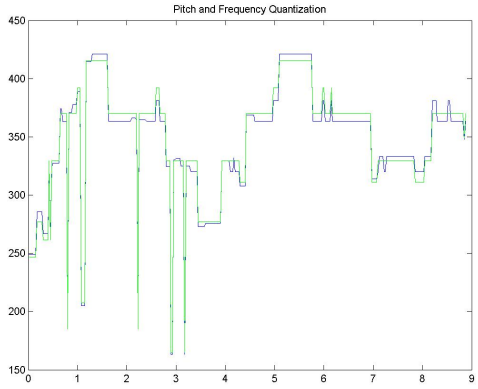

Figure 10: Step 2. Frequency Quantization (in green, the result of the quantization).

There are still some improvements to expect, especially since some values in the series are obviously aberrant, such as protrusions or notes that last less than what was defined as our final resolution.

3.3 Time Quantization

The sequence of pitches we obtain is, in a certain extend, too precise. In order to have a good precision in the rhythms, we have computed the former pitch detection algorithm with a higher time resolution than we actually need. This may have led to a pitch sequence that possibly contains improbable sequences. We detail some examples below. We have set some heuristics that can help us to find the final musical score we will use, without being computationally too expensive.

3.3.1 Some Examples of undesired situations in the pitch sequence

The pitch detection algorithm is far from being perfect, and we need to "smooth" the obtained signal. This smoothing is quite specific, since we have a signal which values are discrete, and we want to stay in a discrete space. That is why we can not, at first sight, apply a classical smoothing techniques.

We can consider some cases, as we can see on the following figures:

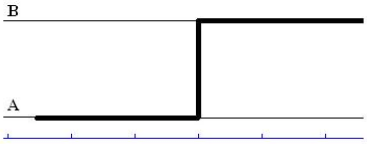

Figure 11: Ideal case

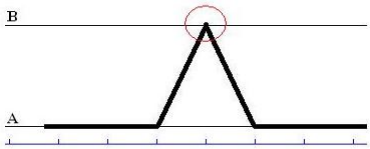

Figure 13: One “error” in the sequence

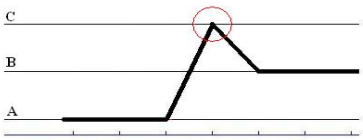

Figure 12: Error during a transition

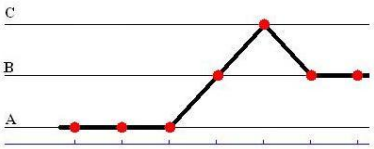

Figure 14: Error just after a transition

We can try to figure out how the program should decide whether to consider a note as an abberation or not by seeing how our perception works. We assume that we want notes that are at least two-samples long, which means that notes that exist only on one sample are either “errors” or “good” notes that are preceded or followed by an abberation.

First, on figure 11, we can see the ideal case, “AAABBB”, where we would not have any problem to decide. If we compare with the case on figure 13, “AABAA”, we clearly see that the central value, B, should be discarded, since it appears only for a short time (one sample), and replaced by a A. And we know that it is not the other way round - all the ‘A’s becoming ‘B’s - especially since B is “surrounded” by several ‘A’s.

Now figure 12, “AACBB”, reminds us of the previous one, except the sequence finished with a value that is different from that of the beginning. We can still say that C is surrounded by several A and several B. The last case we consider here on figure 14, “AABCBB”, is slightly more difficult. If we see the whole sequence, then, we can unmistakably decide that the value C is not on the right place. However, if we think that we will have to check for the abberations in a sequential order, then, this last case and the one before will not be different, say, up to the sequence “AACB”. At such a step, it is not possible to decide which one, between C and B, one should discard. We need the next element in the sequence to be able to make the decision. We actually use it in the heuristic shown in the next section.

Another case can show up, which is “ABABAB”. In that case, we can not say what value should be kept. We can only leave the decision to a defined heuristic, set without any indication as concerns how to address the latter situation.

3.3.2 Heuristics to address the above issues

Why should we use a heuristic, instead of actually compute the frequencies ? Actually, what could seem more straight-forward and maybe more reliable would be to take segments which are as long as the resolution wanted permits them and then compute the mean value on these segments, also

allowing overlapping, to smooth slightly the result.

However, for our application, we noticed that this kind of algorithm does not lead to a good result. What we want is to get rid of the abberations, which can be really far from the actual value. If we compute a basic mean value, taking in account the abberations would lead to an even worse result. We can not smooth the signal with an algorithm using a mean value or a median value. Some tests showed that it was not really satisfying.

That is why we decided to implement a simple heuristics that could use some rules (those defined above) to assign what was supposed to be the right tone for a given segment.

We check whether the data s at the sample i is closer to the previous sample or the next one. If $|s(i) - s(i-1)| \leq |s(i) - s(i+1)|$, we can decide to affect $s(i-1)$ to $s(i)$: $s(i) := s(i-1)$. Since it is an algorithm that runs from the beginning to the end of the data, this way of doing is reasonable. On the contrary, if $|s(i)-s(i-1)| \geq |s(i)-s(i+1)|$, then we have two possibilities: first, if $s(i+1) = s(i+2)$, then we keep this value for $s(i)$ as well. $s(i) := s(i+1)$. Else, if $s(i) = s(i+2)$, then we modify $s(i+1)$: $s(i+1) := s(i)$.

- If $|s(i) - s(i-1)| \leq |s(i) - s(i+1)|$, then $s(i) := s(i-1)$;
- else
  - if $s(i+1) = s(i+2)$, then $s(i) := s(i+1)$,
  - if $s(i) = s(i+2)$, then $s(i+1) := s(i)$.

We note that we do not do anything in the case when the three values, $s(i)$, $s(i+1)$ and $s(i+2)$ are different. We wait until we know the next sample at $i+3$, which is done by going on progressing on the data series. Concretely, if we take the above examples:

```
AAABBB  --->    AAABBB
AABAA   --->    AAAAA
AACBB   --->    AABBB
AABCBB  --->    AABBBB
ABABA   --->    ABBB?
```

As we can see, this algorithm gives rather good results. It essentially helps to avoid that the basic "noise" in the time series. Now we have to proceed to the time quantization itself. We assume that the wanted resolution corresponds to a window that is at least bigger than 2. This means that the resolution for the pitch detection algorithm has to be higher than for the time quantization. We first eliminate the protrusions with the above algorithm. Then we run the following algorithm, with a sliding window:

```
for i=0:Nwindow-1
    window = pitch(1 + i:Lwindow + i);
    if window(1) == window(end)
        window = ones(size(window))*window(1);
    else
        for j = 2:Lwindow-1
            if (abs(window(j)-window(Lwindow)) < abs(window(j) - window(1)))
                window(j) = window(end);
            else
                window(j) = window(1);
```

```
            end
        end
    end
    pitch(1+i:Lwindow +i) = window;
end
```

In the above lines, which are in MATLAB code, `Lwindow` is the size of the window (dependent on the resolution and the sampling rate, as well as dependent on the previous resolution, because we take the output of the pitch detection part), `window` is the window currently being processed, and `pitch` is the output of the pitch detection algorithm.

At each step we check if the first and the last elements of the analysis window are equal. If they are, then we affect the value of the first element to the whole window. We are given a resolution, so that we should have, ideally, in the end, only fragments of signal that are at least `resolution`-long. If the two extremes of the window are different, then, for every element between them, we choose the side that is closer to their value and affect it to them as being their new value.

3.3.3 Application to the Chinese Pop Song

At last we can complete the pitch tracking of our Chinese Pop Song. Figures 15 and 16 exhibit the last two heuristics and how they help to retrieve a sequence that is very likely to be a realistic transcription of the intended melody (in red on figure 16).

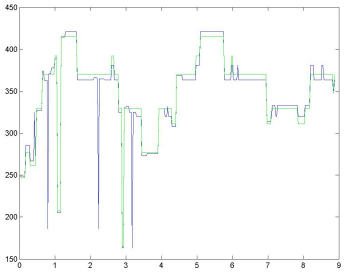

Figure 15: Step 3. Eliminating the protrusions (in blue, the original pitch sequence, in green the output of this step).

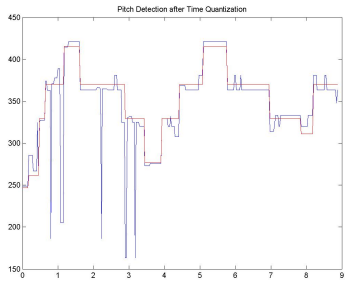

Figure 16: Step 4. Time Quantization (in red).

4 Representation of the Songs

The representation problem in song matching is critical. As we will show in this section, this is probably the main issue to be addressed. We expose here the ideas and the solutions we tried. In the results section, later, we will analyze their respective efficiency and see if they are worth investigating further more.

4.1 Key-Issues in Representing a Song

4.2 Choice of a Pitch Representation: Solving the Transposition Problem

Without talking about finding the exact tone in which the song is played or sung, we still have to process the data in order to avoid key issues. This is a transposition problem.

The tone in which you sing a song is the main tune in which the song is written. However, talking about melodies, it seems obvious that, for a human, if you sing or play a song in certain key and then play it in other key, you will still be able to recognize it. Our system is intended to recognize the songs as much as possible as would do a human "expert". In particular, a singer without training will not be able to produce the melody exactly in the original key. In that extend, we need an algorithm that can "adapt" the key of the songs in the database to the tone used in the query.

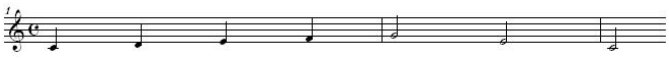

Figure 17: A 'C' (Do) scale

Figure 18: A 'F' (Fa) scale, which could be seen as the above 'C' scale shifted in pitch.

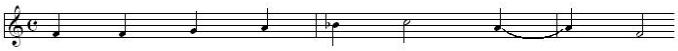

Figure 19: The above 'C' scale shifted in pitch and time

The figures 17, 18 and 19 show some situations that can occur. In particular, the figure 18 shows a "vertical shift" of the scale in 17, that is to say a shift in the frequency domain. This case happens when the singer sings exactly with the same tempo, but in a different key. Here, instead of singing a 'C' as first note, he begins with a 'F' note. For that example, let us use a simple notation to explain the basic idea to avoid the problem of transposition.

We adopt, as notation, a time series representation, where each time unit is a quarter note. We can therefore represent the first figure 17 as: `[0, 2, 4, 5, 7, 7, 4, 4, 0, 0]`. Here we are not making distinctions as wether a sound is repeated or not. The numbers correspond to a "MIDI-like" notation, each increment (respectively decrement) of 1 corresponds to an increment (respectively decrement) of a half-tone. 0 is chosen as the lowest note here, ie 'C3'.

The second figure 18 is represented by the vector: `[5, 7, 9, 10, 12, 12, 9, 9, 5, 5]`. The last figure is then: `[5, 5, 7, 9, 10, 12, 12, 9, 9, 5]` (let us say all those fragments have the same size, ie 10 time units).

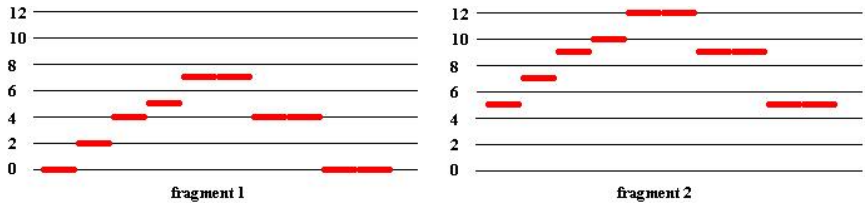

Figure 20: Symbolic representation of the first and second fragments.

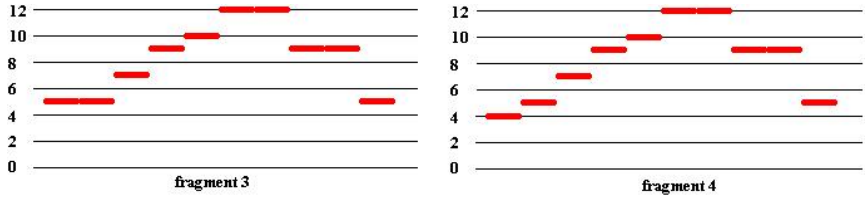

Figure 21: Symbolic representation of the third fragment and a modified version of it.

Figures 20 and 21 show a graphical representation of the vectors explained above. The following sub-sections exhibit some strategies that we can use to compensate the transposition problem in those cases.

4.2.1 A Global Reference Strategy

A first strategy is to consider that every fragment should begin with the same note. This is motivated by the idea that if a singer (rather good) begins with a note, he will sing up to the end with a reference that is the same as for this first note. In the terms we already used in the previous section (the Pitch Detection algorithm), this corresponds to the idea of global reference. In some cases, although this way of doing can solve our problem. Let us say the first note will be 0 for every fragment. this gives us the 3 fragments: `[0, 2, 4, 5, 7, 7, 4, 4, 0, 0]`, `[0, 2, 4, 5, 7, 7, 4, 4, 0, 0]` and `[0, 0, 2, 4, 5, 7, 7, 4, 4, 0]`. As we can see, this simply gives the same fragment, and for the last one, it is just the first segment, shifted by one time-unit.

However, as we said, singers that can keep that reference from the beginning to the end are very few. Let the last example, the shifted version, begin with an hypothetical previous note, let

us say a 'E' (4 for our notation)note: `[4, 5, 7, 9, 10, 12, 12, 9, 9, 5]`. For this fragment (fragment 4 on figure 21), we obtain the corrected vector: `[0, 1, 3, 5, 6, 8, 8, 5, 5, 1]`. As we can imagine, the success of this method is highly dependent on the fact that the first note is representative of the rest of the fragment. Therefore, the more this first note is "false", the more the shift will be heavy, and then bias the final result. That is why another strategy seems necessary in order to represent more trustfully the pitches.

4.2.2 A Relative Reference Strategy

The next idea is to compute the relative pitch differences. This is compatible with the relative reference we introduced earlier. Instead of directly using the pitch series we take the successive differences of pitch. For the first fragment, we obtain: `[2, 2, 1 , 2, 0, -3, 0, -4, 0]`. The second fragment gives: `[2, 2, 1 , 2, 0, -3, 0, -4, 0]`. For the first sample, we keep a possible identification. The last example also gives a fragment that can match together with the first one: `[0, 2, 2, 1 , 2, 0, -3, 0, -4]`.

At last for the fragment with a "wrong" first note, we obtain: `[1, 2, 2, 1 , 2, 0, -3, 0, -4]`. This time, only the first element of the series can induce an error. Apart from that, the series is exactly the same as the preceding one. What is more, this algorithm keeps a satisfying representation even if, during the fragment, there is a shift.

If there ever is a change of the reference in the middle of the fragment, but if all the pitch ratios stay the same as in the original (we are talking about the *log* ratio as defined previously), apart from that error, then the first strategy would just return a fragment that is true half of the time and then completely wrong, while the latter strategy would just find one error and assume the rest is right. One should however note that if the singer is too approximative in his performance, especially for some songs where the melody "moves" a lot, then it might be possible that the second strategy would lead to a wrong interpretation of the fragment. For example, if every two note is wrong while the rest is right, the result will seem random to the program.

We later on this paper do some tests on these representations. We can however say that even if, musically speaking, the second strategy of representation seems more accurate and close to reality, it would be interesting to research more this aspect and check real audio recordings of a large panel of people singing, in order to know which representation, global pitch representation or relative pitch difference would fit more our application. Since this kind of study needs a lot of singers and samples to be relevant, as well as a lot of work - which can hardly be automatic, and because this is not directly the purpose of this study, in a short term, we chose not to further investigate this matter.

4.3 Segmentation

When the user sings a query, the system is not supposed to know which part of which song he is singing. That is why, we have to find a way for the algorithm to be able to identify the query sequence even if it is in the middle of one song in the database.

4.3.1 Our Model, SoundCompass

In order to complete that task, we based our system on the segmentation done in the SoundCompass system [6]. We thought it was the most interesting way of thinking.

In [6], the authors explain how they cut the MIDI files in order to make the segments of the database. There are two different segmentations, the first one is held with a constant sliding window, the CSTPV (constant slide-length tone-in phonetic value) and the other one with a variable sliding window, the VSTPV (variable slide-length TPV). "TPV" (Tone-in Phonetic Value) actually is a notation which our notation in the previous section is based on: each note in the MIDI track is given its MIDI value (an integer from 0 to 127) and the durations of the notes are represented according to the time unit. This time unit (called "pvrs" in [6], "phonetic value rates") is evaluated thanks to the rhythm that appears the most often in the track. Then every other rhythm is quantified after that value. The authorized values are, for rhythms in SoundCompass: 0.5, 1.0, 1.5, 2.0, 3.0, 4.0, 5.0 and 6.0. Over 6.0, the value are diminished to 6.0, because it seems unlikely that singers could sing such long notes accurately.

To sum up, the CSTPV can be seen as segments that are extracted from the song, all with the same length, and with a constant overlap, which means that each segments begins exactly at a constant slide from the previous one. For example, each segment could begin at a new bar. However, since this way of doing can seem slightly arbitrary, the authors imagined another solution.

VSTPV segments are segments which do not begin at predetermined places. The common sense tells us that when singing a song, the user will probably begin with some relevant passage in the song, thus the importance of the concept of distinguishable for the notes. The authors of [6] noticed that the best retrieval accuracy for their system was obtained when taking the highest tone as the most distinguishable, in a segment. The idea for the VSTPV is to take these most distinguishable notes as beginning notes for the segments.

4.3.2 Our Segmentation solution

Following the SoundCompass method to segment the files, we implemented a segmentation algorithm that cuts the input time series into segments of same size (or possibly two sizes, one for the constant slide-length and one for the variable slide-length segments). We kept the dual idea of constant/variable slide-length, because the concept of "distinguishable" notes seemed interesting and promising. We define four variables, `staticLength`, `staticOverlap`, `varLength` and `varOverlap`. Actually, for now, we use the same values for `staticLength` and `varLength`, same thing for `staticOverlap` and `varOverlap`. Changing this fact could be the object of other tests, but we did not try to test these parameters independently.

As opposite to the SoundCompass system, we wanted to try to apply the segments without the time normalization proposed. That is to say, our unit is directly time. We assumed that the durations should be a decimal number with only one decimal. Thus by taking the durations multiplied by 10, we obtained the number of samples unit for the current note. Let our input data be the output of our pitch series extraction. We have then n couples of the form (f_i, t_i). We only allow time durations t_i with one decimal, thus they have a precision of one tenth (0.1) second. To obtain the time series we will work on, we produce a sequence such that we transform each couple (f_i, t_i) into a sub-sequence $\underbrace{(f_i, f_i, \ldots, f_i)}_{10*t_i \text{times}}$. In the end, we obtain a sequence of the form:

$$\left(\underbrace{f_1, f_1, \ldots, f_1}_{10*t_1 \text{times}}, \underbrace{f_2, \ldots, f_2}_{10*t_2 \text{times}}, \ldots, \underbrace{f_n, \ldots, f_n}_{10*t_n \text{times}} \right)$$

For the constant slide-length segments, let us call them "CSS", we use a very simple algorithm.

We store the indexes that correspond to the head (the beginning) and the tail (the end) of each segment. For the "CSS"s, it is very simple, since it is possible to compute their number when we know the length of the whole sequence. Let L be that length. We want as many segments as allowed by the arguments `staticLength` and `staticOverlap`, plus a last segment which end actually is the end of the sequence (and for which we thus do not consider the overlap with the preceding segment). Let N be the number of "static" segments possible out of the sequence. Then $N = floor\left(\frac{L - staticLength}{staticOverlap}\right) + 2$. We add 2 segments in the end because we have to add the segment made in the end, plus the one that was taken off while counting the number of overlaps that are possible. Each of the N segments is therefore characterized by a beginning at $i * staticOverlap$ and an end at $i * staticOverlap + staticLength - 1$, for $i \in [0, N-2]$. The last segment begins at $L - staticLength + 1$ and ends at L.

At last, the variable slide-length segments, "VSS", are computed as beginning at only most "distinguishable" notes in the sequence, we took the same criteria for them as in the SoundCompass system, that is to say we took the highest pitch in the given range. To search for those distinguishable notes, we proceeded as follows: in the current CSS, check which note between the head of the segment and the place where the next CSS segment begins (if the first segment begins at the sample i then the second one begins at $i +$ `staticOverlap`) is the highest one. The new VSS begins at that exact note. Then, we leave a space equal to the given `varOverlap`, and check for the highest pitch from that new position to the static overlap. If the new position goes further than the overlap for the static segments, we jump to the next CSS. If not, then we go on, find the highest pitch, define the new VSS and go to a new position. This way, we are able to catch some "local" maxima in the pitch sequence.

We could imagine some other ways to define segments, for example, we could choose random places in the sequence and this way construct the new segments. This could be relevant for instance when processing with the recorded data, and choosing a random strategy that privileges the "center" of the sequence, since we could consider that the borders are not relevant. They often correspond to moments when the user usually do not sing, adjusting the microphone, his voice, or simply pushing the button to begin or stop the recording.

5 Song Matching

In order to match the song the user is singing with one of the songs in the database, we chose the DTW algorithm (Dynamic Time Warping). We first introduce some algorithms and possible enhancements and then discuss about evaluation issues, that is to say, discuss the different possibilities of evaluating a candidate song.

5.1 The Matching Algorithm

As we have seen previously, we have several segments in our database, all of which coming from the songs we entered in the system. At the same time, the song the user sang is segmented the same way (or actually it could be some other way). We want to know if the segments coming from the user's query "match" some segments in the database, that is to say, according to some similarity measurement, find out which segments in the database are more likely to be the desired ones.

5.1.1 DTW: Dynamic Time Warping

Dynamic Time Warping is a well known algorithm since it can be used for a lot of purposes. The idea is simple: find in a database the most similar sequence to a query sequence. The query sequence can be slightly different from the "original" in the database. It is especially well adapted to music queries, since an amateur singer (which would be our standard user) will probably not sing exactly with the same tempo and same rhythms as in the original we have.

In such case when the query sequence is allowed to have distortion in time, a mere comparison "sample-by-sample" with the original would be awkward. Let us take an example. Let two sequences `3 3 5 5 5 1 1` (sequence 1) and `2 2 4 4 3 3 3` (sequence 2). We make the assumption that every sequence has the same length (which is the case in our program, but this condition is unnecessary when using the DTW algorithm). We define the similarity distance between two sequences as an euclidian distance between the two "vectors".

$$d(seq_1, seq_2) = \sum_k (seq_1(k) - seq_2(k))^2$$

Using that definition, the distance between sequence 1 and sequence 2 is 16. Let us consider the following sequence, which is a modification of sequence 1: `3 3 3 5 1 1 1`. It is still obvious that this sequence, compared with a reasonable similarity distance should be closer to sequence 1 than from sequence 2. However, a quick calculus shows that the distance between this query and sequence 1 is already 20 and the distance with sequence 2 is still 16! According to this similarity distance, we should then say that the query sequence matches sequence 2 and not sequence 1, as expected. The straight-forward algorithm fails at addressing our sequence matching problem.

```
seq1        3   3   5   5   5   1   1
query       3   3   3   5   1   1   1
distance    0+  0+  4+  0+  16+ 0+  0   = 20

seq2        2   2   4   4   3   3   3
query       3   3   3   5   1   1   1
distance    1+  1+  1+  1+  4+  4+  4   = 16
```

That is why we need a more reliable algorithm to match sequences. The **DTW algorithm** gives us a convenient solution.

As we have seen above, the query sequence can be the modification of an original sequence, with possible errors in the sequence of values ("replacement") or time delays (not necessarily with constant delay through the sequence). The idea in DTW is to find a "path" between the two sequences that minimizes the distance. Choosing a path corresponds to choosing which element of the first sequence should be compared to which element in the second one. Ideally, considering our first example, all the 3s should be compared together, all 5s as well and same for the 1s. In the end, it would lead to a similarity distance of 0. Of course, one cannot choose randomly which element to compare with which, there has to be rules, there should for example be restrictions as concerns the set of elements of one sequence you can compare with each element of the other.

In principle, we have to use two matrices. The first one is stores the distance values while the second one is used to store the "up to now" similarity distances. We first compute the distance matrix d, whose element $d[i, j]$ is the distance between the element i of the first sequence and the element j of the second. This distance can be any distance, we chose for our final algorithm the absolute value of the difference of the two elements:

$$d(a, b) = |a - b|$$

However to illustrate this section, we will still use the euclidian distance. Once we have this distance matrix, we can initialize the other one.

$$d = \begin{bmatrix} & 3 & 3 & 3 & 5 & 1 & 1 & 1 \\ 3 & 0 & 0 & 0 & 4 & 4 & 4 & 4 \\ 3 & 0 & 0 & 0 & 4 & 4 & 4 & 4 \\ 5 & 4 & 4 & 4 & 0 & 16 & 16 & 16 \\ 5 & 4 & 4 & 4 & 0 & 16 & 16 & 16 \\ 5 & 4 & 4 & 4 & 0 & 16 & 16 & 16 \\ 1 & 4 & 4 & 4 & 16 & 0 & 0 & 0 \\ 1 & 4 & 4 & 4 & 16 & 0 & 0 & 0 \end{bmatrix}$$

There are different ways of initializing this DTW distance matrix D. This matrix has the same size as the former one. Let us say the lines correspond to the elements of the first sequence and the columns to the ones of the second sequence. The most basic way of initializing is to set the first column and the first line as the cumulative sum of the elements of the distance matrix, thus giving:

$$D = \begin{bmatrix} & 3 & 3 & 3 & 5 & 1 & 1 & 1 \\ 3 & 0 & 0 & 0 & 4 & 8 & 12 & 16 \\ 3 & 0 & & & & & & \\ 5 & 4 & & & & & & \\ 5 & 8 & & & & & & \\ 5 & 12 & & & & & & \\ 1 & 16 & & & & & & \\ 1 & 20 & & & & & & \end{bmatrix}$$

To fill in the rest of the matrix, the algorithm works a little bit like the Viterbi algorithm. At each step, we compute the best distance so-far up until the whole matrix is complete. Then we return the last element. In order to compute that distance, we have to define the set of possible

choices when considering an element. Let the length of the first sequence be n_1 and the length of the second one n_2. Let us have a look to the algorithm itself:

- for i from 2 to n_1
  - for j from 2 to n_2
    1. Choose $(k, l) \in \text{set}_{(i,j)}$ so that $D(k, l)$ is minimum.
    2. $D(i, j) = D(k, l) + d(i, j)$

The set $\text{set}_{(i,j)}$ depends on (i, j), and can be, for instance: $S_1 = \{(i-1, j-1); (i-1, j); (i, j-1)\}$ or $S_2 = \{(i-2, j-1); (i-1, j-2); (i-1, j-1); (i-1, j); (i, j-1)\}$. The figure 22 shows how the algorithm works with these sets. During the algorithm process, we choose the "predecessor" that has the minimal value.

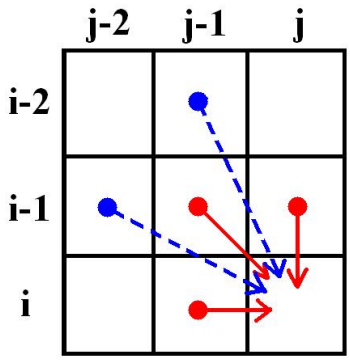

Figure 22: Sets for the DTW algorithm. In red continuous arrows: S_1 and in blue discontinuous arrows: S_2

One should bear in mind that if $(k, l) \in \text{set}_{(i,j)}$, then we should have $k \leq i$ and $l \leq j$, and (i, j) is not in $\text{set}_{(i,j)}$. This would meant to avoid to go backwards.

All in all, the outline of the whole algorithm is given below:

1. Compute the distance Matrix d: $d(i, j) = d(seq_1(i), seq_2(j))$, for $(i, j) \in [1, n_1]x[1, n_2]$
2. Initialize the matrix D:
   - $D(1, 1) = d(1, 1)$
   - for $i \in [2, n_1]$, $D(i, 1) = D(i-1, 1) + d(i, 1)$
   - for $j \in [2, n_2]$, $D(1, j) = D(1, j-1) + d(1, j)$
3. Fill in the matrix D:
   - for i from 2 to n_1
     - for j from 2 to n_2
       (a) Choose $(k, l) \in \text{set}_{(i,j)}$ so that $D(k, l)$ is minimum.

(b) $D(i,j) = D(k,l) + d(i,j)$

4. Return $D(n_1, n_2)$.

For our example, we can then compute the matrix D for sequence 1 and sequence 2:

$$D_1 = \begin{bmatrix} & 3 & 3 & 3 & 5 & 1 & 1 & 1 \\ 3 & 0 & 0 & 0 & 4 & 8 & 12 & 16 \\ 3 & 0 & 0 & 0 & 4 & 8 & 12 & 16 \\ 5 & 4 & 4 & 4 & 0 & 16 & 24 & 28 \\ 5 & 8 & 8 & 8 & 0 & 16 & 32 & 40 \\ 5 & 12 & 12 & 12 & 0 & 16 & 32 & 48 \\ 1 & 16 & 16 & 16 & 16 & 0 & 0 & 0 \\ 1 & 20 & 20 & 20 & 32 & 0 & 0 & 0 \end{bmatrix}$$

$$D_2 = \begin{bmatrix} & 3 & 3 & 3 & 5 & 1 & 1 & 1 \\ 2 & 1 & 2 & 3 & 12 & 13 & 14 & 15 \\ 2 & 2 & 2 & 3 & 12 & 13 & 14 & 15 \\ 4 & 3 & 3 & 3 & 4 & 13 & 22 & 23 \\ 4 & 4 & 4 & 4 & 4 & 13 & 22 & 31 \\ 3 & 4 & 4 & 4 & 8 & 8 & 12 & 16 \\ 3 & 4 & 4 & 4 & 12 & 12 & 16 & 16 \\ 3 & 4 & 4 & 4 & 16 & 16 & 16 & 20 \end{bmatrix}$$

That is why, we can say this time without ambiguity that the query sequence is more similar to sequence 1 than to sequence 2, since $D(7,7) = 0$ and $D_2(7,7) = 20$.

It was not necessary for our case, but one can also store the predecessors for each element of the matrix D so that in the end, you are able to rebuild the best path between the two sequences. Some enhancements are however to be discussed. In fact, this algorithm has a huge cost in terms of computation time, and we need to save time in order to give a result in a reasonable time. To compare two sequences, we need to make computations in a $\mathcal{O}(n_1 * n_2)$ time. Since we have potentially a lot of songs, leading to a lot of segments each of which we will have to compare to the query segments, with length n to be determined. Let the number of segments in the database be N, the number of query segments be M, this means that we will have to make computations in $\mathcal{O}(N * M * n^2)$. This prohibitively expensive in computation time, and finding a faster way of computing is critical for the matter at hand.

Next, we introduce two methods to enhance the DTW algorithm. The first one, the "early stopping" algorithm tries to reduce the computations during one comparison, while the next one, the "FTW" (Fast Time Warping) algorithm is a synthetic algorithm gathering any possible way of accelerating the process, especially by reducing the number of comparisons in the end. Both are adapted from [].

5.1.2 Restrictions on the scope of the path and the Early Stopping Algorithm

A first way of enhancing the average speed of the DTW algorithm is to put restrictions on the scope of the path. It means that instead of investigating the whole matrix to find the path between the two sequences, we restrict it to a certain part of the matrix. We hope that the two sequences we want to compare are not too different in terms of time warping, that the query is just a slight

variation of the original. In such case, the path cannot be too far from the first diagonal of the matrix.

The figure 23 shows two types of restriction. The first one limits the warping scope to a band around the first diagonal of the matrix. The other one limits the scope to a parallelogram. The first solution allows the sequences to match even if they do not begin at the same point, and can allow at the same time to finish at a different point from (n_1, n_2). Both algorithms put constraints on the time warping, limiting the variations to be between one half the original speed and twice as much as this speed.

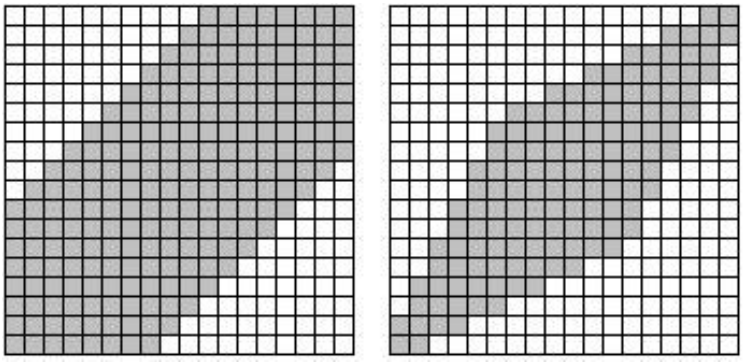

Figure 23: Constraints on the warping scope.

In practice, we can take the same algorithm as explained before. During the initialization, you should also fill the rest of the matrix D with ∞. What you compute at each step is $D(i, j)$ only if (i, j) belongs to the warping scope. Concretely for the first constraint, the formula is:

$$\frac{n_2}{n_1}i - n_2\alpha \leq \quad j \quad \leq \frac{n_2}{n_1}i + n_2\alpha$$

The parameter α is a ratio between 0 and 1. The bigger α is, the bigger the scope is. For the second type of constraint:

$$\begin{aligned} \frac{n_2}{2n_1} \leq \quad & \frac{j}{i} \quad & \leq \frac{2n_2}{n_1} \\ \frac{n_2}{2n_1} \leq \quad & \frac{j - n_2}{i - n_1} \quad & \leq \frac{2n_2}{n_1} \end{aligned}$$

The **"Early Stopping"** algorithm as shown in [11] is very much like the constraints we have just seen. However, what is really different is that this algorithm adapts the scope according to a given threshold d_{cb}, which stands for "current best". Typically, if we assume that we keep an updated list of the current k best sequences as well as the DTW score they obtained, we can define the d_{cb} as the k^{th} DTW score of the list.

The principle of the Early Stopping algorithm is that, if we can be sure that some "track" in the matrix has no hope of returning a better result than the chosen threshold d_{cb}, we stop the searching of the path on this track. For that purpose, we have to dynamically update two vectors, `begin` and `end`. For each line i, we compute the elements of $D(i, :)$ in the scope `begin`$[i]$ and `end`$[i]$, and

according to the values we obtain, we update the vectors `begin` and `end` to dynamically adapt the scope. Below the algorithm outline as given in [11]:

- Initialization of `begin` and `end`:
  for $i = 1$ to n_1 do
  1. `begin`$[i] = 1$
  2. `end`$[i] = n_2$
- Compute the DTW distance:
  for $i = 1$ to n_1 do
  - for $j =$ `begin`$[i]$ to `end`$[i]$ do
    compute $D(i, j)$
    * if $i \neq 1$ and $i \neq n_1$ then
      · if $j >$ `end`$[i-1]$ and $D(i, j) > d_{cb}$ then
        1. `end`$[i] = j$
        2. break
  - if there is no j such that $D(i, j) \leq d_{cb}$ then
    return $D(i,$ `end`$[i])$
  - else
    `begin`$[i] = min\{j | D(i, j) \leq d_{cb}\}$
    `end`$[i] = max\{j | D(i, j) \leq d_{cb}\}$
    if $i \neq n_1$ and `begin`$[i+1] <$ `begin`$[i]$ then
    `begin`$[i+1] =$ `begin`$[i]$
- return $D(n_1, n_2)$.

As we can see on the process of the algorithm, this way of defining the vectors `begin` and `end` do not allow to go back (by updating at each step i the value of `begin`$[i+1]$) and stops earlier (thus the name of the algorithm) in the line when we are sure that going further would lead to a result higher than d_{cb} (by stopping when $D(i, j) > d_{cb}$ and that we went over the previous `end`).

If we apply this algorithm to our example, in the first section, with a threshold of 4 (for instance), then we obtain a matrix D for which we did not have to compute every elements. This threshold means that we already compared the query with another sequence for which the result was 8.

$$D_1 = \begin{bmatrix} & 3 & 3 & 3 & 5 & 1 & 1 & 1 \\ 3 & 0 & 0 & 0 & 4 & 8 & 12 & 16 \\ 3 & 0 & 0 & 0 & 4 & 8 & - & - \\ 5 & 4 & 4 & 4 & 0 & 16 & - & - \\ 5 & 8 & 8 & 8 & 0 & 16 & - & - \\ 5 & - & - & - & 0 & 16 & - & - \\ 1 & - & - & - & 16 & 0 & 0 & 0 \\ 1 & - & - & - & - & 0 & 0 & 0 \end{bmatrix}$$

$$D_2 = \begin{bmatrix} & 3 & 3 & 3 & 5 & 1 & 1 & 1 \\ 2 & 1 & 2 & 3 & 12 & 13 & 14 & 15 \\ 2 & 2 & 2 & 3 & - & - & - & - \\ 4 & 3 & 3 & 3 & 4 & 13 & - & - \\ 4 & 4 & 4 & 4 & 4 & 13 & - & - \\ 3 & 4 & 4 & 4 & 8 & 8 & - & - \\ 3 & 4 & 4 & 4 & 12 & - & - & - \\ 3 & 4 & 4 & 4 & 16 & 20 & 24 & 28 \end{bmatrix}$$

5.1.3 FTW: Fast Time Warping

In order to compute even faster the similarity distances and find the most similar song to the query, we can enhance the computing of the distance itself, as we have seen before, and we can try to prune as soon as possible sequences (segments from songs) that are less similar to the query. If we can prune some of the segments early enough, we could significantly improve the computing performances.

The principle for the FTW (Fast Time Warping) algorithm is to combine another similarity measure, the Lower Bounding distance with Segmentation (LBS), the Early Stopping algorithm and a technique of refinement. We will detail further more each of these techniques, as they are explained in [11].

1. *LBS, similarity measure for the FTW*
   We consider a sequence S, with n elements. An element of S is noted as $S[i]$, for $i \in [1, n]$. If we want to compare two sequences S_1 and S_2, according to the distance measure of the first section, we should have a computation time in $\mathcal{O}(n^2)$. In order to save time when we have a large database, we can imagine to compute the distance first on approximations of the sequences then prune at early stage the candidate sequences for which the similarity distance overpass a given threshold. This process will be explained in the following sections. What we need now is to define a distance that would allow us to compare the approximations of sequences.

   We need to specify a distance that would be more general than the distance we use for the final DTW, when we compare the full sequences (and not just the approximations). In order to do so, the proposed distance in [11] is based on the segmentation of the sequences into equal segments. For each segment, we give its maximum and its minimum. These quantities characterize the segment.

   When we have two sequences to compare, say S_1 and S_2, we first compute their approximate versions: S_1^A and S_2^A. Let us say these versions respectively have m_1 and m_2 segments. One segment is noted as $S^A[i] = \{S^A_{min}[i], S^A_{Max}[i]\}$, where $S^A_{min}[i]$ is the minimum on $S^A[i]$ and $S^A_{Max}[i]$ the maximum. We define the distance between $S_1^A[i]$ and $S_2^A[j]$ as follows:

$$d_{LBS}(S_1^A[i], S_2^A[j]) = \begin{cases} S^A_{1,min}[i] - S^A_{2,Max}[j] & \text{, if } S^A_{1,min}[i] > S^A_{2,Max}[j] \\ S^A_{2,min}[j] - S^A_{1,Max}[i] & \text{, if } S^A_{2,min}[j] > S^A_{1,Max}[i] \\ 0 & \text{, otherwise.} \end{cases}$$

   If we note the DTW computing by D_{exact}, as being the exact similarity distance between the query and the sequence from the database, and if we note D_{LBS} the DTW using the LBS

distance, we can prove that this distance obeys:

$$D_{LBS}(S_1^A, S_2^A) \leq D_{exact}(S_1, S_2)$$

This means that we can use a DTW using the LBS measure as a first approximation for the similarity distance, it is more general than the "normal" distance. This also means that by applying the D_{LBS} distance, and sorting the sequences according to this distance will not lead to pruning the better sequences, since it will globally keep the order of the "true" DTW. We could also prove that applying this distance first for a coarse version C of the sequences, D_{LBS_coarse} and then to a tighter version (with smaller segments) T of these sequences, D_{LBS_tight}, leads to the same hierarchy:

$$D_{LBS}(S_1^C, S_2^C) \leq D_{LBS}(S_1^T, S_2^T)$$

This result is important for the FTW algorithm as we will see later. It proves that with a tighter segmentation of the sequences, the similarity distance will tend to grow up to the exact distance.

2. *The Early Stopping Algorithm*

   As seen before, we could use the Early Stopping algorithm in a basic DTW to improve the overall speed. Here, we can do the same and implement this algorithm during the process thus restraining the warping scope, even for the approximate sequences. The algorithm works exactly the same way as explained before.

3. *Progressive Refinement*

   A progressive Refinement strategy corresponds to what we were talking earlier. We first compute an approximate estimation of the distance between the candidate sequences and the query sequence, then we prune the candidates that have a score above a given threshold (eventually a dynamic one) and at last we compute the exact distance between the rest of the candidate sequences and the query sequence. This would hopefully reduce the number of computations necessary for each query.

   In order to use this idea, we will need to store some pieces of information continually during the calculus. What is more, we can extend the application by creating several approximation resolutions, from the coarsest to the finest, up to the exact original sequence. Let c be the "coarse degree", $c = 1 \dots C$. Each version of a sequence S is noted S_c, and the length of the segments of S_c, noted l_c, obeys the relation:

$$l_1 = 1 < l_2 < \ldots < l_{C-1} < l_C < n$$

   where n is the length of S. We see that S_C is the coarsest version while $S_1 = S$ is the original series. When computing the similarity distance, we use the Lower Bounding distance introduce earlier. What is more, we have the result, for a query sequence Q and a test sequence P in the database:

$$D_{LBS}(Q_C, P_C) \leq D_{LBS}(Q_{C-1}, P_{C-1}) \leq \ldots \leq D_{LBS}(Q_1, P_1) \leq D_{exact}(Q, P)$$

   Intuitively, we would then compute the distance for the coarsest, because it leads to less calculations. Then, for those of the k first sequences that are below the threshold, we compute

the exact similarity distance and sort the results. The last distance (ie the biggest), say the k^{th} distance, is taken as first d_{cb}.

The next step is to check that none of the sequences that were put aside are not better than the ones we kept. Thus we compute the distances for these sequences. We begin with the coarsest versions and, while the distance is less than d_{cb}, we compute this distance for even finer versions of the sequence. If, in the end, the distance is less than the given d_{cb}, we update the list of k best sequences and potentially obtain a new d_{cb}.

Concretely, the outline of the algorithm as given in [11] is: (where Q is the query sequence and k is the number of wanted answers)

(a) For each sequence P in database,
- compute $d_{coarse}(P) = D_{LBS}(P_C, Q_C)$
- if $d_{coarse}(P) < tempList[k].dist$, add P in $tempList$ with $d_{coarse}(P)$ as its $dist$ field and sort.

(b) For each P in $tempList$, add P and $D_{exact}(P, Q)$ to NNL and update d_{cb}. NNL is the k-nearest neighbor list.

(c) For each P in the database,
- if P is not in $tempList$ and $d_{coarse}(P) < d_{cb}$,
  - For $i = C - 1 \dots 1$
    - $d_{approx} = D_{LBS,d_{cb}}(P_i, Q_i)$, The $D_{LBS,d_{cb}}$ is the similarity distance computed with the early stopping algorithm, d_{cb} is the current threshold.
    - If $d_{approx} > d_{cb}$, break.
  - If $d_{approx} \leq d_{cb}$, then compute $d_{exact} = D_{exact,d_{cb}}(P, Q)$
    - If $d_{exact} \leq d_{cb}$, then add P and d_{exact} to NNL and update d_{cb}.

(d) Return NNL.

As we have seen, the FTW algorithm in theory allows to reduce the computation time by reducing the number of computations in one comparison (thanks to the Early Stopping Algorithm) and the number of comparisons itself thanks to a mechanism using the threshold to prevent computing comparisons that would anyway lead to even worse distances. We will see in the results section if the use of this algorithm is critical or not, especially since it did not seem to be developed for this particular application.

5.2 Evaluation Procedures

Evaluating the results can be important for our application, since we are using several segments for each song. This means that, ideally, when a song and the query match, they have a lot of segments that have a rather low similarity distance. We adopted several ways to compare the final results. We mainly make the difference between evaluating the similarity directly from the segments and evaluating through a statistics for the whole song.

5.2.1 An Evaluation with Respect to the Segments

A first way of evaluating is the straightforward strategy. We have a list of the best segments tested over the whole database and we update it each time we compute the distance for a new segment. This algorithm does not take in consideration wether the segment belongs to a specific song or not. It can therefore lead to a result table where there is only one song (ideally the right one...). We reproduced below the first 16 best segments for the song "Obla Di, Obla Da". The correct target comes as 13^{th} answer.

```
Testing the file ObLaDiObLaDa_2_couplet.wav.midi.dat
this is eval1:
rank: 0 SgtPepperReprise_3_.txt 92 0 99 420 519
rank: 1 ADayInTheLife_6_.txt 99 0 99 760 859
rank: 2 ADayInTheLife_6_.txt 101 0 99 750 849
rank: 3 AcrossTheUniverse_1_.txt 103 0 99 90 189
rank: 4 SgtPepperReprise_3_.txt 107 0 99 300 399
rank: 5 PaperbackWriter_2_.txt 107 0 99 30 129
rank: 6 PaperbackWriter_2_.txt 111 0 99 90 189
rank: 7 Help_4_.txt 114 0 99 420 519
rank: 8 BackInTheUSSR_1_.txt 116 0 99 280 379
rank: 9 PaperbackWriter_2_.txt 116 0 99 210 309
rank: 10 Help_4_.txt 118 0 99 240 339
rank: 11 Blackbird_4_.txt 121 0 99 150 249
rank: 12 ObLaDiObLaDa_1_.txt 121 0 99 450 549
rank: 13 WhenIm64_3_.txt 122 0 99 316 415
rank: 14 LucyInTheSkyWithDiamonds_3_.txt 122 0 99 60 159
rank: 15 AHardDaysNight_2_.txt 125 30 129 0 99
...
```

However, we can imagine that, with a similarity distance that would be too general, this algorithm will return a list that actually is full of segments for which the similarity distance with a segment of the query song is 0. This could be inoffensive for our strategy, except that, with only 0 to feed our list, we can not in the end rely on that sorted list. In the end, this list would have, in the worst case, the same order as the order of exploration of the database. After our tests, we will draw some conclusions about this algorithm.

In order to avoid this problem, we tried another evaluation technique, based on a statistics over the songs.

5.2.2 Evaluation introducing Statistics for each Song

To avoid the situation where a lot of the segments would have as similarity distance 0, we decided to keep, for each song, another table of the k first best segments. Once the computing for the current test song of the database is finished, we compute the mean value of the results for those k first segments. We consider this value as being the similarity distance between the query song and the test song.

At last, we compare this similarity measure to the other ones obtained and sort the k best results. The last similarity distance (ie the distance for the worst of the k songs) is taken as the threshold for the Early Stopping Algorithm. It seems general enough to allow the algorithm not to

restrain too much and too fast the field of research for the best match for the query humming. The results using this evaluation technique are:

```
Testing the file ObLaDiObLaDa_2_couplet.wav.midi.dat
...
this is eval2:
rank: 0 SgtPepperReprise_3_.txt 178.533 0 99 420 519
rank: 1 YellowSubmarine_2_.txt 182.9 0 99 930 1029
rank: 2 Help_4_.txt 185.7 0 99 420 519
rank: 3 ADayInTheLife_6_.txt 200 0 99 760 859
rank: 4 ObLaDiObLaDa_1_.txt 209.8 0 99 450 549
rank: 5 Blackbird_4_.txt 223.233 0 99 150 249
rank: 6 HeyJude_1_.txt 229.733 33 132 300 399
rank: 7 GoodMorningGoodMorning_3_.txt 230.467 0 99 390 489
rank: 8 PaperbackWriter_2_.txt 243.567 0 99 30 129
rank: 9 GettingBetter_3_.txt 247.5 0 99 30 129
rank: 10 BackInTheUSSR_1_.txt 247.9 0 99 280 379
...
```

As an example, figure 25 shows two segments that matched during our tests. These segments correspond to a segmentation with global pitch representation, relative time expression, with 160 sample-long segments (which corresponds here to around four bars in a music score). The segments actually match since they correspond to the part *"... and make it better. Remember to let her into your heart, Then you can start..."* in the song **"Hey Jude"**. Of course, this time we were quite lucky but sometimes (probably most of the time for our program , as our tests seem to confirm) we can find very similar segments from other songs, as we can see on figure **??**. The segment for the database song **"Lucy In The Sky With Diamonds"** (left on the figure) still has a better evaluation mark than the segment for **"Love Me Do"**.

We can find the correct answer ranked fifth best matching song, which is better than the above result. However, as we will see in the next section 6, this second evaluation is not always the best one.

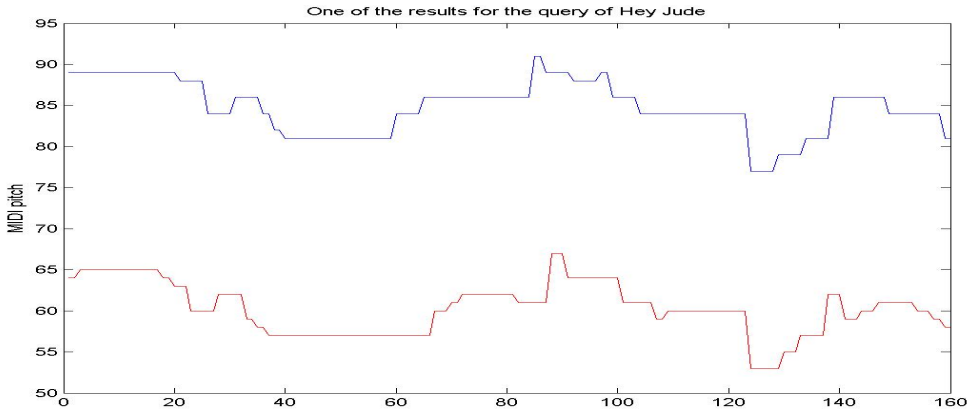

Figure 24: Two patterns that matched in our tests: in blue above the pattern from the database, in red below the pitch sequence extracted from the audio data (the performed song is "Hey Jude").

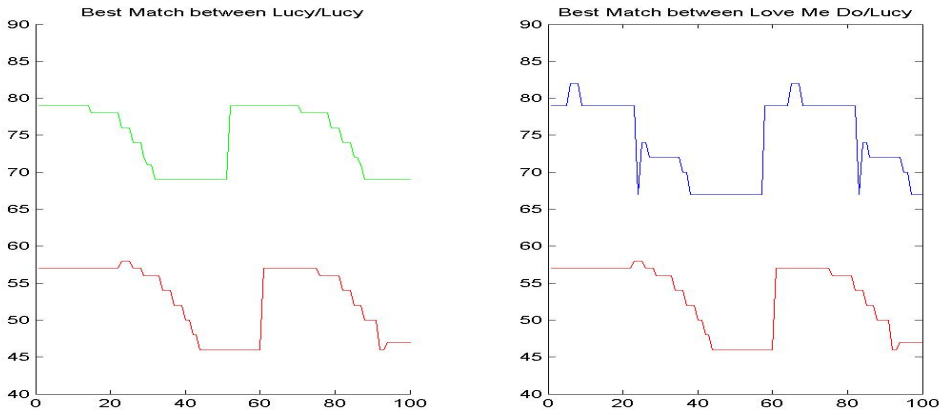

Figure 25: Query song: "Lucy In The Sky With Diamonds". Two matching patterns with high similarity scores: "Lucy In The Sky With Diamonds" (left) and "Love Me Do" (right)

6 Results

We made some tests to measure the performance of our system. We essentially wanted to test some features and different settings for the program and see what would happen. Our first test concerns the lengths and the overlap we should use for the segments. Then we tried to consider which database was the most successful. We also tested the different representation available to determine which one was the best for our application. At last, during our tests, we kept a recording of the times spent by the different possible implementations, and we try to compare the computing times, since the speed issue would be one of the key problem if we wanted to make a "commercial" application.

We tested several configurations of database and test files. Our main databases are formed by songs of the beatles. The first one is the collection of the tracks that contain the melodies of 101 Beatles songs, for a total of 148 files. We call this database `source_melodies`. A second collection is extracted from the first one, but with only one file per song, containing all the characteristic parts (for instance melodies for the verse or the refrain) of the song. These melodies were extracted manually, there are 33 songs (thus 33 files) in this database, called `melodies`. At last, we have a database where we stored the separated melodies for each song. We extracted 53 melodies out of 28 Beatles songs. This database is called in this paper `sep_melodies`.

For the test files, we basically used the above database, `sep_melodies`, since it is likely to be what a user would sing if he wanted to retrieve one of our songs. We also tested the program in a "real" environment, that is to say with recordings of users. We had two users singing songs of the Beatles, the first user sang 17 melodies, from 2 of the songs in the database: "Penny Lane" and "Obla Di Obla Da". The second user sang 87 melodies from 10 songs. We thus have a total of 12 songs, in 104 WAV files. The users were to sing each of the wanted melody in several manners, without or without the music in background, and with or without the lyrics. Having the music in background would help to sing in the same tone and tempo as in the original file. When singing with the lyrics the user sometimes seems to forget about the singing itself, focusing on the lyrics themselves. That is why it is interesting to have recordings on syllabus such as "na" or "ta", where the user concentrates more on the melody. At last singing without the music can give us some interesting samples, because we can get melodies which tune is different and which rhythms are less precise than in the MIDI file, for example.

The melodies were recorded thanks to devices such as computer microphones or mp3 players. The quality of the audio was in average quite high for the computer recordings (Fs = 44kHz, 16 bits resolution) while the mp3 device recorded was lower (16kHz with 4 bits resolution). The collection of those recorded samples will be called "mat" (standing for Matlab, since we used our Matlab program to extract the pitch series) or "wav" (because the original recordings were all WAV files). Up to now, the pitch detection algorithm written in C/C++ does not give as good results as in Matlab. The improvement of that part and its final implementation in the final system could be one of the next steps for this project.

Except if it is mentioned, when talking about the direct evaluation technique, we refer to it as **"eval1"** while when talking about the evaluation that gives a candidate song the mean value of the previous evaluations over all the segments for that song, we will refer to it as **"eval2"**.

6.1 Features for the Segments: Lengths and Overlap Issues

We ran the program with different settings as concerns the length and overlap of the segments. We wanted to see if there was any correlation between these parameters and the success rate. In the same time, we checked for which setting the results were the best, so that we could go on with our tests.

We tested several configurations. First, the **segment lengths** that were tested are: **40, 60, 80, 100 and 120**. One should keep in mind that, with the representation we decided above, these lengths correspond to segments which length, in seconds, is 4, 6, 8, 10 and 12 seconds. There is no specific reason why we chose these values more than others. We only expect most of the songs to have melodies that are in this range of duration.

For the overlaps, we took all the possible values between 5 and the lengths of the segments, taken every 5 sample. It means that, for the 40-sample-long segments, we tested the following overlaps: 5, 10, 15, 20, 25, 30, 35 and 40.

1. **Having too many segments leads to biased results:**

   We have noticed that, as concerns eval1, the results for all the databases tested on sep_melodies were all rather high: almost 100% for any length, any overlap. This means that the recall ratio is good. There is however one case when this is not true: when the lengths are small, with a little overlap. Actually this situation also means that the number of segments is rather big. On figure 26, it is obvious that the results are not as good for the small segments (40 sample-long), and getting worse with decreasing overlap. We could explain this thanks to the number of segments involved. Figure 27 shows how many segments are made in average per file in the database.

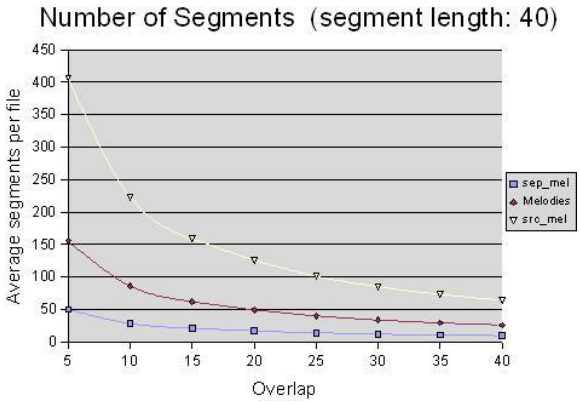

Figure 27: Number of segments of length 40 for the three databases, in function of the overlap.

   This problem can be due to redundancies in the database, with the presence of segments that are similar to each other. At that time, if we try to find the most similar of all the segments, when computing the DTW, we might find several candidate segments for which the result is low, the worst case being when a lot of segments are so similar that every segments gets a similarity distance of 0. This could happen for example if there is a note that last too long in the song.

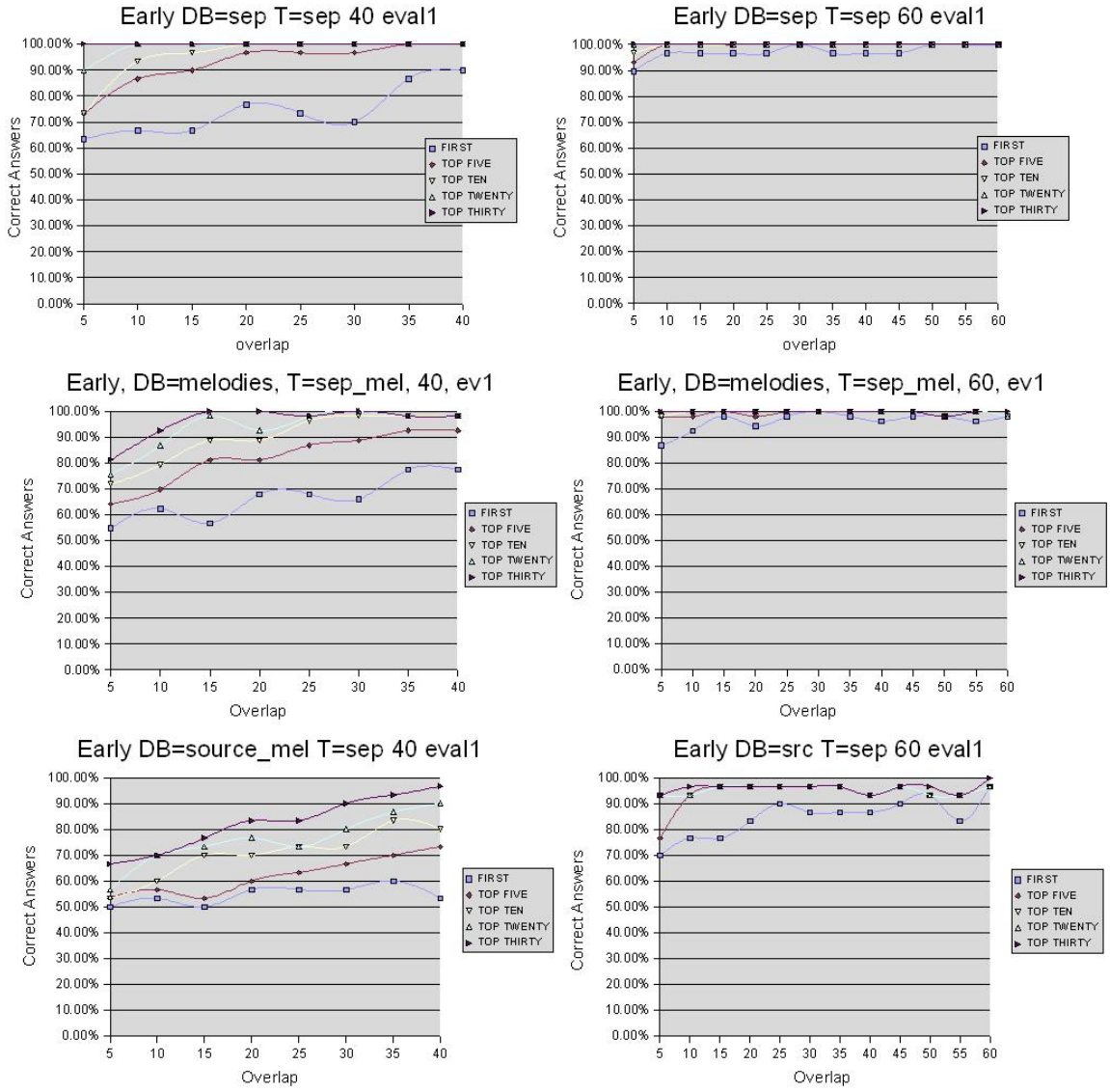

Figure 26: Some test results: success rate in function of the overlap for several length values and database configurations.

When sorting the results, if we meet two segments with the same score, we cannot rely on any other way but the order of analysis. That is probably why the results we see on figure 26 are not outstanding. If we want to avoid this situation, we had better avoid small lengths (such as 40 and even 60 samples) coupled with small overlaps. On the contrary if we have not so many segments, what is more if they are big enough, then the probability that we find another good match for it (except itself) [1]is very low. That is why, when testing the databases on the sep_melodies set, the best fit is often the correct target (thus a success rate of 100%).

But considering this result, does that mean that we should take the biggest segment possible, with the biggest overlap as well? We try to see further down what can be done and discuss about the best configurations.

2. **The number of average segments should be big enough to support the statistics in eval2:**

   Another important result concerns our way of evaluating. We talked about our system of evaluation, based first on the direct result for the segments, while the second possible evaluation is based on statistics run over the candidate songs (and not only just one segment). We were quite astonished to see that the second evaluation, eval2, did almost always get worse results, as we can see on figure 28. That was going against our general understanding of the problem.

   We think that the reason why the results are quite poor when the overlap increases is that there are less segments to compare. With less segments, the mean value that is used to represent the score of the candidate song is based on too few comparisons to be really representative. What is more, our first thought was to compute the mean value over the whole list, with at the beginning the first 30 best fitting segments. If there were less than 30 segments in the list (that is to say too few segments to compare from both the candidate song and the query), we completed the summation with the maximum value allowed in the program. This proved to be a bad idea, as the results showed us.

   When we compute the average score for one song, we include all of the segments, that is to say even the worst ones, if there is not so many segments. There can be cases, which do not seem to be so rare, according to our results, when the average on the target song gives a worse result than the average on another candidate song. For instance, for the correct target song, let us say that only one segment is really similar to the intended melody and that the rest just gives bad results. Let us assume that for some another candidate song, even if there is no segment that is as good as the one from the correct song, the segments in average have not so bad results , or at least, better than the other segments of the correct target song. A mean value would then create an unbalanced situation, a biased situation, where the target song has lower score than this “average” similar song.

3. **Behavior of the Application for the audio queries**

   We also tested the different databases on the audio queries we had recorded. The figure 29 shows the results we obtained for the database “melodies”. We wanted to check first that our algorithm was performing better than a random drawing (that is to say an algorithm that would just return randomly one of the songs of the database). We chose for this experiment

[1]Up to now, we are comparing the database to itself - sep_melodies is based on melodies extracted from the two other databases, so that every segment in it can potentially be present in any database

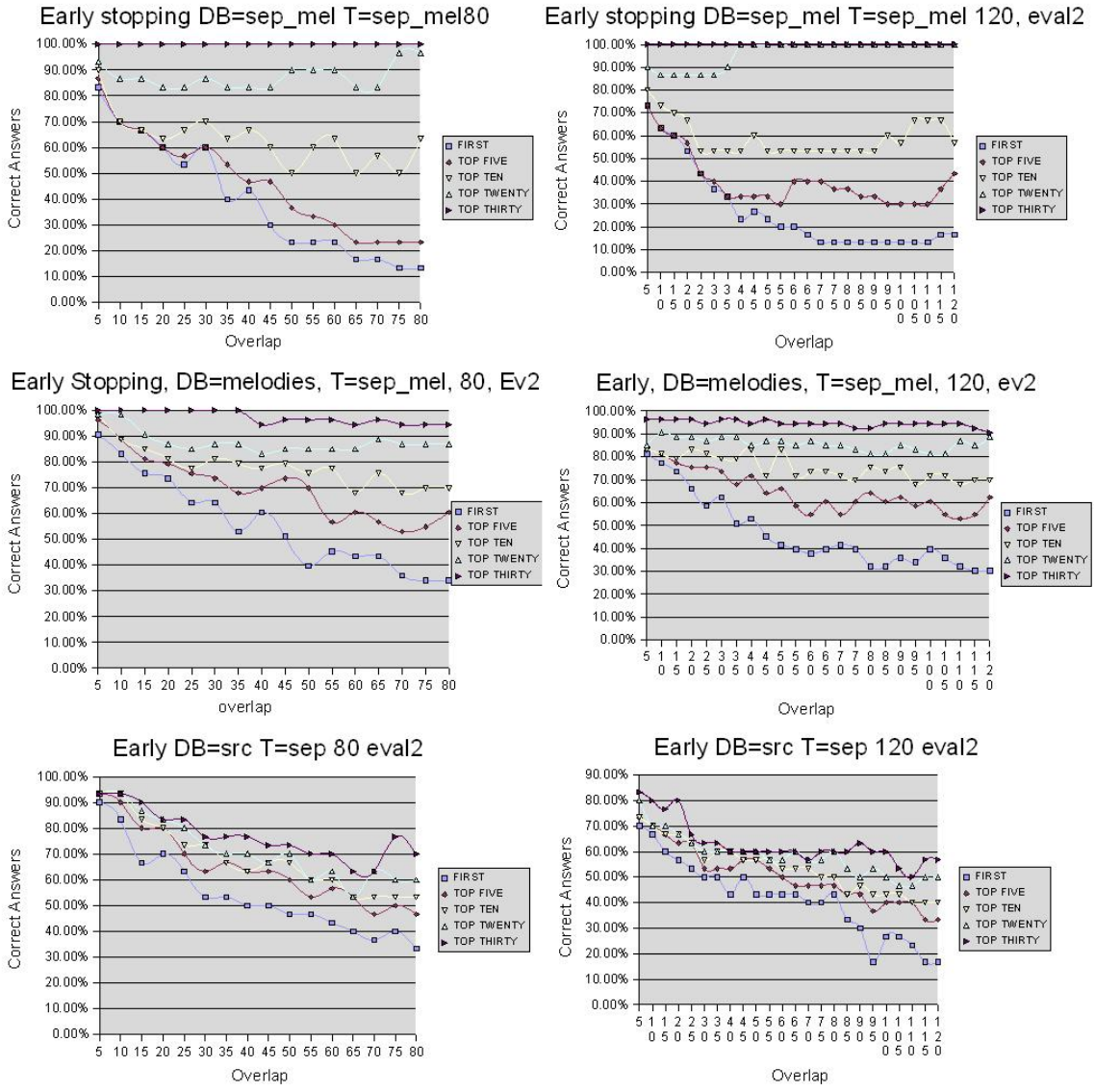

Figure 28: Some test results: success rate in function of the overlap for several length values and database configurations, evaluation form: eval2.

the configurations of length/overlap that were giving the best results for the top ten (correct answer in the first ten best matching segments).

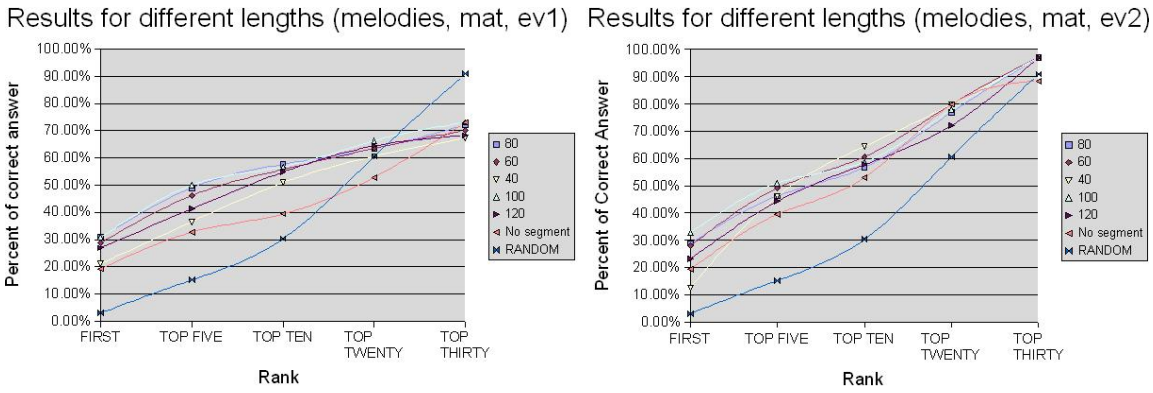

Figure 29: The success rates for eval1 and eval2, for different lengths. The database is "melodies" and the test files are "mat". For comparison, there are also the random probability of success as well as a pattern matching that does not make segments (ie each file has only one segment)

For eval1, the correct target was returned within the tenth rank in general in more than 50% of the cases. However, if we check in the top thirty (the first thirty best fitting segments), we see that it did not return as many correct answers as it should. Actually, with eval1, the program was allowed to return the same song several times, so that it is not so astonishing not to find the correct song in the list. For example, if we have one song for which the similarity is good for several segments, this song will appear several times, reducing the "chance" of obtaining the correct answer (if the first song was not the correct one).

We notice that eval2 performs slightly better, with a success rate between 55% and 65% for the top ten. For this purpose, we had also changed the way eval2 was computing the mean value, we tried several configurations such as evaluating it only over the segments that were actually put in the list (which is of course not very balanced, since this number is different for each song) or taking only the best match for each song (and therefore each song would at most appear once in the list). The first evaluation technique seemed better, so we chose it.

As concerns the evaluating system, we also tried to compute an evaluation that would take the best match for each segment of the query, and make a list of these results, for each segment. We compute the average rank obtained by each song and return this list. However, we did not have time to seriously test this evaluating system, and the first results we obtained are still at most as good as what we already obtained. However, we believe this evaluation technique would worth more attention and adjustments: the average ranking might not be the most relevant similarity measure there, for instance.

It might ne interesting to test other lengths, bigger than the ones we tried, in order to check if we cannot find even better configurations. However, seeing these first results, the length of the segments does not seem to be a critical element for our program. We limited it to a value around 100 samples, because it corresponded to 10 seconds. It would probably be more important to adjust

this parameter in a case where the length of the segments is related to the rhythms themselves. We will discuss about this later in the section 6.3. What is more, since the query, coming from raw audio data, and the songs of the database are by essence different, it might be interesting to try systems that include a hybrid segmentation: segment the database, so that we have the most possible melodies we can, and take the query as one segment, because it is more likely to be one melody (and not a "collection" of melodies as a song could be).

6.2 The Choice of the Database

We were also wondering in what extend changing the database would affect the result. Taking the results shown on figures 26 and 28, we can draw the following conclusions. Figure 30 shows the results for some of the best configurations we could find.

1. **The chosen databases do not seem to have many redundancies within them:**

   As we can see on the figure 26, for eval1 and even for quite small length, that is to say quite a lot of segments, the result is still acceptable. This means that the probability, given a segment to identify, to find a segment other than the original and for which the similarity is better than this original is rather small. This also means that the redundancy within the databases is low. Of course, as we said, we had better avoid lengths that are too small.

   What is more, we know that the files in source_melodies are longer than the ones in the other databases, which leads to more segments per files, thus, according to the conclusions from the last part, to more possible redundancies. In this case the use of eval2 becomes mandatory and according to our data, it is more interesting to consider eval2 for segments that are 40 sample-long or 60-sample-long, with overlaps lower than 20 samples.

2. **Which database achieves the best results?**

   Considering the figure 30, it seems quite obvious that the using the first database, **sep_melodies**, with the eval2 evaluation leads to the best results, between 60% and 70%. The overall best result is found for the following configuration:

   - Segment length: 80;
   - Overlap: 5;
   - Database: sep_melodies;
   - Evaluation: eval2;
   - Representation (pitch and duration): global pitch and no normalization for the durations.

   Up to now, this configuration is the best we had. However, we still have to test the representation issues and be careful of the computation time parameter.

6.3 Representation of the sequence: Global and Relative Pitches and Durations

1. **Relative pitch representation**

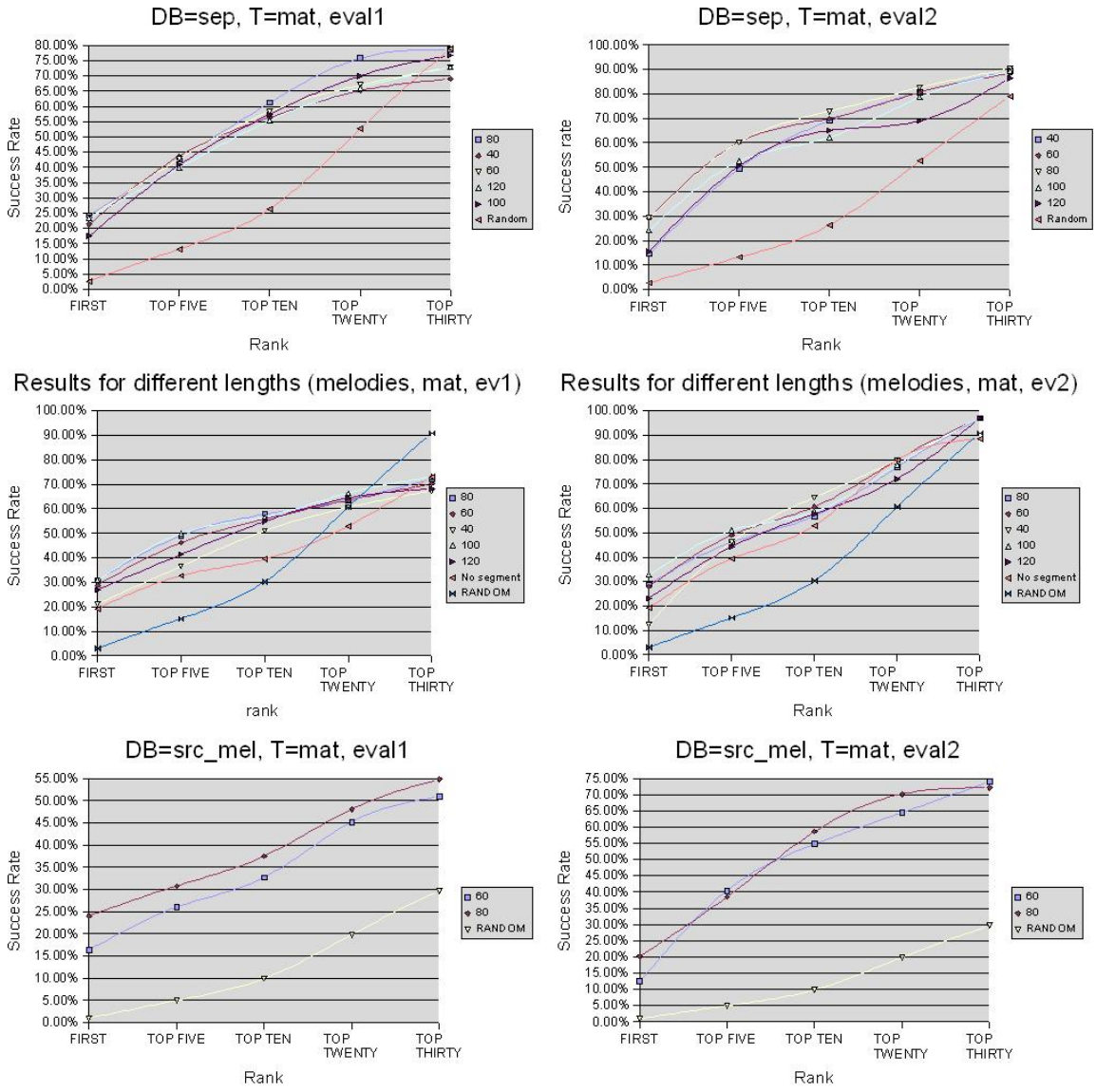

Figure 30: Success rates, probability of presence as first choice, in the top five, ten, twenty and thirty for several databases and segment lengths.

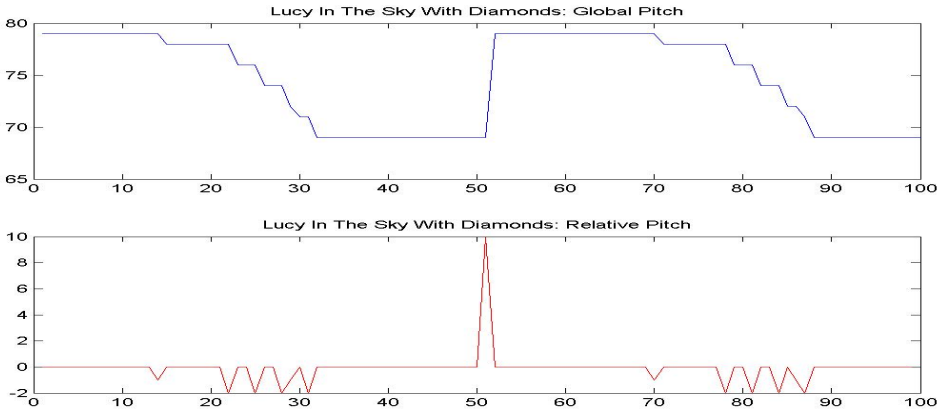

Figure 31: Global and Relative Pitch representation for "Lucy In The Sky With Diamonds".

Instead of computing the similarity distance directly on the given pitches, we compare the successive tone differences, as introduced in section 4 as the "relative pitch representation" (see figure 31). We did some tests there, but the current state of our algorithm does not show any significant difference between the global and relative pitch representation methods.

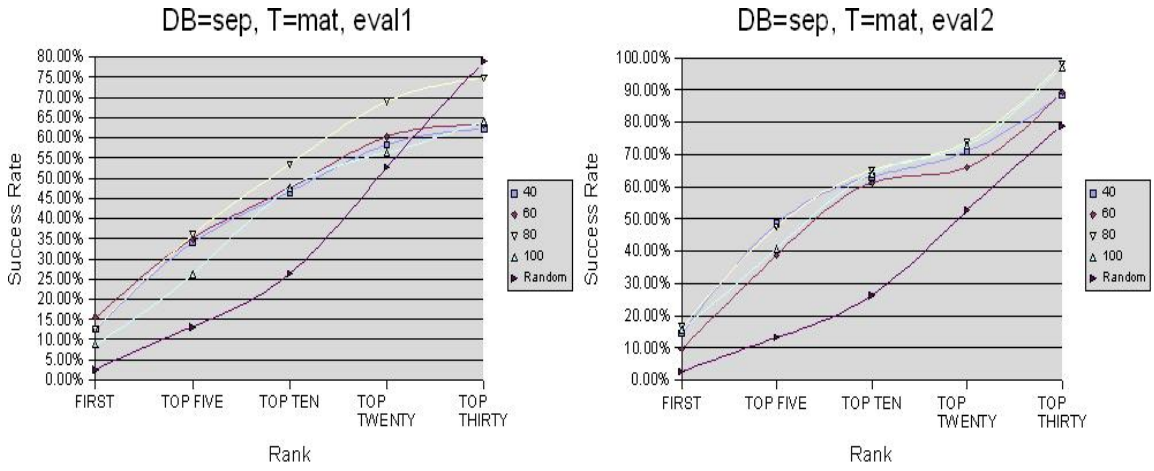

Figure 32: Results of the QBH system with a relative pitch representation.

The tests we did were held for lengths from 40 to 100 samples for each segments, the database was sep_melodies and the queries were the pitch series extracted from the audio queries. This time, the database had 38 files. As we can see on figure 32, we obtain more or less the same result as on figure 30. The evaluation eval2 obtains once again a success rate of 60% of correct

target in the first ten answers. It means that, at least, we can still hope to improve the results with this technique.

The fact that the result is not so different might have another explanation. A singer can make mistakes, for example going out of key, or changing its reference (going from the global reference to some relative reference). However, we saw earlier two strategies to get rid of this problem. The first one was called "global" pitch strategy. With short segments at hand, it would be more exact to say "locally global" pitch, since we shift the segments so that they have the same starting note, only for this instant, each segment is being processed independently. On the other hand, the "relative" pitch strategy proposes to change the reference from a note to another. However, we assume that a singer usually does not change its pitch reference every two note, so that locally, he actually uses a sort of global reference. For this reason, with rather small segments, the two strategies would actually almost give the same result. What is more, our pitch series extracting algorithm does already have a "relative pitch" way of doing its task.

2. **Relative Duration representation**

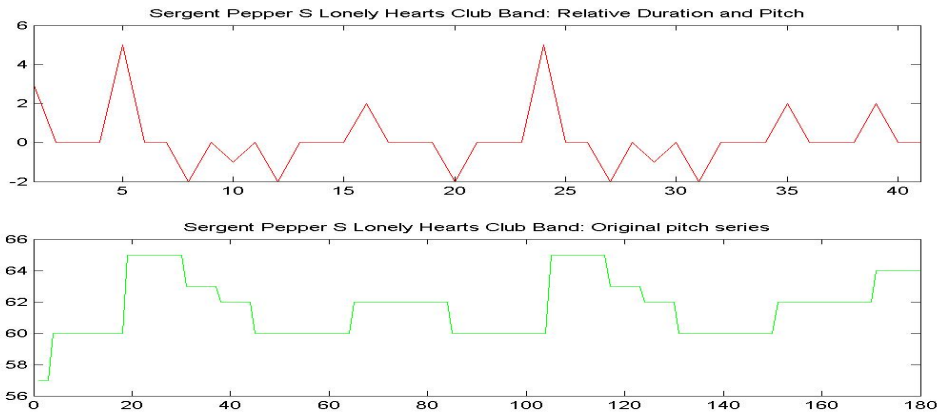

Figure 33: Results of the QBH system with a relative pitch representation.

The figure 33 represents our last try to improve the success rate of our system. We did not test this technique enough to adjust it, so that the results might not be optimal. We followed once again the authors of SoundCompass ([6]) and tried a normalization solution. We took, for each segment, the median duration value and used it as the time unit (or rhythm unit). We then normalize all of the durations by dividing each of them by the time unit. At last, we quantize them to stay within "normal" rhythms. Here, we assumed the time unit could be the duration of one eighth note (this result should actually be verified by doing some statistics over the database). Let us say quarter notes get a normalized duration of 1, then the time unit should be $\frac{1}{2}$ and so on.

Hopefully this method could also help reducing the size of the songs, and reduce the number

of segments, thus reducing the computing time. However, we did not notice a big difference between the database before and after the transformation.

The results we obtained are also slightly under the ones we obtained without the rhythm normalization. Actually, one should keep in mind that, since we are using a DTW algorithm, the time parameter is quite special. We need a way to penalize the segments if the durations do not match. The DTW algorithm, in its usual form in fact accepts time warping, without distinction of knowing whether the rhythms inside the melody are the same. Basically, our first implementation of the DTW was not taking into account the duration parameter. Then, in the end, we tried To put some rules, such as adding 1 to the result in the DTW process when the best choice for the preceding sample was not on the first diagonal. We are this way compensating errors such as omissions of notes or addition of note. Considering the figure 22 from the section 5, it means that, computing the similarity for the couple (i, j), we do not penalize it if the preceding sample is $(i-1, j-1)$. For any other settings, there should be some way to measure it. We decided to add 1 each time it was not in the above situation. However, up to now the results do not seem to show any significant improvement.

6.4 Speed Issues

At last we talk about the speed issues, and about the computing time.

1. **The Early Stopping Algorithm beats the simple DTW Algorithm**

   Each time we ran the program we kept some data giving the computing time. The figures on 34 show how long some of our test lasted on our computer (processor 1.60 GHz).

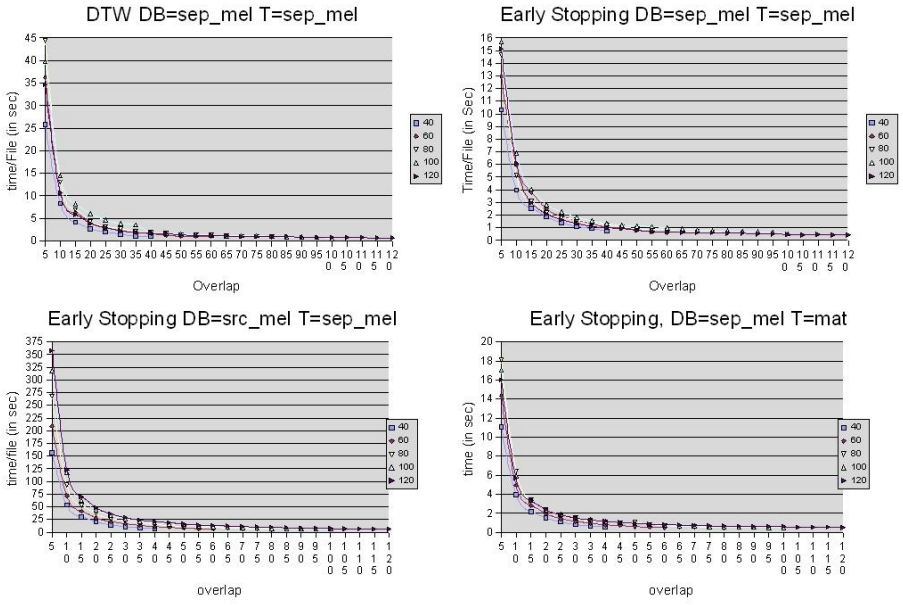

Figure 34: Computation times for several test configurations.

As expected, the Early Stopping algorithm helps to decrease the computing time. What is more, when we compared the success rates obtained by the two algorithms, we found there was almost no difference. That is why along our study, we essentially used the Early Stopping Algorithm. As we can see, for the database sep_melodies, we can obtain the answer in reasonable time (around 10 seconds). Of course, it would be better to fall under one second of processing in order to make a system that would fully satisfy a user.

This figure also highlights the biggest problem we have with the database. In fact, the larger the database, the longer it will take to get an answer. But as we can see here, for the database source_melodies, with 148 files, we need between 1 and 5 minutes to get the answer! This would tend to prohibit the use of this database in this application. However, the principle of QBH system is to cover a maximum of possible songs, so that finding a fastest way to measure melody similarity would be appreciated.

2. **An FTW Algorithm that does not seem to fit our goal:**

   In an early stage, we tried the FTW algorithm. However, we found out, after a few tests that the results were quite bad. Due to some errors in the implementation, the results were not as general in the first iteration as they should have been. Since we did not have enough time to correct these mistakes, we could not go on testing and improving this algorithm.

   Although the system did not return good results, we were still able to identify some problems with the algorithm, more fundamental ones. First, we noticed that with our implementation of the algorithm, the program was running slower than with a normal DTW. Actually, we believe it was taking more time, and it would if we were implementing it again, because, as explained in [11], the sequences have to be quite big for the algorithm to be efficient (around 2048 according to the paper). The segments we have here are at most 120 big. This, even with the FTW way of cutting the sequences, making coarser fragments to be compared and so on, is not big enough to be able to feel the difference. Reducing the dimension of the sequence is meaningful only with large sequences. Let n be the size of the sequence. If we propose a reduction by 2 of this sequence, it leads only to a reduction by 4 of the operations. However, in [11], the authors first reduce the data sequences by 128, thus a reduction by $128^2 = 16384$ of the operations! That is why, instead of speeding up our program, that algorithm was slowing it down, because it was as if we were doing the DTW several times, instead of just once.

7 Conclusion

As a conclusion we can say that, even if our system is not the best one, it can still retrieve more than half of the queries that are made, within the tenth rank , for our best configuration, it can reach 70% in the top ten. The segment length and overlap do not seem to have a big influence on the result, except that smaller segments leads to slightly worse results while smaller overlaps have a tendency to slow down the system.

The database are also not perfect, since they were essentially manually made, out of MIDI files found on the Internet . Maybe the fact that some songs had MIDI files bigger than others have led to biased results. A small database, very specialized such as sep_melodies is enough to retrieve the queries, however, it requires a lot of energy to build, because it has to be done manually. At our knowledge, there is still no reliable melody extraction algorithm that could tell which track in a MIDI file should be the melody, and which one is not.

At last, the representation matter has still to be tested. Especially since the evaluation form and the similarity measure have to be designed to give better results. We have said that the DTW was not allowing us to measure the distortion in the durations, which actually is critical in our application. What the system up to now does is to check and find the sequence of pitch that is similar to the query series, almost without considering the length of the notes. We believe that there should be some thinking about this matter before going on designing this project. The DTW pattern matching can still be used, but probably in another way. But we could also imagine another similarity measure that takes into account the pitch *and* the duration information. This measure can potentially include the relative pitch and duration theories explained before, the previous tests did not determine whether it was critical or not to use them, but the musical sensitivity and common sense would certainly lead to such a solution.

A Query-By-Humming system is a big system, because it has to deal with at least two major problems. First it has to interpret the query and transcript it into some symbolic representation, which is not as easy as it seems. Second, the pattern matching and the similarity measure to apply has to be designed so as to give the best results, in less time as possible. It is hard to carry on such a project but we believe that in the near future, such technologies will be implemented into everyday devices such as mobile phones or mp3 players. The devices are ready, but the algorithms, even the state-of-the-arts ones (SoundCompass, CubyHum, RePReD, MUSART, and so on), still need to affine their results. A worthwhile question has to be raised: is it actually possible to do better than what we do now? Considering what the authors in [7] say, the task might not be as simple as it seemed in the beginning. Can a computer know music better than a human?

8 Acknowledgements

I would like to thank my teacher XU MingXing for his help and advise, and for allowing me to make these researches on a subject I was really interested in.

I also would like to thank my fellow Chinese students LIN Song and ZHANG KaiXu who, although they had to attend to other lessons and were already very busy, kindly helped me to lead this project, taking part of the discussions about the techniques and helping a lot with the programming part.

At last, many thanks my friend Gwen for lending me his voice and allowing me to carry on my tests on real audio queries.

9 Annexes

We describe in this annexe some tools we had to use. They are in a sense out of the study, however their understanding puts a lot of constraints in the results of our project. For example, we decided to take MIDI files as raw material for our database, so that we had to find a way to extract the data out of these. What is more, we needed a program that could write a MIDI file (simple one), in order to understand more easily what was happening in the pitch detection system. We could see curves, of course, but a "listenable" result was helpful. What is more, it could provide a feedback to the user (so that he could know what the program understood from his query).

9.1 Midi file Reader (in Java)

We need to be able to read midi files, so that we can build our database. The specifications here are that the program should be able to extract the sequence of notes out of a midi files. Since we do not really need any other information, we limited our program to extract the melodies from each track of the file, thus obtaining the sequence of notes. This sequence is formatted as follows: for each note, we first give its pitch, and then give its duration in seconds. For example:

```
69      0.25
74      0.5
78      0.5
74      0.5
69      1.0
...     ...
```

To do so, we decided to use Java since it has an integrated package that deals with midi files: `javax.sound.midi`. Thanks to the functions found in that package, we can easily find these pieces of information. The classes and functions we used are listed below:

```
class MidiSystem
    Methods:
        static Sequence getSequence(File file)
            obtains a MIDI sequence form the specified File
class Sequence
    Field:
        protected Vector tracks
    Methods:
        Track[] getTracks()
            obtains an array containing all the tracks in this sequence
public class Track
    Field:
        protected vector events
    Methods:
        MidiEvent get(int index)
            obtains the event at the specified index
        int size()
            obtains the number of events in the track
public class MidiEvent
```

```
    Methods:
        MidiMessage getMessage()
            obtains the MIDI message contained in the event
        long getTick()
            obtains the time-stamp for the event, in MIDI ticks
class MidiMessage
    Field:
        protected byte[] data
        protected int length
    Methods:
        byte[] getMessage()
            obtains the MIDI message data
class ShortMessage
    Field:
        static int NOTE_ON
            Command value for Note On Message (0x90, or 144)
```

We still have to explain how, from the time-stamps we get from the function `getTick()`, we can obtain the duration in seconds. The returned value for `getTick()` is expressed in MID ticks. We actually decided, as first try, to take a tempo, 100 ppqm (pulses per quarter note and per minute), from which we would compute the duration in seconds of the note. The formula we used is:

$$\Delta t = \frac{(e.getTick() - last) * 60}{tempo * 18 * 8}$$

where $\frac{tempo}{60}$ is the tempo in ppqs (pulses per quarter note and per second), the denominator $18 * 8$ corresponds to a general case in MIDI files, for which each quarter note has $18 * 8$ ticks. We could improve the exactness of the extraction of the MIDI files by determining all of these values directly from the files (and not by default). However, because of a lack of time, we did not try to improve this aspect.

9.2 Midi file Writer (in C++)

This part of the program is not necessary. However, as seen in some works, a feedback for the user to know how the program transcribed what he was singing. We wrote a simple "WAV-to-MIDI" program. Actually, what we wanted to do was to obtain a basic MIDI file out of the sequence of pitches that the pitch detection algorithm returns. We first explain some basic knowledge about the MIDI file format and then give our solution to the problem.

9.2.1 The MIDI file format

We based our knowledge essentially on web-sites such as [15] or [16] explaining this format and on the direct observation of MIDI files (thanks to a hexadecimal viewer). We understood, to a certain extend, how the MIDI files were formed and how we could write a simple one when we only have the information corresponding to the pitches and their durations of the notes.

To make it easy, let us say a MIDI file is composed of a first chunk, the header chunk (MThd), followed by other chunks, Track Data chunks (MTrk). The first one helps programs dealing with MIDI files to acknowledge if the given file is or not a MIDI file. The Mtrk is used to store the song

data itself as well as some other details (meta-events) about the song such as tempo indications, time signature, or none musical events such as track names, lyrics, and so on. Usually, you can find some general meta-events in a first Mtrk chunk, while the musical data are stored in another MTrk (one MTrk per Track).

- *The Header chunk (MThd)*:

  The header chunk should be always constituted by 14 bytes. These are (the values are the hexadecimal values):

```
4D  54  68  64  00  00  00  06   ff  ff       tt  tt        dd  dd
M   T   h   d       (length)     (format)(number of Tracks)(Division)
```

  The first four bytes are the signature for a MIDI file: it identifies the file as one of them. the four following ones represent the length of this chunk: 6 bytes.

  The next two bytes define the format for the MIDI file, ie the kind of MIDI file it is. At this point, it only tells us if the MIDI file has one Track (00 01), one or more simultaneous tracks (01 01) or if it has several tracks, all of which independant from each other (00 02). The first format fits the case where there is only one instrument playing (our case). The second one is used when there are more than one instrument and the last format can be used, for example, to build a compilation of songs.

  The `tt tt` bytes tell the number of Tracks for the MIDI file and the `dd dd` bytes give the "PPQN", the "pulses per quarter note", which is also the resolution the time-stamps are based on. It can be seen as the "division of a quarter note represented by the delta-times in the file" (cf. http://jedi.ks.uiuc.edu/ johns/links/music/midifile.html [15]). For example, a value of 96 here would mean that, later, when expressing the delta-times between the events (that is to say the time between two events), each delta-time period of 96 corresponds to a quarter-note. At the same time, there is a possibility of expressing this concept another way. You can find the explanation in the webpage noted above. To put it in a nut shell: the first byte is used to precise the number of frames per seconds and the second byte to precise the resolution within one frame. Since this is not the objective of this study, we will not develop more this subject. We used the first version. We see in the next section how we can compute the delta-times out of the times in the input file thanks to the `dd dd`.

  Finally, we opted for the below header chunk (hexadecimal values):

```
4D  54  68  64  00  00  00  06   00 01       00 02          00 78
M   T   h   d       (length)     (format)(number of Tracks)(Division)
```

- *The first Data Track Chunk: general features*:

  When creating a format 1 MIDI file, it is convenient (and recommanded) to make a first Data Track containing the features for the musical data, including the tempo and the time signature. For this kind of events, the MIDI file code is as follows:

  1. Tempo event: `FF 51 03 tt tt tt`, where `tt tt tt` is the tempo expressed in microseconds per quarter note. If there is not that indication in the MIDI file, the tempo is assumed to be at 120 bpm.

We have the durations of the notes in seconds, so we can use the following formula to obtain the number of MIDI clocks (because the delta-times are expressed in clocks) required for that duration:

$$\text{delta-time(clocks)} \quad = \quad \text{time(s)} * \frac{\text{dd dd}}{\text{tt tt tt}} \tag{10}$$

At last, we chose a 100 bpm tempo for every file. Actually, it will not change the result, since we are only considering the time and not the possible rhythms that are involved. If we had to make a "real" WAV-to-MIDI aplication, we would have to be more careful on that point. 100 bpm corresponds to $600000\mu s$ for one quarter note, $0x09, 0x27, 0xC0$ in hexadecimal: `FF 51 03 09 27 C0`

2. Time Signature event: `FF 58 04 nn dd cc bb`, where `nn` and `dd` represent the time signature, `cc` is the number of clocks per metronome click and `bb` is the number of 32^{th} notes in a MIDI quarter note.

   The time signature in music is expressed by a fraction, the numerator is the number of units in one bar and the denominator defines this unit. For example, a $\frac{4}{4}$ means 4 quarter notes in one bar. $\frac{6}{8}$ means 6 eighth notes in one bar. `nn` is that numerator, while `dd` is the power of two corresponding to the denominator. Thus, for the two preceding examples, it would be, respectively: `04 02` and `06 03`.

   As for the `bb` part, it allows the creator of the MIDI file to identify any value as a MIDI quarter note.

   In the end, we chose $\frac{4}{4}$ as time signature and usual values for the rest: `FF 58 04 04 02 18 08`

- *The second Data Track Chunk: musical data*:

  The main difficulty here is to compute the delta-times, especially since the MIDI file format allows them to have variable sizes. We found on the Internet some ressources that helped us to compute them. We limit our study to a very simple form of music data track: the data chunk is then composed by a header, the length (in bytes, on 4 bytes) of the track and the data events.

  ```
  M   T   r   k   ll  ll  ll  ll  xx  xx  xx...
  ```

  We only chose two data events, the basic ones: NOTE_ON and NOTE_OFF. We chose the following code for each:

  ```
  NOTE_ON:    tt tt tt tt 90 ff 64
  NOTE_OFF:   tt tt tt tt 90 ff 00
  ```

  Before each event, one have to precise the delta-time when it occurs, that is to say the time (in clock) elapsed after the previous event: `tt tt tt tt` (we explain later how the variable length works). Actually, since we just have notes and their durations, and since our pitch detection algorithm does not detect the silences, we decided that for each note, there would be two events, NOTE_ON and NOTE_OFF, the first event happening directly after the previous event, delta-time is set as 0 and the second event is happening after the NOTE_ON, with a delta-time corresponding to the duration of the note. The `ff` section stands for the MIDI value of the frequency.

The delta-time computing is really important, though complicated. It is expressed at most by 4 bytes, the last byte has a 0 as the biggest bit, while the other ones have 1 as biggest bit. To find the value, one have to read all the bytes without the biggest bit of each byte and by concatenating the bits together. Thus the maximum authorized delta-time is `0F FF FF FF`, corresponding to the delta-time `FF FF FF 7F`. The following C routine can write a value as a variable length delta-time:

```
register unsigned long WriteVarLen(register unsigned long value)
{
    int count =1, i=1;
    register unsigned long buffer;
    buffer = value & 0x7F;
    while ( (value >>= 7) )
    {
        buffer <<= 8;
        buffer |= ((value & 0x7F) | 0x80);
        count++;
    }
       return buffer;
}
```

and to write it in the sequence of bytes of the MIDI file:

```
while (1)
{
    DataMTrk2[counterSizeTrack] = (unsignedchar)(varLen);
    counterSizeTrack += 1;
    if (varLen & 0x80)
        varLen >>= 8;
    else
        break;
}
```

9.2.2 Our "WAV-to-MIDI" program

You can ask for the source code by asking us at jean-louis.durrieu@m4tp.org. The outline of our program is:

1. Using MATLAB, extract the time series corresponding to the raw audio query. The output we are expecting has the form explained in 2.
2. Running the C/C++ program that takes as input the obtained file and transforms it into a MIDI file.

The MIDI file returned is rather simple, but some improvements are still possible, especially about the tempo and the time signature wanted. A pre-processing is also possible, following or example the rhythm detection (quantization) introduced for the SoundCompass system ([6]). We give further in this section an overview of a tempo tracking technique which can also be used to

adapt the tempo for the MIDI file (actually, it would give a better result to take a constant tempo through the MIDI file, and adapting the durations of the notes according to the tempi detected).

9.3 WAV file Reader (in C++)

In order to write a program that could handle most of the tasks in our study, we wanted to make it possible for the C/C++ program to process the WAV file. The WAV format is quite hard to implement and read. We based our study on the following web-page [17]:
http://www.sonicspot.com/guide/wavefiles.html.

As concerns the organization for a basic WAV file is:

Chunk ID "RIFF"
Chunk Data Size
RIFF type ID "WAVE"

Chunk ID "fmt " (note that there is a space character in the end of "fmt ")
Chunk Data Size (16 plus extra format bytes)

Size (bytes)	Content
2	Compression code
2	Number of Channels
4	Sample Rate
4	Average Bytes per Second
2	Block Alignement
2	Significant Bits per Sample
2	Extra Format Bytes
...	Extra Format Bytes

Chunk ID "data "
Chunk Data Size
Digital Audio Samples

Most of the above fields are explained on [17]. Some particular fields are essential to be able to read efficiently the WAV file, such as the Sampling Rate Fs, important for us so that we can retrieve the pitch in Hertz (without Fs we would only know the pitch - or period - in terms of samples). The significant bits per sample is also fundamental, for it gives the information of how we have to read the audio samples. The usual values for this field are 8, 16, 24 or 32 bits per sample (bps).

Some remarks, concerning the WAV format: as concerns the audio data samples, if the number of channels is bigger than 1, then the data samples are interlaced. That is to say, if we have two channels, the first data samples encountered corresponds to the "left" channel, the second one corresponds to the "right", both for the same time, let us say time 0. The next sample is for the "left" channel, for the next time stamp, time 1, and then the "right" channel and so on...

Because it was set a while ago, the WAV format uses the little endian convention to represent the bytes, which means that the first byte for a value is actually the lowest byte. In practice, it means that, in order to read the values in the WAV file, we have to "flip" all of the byte for one value.

Another "funny" thing about the WAV format is linked to the significant bits per sample. For 8 bits, the conversion is quite easy, each byte has to be read as an unsigned char and you obtain the value of the wave for the corresponding time. However, above 8 bps, it becomes more complicated. For example, for a bps of 12, the figure 35 gives an insight of what has to be read. The values on more than 8 bits are to be read as signed values.

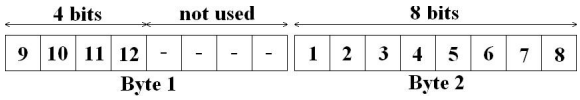

Figure 35: How to read a data sample in a WAV file. To reconstruct the right value, here encoded on 12 bits, we have to put the byte 2 before the byte 1 and cut the last 4 bits.

Thereafter, we wrote a program which first verifies that the input file is a WAV file, checking for every header possible, then checks that we have a non-compressed WAV file, and at last, take the values of the wave time series. The program returns then the time series as well as the sampling rate, which, as we said, has to be known for restitution issues and processing issues.

9.4 Rhythm and Tempo Recognition: Sequential Monte Carlo Algorithms

We will not develop this technique here, since it is not the purpose of this study. What is more, the technique itself is quite complicated. We will just highlight the model, which we think seems a good one for Tempo Tracking applications as well as Rhythm Detection applications.

Of course, those two kind of applications are linked, since the production of a note is conditioned by the Tempo of the song and by its rhythm. Such that what the audience hears is the mixture of the two concepts. That is what the authors in [2] have modeled.

The model they chose is rather complex and allows us to write the problem of tempo tracking into a "switching state space model" (or "conditionally linear dynamical system", etc.). We can see on the figure 36 they used in their paper the model and the different variables that they used.

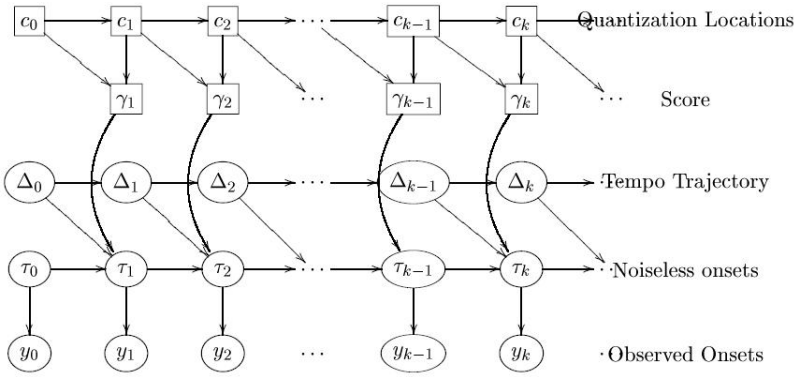

Figure 36: Model for the Tempo and Rhythm tracking algorithm. The lower layer represents the observations while the upper layer are the "quantization locations".

We can enumerate the different variables and see their role:

- y_k

  The y_k are the observed onsets. When a user is playing, we assume that we are able to know exactly when he plays a new note. These values are the only observable ones. All the other variables are hidden. Let us assume the y_k are in seconds.

- c_k

  A c_k represents the location of a note on a musical score. It is quantized, and corresponds to the time when the note appears. The unit is, for example, one metronome click.

- γ_k

  The γ_k are the actual duration of the notes on the score . To be more precise, they are the inter-note intervals, still in metronome clicks. They linked to the c_k by the relation:

  $$\gamma_k = c_k - c_{k-1}$$

- Δ_k

  $\Delta_{0:K}$ is the Tempo trajectory. The tempo is usually expressed in clicks per minute (or beats per minutes, bpm), or clicks per seconds (bps). We assume that the tempo is slowly varying, that is to say:

  $$\Delta_k = \Delta_{k-1} + \zeta_{\Delta_k}$$

  Where the ζ_{Δ_k} are independent and identically distributed with $\mathcal{N}(0, Q_\Delta)$. This is actually a first order Gauss-Markov process.

- τ_k

  At last, the τ_k are the "real" or "intended" timings for the notes. They are conditioned by the tempo Δ_{k-1} and the rhythm γ_k. In fact, we assume that, from the sample $k-1$ to the

sample k, the tempo is Δ_{k-1}. The duration of the current note is γ_k (thanks to the above definition). The next note, at sample k, appears when the time is equal to τ_k:

$$\tau_k = \tau_{k-1} + \gamma_k \Delta_{k-1} + \zeta_{\tau_k}$$

The authors modeled this value with $\zeta_{\tau_k} \sim \mathcal{N}(0, Q_\tau)$, which represents the expressiveness of the note and the freedom given to the performer. y_k is the observation, such that we can assume that it is slightly more noisy than τ_k:

$$y_k = \tau_k + \epsilon_k$$

As we can see with these variables, we can build a markov process that models the tempo tracking problem. Let $z_k = (\tau_k, \Delta_k)^T$ and $\zeta_k = (\zeta_{\tau_k}, \zeta_{\Delta_k})^T$. We can rewrite the above equations:

$$z_k = \begin{pmatrix} 1 & \gamma_k \\ 0 & 1 \end{pmatrix} z_{k-1} + \zeta_k$$

The resolution of the problem of quantization (or rhythm transcription) can then be seen as a "MAP state estimation problem":

$$\gamma^*_{1:K} = argmax_{\gamma_{1:K}}\, p(\gamma_{1:K}|y_{0:K})$$

$$p(\gamma_{1:K}|y_{0:K}) = \int dz_{0:K}\, p(\gamma_{1:K}, z_{0:K}|y_{0:K})$$

and for the tempo tracking problem, it can be seen as a filtering problem:

$$z^*_k = argmax_{z_k} \sum_{\gamma_{1:K}} p(\gamma_{1:K}, z_k|y_{0:K})$$

The calculus and the algorithm that can be used to estimate the different probabilities in the above equations are explained in [2]. The techniques used are essentially based on the Monte Carlo Simulation. This technique allows to simulate a process for which we do not know the exact distribution. By generating samples thanks to a given distribution of probability and then refining the result (by resampling, that is to say keep the best samples) thanks to the observations (which provide information about the true probabilities).

We believe this model can produce good results, because it seems very close to reality and can model a lot of parameters (especially the expressiveness). We think this algorithm would worth a try in future works, in particular for "WAV-to-MIDI" applications and researches. This method will however probably not fit our QBH project, because it needs some computational time, which is a resource we need for other parts in our application.

References

[1] Nancy Bertin, Indexation Scalable des Documents Sonores, Université Pierre et Marie Curie, 2005.

[2] Ali Taylan Cemgil, Bert Kappen, Monte Carlo Methods for Tempo Tracking and Rhythm Quantization, in *Journal of Artificial Intelligence Research 18*, pages 45 - 81, January 2003.

[3] Roger B. Dannenberg, Ning Hu, Understanding Search Performance In Quer-By-Humming Systems, 2004 Universitat Pompeu Fabra.

[4] David C. De Roure, Steven G. Blackburn, Content based navigation of music using melodic pitch contours, 1998.

[5] Richard L. Kline, Ephraim P. Glinert, Approximative Matching Algorithms for Music Information Retrieval Using Vocal Input, ACM-MM'03 2003.

[6] Naoko Kosugi, Yasushi Sakurai, Masashi Morimoto, SoundCompass: A Practical Query-By-Humming System, in ACM SIGMOD, June 2004.

[7] Bryan Pardo, William P. Birmingham, Query by Humming: How good can it get?, university of Michigan, 2002 (?).

[8] Steffen Pauws, CubyHum: A Fully Operational Query by Humming System, 2002 IRCAM . Centre Pompidou

[9] Gael Richard, Traitement de la Parole, Brique PAMU, Module PAROL, ENST Télécom Paris, 2003-2004.

[10] PAMU module, lesson transcript MSA 2003, ENST Télécom Paris (http://www.telecomparis.com).

[11] Yasushi Sakurai, Masatoshi Yoshikawa, Christos Faloutsos, FTW: Fast Similarity Search under the Time Warping Distance, in ACM PODS, June 2005.

[12] Rainer Typke, "MIR systems" (Music Information Retrieval) website, http://mirsystems.info/index.php?id=mirsystems

[13] Yunyue Zhu, Dennis Shasha, Warping Indexes with Envelope Transforms for Query by Humming, ACM-SIGMOD 2003.

[14] Yunyue Zhu, Dennis Shasha, Xiaojian Zhao, Query by Humming in Action with its Technology Revealed, ACM-SIGMOD 2003.

[15] The MIDI File Format Spec (http://jedi.ks.uiuc.edu/ johns/links/music/midifile.html), John E. Stone, (or The MIDI File Format Spec - http://jedi.ks.uiuc.edu/ johns/links/music/midifile.htm).

[16] MIDI File Format (http://www.sonicspot.com/guide/midifiles.html), The Sonic Spot.

[17] Wave File Format (http://www.sonicspot.com/guide/wavefiles.html), The Sonic Spot.