



Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

János Balázs

**BEAT DETECTION AND
CORRECTION FOR DJING
APPLICATIONS**

M.SC. THESIS

SUPERVISOR
Dr. Balázs Bank

BUDAPEST, 2013

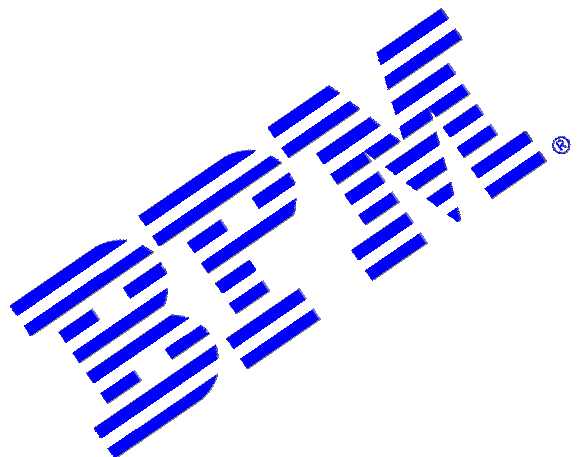


Table of Contents

Összefoglaló	5
Abstract.....	6
1 Introduction.....	7
2 Tempo detection	10
2.1 The Beat This algorithm	10
2.2 Tempo detection module of my project	13
2.3 Performance of the tempo detection module	19
2.3.1 Comparing with Beat This	19
2.3.2 Quantitative verification of the tempo detection module	20
3 Beat detection	22
3.1 Beat detection of Beat This.....	22
3.2 Statistical streaming beat detection.....	22
3.3 Beat detection of electronic music	23
3.4 Beat detection of acoustic recordings	27
3.4.1 Beat detection of a drum pattern.....	27
3.4.2 Beat detection of live recordings	32
4 Pre-processing for beat correction	37
5 Time stretching	39
5.1 Resampling	39
5.2 Phase vocoder	41
5.3 The Synchronous Overlap and Add (SOLA) algorithm	43
5.3.1 Key features of the SOLA algorithm.....	43
5.3.2 SOLA algorithm for stereo signals	47
6 Implementation in C/C++	48
6.1 Mex files	49
6.2 Developing the C/C++ functions	50
6.2.1 Decimation.....	50
6.2.2 Comb filter.....	51
6.2.3 Computation time comparison.....	54
7 Graphical User Interface.....	55

8 Results and possibilities of development.....	60
8.1 Listening to the outputs of the program.....	60
8.2 Summary.....	61
8.3 Possibilities of development.....	63
9 Acknowledgment.....	64
10 Readme.txt file of the supplement CD.....	65
11 List of references.....	67
12 Referred figures.....	69
List of figures.....	70
List of tables.....	71

HALLGATÓI NYILATKOZAT

Alulírott Balázs János, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2013. 05. 26.

.....
Balázs János

Összefoglaló

A számítástechnika fejlődése lehetővé tette, hogy a korábban analóg lemezjátszót illetve CD játszót és keverőt tartalmazó DJ összeállítást szoftveres megoldások váltsák fel. Ennek fő oka, hogy manapság a zeneszámok többsége a korábbi CD vagy LP kiadványok helyett csak fájl formátumban, internetes áruházakon keresztül érhető el. Emellett a DJ szoftverek más előnyökkel is rendelkeznek a hardveres megoldásokhoz képest, mint például az átkeverendő számok tempójának automatikus szinkronizációja, a zenei adatbázis átláthatóvá tétele, és az ebből következő gyors számcsere lehetősége.

A DJ szoftverek feltételezik, hogy a zene állandó tempójú és a negyedütések pontos időközönként követik egymást, hiszen alapvetően elektronikus zenei alkalmazásra készültek. Ahhoz, hogy ezekben a szoftverekben akusztikus zenét is felhasználhassunk, szükség van az azokban fellelhető esetleges tempóegyenlenségek kijavítására.

Dolgozatomban egy olyan szoftver megvalósítása volt a célom, mely a betöltött zeneszám negyedütségeit automatikusan meghatározza és az időben pontatlan ütések time-stretch algoritmus segítségével a helyükre húzza.

Az algoritmus fejlesztését MATLAB környezetben végeztem, melyet azután C++ nyelven implementáltam.

Abstract

By the evolution of computer technology the classic DJ setup consisting of turntables or CD players with mixers have been replaced by computer DJ softwares. The main reason for changing equipments is that most of the electronic music of today is released in digital audio files replacing the former vinyl or CD formats. By using DJ softwares, tempo analysis and beatmatching of songs has been automated. Moreover the music database is handled by the computer, making it much quicker and easier to find and load tracks into the virtual track decks.

DJ softwares presume a constant tempo throughout the music tracks, as they have originally been designed to handle electronic music. In according to load and mix acoustic or instrumental music with computer DJ softwares, these kinds of tracks need their tempo to be corrected.

The aim of my project was to develop a software tool that is able to analyze acoustic and instrumental recordings by finding the location of beats. Did the beat grid of the track contain any imperfections; the algorithm should correct it by time stretching procedure.

For developing the algorithm I have used the MATLAB environment, and I have implemented the program in C++.

1 Introduction

Computer DJing acquired popularity in the last two decades. The reason of its success is that in compared to traditional DJ techniques such as turntables or CD players, DJ softwares provide graphical display for the DJ, store the music library in database making it easy to search for a track or tag a file with personal notes, and the softwares are also able to synchronize the tracks to be mixed.

By 2012, DJ softwares gained a large market; their software (and hardware) components can be found on any website selling DJ & audio equipments. The most popular softwares of today are [1] [2].

- Traktor DJ
- Ableton Live
- Virtual DJ
- Serato Scratch DJ
- Mixvibes Cross DJ

As a crossover platform, vinyl and CD emulation systems appeared on the market in the mid 2000's. These systems allow manipulation and playback of digital audio files using vinyl turntables or CD players via special timecode vinyl records or CDs. With these timecode records, DJs can enjoy the convenience of using software DJing systems, facilitated by the traditional interfaces, such as turntables or CDs and external mixers. The market of these timecode systems also became considerable in the past few years in correspondence with the prosperity of DJ softwares. I personally prefer vinyl timecode system, as they not only allow computer DJing with turntables, but both digital and analog records (actual vinyls) can be used one after the other.

For software DJing only MIDI controllers are used as input interfaces. MIDI controllers operate like computer input devices in general, using the computer's main bus system, (e.g. PCI Express) sending input commands on action. The input latency of MIDI devices is very low to fulfill the timing requirements of play/pause and cue actions. Still, according to my experiences on Traktor Scratch systems, MIDI controllers' latency can be twice as large as the CD players' latency. This is an important feature that made timecode systems more popular than computer-only-systems among pro DJs.

The latency of the playback is input peripheral independent; both software-only and timecode systems use ASIO low-latency sound card driver protocol and so have the same output latency.

In general, software DJ programs have track decks that are responsible to play, pause and cue the records, along with setting the records' tempo and pitch. The track decks are the most important elements of the software, all the accuracy of the DJ mixing depends on the track decks' playback accuracy. Track decks are the software equivalents of CD players and turntables. In Figure 1 the track deck of Traktor Scratch can be seen. [1] The track deck displays the sound waveform by blue and the song's beat grid by grey lines. It also shows the current tempo of playback in the upper right corner (140.00) while the original tempo can be found at the bottom of the track deck (143.000).

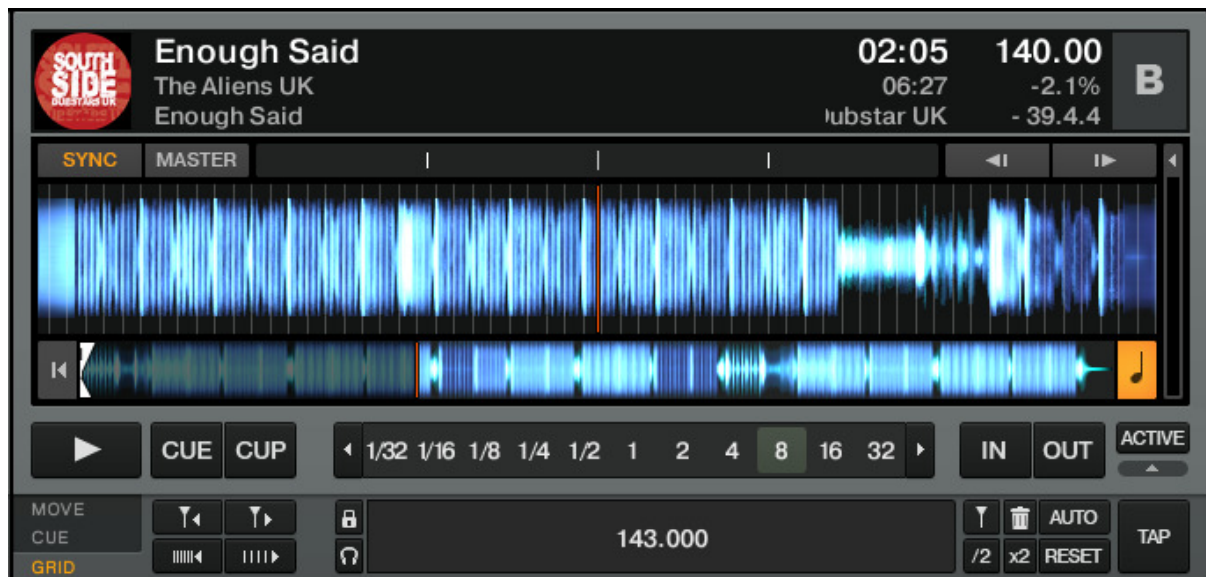


Figure 1: The track deck of Traktor Scratch

DJ softwares have an important feature that traditional DJ setups don't: synchronizing the records in their track decks. Synchronizing means that the two records to be mixed are set to the same tempo and to the same phase (tracks beat synchronously). If needed, the automatic synchronization feature can be switched off, granting the DJ the possibility to perform synchronizing manually.

To synchronize the records, the track decks need to analyze the songs first, i.e. they need to find the songs' tempo and the exact locations of their beats in time. The analysis process is run when the tracks are added to the library, and it usually takes about 5-10 seconds before the user gets access to the tempo information. The softwares of today are able to calculate a song's tempo with the accuracy of 0.01 BPM (beat per minute).

However, once the software calculated the tempo of a song, even though it consists of blocks with different BPM values (e.g. 140/110 BPM songs) only one of the two BPM values will be found and displayed for the DJ. This property is a notable disadvantage.

Another shortcoming of DJ softwares is that they are unable to handle imprecision of tempo in a sense that constant tempo rate is presumed throughout the song. This feature does not affect the playback and mixing of electronic music as their beats are perfectly accurate. Mixing acoustic live takes or non-electronic music on the other hand, demands the detection of these slight imperfections in tempo. DJing softwares can't perform this precision analysis of songs; therefore can't correct songs to a fixed tempo either.

These led me to the idea to create a program that is not only able to find the tempo of electronic records but acoustic music as well, correcting them beat by beat and making them ready to be DJ mixed.

The design project consisted of solving the following problems:

1. Finding the tempo of the songs (Tempo detection)
2. Finding the beats in time (Beat detection)
3. Calculating the corrected beat grid for the songs (Beat correction pre-processing)
4. Performing the modification of the beat grid (Beat correction by time stretching)

In Chapter 2 the tempo detection module, in Chapter 3 the beat detection procedure is explained.

In Chapter 4 the beat correction pre-processing is discussed, Chapter 5 explains the beat correction method by time stretching in details.

Once the algorithm design in MATLAB was ready, I implemented it in C/C++ Matlab Exchange (mex) format, covered in Chapter 6.

Chapter 7 deals with the MATLAB graphical user interface that has also been added to the project.

Chapter 8 summarizes the results and talks about the possibilities of development.

2 Tempo detection

Plenty of articles, papers and webpages trade with the problem of beat detection in some respect. Among these the most referenced paper is Eric D. Scheirer's Tempo and beat analysis of acoustic musical signals from 1996, MIT Media Laboratory, Cambridge, Massachusetts. [3] Based on Scheirer's paper, a project has been published, titled 'Beat this' by researchers of RICE University, Houston. [4]

The project's webpage discusses the MATLAB implemented algorithm in details. In the beginning my aim was to understand this tempo detecting method and improve its performance.

2.1 The Beat This algorithm

Following through the steps of Schreirer's procedure of tempo detection by the Beat This algorithm, their steps are as follows:

1. Picking a short interval from the song to be analyzed

Regardless of the content of the short signal from the song, the algorithm simply cuts a few seconds out from the middle of the track.

2. Breaking the signal into frequency bands

To investigate the tempo of the different instruments (supposedly being in different frequency bands) the algorithm separates the frequency spectrum into six bands, spanning one octave each.

- | | |
|------|----------------|
| i. | 0-200 [Hz] |
| ii. | 200-400 [Hz] |
| iii. | 400-800 [Hz] |
| iv. | 800-1600 [Hz] |
| v. | 1600-3200 [Hz] |
| vi. | 3200-4096[Hz] |

3. *Envelope generation*

In this step the signal is first full-wave rectified. Then the signal is convolved by the right half of a Hanning window with a length of 0.4 seconds. The signal and the window are transformed into frequency domain, multiplied, and inverse transformed to decrease computation time. This is done for each band separately.

4. *Diff-Rect*

Next, the envelopes are differentiated in order to have more emphasis on amplitude changes. The largest changes should correspond to beats as a beat can be considered as a major increase in sound amplitude. The six envelopes are differentiated in time, then half-wave rectified so that only increases in sound can be seen.

5. *Comb Filter*

This is the computationally most intensive step. The differentiated envelope signals get convolved with various comb filters to determine which tempo yields the highest energy. A comb filter is a series of pulses that occur periodically, at the specified tempo. Convolution with a comb filter (consisting of three impulses) with the investigated signal produces an output that has higher sum energy when the tempo of the comb filter is matching the tempo of the song.

Squaring the output of the convolution gives the energy of the signal.

6. *Peak picking*

As the last step the tempo yielding the highest energy is chosen as the track's tempo.

The block diagram of the process above can be seen in Figure 2:

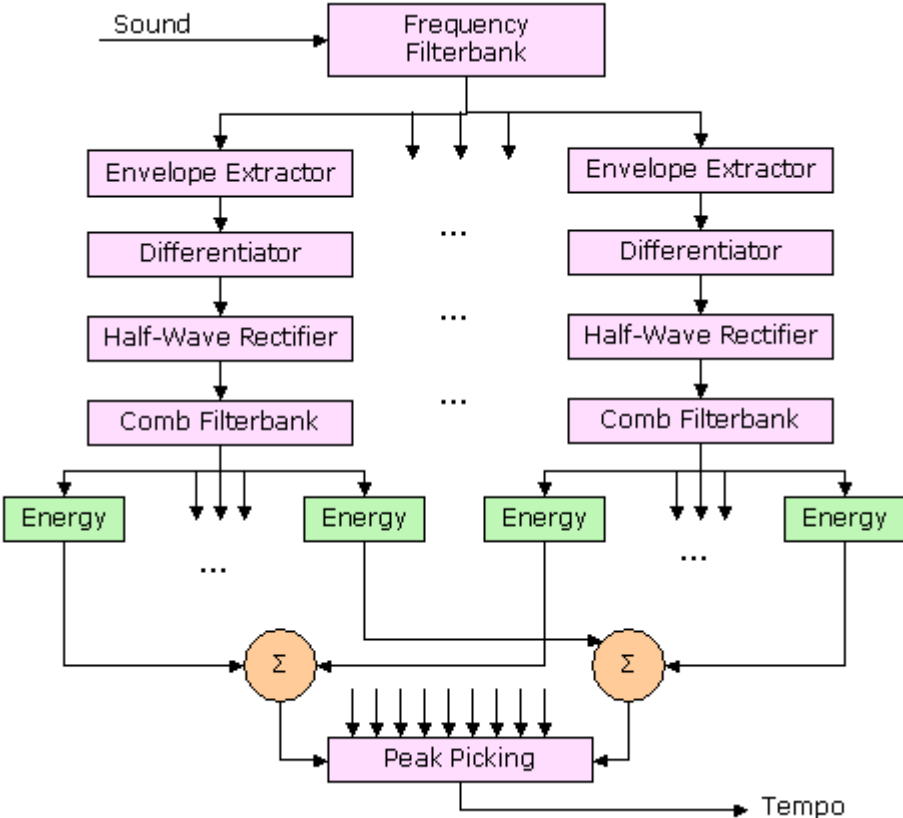


Figure 2: Block diagram of the Beat This algorithm (Figure taken from [4])

2.2 Tempo detection module of my project

After investigating the Beat This algorithm, I tried to beat it.

The performance of the Beat This algorithm had been published on their project's website, [4]. Their algorithm was run on plenty of tracks, calculating their tempo. The algorithm-estimated and the 'real', metronome-calculated tempo of the tracks have been compared, so that a hit could be defined for the algorithm. After implementing my algorithm, I ran it on the same songs finding their tempo to see if I could improve the hit performance.

As for the start I followed the steps of the original Beat This algorithm and investigated if the certain steps are beneficial or necessary in it. I tried to expand its hit probability and improve the execution time.

The steps of the beat detecting module of my project are as follows:

0. Reading the input sample

Previously the algorithm was only able to read wav files, although quite a few songs are released in original waveform format. Most of the tracks are released in some compressed format, such as mp3 or m4a. Therefore I searched for Matlab extensions with which compressed songs could be processed. On Matlab Central I found the *Mp3read and mp3write* library. The *mp3read* command uses the Lame encoder to uncompress mp3 songs, creating temporary waveform files from the mp3's. By applying this library in my project the amount of tracks I could use for testing highly increased, [5].

1. Decimating the input sample

In according to handle longer cuts from the songs in the computer's memory, the sampling frequency had been reduced to its sixth. Lowpass filtering and picking every sixth element from the input can be performed by Matlab's *decimate* command. Reducing the sampling frequency to 7350 Hz will keep frequencies only under 3675 Hz. Although this means loosing a large amount of important data, longer sample cuts could be loaded into memory for analysis.

2. Smoothing

The absolute value of the sample is taken, and then it is lowpass filtered at 10 Hz in order to retain only the trends of the sound energy. A 6th order lowpass Butterworth filter is designed by the *butter* command; sample is filtered by the *filtfilt* command. This reduces computation time compared to convolving by a Hanning window.

The filterbank has been omitted from the project as it didn't increase the hit count significantly, although it highly increased the computation time.

3. Diff-Rect

The difference from one sample to the next is found for the smoothed signal. Next it is half wave rectified to retain only if difference is positive. This step has not been changed from the Beat This algorithm. I tried to omit this step, but the hit performance has been radically decreased without it. In Figure 3 and Figure 4 the differentiated envelope $d[n]$ can be seen.

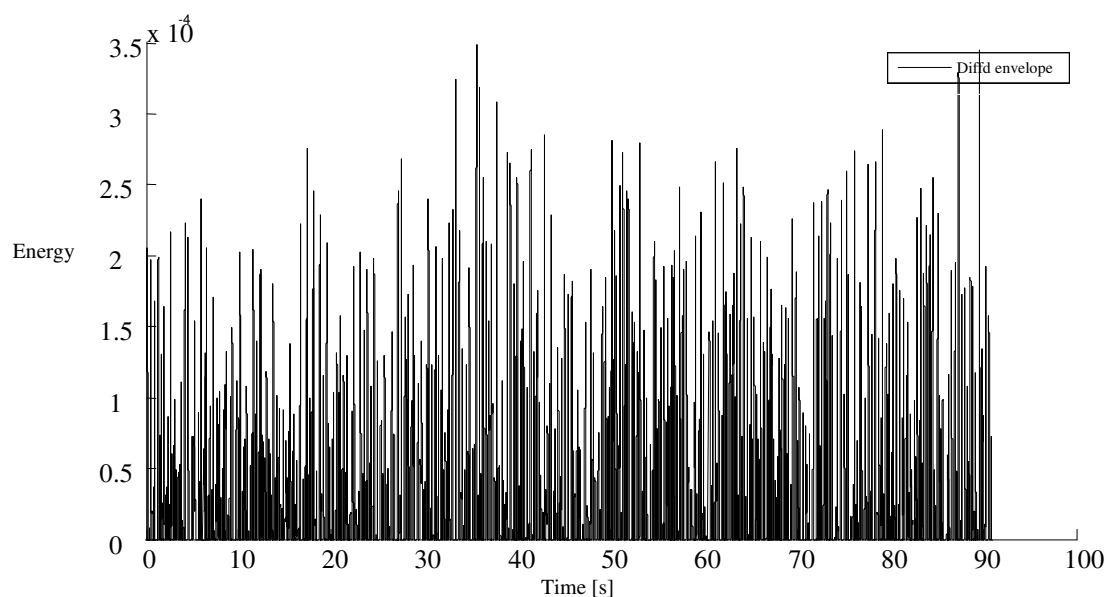


Figure 3: Differentiated envelope

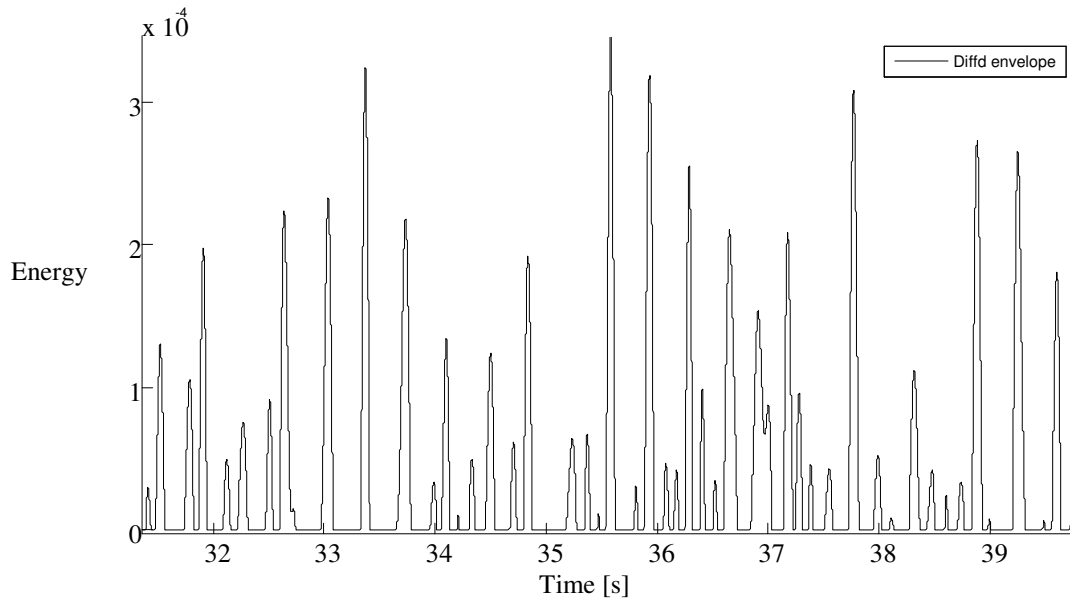


Figure 4: Differentiated envelope, closer look

4. *Picking a sample cut to be analyzed*

As the sampling frequency was reduced to its sixth, the amount of data to be loaded into memory can be increased. For initial estimations of tempo the first minute is cut for analysis.

5. *Comb filter*

Similarly to as discussed in section 2.1.5, the differentiated signal envelopes are convolved with various comb filters to determine which tempo yields the highest energy. Convolution is performed in the frequency domain; the input sample is windowed before the Fast Fourier Transformation. The number of pulses in the comb filter was increased to 15 from 3; the tempo interval of interest got expanded from 60 BPM till 210 BPM. The algorithm at this point finds the tempo with the accuracy of 1 BPM.

The output of step 5 for two songs can be seen in Figure 5 and Figure 6.

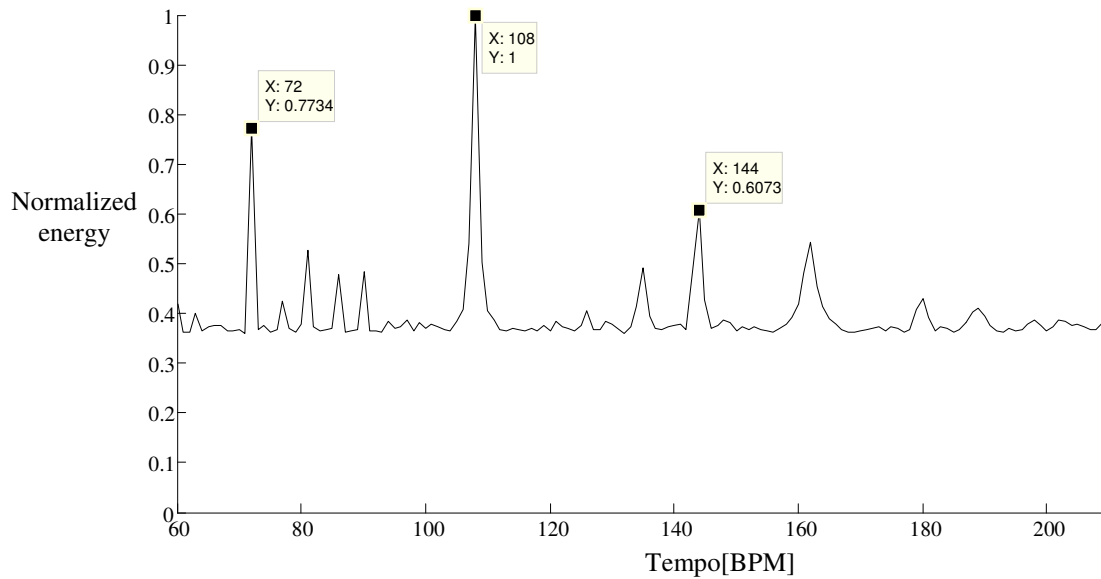


Figure 5: BPM prediction of Will Smith – Gettin’ Jiggy With It

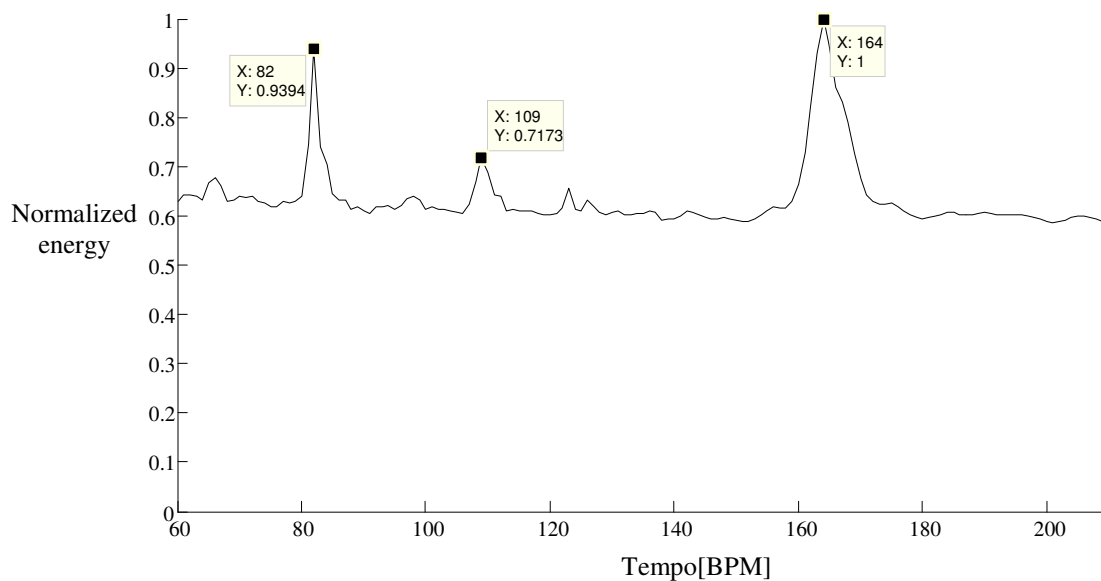


Figure 6: BPM prediction of Csík Zenekar – Csillag Vagy Fecske

In Figure 5 we can see a sharp peak in the prediction diagram of an electronic music track at 108 BPM. From the sharp peak we can presume that the tempo of the song is quite accurate.

Contrarily, in Figure 6 the tempo prediction diagram belongs to a live recorded, acoustic song that supposedly fluctuates in tempo over time, hence the wider lobe at 164 BPM.

6. Fine resolution comb filter

After finding the tempo of the song's first minute by 1 BPM resolution, the algorithm calculates the tempo for each 18 seconds interval with an overlap of 9 seconds, using the same method as discussed in step 5. The tempo is now sought by the resolution of 0.05 BPM around the current tempo in the interval of +/- 4BPM. The outputs can be seen in Figure 7 and Figure 8.

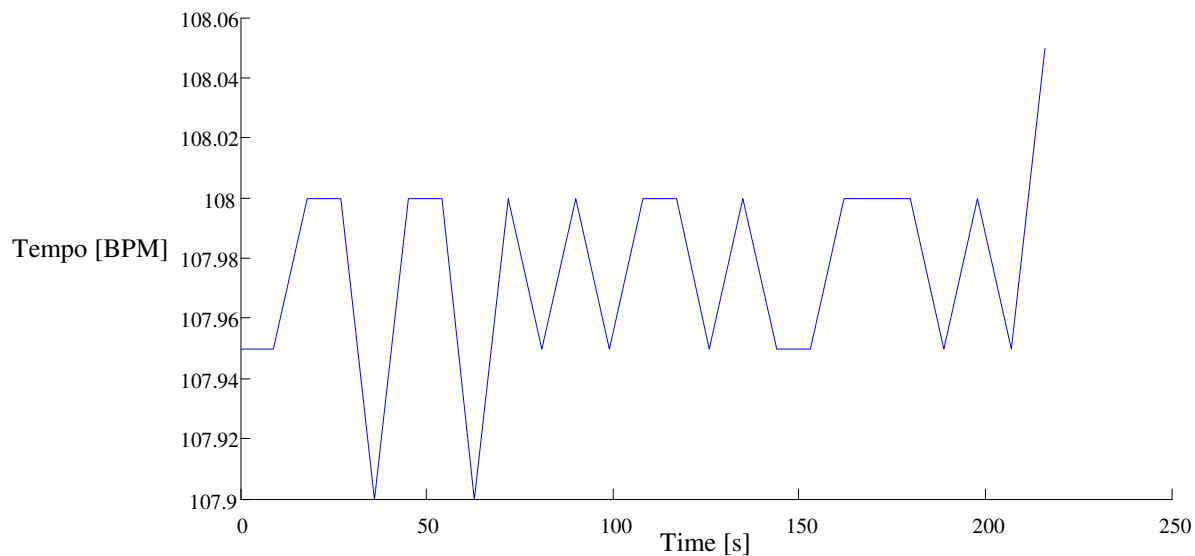


Figure 7: Fine BPM prediction over time for Will Smith – Gettin' Jiggy With It

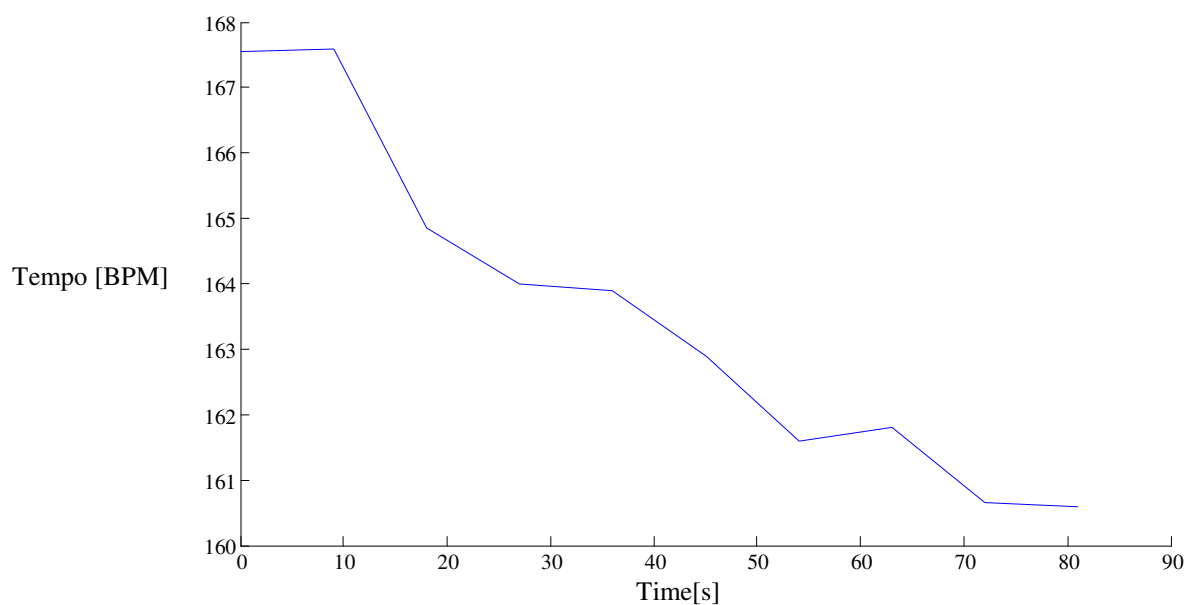


Figure 8: Fine BPM prediction over time for Csík Zenekar – Csillag Vagy Fecske

In case of electronic music recordings the tempo usually stays on the same speed or slight differences might be found over time. See Figure 7, where the estimated tempo fluctuation is within 0.15 BPM.

In case of acoustic recordings, the deviation from the starting tempo can be considerable. As drummers usually don't use metronomes during live takes, a drift in tempo is quite usual. See Figure 8, where the estimated fluctuation is more than 7 BPM, and the music is slowing down continuously.

2.3 Performance of the tempo detection module

2.3.1 Comparing with Beat This

The Beat This group published a chart about their tempo detection performance on their homepage's results section [4]. They ran the algorithm on 11 tracks from which 8 tracks contain drum beats, the rest were symphonic pieces. To compare the performance of the tempo detection algorithms, I looked up the same songs and ran the tempo detection tests on them. For verification purposes I also ran the tests in *Mixed In Key* and *Traktor Scratch Pro* softwares to find their tempo prediction for the songs as well. *Mixed In Key* is only able to estimate the tempo by 1 BPM resolution.[6]

The test results can be seen in Table 1.

Track title	Own module	Beat This	Traktor	Mixed In Key
Beverly Hills Cop theme	117.38	160.12	117.52	118
Lil' Jon - Bia Bia	160	119.02	160.11	160
Venga Boys - Boom	138.5	140.16	138.48	138
Will Smith - Gettin Jiggy With It	107.95	109.5	107.9	108
Stan Getz - Girl from Ipanema	130.32	136.73	130.61	130
Limp Bizkit - Rollin	195.07	186.19	194.738	192
Green Day - Longview	143.04	155.34	148.64	146
Green Acres theme	119.39	125.96	119.90	120

Table 1: Comparison of beat detection algorithms

In Table 1 cells that have been multiplied by two for the comparison can be seen with grey background. From DJing point of view multiplying tempos by two don't cause any error as the multiplied tempo can be considered as its own harmonic. Nevertheless, by the optional multiplication of certain cells the measurement data could be compared.

In Table 1 values that are different from their row by more than 4% are marked with red. (In general, the used commercial softwares can be trusted as reference.)

It can be seen that the accuracy of the module discussed in this chapter is in correspondence with the commercial softwares, where as the results of Beat This got marked 4 times out of 8.

2.3.2 Quantitative verification of the tempo detection module

For verification of the module, the algorithm was run on computer aided/studio recorded (not live acoustic) tracks from different music genres. The test results can be seen in Table 2.

	Track title	Own detection module	Traktor
1.	01 Kenny Leaven - Feeling Spicy.wav	125	125
2.	01-umek_-_carbon_occasions-etImp3.wav	129	129
3.	02 Delon & Dalcan - Ultracolor.wav	128	128
4.	02-lindstrom-i_feel_space__mandy_remix-tsp.wav	124	124
5.	02-shlomi_aber_ft_lemon_-_moods_(valentino_kanzyani_remix).wav	126	126
6.	03-moonbeam-cocoon_(sunset_mix)-loyalty.wav	125	125
7.	05-barem-link-hqem.wav	127	127
8.	06-paul_kalkbrenner-gebruenn_gebruenn_(alexander_kowalski_rmx).wav	130	130
9.	07 gui boratto - the blessing.wav	127	127
10.	07-paul_kalkbrenner-gia_2000_(modeselektor_rmx).wav	114	115
11.	120 justice_-_d.a.n.c.e_justice_remix.wav	120	120
12.	187 lockdown - gunman (original mix).wav	128	128
13.	Artful Dodger - R U Ready.wav	135	134
14.	Audio - Emissions.wav	172	172
15.	B-15 Project Ft Crissy d & Lady G - Girls Like Us.wav	130	130
16.	Beverly Hills Cop Theme.mp3	117	118
17.	DJPalotai@KáoszKemping.wav	98	98
18.	Daft Punk - Robot Rock.wav	112	112
19.	Green Acres theme.mp3	119	120
20.	Green_Day-Longview.mp3	143	149
21.	Limp Bizkit – Rollin.mp3	195	195
22.	Ratatat - Ratatat - 01 - Seventeen Years.wav	116	115
23.	Sub Focus - Deep space.wav	174	174
24.	Sweet Female Attitude - Flowers.wav	132	132
25.	The Knife - Heartbeats (Rex The Dog Mix) .wav	117	117
26.	The Sounds - Bombs Bombs Away (Teenage Battlef.wav	140	139
27.	a1 petter - all together.wav	125	124
28.	a1-marc_houle-business-cute.wav	123	123
29.	crystal castles -04- dolls.wav	131	132
30.	helicopter showdown - dramatron.wav	140	140
31.	excision & datsik - invaders.wav	140	140
32.	jurgens - love it (final mix).wav	125	125
33.	moby - Lift Me Up (Mylo Mix).wav	125	125
34.	Venga Boys – Boom.mp3	139	138

Table 2: Beat detection module vs Traktor

As it can be seen in Table 2, comparing the beat detection module with the output of Traktor Scratch 2, the difference of the tempo results have been greater than 1 BPM only once out of the 34. The outlier element of Table 2 has been analyzed with fine resolution tempo detection and gave the results of Figure 9.

It fluctuates in tempo so heavily that tempo prediction goes into saturation around the ~90-99sec and ~162-180sec intervals. The tempo prediction based on its first minute was 141 BPM, therefore the tempo through the song has been sought only in the 137 – 145 BPM interval. I have checked the tempo “manually” using metronome, and found that the song’s chorus is on 146-149 BPM, while the verse is only on 140 BPM.

The prediction of my module was eventually 143 BPM, the mean of the tempo of sections throughout the track. Traktor’s estimation gave the tempo of the last minute in the song.

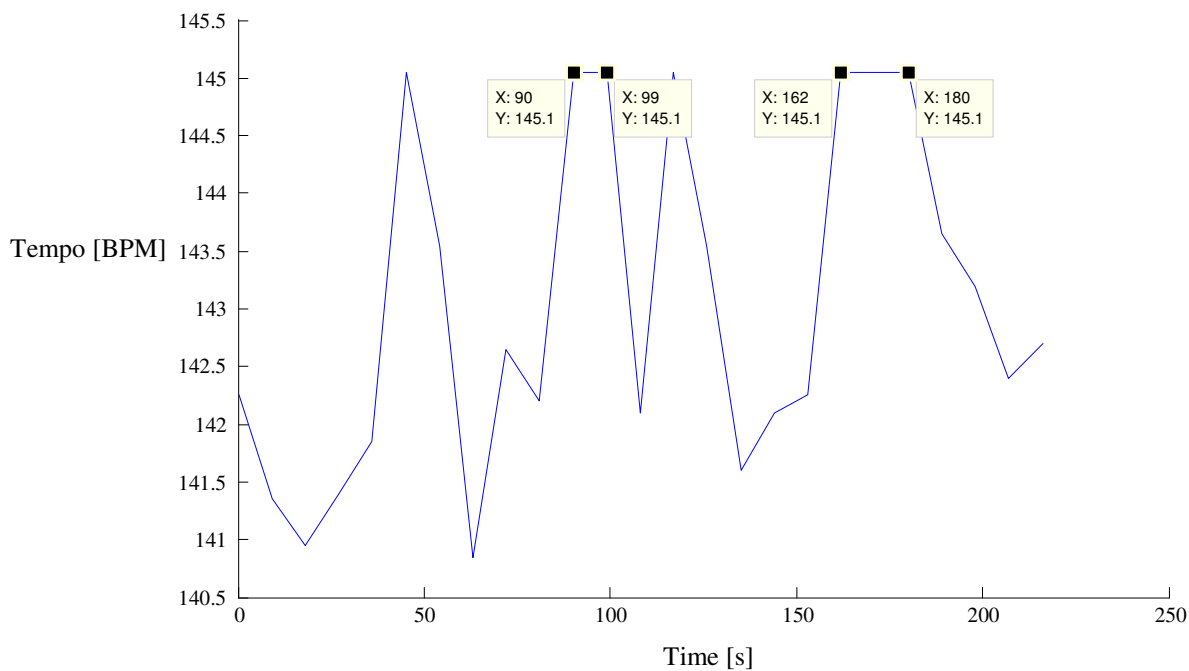


Figure 9: Fine BPM prediction over time for Green Day – Longview

Among the test tracks various music genres can be found including midtempo, techno, hip-hop, dubstep, drum’n’bass, UK garage, metal, rock and punk.

It can be seen that the algorithm works perfectly except for songs having intentional tempo changes, such as in Longview from Green Day, or in 110BPM → 140 BPM switchup songs. Finding the sections with different tempo in the track could be an advanced feature of the algorithm; this is a possibility of further development in the project.

3 Beat detection

For correcting the micro tempo inaccuracies in a track first the beats need to be found. Finding just the tempo of a track is not satisfactory as tempo does not tell the exact locations of beats over time, although it can help us where to look for it. If we can find one beat somewhere in the track and we also know its tempo, a map can be created about the locations where we suppose to find beats. This is the main idea behind the Beat This team's *phase alignment* method [4].

Ignoring the tempo data and looking for quick increases in sound amplitude in some way is the idea behind the *statistical streaming beat detection* [7].

In my work I tried to take advantage of both algorithms as I came to find beneficial details in both of them.

3.1 Beat detection of Beat This

The algorithm starts by running a music track through all the steps of the beat-matching algorithm until the comb filtering step. This gives the signal strong peaks where beats occur. Since this frequency at which the beats occur is known as the algorithm has just calculated it, it is also known which comb filter corresponds to the track.

At the first part of the comb filter output a local maximum is sought. If found, seeking for the next beats is carried on in distances of the track BPM reciprocal. The found beats' locations of the envelope are recorded. The beat positions in the envelope and in the original track are supposed to be at the same places.

3.2 Statistical streaming beat detection

Frédéric Patin's work discusses another simple method for beat detection [7].

This algorithm does not use any a priori information about the tempo, only looks for great amplitude changes in the input signal. The input signal is divided into "moments", approximately 17 ms fractions. Over time a threshold value is maintained, defined as the sum of the mean plus a fraction of the standard deviation of the last second. Where the signal rises above the threshold value for a moment, the algorithm records a beat with that timestamp.

3.3 Beat detection of electronic music

The tempo detection algorithm is able to calculate the tempo by the accuracy of 0.05 BPM. For finding the beats in the track, we can use this tempo information. Following the idea discussed in section 3.1 first we try to find one beat in the track. We know that the previous and oncoming beats are to be expected in $1/\text{BPM}$ distances from the beat what we have found. If we succeeded, a simple grid can be put over the track marking the beats in the signal.

The beat detection algorithm for electronic music reads in two inputs: the envelope and the estimated tempo of a track. From these the algorithm creates the beat grid for the track, producing a vector containing the locations of the beats over time.

In steps the algorithm performs the following:

1. A comb filter is created (as discussed in section 2.2) having the same tempo as the track being analyzed.

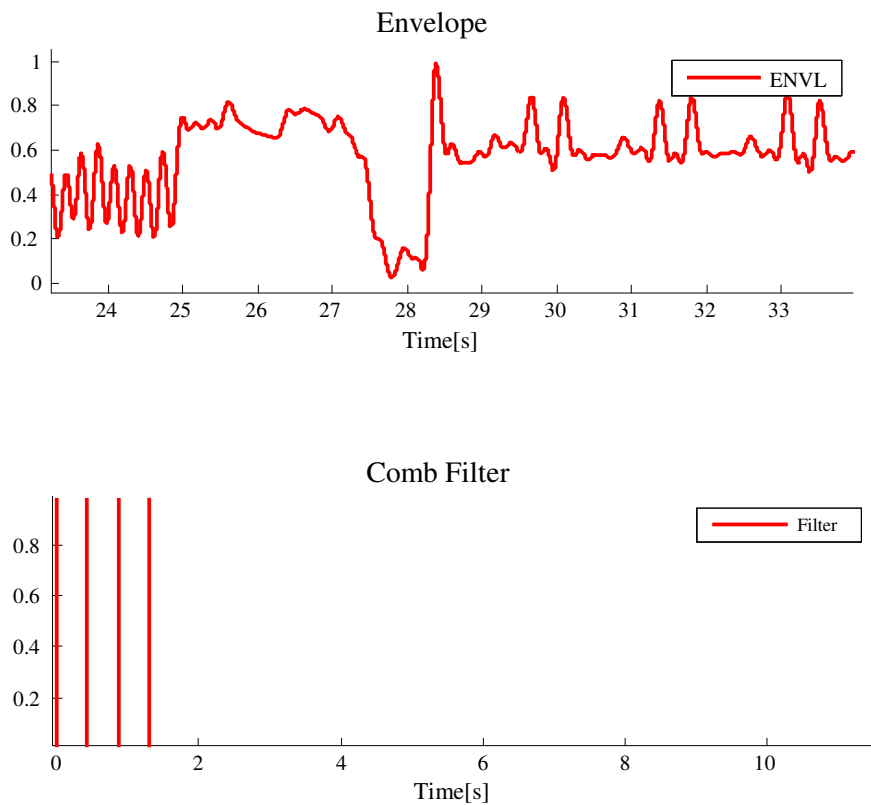


Figure 10: Envelope $e[n]$ of a track and its corresponding comb filter

2. The envelope of the song $e[n]$ (top signal of Figure 10) is filtered by the comb filter of four pulses (bottom signal of Figure 10) producing the ‘sharpened’ signal $s[n]$ (black signal in Figure 11).

Notice the peaks in the black signal at locations where there weren’t any in the red signal. This helps finding the beats, by emphasizing the peaks.

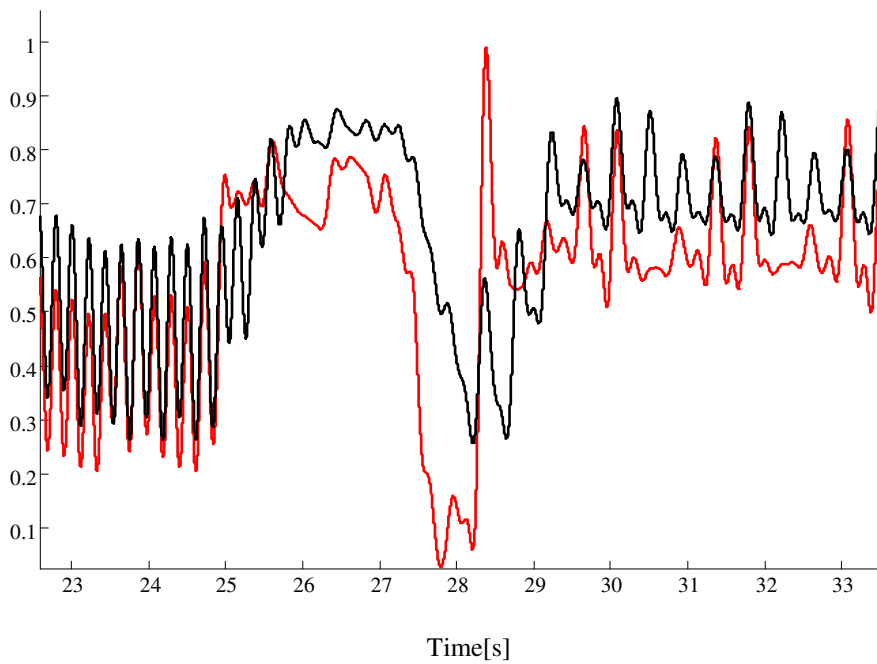


Figure 11: Sharpened envelope $s[n]$ (black) and envelope $e[n]$ (red) signals

3. We find the location of the maximum amplitude in the sharpened $s[n]$ signal. This location will be considered as a beat, and this will serve as the foundation of the beat grid $g[n]$.

4. We calculate the locations being at multiples of $1/\text{tempo}$ distance from the fundamental beat in both forward and backward directions. (Marked by red stems in Figure 12)

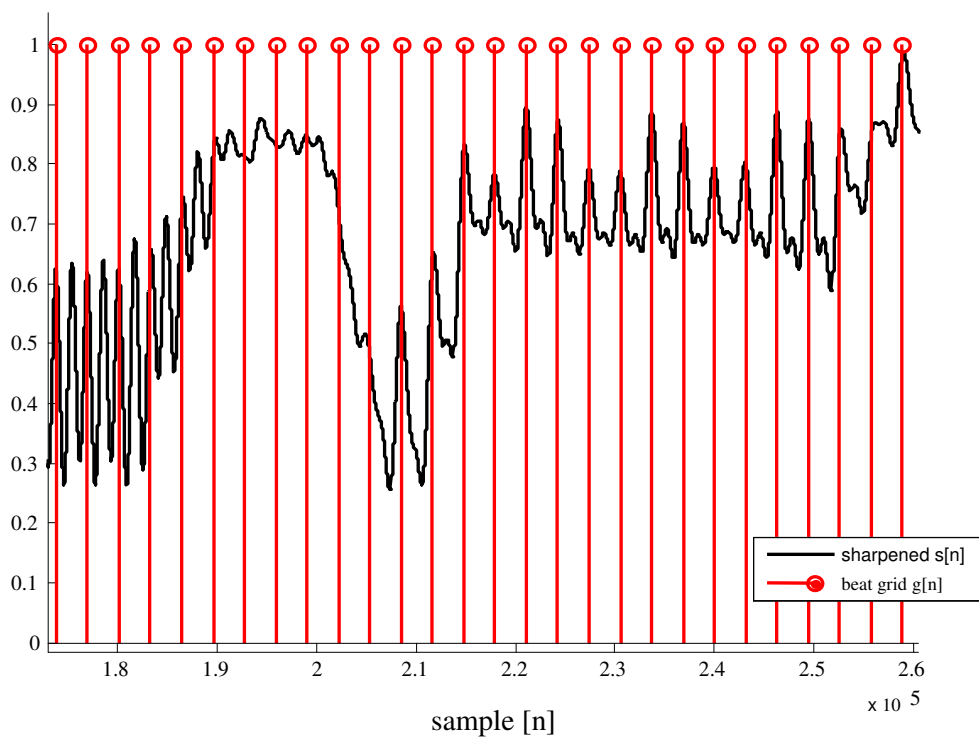


Figure 12: The beat grid on track

As a fifth step I tried to optimize the beat grid. I've compared the beat grid to the local maximum places of the envelope around the beats. If a local maximum was not at the exact location of the calculated beat grid point, the grid beacon has been shifted to the location of the local maximum. Although this idea seemed reasonable, it has not brought success at all. I checked the operation of the algorithm by cutting 4 / 8 / 16 beats long samples out of the track and looped them, using the beat grid points to define the loops' start and endpoints. At the ends of the loop samples the beats did not always match.

Although beat by beat electronic music track beats may not be strictly on a beat grid, in a medium range of 8-16 bars they are accurate, this comes from the algorithm also.

3.4 Beat detection of acoustic recordings

Creating the beat grid for electronic music only needed a single maximum energy point in the envelope to be found, and the grid could be placed over the record straight away. Does this method work in the case of acoustic music as well? Unfortunately, it does not.

When looking for the beat grid of non-computer aided music, we can always find slight inaccuracies in tempo. If a band plays live in the studio for album recording, they play with a metronome, and if needed, sound engineers assist them making their patterns accurate in tempo afterwards. The majority of acoustic songs played in the radio have already gone through this process. Without fixing these slight inaccuracies the songs can't be beatmatched with DJ softwares.

In fact, most of the tempo-inaccurate recordings are live takes. Live and/or unplugged recordings that have not gone through any tempo restoring post processes are the target of this project, such as the tracks of Hungarian Radio 2's Akusztik series. [8]

Correcting the live recordings in tempo seemed to be difficult; therefore first I tried to correct the tempo of a simple drum pattern.

3.4.1 Beat detection of a drum pattern

To create a valid inaccurate sample to work on, I myself created a drum pattern hitting my laptop case gaining a snare-like sound. To make it more similar to a real live drum pattern I replaced certain beats by kick drum and hi hat sounds from libraries. This has become the recording I used to test my beat correction algorithm.

The first difficulty of the correction process arose straight when I tried to create the envelope of the drum pattern. The method discussed in section 3.1 cannot be used, as the convolution process would average the slight inaccuracies of the recording, producing a slightly corrected tempo for the convolution output.

Therefore, to create the envelope of the drum pattern I have used the *peak* measurement method from Udo Zölzer's book, DAFX [12].

The block diagram can be seen in Figure 13.

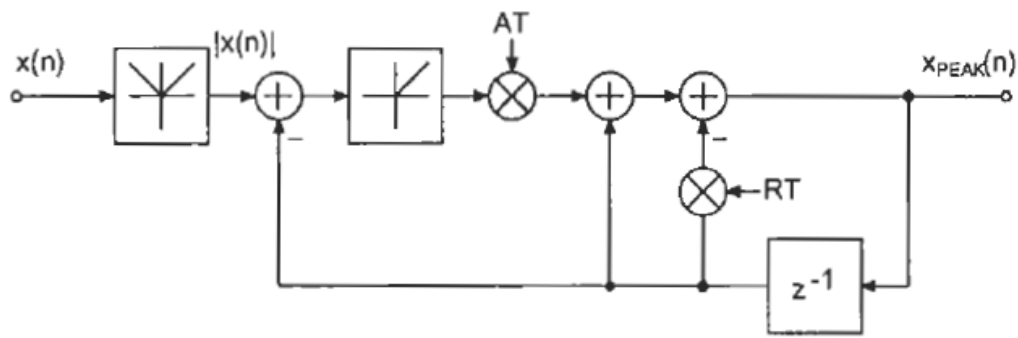


Figure 13: Peak measurement system (Figure taken from [12])

The system can be considered as an IIR filter. Creating the envelope $e_{\text{peak}}[n]$ from the original track $y[n]$ can be defined by the equation of:

$$e_{\text{PEAK}}[n] = \text{pos}(|y[n] - e_{\text{PEAK}}[n-1]|) * AT + e_{\text{PEAK}}[n-1] * (1 - RT) \quad \text{Eq.1.}$$

The system has two parameters, attack time (AT) and release time (RT).

By fine adjusting the values of these parameters we can achieve the envelope in Figure 14 (RT = 0.005, AT = 0.3).

The original waveform of the track $y[n]$ can be seen by red, and its envelope $e_{\text{peak}}[n]$ by black.

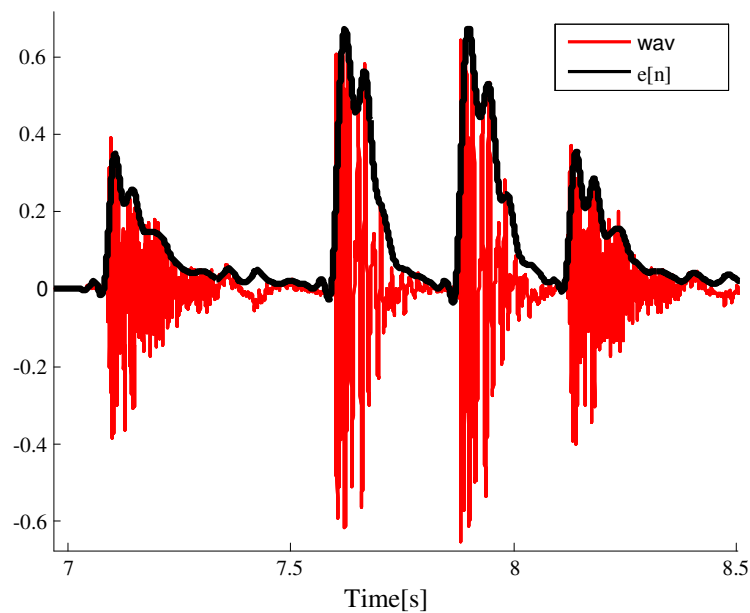


Figure 14: Envelope $e_{\text{peak}}[n]$ signal, using the peak measurement system

The algorithm for creating the beat grid for the drum pattern is as follows:

1. The tempo of the envelope is calculated. The general tempo is specified by the accuracy of 0.05 BPM.
2. We need to find the location having the highest energy in the envelope. This step is the same for finding the beat grid of electronic music recordings.
3. Beats are being sought in eight note distances, (backwards in time first, then in forward direction). For defining a beat's supposed location, the algorithm checks the derivative of the envelope $d[n]$. The reason for investigating the derivative signal $d[n]$ is that it is peaking sharper than the envelope.

When hitting a kick drum and a snare at the same they don't peak at the same time, even though they had been hit at the same time. Using the derivative of the envelope slightly reduces the impact of this phenomenon. In Figure 15 we can see the derivative signal $d[n]$ of the envelope by blue.

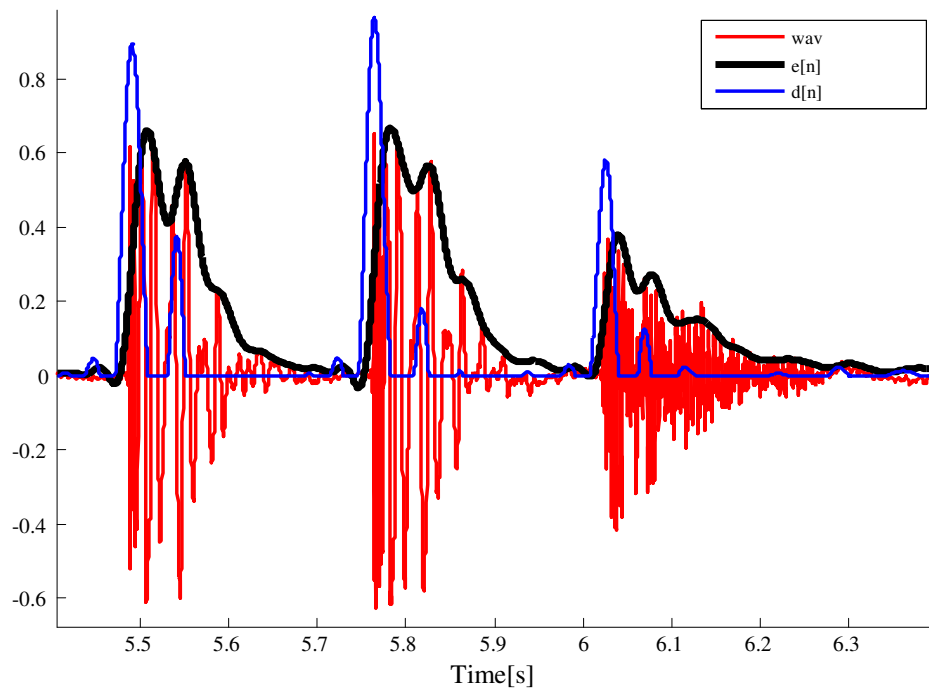


Figure 15: Envelope signal $e_{\text{peak}}[n]$ (black) and derivative of the envelope $d[n]$ signal (blue)

4. We check a small interval around the location where the beat is awaited. If we find a beat we compare its location to the location it should be (where the beat was awaited exactly).

5. If the presumed location of the beat and the actual location of the beat, i.e. the local maximum of the derivative signal are not at the same location (which is quite likely) the whole bar is shifted altogether to the right position.

For that, the certain bar of the original file is cut out and multiplied with a (bar length) window $w[n]$ then placed to the right position in a new output file.

The window signal $w[n]$ with on a bar can be seen in Figure 16.

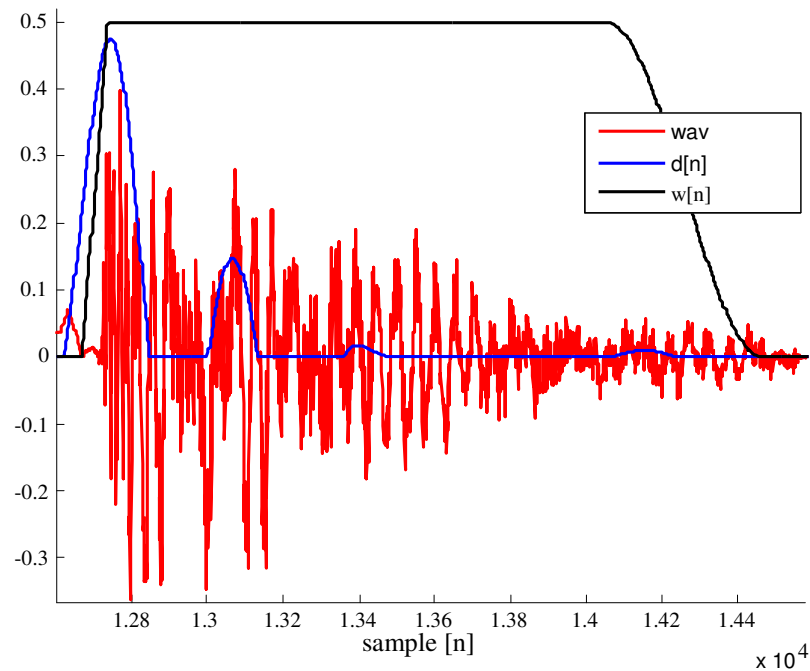


Figure 16: Window $w[n]$ signal (black), on a bar of the original file $y[n]$ (red), at the peak of the diffd envelope $d[n]$ local maximum (blue)

6. Every time a beat is found and corrected, the oncoming beat is sought in one eighth note distance, measured from the position where the last beat has been found. Using this method, errors in tempo are not cumulated but beats are corrected locally beat-by-beat.

If we can't find a beat at one eighth note distance from the last beat, the algorithm looks for the next beat at one quarter note distance. If no beat could be found there either, the algorithm carries on, seeking for a beat at $3/8$ note and so on. Unfortunately, with this method if there is a long break in the pattern and the musician comes back from the beatdown at the wrong time (making the hiatus too short or too long), the algorithm fails correcting it for being blind for beats out of its scope.

The output of the algorithm can be seen in Figure 17. In order to make the comparison of the original and the corrected files easier, the envelopes are compared instead of the unmodified, raw waveforms.

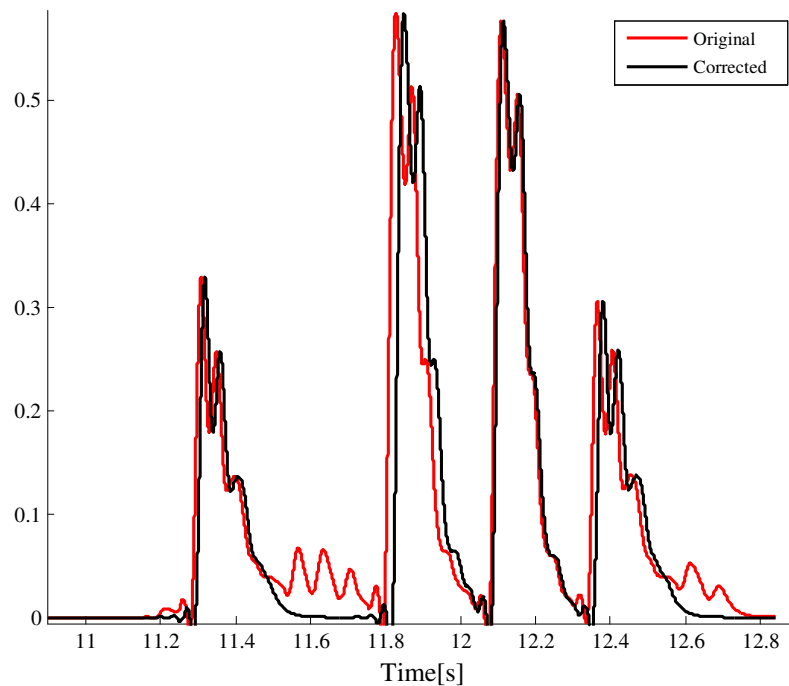


Figure 17: Comparing the envelopes of the original and the modified signals

As we can see, in the corrected file the beats are put to the right location. It is notable that the lower amplitude ripples of the red signal are not copied into the corrected black signal as they were not considered as beats; the algorithm hence skipped copying those bars, inserting silence into the output file instead.

Although this method can be used to correct percussive input signals, it can not be applied to acoustic music for skipping bars in case of no beats. Acoustic music usually contains long held instrument sounds that shall not be cut at the ends of bars. Therefore I had to look for another solution for correcting acoustic music.

3.4.2 Beat detection of live recordings

As the method of section 3.4.1 could not be applied to live acoustic recordings, I focused again on the beat detection algorithm for electronic music, and tried to make it somehow suitable for acoustic music as well.

The main problem of the algorithm for electronic music is that it assumes a perfect tempo through the song, and finds only a fundamental beat for the beat grid.

However, without the aid of metronomes or computers, man can't play music without faults in tempo (or at least very few of them), therefore every single beat has to be found consequently. Tempo information of the songs still can be helpful for finding the beats; only amendments need to be applied during the process.

The algorithms discussed in section 3.1 and 3.2 introduced two completely different solutions for beat detection. As I found beneficial both, I tried to merge the algorithms to get a more robust and reliable solution.

1. Calculating the threshold

First I focused on Frederic Patin's work, discussed in section 3.2. According to his paper, for recording the presence of a beat a large increase in magnitude must be observed. The question is: how large the increase needs to be to find the presence of a beat?

The answer is first we need to define a threshold value in some way, and whenever the signal rises above the threshold we have a beat [7].

A pre-defined constant level undoubtedly can not serve for threshold value, as the sound level of recordings depend highly on the studio environment, mixing, used instruments, etc.

Instead, the first and the second moments of the signal can be calculated, based on the last one second. In my solution for threshold value the linear combination of the moving average and the standard deviation is calculated. During the optimization process I tried to find the best coefficients for the moving average and the standard deviation. Eventually both coefficients have been set to 1, simply defining the threshold value as the sum of the mean and standard deviation of the last second.

Although Patin's work says that every time the signal rises above the threshold value a beat is found, I have modified my algorithm from this point on. Instead of investigating

the signal itself, I calculated the threshold $th[n]$ from the differentiated envelope $d[n]$ and have compared the differentiated envelope $d[n]$ to the threshold signal $th[n]$, cutting off the sections under the threshold value.

In Figure 18 the “cleaned” differentiated envelope $c[n]$ can be seen. The cleaned differentiated envelope $c[n]$ (black signal in Figure 18) only contains peaks above the threshold.

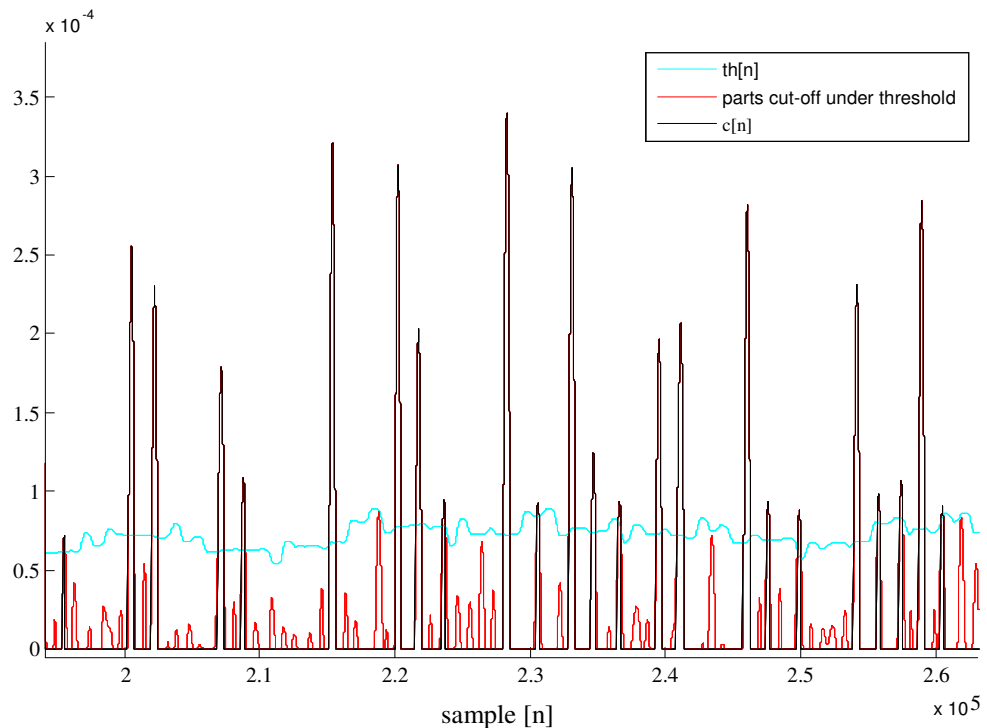


Figure 18: Cleaned up derivative of envelope $c[n]$ (black) and threshold signal $th[n]$ (cyan)

2. Finding the first beat

The next step in the beat detection of electronic music was “sharpening” comb filtering. In acoustic songs beats can be fluctuating in tempo; therefore I omitted the sharpening step, as the sharpening process (in contrast to its name) can smear away the sharp peaks of the cleaned signal $c[n]$.

Instead, to start looking for the beats, first the fundamental beat with the highest peak is selected in the cleaned differentiated envelope $c[n]$ signal.

3. *Creating the beat grid*

As the first beat was found in the signal, the beat grid can be constructed. We know the tempo of the signal, and with that we know where we are expecting the next beats. As we also expect tempo imperfections, we await the next beat within an interval. The radius of the interval in which we expect the next beat is a system parameter. If the parameter is too strict (too low), beats might be left out; if the parameter is loose (too high) false beats might be found. Setting the value of parameter *loose* therefore demands deliberation. The set parameter is: *loose = 1500 samples*, which can detect ~2 BPM error from one beat to another in case of a 60 BPM song.

The beats are sought from each other in varying distances over the length of the track. The oncoming distance where we expect the next beat is calculated by reading the fine resolution BPM vector's corresponding element to the certain location of the envelope. This can be thought of as if the fine bpm vector with its 20-30 elements were stretched to be the same length as the cleaned envelope and to each sample of the cleaned envelope a tempo value belonged. This tempo information is used to find the next beats one after the other.

As we know where to start building the beat grid, where to expect the oncoming beats and how strict the algorithm shall be, we can build up the beat grid of the song.

Figure 19 and Figure 20 display the outcome of the beat detection algorithm. If the algorithm successfully found a beat at one location, a pulse with unity amplitude is recorded in the beat grid vector $b[g]$ (black), if no beats found within the interval the amplitude of the pulse is set to half. If no beats were found, the algorithm carries on looking for the next beat as if the last beat had been at the right position.

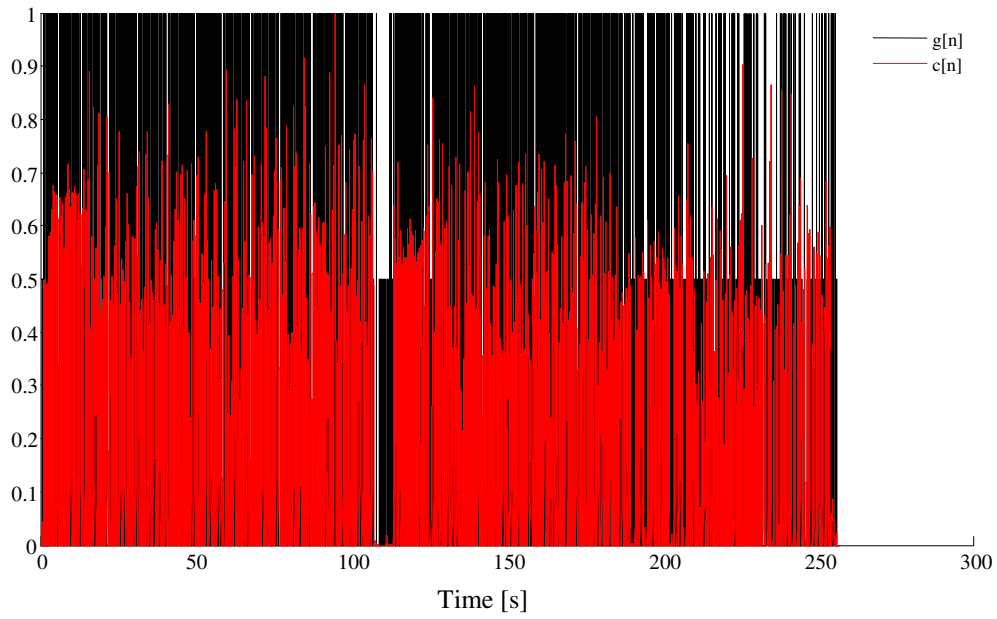


Figure 19: Beat grid $g[n]$ (red) in the cleaned envelope $c[n]$ (black)

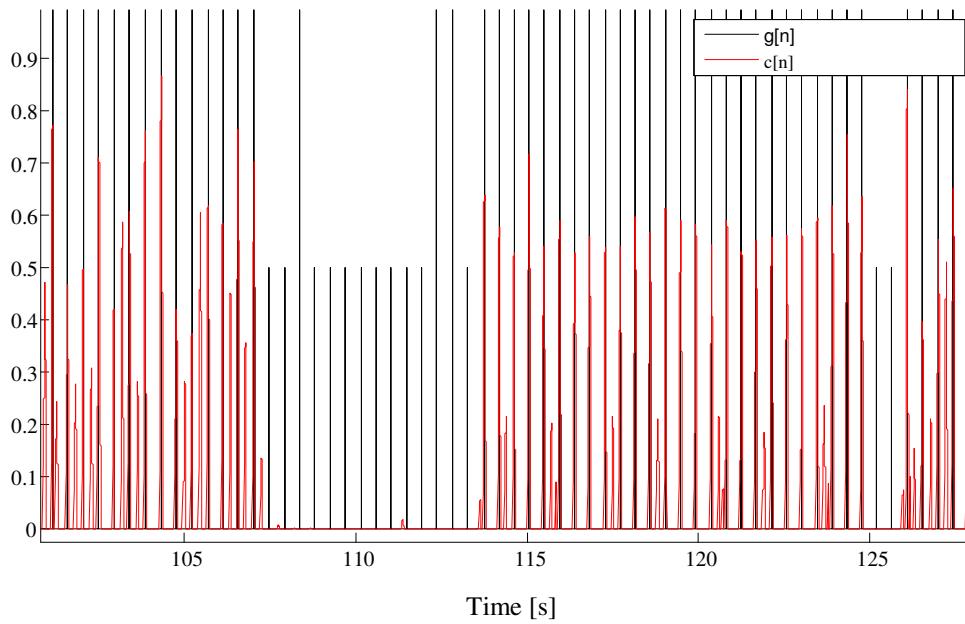


Figure 20: Beat grid $g[n]$ in the cleaned envelope $c[n]$, a closer look

To check the functionality of the beat detection algorithm I created a beep sound and mixed it to the original music file at positions where the beats were found in the envelope. The sound output can be seen in Figure 21 for the song *Bélamühely – Tekno2*.

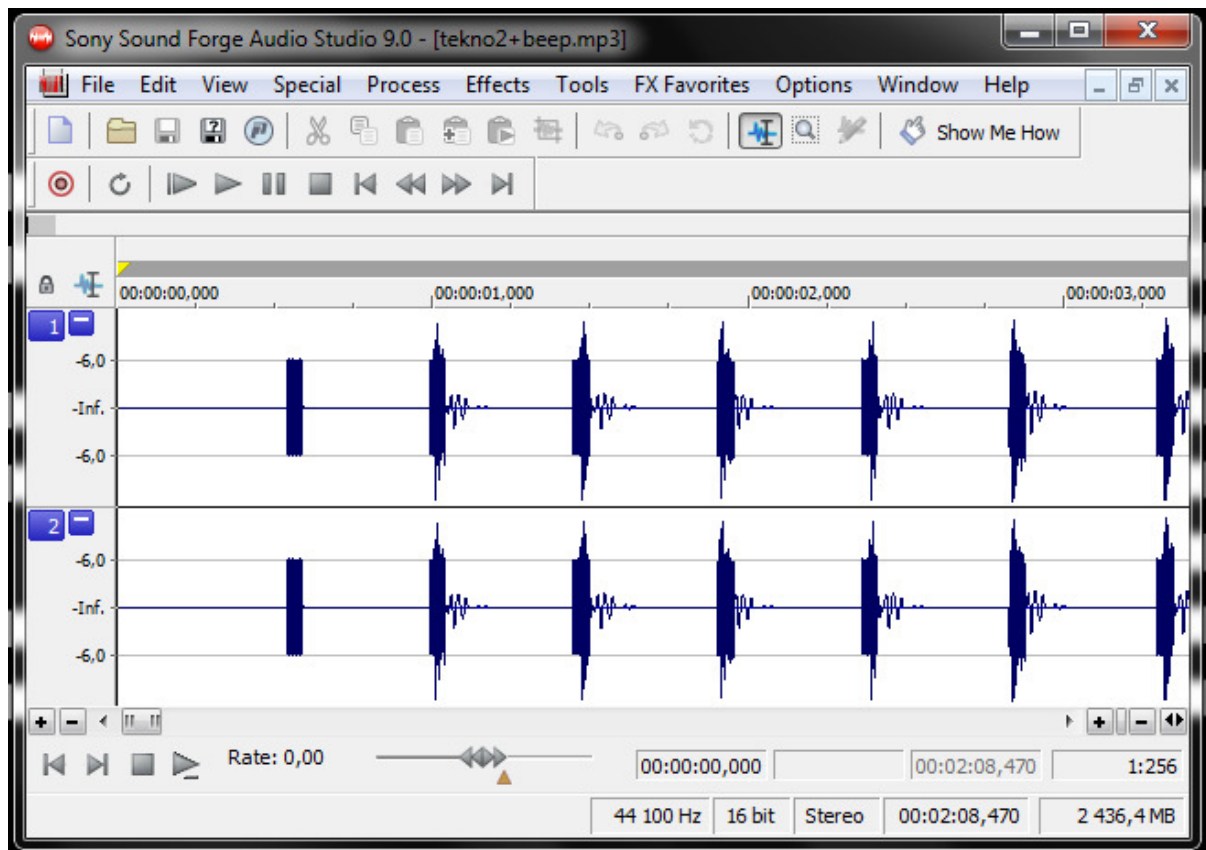


Figure 21: Beeps mixed on beats (song: Bélaműhely – Tekno2)

In Figure 21 the beeps are the rectangle-shaped parts, while the kickdrums of the original song are the decaying sinusoids. The file can be found on the supplement CD.

When creating the beat vector for a track an approximately constant tempo is presumed only with a parameter-given flexibility. If there is a greater difference in the tempo from one beat to another than it is permitted, e.g., a band comes back from a beatdown section at the middle of two beat grid points, the algorithm will fail finding the beats for a long period, as from that point on all the beats will be consecutively at the middle of two grid points. The beats will be consecutively out of scope until the tempo of the song becomes faster or slower than the expectations of the algorithm so much that the beats get into the algorithm's scope again. The width of the scope is two times the *loose* parameter. We can understand the importance of selecting this parameter well; too strict algorithm might lead to losing track, on the other hand being too permissive might generate fake grid points.

According to my experiences on the algorithm, with the *loose* parameter set to 1500, seeking the tempo in 15 second intervals through the song with the accuracy of 0.05 BPM, the beat grid can usually be built successfully.

4 Pre-processing for beat correction

Once the beat grid is ready, we know the locations of beats over time. If the beat grid was absolutely flawless, the distances between beats would be the same length every time. By calculating the differences beat by beat we can see how much they stray in length (Figure 22):

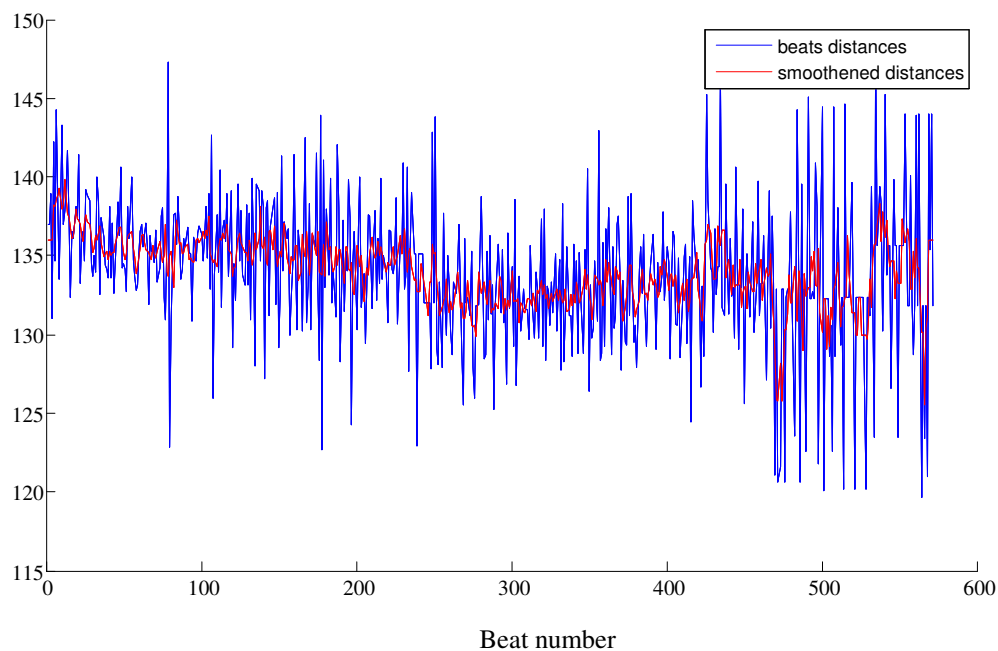


Figure 22: Beat distances over time (exchanged to BPM)

In Figure 22 we can see the beat distances over time exchanged to BPM measures. By blue we can see the calculated, unmodified trend of the beat distances; by red we see its smoothed version. The smoothed red curve is carried out by taking the mean of the adjacent beat distances (bar lengths) in 2 beat radius, 5 samples altogether.

Once we know the positions of the beats, our last step in correcting the sound file in tempo is to stretch the beats to even lengths over the song. For that the length of the mean beat distance has to be calculated first, and then we can adjust every beat to be mean length long.

In Figure 22 we can see that the measured tempo through the song is heavily fluctuating. Correcting it beat by beat would mean large modifications in the beat lengths as it strays from the mean value in a big extent. This is why the red curve was carried out as well.

When taking the red signal, we suppose that the found beats are not exactly at the locations where we originally found them, but at locations prescribed by the smoothed, red curve. By summing up (discrete integrating) the red signal, we gain the beat positions of the smoothed beat grid.

The aim of smoothing the beat grid is to reduce the extent of the necessary modification in the next step, where the time stretch algorithm tries to make the tempo correction as inaudible as possible. The lesser the needed modification is, then better output sound quality we gain eventually. Furthermore, the aim of the algorithm is to reduce the tempo fluctuation throughout the song, not correcting every single beat. For that the smoothed signal is used. As all the information we needed about the beat locations is known, the last step in the project is to actually produce our output, the tempo corrected version of the input file.

5 Time stretching

Time stretching is the method of changing the tempo (or length) of an audio signal without modifying its pitch. Pitch scaling or pitch shifting is right the opposite: the process of changing the pitch without modifying the speed [12].

In general, time stretching algorithms are often used to match the pitches and/or tempos of two pre-recorded clips for mixing when the clips cannot be re-performed or resampled. They are also used as sound effects.

5.1 Resampling

The easiest way to change the length or pitch of a digital audio clip is to resample it on a new frequency and play it back on the original frequency.

Resampling is an operation that rebuilds a continuous waveform from its samples and then samples the waveform again at a different rate. When the new (resampled) signal is played back at the original sampling frequency, the audio clip sounds higher or lower (producing a monstrous voice or the infamous chipmunk sound). This is analogous to speeding up or slowing down vinyl records on a turntable.

One way to perform resampling is to apply the Lanczos resampling. [9] The Lanczos resampling or Lanczos filter is typically used to increase the sampling rate of a digital signal, or to shift it by a fraction of the sampling interval. It is also used for multivariate interpolation, for example to resize or rotate digital images.

Mathematically, Lanczos resampling is a procedure that maps a stretched windowed sinc function on each sample of the given signal, positioning the central hump of the sinc onto the sample. The sum of these various sincs is then evaluated at the desired points. In Figure 23 a digital input signal can be seen by black dots, containing samples at the integer positions [10].

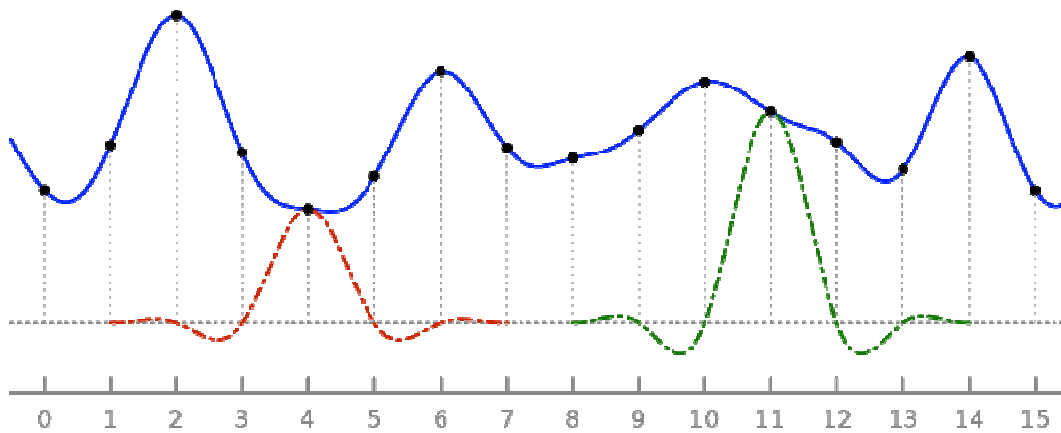


Figure 23: Resampling with sinc functions (Figure taken from (3))

By red and green colors we can see sinc functions positioned on samples 4 and 11 having a size parameter of 3, reaching until the distance of 3 samples into both directions (this property is also known as to be seventh order). In Figure 23 by blue we can see the continuous resampled signal, can be carried out by the sum of all the sines put on the samples (by black). If we wanted to know the value of the resampled signal at a fractional point somewhere between two samples, we first would need to calculate the value of the 7 adjacent sines at that certain position and eventually sum them up.

5.2 Phase vocoder

One way of stretching the length of a signal without affecting its pitch is to use a phase vocoder after the work of Flanagan, Golden, and Portnoff, also discussed by Udo Zölzer. [11] [12]

The phase vocoder handles sinusoid components well, but it may produce audible “smearing” on beats at all non-integer compression rates. The phase vocoder technique can be used to perform pitch shifting, time stretching, chorusing, and it is a widely used effect for online voice modulating, e.g. with the quite popular Auto-Tune software.

The three main steps of creating a phase vocoder are:

1. The discrete Fourier transform of a short, overlapping and windowed section of samples is calculated
2. The required modification on the signal is applied in frequency domain, processing the Fourier transform magnitudes and phases.
3. We perform the inverse discrete Fourier transform by taking the inverse Fourier transform on each chunk and adding the resulting waveform chunks by the overlap and add algorithm (see 5.3) [13].

The block diagram of the phase vocoding procedure can be seen in Figure 24.

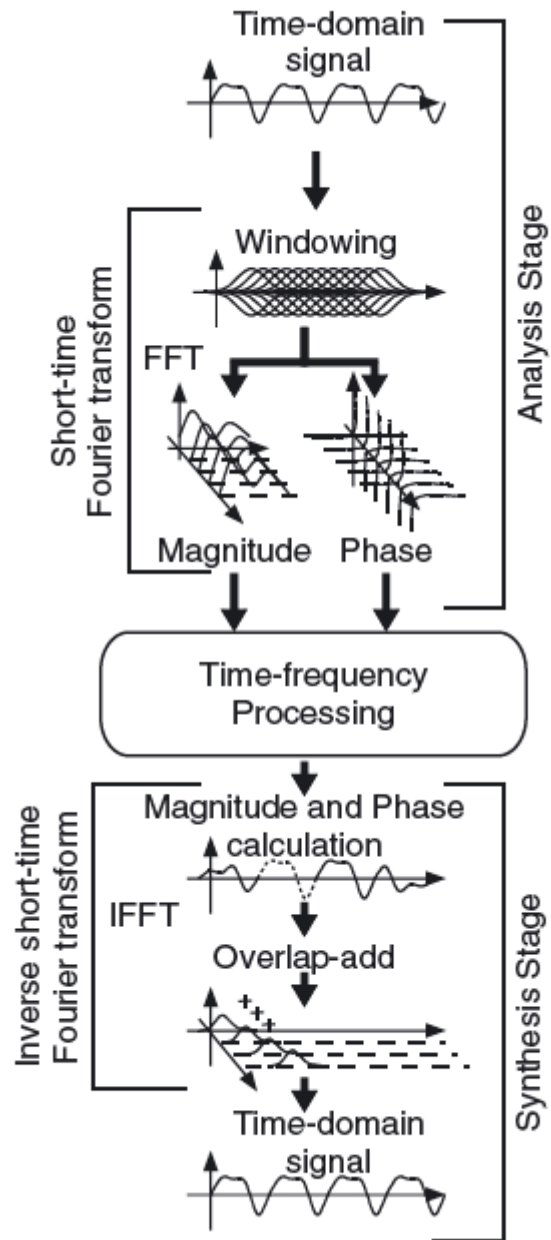


Figure 24: Steps of performing phase vocoding (Figure taken from [12])

5.3 The Synchronous Overlap and Add (SOLA) algorithm

5.3.1 Key features of the SOLA algorithm

The synchronous overlap and add algorithm is a time stretching method working in time domain. The basic idea of the algorithm is to segment the music signal into short overlapping segments. At the ends of segments where the segments overlap, we shrink or dilate the overlapping sections to speed up or slow down a track. This is the so called Overlap and Add method. The drawback of this method is that during the overlapping sections where the crossfading occur the sound distorts, producing audible, considerable noise in the signal because of the phase distortion. (See Figure 25)

The Synchronous Overlap and Add method positions the overlapping sections onto each other in according to gain maximum correlation at crossfading, minimizing the noise of the OLA algorithm. This is done by finding the peak of the autocorrelation of the overlapping sections, and positioning the overlapping sections so, to gain this maximum autocorrelation. [12] [14]

The number of autocorrelated samples and the number of calculated autocorrelations are system parameters.

During offline time stretching an output file is created; into the output file the input signal sections are written pushed or pulled together. Windowing is needed to keep the signal level constant at overlaps.

In Figure 25 we can see two sections cut out and windowed from a pure sinusoidal input signal. The red, framed areas show the overlapping sections that are going to be mixed on each other during the time stretch.

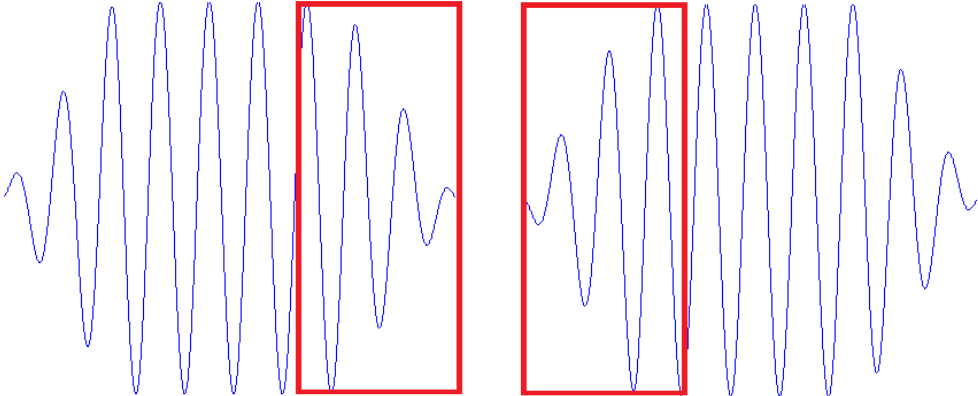


Figure 25: Adjacent sections of a sinusoid signal after windowing

During the OLA algorithm no synchronization is performed at the overlapping sections. This might lead to distortion in the output signal. In Figure 26 we can see the two sections of Figure 25 mixed together. In the overlapping section we can see how much the signal has been distorted because the mixed parts did not match in phase.

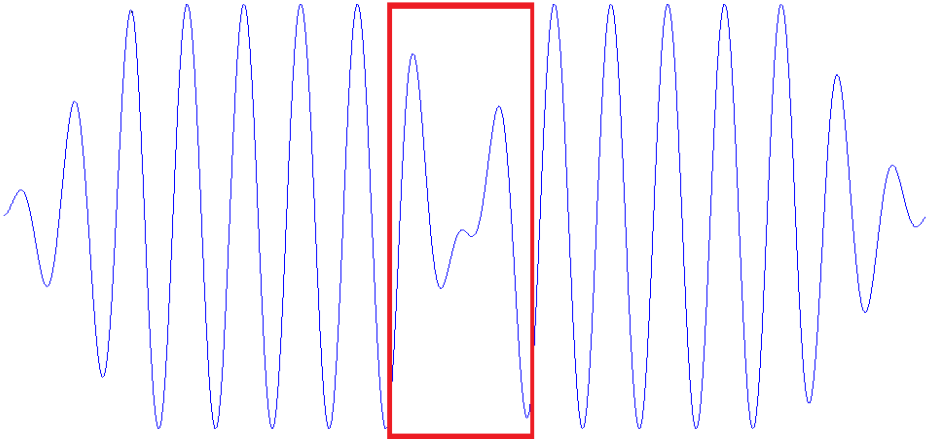


Figure 26: Adjacent sections mixed with OLA algorithm

The SOLA algorithm solves this issue by the discussed synchronization method calculating the best matching for the overlapping sections.

In Figure 27 we can see the two sections when they are selected by the SOLA algorithm in according to match them in phase.

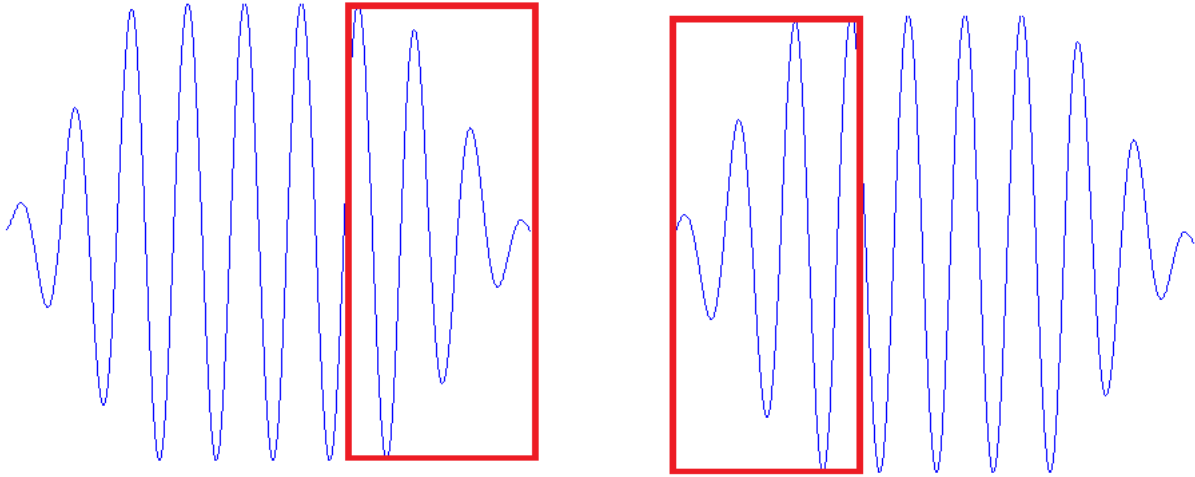


Figure 27: Adjacent sections selected by the SOLA algorithm

In Figure 28 we can see the two sections of Figure 27 mixed together. We can see that in the case of a single sinusoidal signal the algorithm is able to mix together two sections without any faults. Of course for general signals the results are not perfect.

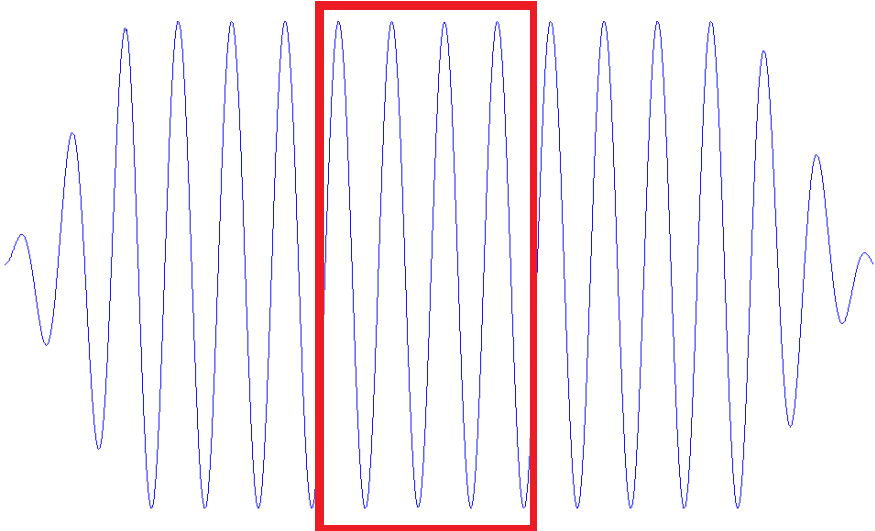


Figure 28: Adjacent sections selected by the SOLA algorithm mixed together

The SOLA method demands less calculation than phase vocoding as no transformation is needed into frequency domain; every calculation is performed in time domain. However, SOLA fails when because of the complicated harmonics in the signal it can not find a good match between the sections. Complicated harmonics mainly occur in orchestral pieces and heavily echoed and effected sounds.

On the other hand, SOLA provides the best results for single-pitched sounds like voice or monophonic instrument recordings. See Figures above.

In my project I applied SOLA algorithm as time stretching for producing good quality output on the expense of relatively low implementation efforts. Nevertheless, I had to find the best parameters for window length (w), window shape, autocorrelation size (n) and number of calculated autocorrelations (r) in according to gain the best sound quality for SOLA as possible. By listening to several songs from many genres of instrumental and electronic music I had determined the following parameter values:

Ideal SOLA parameters in samples	
Window length (w):	2048
Correlation size (n):	256
Seek radius (r):	256

Table 3: Determined ideal SOLA parameter values

My choice on SOLA was also influenced by high-end commercial audio processing packages, as they also combine the SOLA algorithm with other techniques in according to produce high quality time stretching. Zplane's Elastique Pro Time Stretching that is used in most of the DJ / LivePA softwares such as Ableton, Cubase, Mixvibes, Sound Forge, Traktor and Torq use variants of overlap and add method as well. [15]

5.3.2 SOLA algorithm for stereo signals

As discussed in 5.3.1, SOLA may produce good audio quality where its scope is only on a single *mono* signal. However, music recordings are mainly recorded in stereo; therefore I had to investigate the possible methods for finding the best choice for a stereo SOLA application. I implemented four algorithms for modifying the mono SOLA method to make it stereo (the demo music samples can be found on the supplement CD):

1. Applying the mono SOLA algorithm to both channels independently, then concatenating the two channels:

This version fits the channels the best one by one. Although the channels sound great separately, the stereo effect of the signal is highly damaged by this solution. Listening to the output makes the feeling that instruments and vocalists travel around the listener over time, as the stereo effect is based on constant delay between the two channels which has now damaged by the algorithm. This version therefore is not acceptable. The following versions keep the stereo effect.

2. Creating a mono signal from the stereo by calculating the mean of the stereo signal, then applying the mono SOLA algorithm to the mean signal.

The window locations are found in the mono signal, but the SOLA rebuilding is performed using the stereo signal as input.

3. Applying the mono SOLA to one channel and modifying the second channel according to the first channel:

Mathematically, the second channel's quality is not as good as the first channel's quality, although it's not audible.

4. The SOLA performs the correlation on the stereo signal, compromising between the two channels, finding their common best fit. This is the truly stereo SOLA.

The methods 2, 3 and 4 all produce good audio quality. For finding the best model out of the three formal listening tests should be performed. I personally find model 4 somewhat better than the other two, therefore selected it to the final edition of the algorithm.

6 Implementation in C/C++

Once the design of the algorithm was ready in MATLAB, I tried to reduce its computation time by implementing the algorithm in C/C++.

First, I separated the certain steps of the algorithm to implement them as MATLAB functions. These steps are the following:

1. Reading the input signal
2. Decimation
3. Creating the envelope by rectifying and low-pass filtering
4. Differentiating-Rectifying
5. Thresholding
6. Comb filtering (rough, 1BPM accuracy)
7. Comb filtering (high resolution, 0.05BPM accuracy)
8. Finding the beat grid in the cleaned envelope
9. Beep sound generating
10. Beat grid pre-processing
11. Producing the corrected signal by SOLA
12. Writing the output signals:
 - a. Corrected output
 - b. Metronome beeps
 - c. Corrected output + metronome
 - d. Original input + metronome

I have ported steps 2 – 9 from MATLAB to C/C++. The beat detection procedure of the project is the computationally most intensive section, taking about 75-80% of the needed calculation time. Therefore I focused on these steps, and tried to improve the performance of the tempo and beat detection functions first. Time stretching by SOLA didn't demand speed boosting sorely, its performance was satisfactory even in MATLAB.

Furthermore, file I/O operations are much more convenient in MATLAB than they are by calling low level C/C++ audio libraries, hence I kept using `wavread()`, `mp3read()` and `wavwrite()` for reading and writing output files.

6.1 Mex files

To develop the C/C++ equivalents of the MATLAB functions I used Microsoft Visual Studio 2010 environment, the functions have been implemented in Matlab Exchange (mex) file format, using the compiler of VS2010 [16]. The Matlab Exchange format allows users to call C/C++ functions from MATLAB [17]. This possibility has proven very helpful through the development process as I could compare the outputs of the corresponding MATLAB and C/C++ functions by calling both functions with the same parameters from MATLAB.

A mex file consists of two functions: the actual C/C++ code, and a wrapper function, called the *gateway function*. The gateway function serves as interface between the two environments by transforming the MATLAB data types to C/C++ data types and calling the C/C++ function with the transformed data. The signature of a gateway function is the following:

```
/* The gateway function */  
void mexFunction( int nlhs, mxArray *plhs[],  
                  int nrhs, const mxArray *prhs[])
```

The function parameters are:

- `nlhs`: number of output parameters
- `*plhs`: pointer to the array of output parameters
- `nrhs`: number of input parameters
- `*prhs`: pointer to the array of input parameters

In the mex files first the output data allocation and the input data conversion is performed (if needed). Next, all the parameters are passed to the C/C++ function at its calling. The return values of the C/C++ function are passed back to MATLAB through the gateway function again.

During the development once a new C/C++ function was ready in my project; I created a wrapper function calling the function along with its MATLAB equivalent with the same parameters, and plotted the difference of the two outputs to compare their performance.

6.2 Developing the C/C++ functions

6.2.1 Decimation

During the porting of the project I mainly had to re-write my own code from MATLAB to C/C++. All the implemented functions that contain no MATLAB built-in function calls perform absolutely the same, producing zero difference between the MATLAB and C/C++ equivalents.

However, in a few functions MATLAB built-in functions have also been called, that had to be implemented in C/C++ as well. One of them was the *decimate()* function. By default, MATLAB calls an 8th order Chebyshev Type I filter for lowpass filtering, whereas in my *decimate6()* function I applied a 6th order Butterworth filter (decimate6: six for decimating the input to its sixth). To check the performance of my C/C++ decimate6() function I wrote a simple *decimate6()* function in MATLAB as well, applying the same Butterworth filter coefficients in it that I also used in my C/C++ function. The comparison of the two function outputs can be seen in Figure 29. The error between the results is so small that we can't see the difference between them, no matter how close we zoom into the plot.

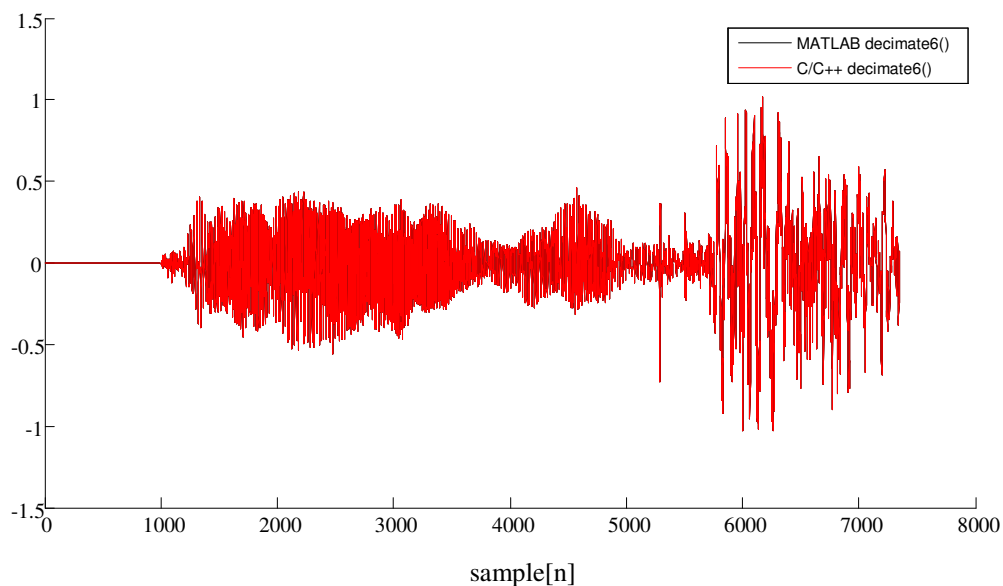


Figure 29: Outputs of MATLAB and C/C++ decimate6() functions

In Figure 30 the difference between the outputs can be seen. The difference has a magnitude of 10^{-13} . Into my C/C++ function I only typed in 15 digits of the Butterworth filter coefficients, hence the small difference between the two outputs.

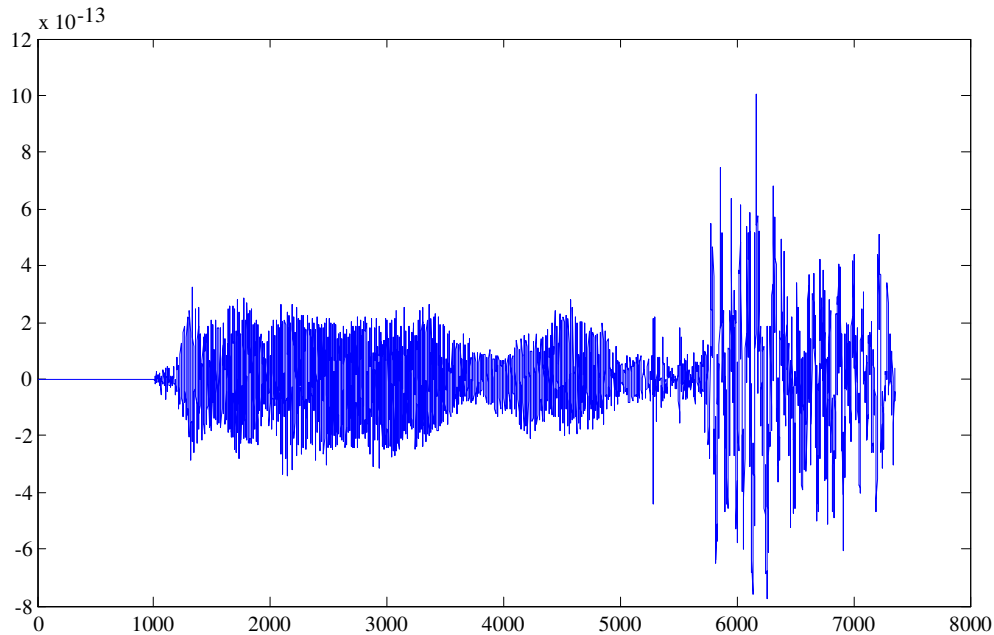


Figure 30: Difference of the MATLAB and C/C++ decimate6() functions

6.2.2 Comb filter

The comb filter of the tempo detection step in the project calls the built-in fast Fourier transform `fft()` function to transform the envelope and the comb filter signals into frequency domain. The transformed signals are then multiplied, and the energy for each comb filter output is calculated.

The comb filter consists of a series of pulses. (See section 2.2.5.) Convolution of the envelope with a shifted pulse can be thought of as if a shifted version of the envelope were added to itself. Convolution of the envelope with a series of pulses can be thought of as if the envelope was summed with its shifted version as many times as many pulses the comb filter consists of.

In case of finding the rough tempo value 60 pulses, in case of finding the high resolution tempo 15 pulses are used in the comb filters. These correspond to adding shifted versions of the envelope to itself 60 and 15 times, respectively.

The output of the two fine resolution comb filter model outputs can be seen compared in Figure 31 (both versions written in MATLAB).

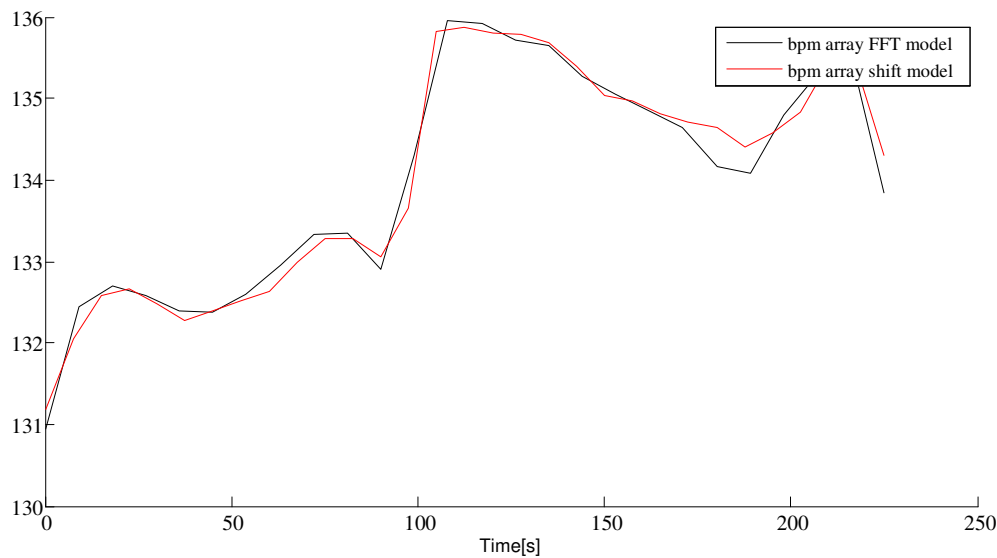


Figure 31: Comparison of MATLAB FFT model and MATLAB shift model tempo predictions

The difference between the models comes from two factors. First, the `fft()` function needs envelope section lengths being powers of two. The length of the analyzed envelope sections is ~17.5 seconds for the `fft` model and 15 seconds for the shift model. This results in different number of sections by the certain models (For Figure 31 the resulting output signals have been scaled accordingly).

Second, the different models use different number of pulses in their comb filter. The number of pulses in the FFT model was 17, while it is only 8 in the shift model.

The difference between the models is negligible, as they both only serve as prediction for finding the beats in the envelope; the beat grid is constructed by finding the exact locations of the beats.

We can see that the shift model can be an alternative of the FFT model. The advance of replacing the FFT model for the shift model is that it reduces both the computation time and coding efforts. To analyze the tempo of a 4 minute song takes about 60 seconds for the FFT model and 50 seconds for the shift model. (On Windows 7, Matlab R2010a, Intel i3 M350 CPU @ 2.27GHz, 2GB RAM).

For the next step I ported the shift model tempo detection from MATLAB to C/C++. The comparison of the MATLAB and C/C++ shift model tempo detection functions can be seen in Figure 32.

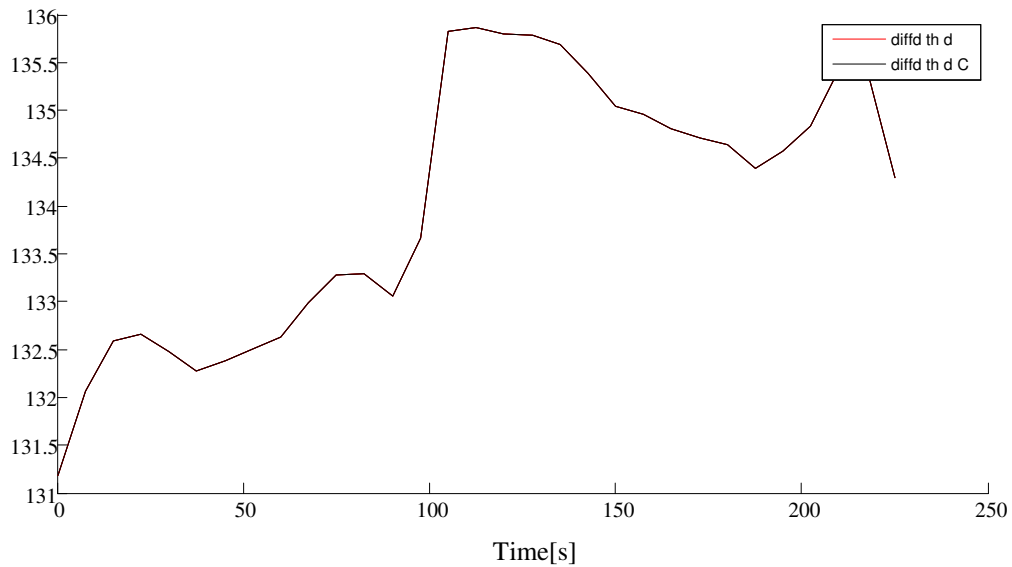


Figure 32: Comparison of the MATLAB and C/C++ shift model tempo detection models

The output is what we expected; without calling built-in MATLAB functions the output can be reproduced exactly after the porting procedure. The difference of the outputs can be seen in Figure 33.

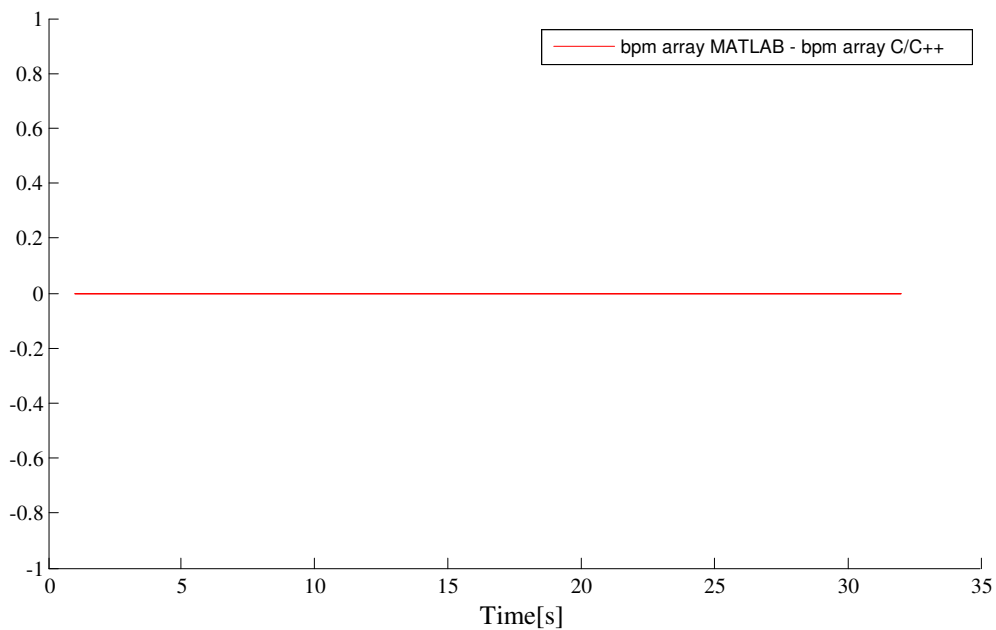


Figure 33: Difference of the MATLAB and C/C++ shift model tempo detection outputs

By porting the tempo detection function from MATLAB to C/C++ the computation time of this function could be reduced from 50 seconds to 8 seconds.

6.2.3 Computation time comparison

The following, non discussed functions produce the same output as their MATLAB equivalents:

- `thresholding_c`,
- `diff_rect_c`,
- `beat_seeking_c`,
- `beep_generate_c`
- `sharpening_c`

The comparison between the corresponding MATLAB and C/C++ runtimes for a 4 minute song can be seen in Table 4:

	MATLAB	C/C++
Decimation	2.3	1.2
Diff-Rect	0.06	0.02
Thresholding	0.31	0.17
Comb filter (rough)	8	6
Comb filter (high res)	50	8
Beat Seeking	0.04	0.02
Sharpening	0.2	0.1
Beep generate	0.04	0.02

Table 4: Computation times [s] for MATLAB functions and their C/C++ equivalents

We can see that apart from the comb filter (rough) function the computation time has been considerably reduced by porting the functions to C/C++. In the case of the comb filter (rough) the C/C++ shift model (summing the signal 60 times in C/C++) could not reduce the computation time considerably, compared to the MATLAB FFT model (2 x FFT, then multiplication in MATLAB).

By porting the functions, in case of a 4 minute song the beat detection computation time could be reduced from 65 seconds to 20 seconds, the global computation time from 80 seconds to 35 seconds. By porting the SOLA time stretching method as well, the needed computation time could be shortened even more, this is a further possibility development.

7 Graphical User Interface

Once the algorithm design and implementation was ready, I designed and added a graphical user interface to the project. The gui has been implemented in *guide*, the MATLAB gui editor tool [18]. In *guide*, common gui elements can be easily added to the main panel of the gui, such as textboxes, pushbuttons, axes, menus, lists etc. First I added the desired elements to the gui and designed the welcome panel, it can be seen in Figure 34.

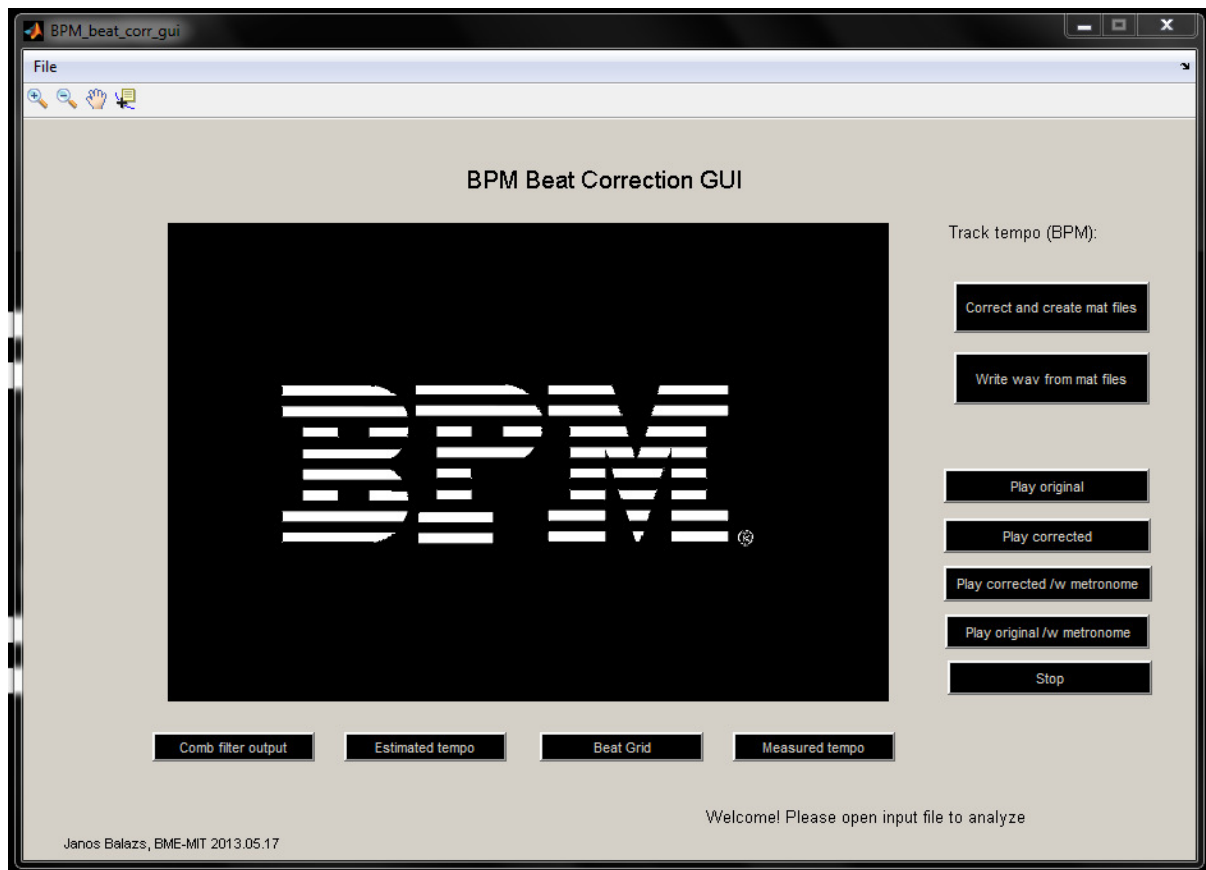


Figure 34: GUI window after startup

An input file can be loaded by opening the File menu and selecting Open ...



Figure 35: Opening the input file

After selecting the input file, the program automatically analyzes it, producing four plots. The plots can be selected for display by clicking the buttons on the bottom of the main display.

The four plots are:

- Comb filter output button: The output of the rough tempo calculation, based on the first minute of the song
- Estimated tempo button: The estimated tempo of the song over time
- Beat grid button: The found beats in the cleaned up, differentiated envelope
- Measured tempo button: The measured tempo of the song over time, based on the found beat locations.

The plot of rough bpm estimation can be seen in Figure 36. The maximum point of the diagram is indicated by a green stem, its value is displayed in the right top corner of the GUI

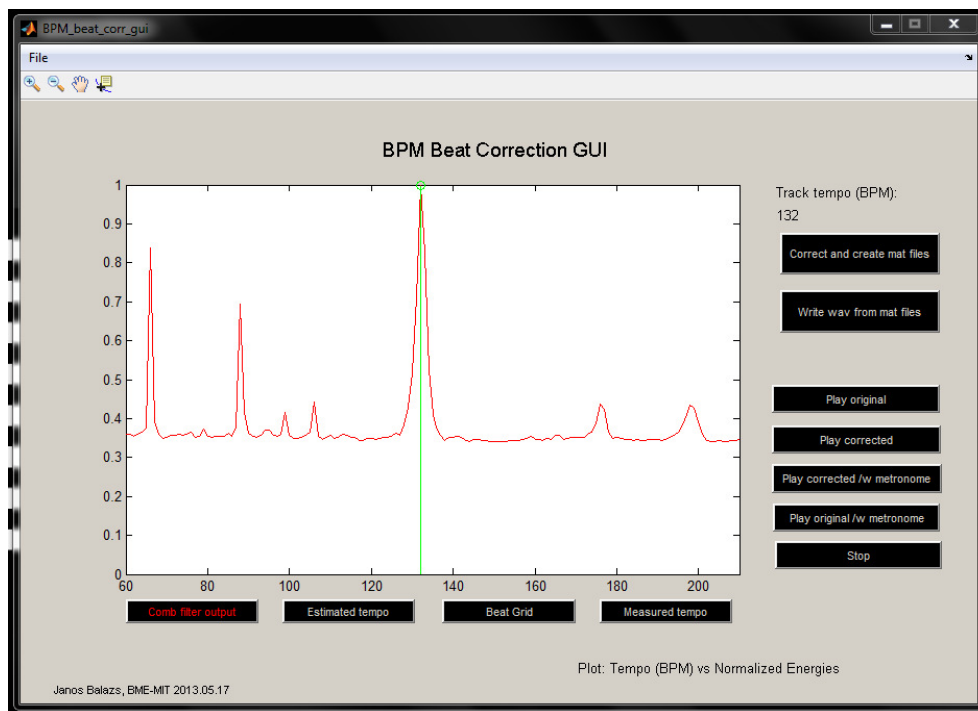


Figure 36: Estimated tempo plot

The fine resolution tempo estimation over time can be seen in Figure 37.

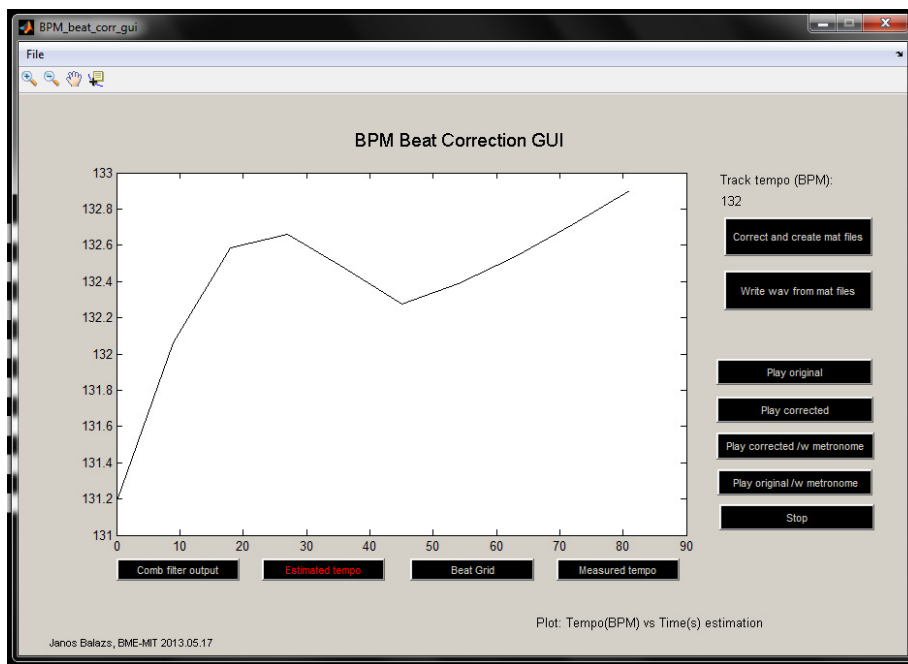


Figure 37: Fine tempo estimation over time

The beat grid plot can be seen in Figure 38. The magnifying glass tool can be useful for investigating the beat grid. The tool can be found right under the File menu.

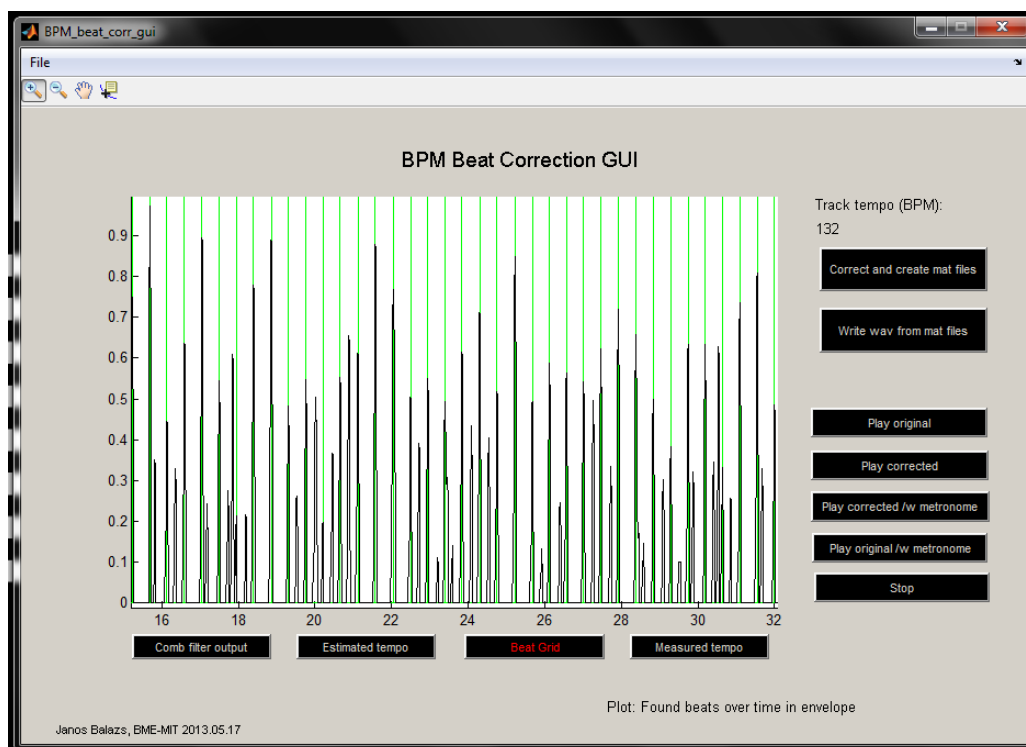


Figure 38: Beat grid plot for the song

The measured beat distances over time can be seen in

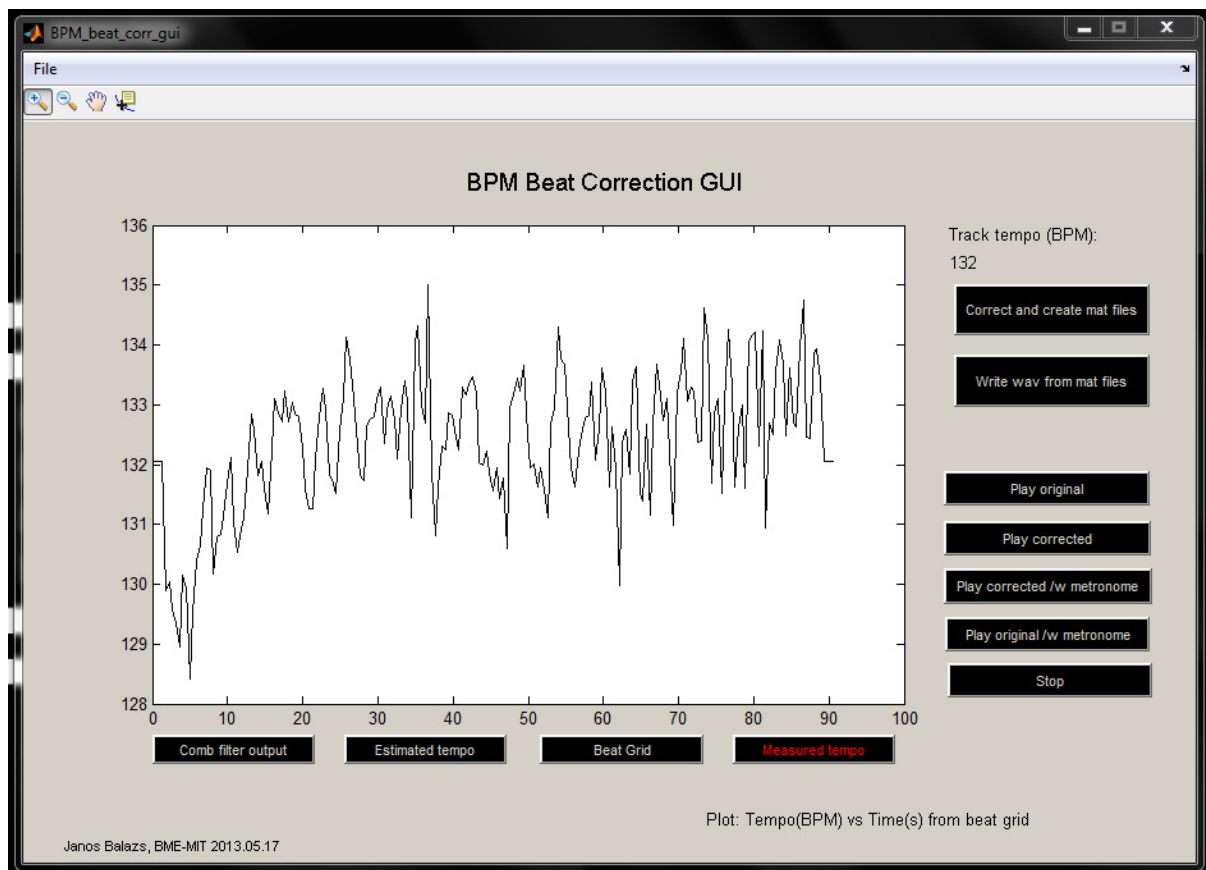


Figure 39: Beat grid plot for the song

Once the analization is ready three mat files are written from containing the following signals:

- original signal
- corrected signal
- corrected signal mixed with its corresponding metronome beeps
- original signal mixed with the metronome beeps

In the beginning I tried to write these signals into wav files straight away, but unfortunately when calling the `wavwrite()` command for signals longer then 5 million samples, it usually returned with *Out of memory error*. Once getting this error message, no operations can be performed in MATLAB until it gets restarted. By first storing the workspace variables in file by the `save()` command, I clear to whole workspace. By clicking the *Write wav from mat files* button the mat files are read back into the clear workspace, the signals are written into wavs

and then the workspace is cleaned again. This procedure is performed for all four signals, one by one.

After the `wavwrite()` function finished writing the output signals, they can be played back by clicking the buttons accordingly:

- *Play original*
- *Play corrected*
- *Play corrected with metronome*
- *Play original with metronome*

By clicking the *Stop* button, any playback can be stopped.

The graphical user interface can be closed by either clicking on the red X of the top right corner of the window, or by selecting Close from the File menu.

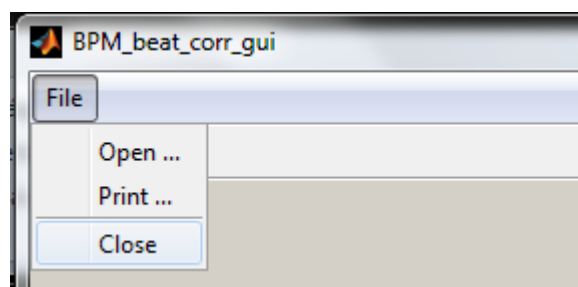


Figure 40: Closing the GUI

8 Results and possibilities of development

8.1 Listening to the outputs of the program

I tested the functionality of the program on tracks that could possibly contain tempo inaccuracies over time. Unfortunately, when MATLAB runs the graphical user interface, only 90-100 seconds of the songs can be read by wavread. Therefore I created ~100 seconds long cuts of the songs. For each song to be analyzed the three waveform outputs are created; the corrected signal, the corrected signal mixed with beeps and the original signal mixed with beep, as discussed in chapter 7.

In the radio show of MR2 Akusztik, groups play their own and fellow bands' songs unplugged [8]. On the official YouTube channel of the series all the archived episodes of the show can be watched. For analysis, I selected three commonly known cover songs:

- Csík Zenekar - Csillag vagy fecske
- Kesh – Mizu
- Punnany Massif – Telik

In case of the songs of Csík Zenekar and Punnany Massif, the corrected output and the mixed beep sounds are perfectly in beatmatch. In case of the song Kesh - Mizu a too long breakdown section (too long hiatus) misleads the beat detection module, therefore in the output the beep sounds fall out of sync following the actual beats of the song. By clicking on the Beat Grid button of the GUI we can see how the program has set the beat grid points and where it fails following the actual beats of the song. See the supplement CD for the wav files. After analyzing the MR2 Akusztik songs I ran the algorithm on Longview of Green Day. In this song, sections with various tempos can be found (discussed in section 2.3.2). Between the tempo of the verse and the chorus there is a difference of 6 BPMs. During the correction of the song, the algorithm eliminates this type of “dynamics” from the signal and sets a constant tempo instead, which is the mean tempo of the input.

I also ran the algorithm on Bélaműhely's track Tekno2, a live recorded techno song, performed on do-it-yourself instruments [19]. The song is quite monotonic and repetitive; the algorithm corrected the 4-5 BPMs of inaccuracy in the song precisely.

Furthermore, I tested the algorithm on beat-boxing as well. The analyzed track was an improvisation of Skiller, the beat-boxing world champion of 2012. The 100 seconds sample of his performance contains dubstep and techno, with tempos at 70 and 140 BPM, respectively. The algorithm always starts constructing the beat grid from the point with the highest energy in the song. In this beatbox signal this point is unfortunately not a beat, but a position in the middle of a bar, where an unaccented “hi-hat” sound is imitated. The algorithm latches on this sound, and corrects the song accordingly. Although the beeps have not put on the beats, but on a sound between them, the algorithm still manages to correct the beat grid well, only there is a phase shift between the beats and the beep sounds in the output signal.

At 0:53 in the song, the artist changes from techno to dubstep, and falls out of tempo for one bar. This causes the algorithm to latch on the actual beats of the song, and from that point on, the beep sounds are mixed to the right positions.

Resolving this issue, i.e. to find the end of a beat grid is another possibility of development. All the input and output sample files can be found on the supplement CD.

8.2 Summary

At the end of the project my experiences on the algorithm were the following:

1. On the market of DJ softwares there is a lack of programs being able to analyze and correct tempo inaccuracies of acoustic songs. None of the DJ programs listed in the introduction are able to perform this process online or offline, see [1] and [2].
2. The reason for DJ softwares lacking the feature of correcting the beat grid can be explained by the focus of the DJ softwares: they are designed to deal with electronic music. The beat grid of electronic music is accurate, there is no need trying to correct it; any effort only makes the beat grid worse. See section 3.3.
3. Based on the works of Eric D. Schreier and RICE University, Houston, I could develop a tempo detection algorithm that performed better than the previous implementation of the algorithm. See comparisons in section 2.3.

4. Beat detection can be performed with or without relying on the tempo information, according to Eric D Schreier and Frederic Patin, [3] and [7]. In my work I tried to merge the two solutions to get a more robust algorithm. See sections 3.1 and 3.2.
5. When finding the beat grid of the song, the degree of flexibility in the algorithm shall be elaborated to avoid fake hits (false alarms) or target miss. See section 3.4.1.
6. The implemented beat detection algorithm works well for songs that stay between the flexibility limits in tempo. On the other hand, it is not able to find the points where the beats exceed the bounds of the scope, in which the algorithm looks for the beats. See section 8.1.
7. When performing SOLA time stretching, it is very important to keep the stereo effect of the song. I have implemented three types of SOLA time stretching, all of them performing very well. The outputs of the corresponding versions can be found for comparison on the supplement CD.
8. Visual Studio 2010 allows programmers to connect VS 2010 to MATLAB, and to compile mex files. By mex files C/C++ functions can be called from MATLAB environment.
9. Porting the MATLAB functions to C/C++ the computation time could be reduced from 50% up to 80% in case of certain functions. For computation time comparisons see section 6.2.3.
10. The MATLAB wavread(), wavwrite() and save commands require too much free memory space and often return with “Out of memory error”, even though the workspace is empty, and there is available system memory space in Windows 7.
11. The MATLAB guide graphical user interface editor is a convenient way of designing a MATLAB gui.

8.3 Possibilities of development

Although the program works in general, there are a few details that could be developed:

- The main issue is that the program can not detect if the track being analyzed falls out of tempo for one bar i.e. there is a bar which is significantly shorter or longer than its environment.
- If there is a great tempo change in the song, such as a switchup from 140 to 148 BPM, the algorithm can't analyze and correct these sections separately. In case of Green Day – Longview this type of tempo switch is eliminated, although a better solution would be if the 140 and 148 BPM sections were found and corrected separately.
- The graphical user interface could display more data about the process being performed.
- The output could be created in compressed audio format, such as mp3 or m4a.
- In order to read, process and write longer audio data than 90-100 seconds, all the MATLAB functions could be replaced by C/C++ functions. Not only longer data could be processed by the algorithm, but computation time could be reduced even more.

9 Acknowledgment

I can not thank enough my family for helping me during my 7 years of attending university, way beyond their possibilities. Without their selflessness I could not focus 100% on my duties and could not have achieved all that I have by now.

I also would like to thank my supervisor Dr. Balázs Bank for supporting my work with his ideas and expertise through four semesters.

10 Readme.txt file of the supplement CD

Contents:

0. Install notes
1. Running the GUI
2. Running the console application
3. Listening to demos
4. Documentation

0. Install notes

Copy the content of the supplement CD to a folder on hard drive. This folder is denoted by <install dir>.

Required version of MATLAB: MATLAB R2010a or later (win32).

1. Running the GUI

1.1 Run <install dir>\supplementCD\3 - MATLAB project\1 - GUI version\BPM_beat_corr_gui.m

1.2 Select file for analysis by File -> Open

1.3 Wait until analysis plots appear

1.4 Click 'Correct and create mat files'

1.5 Wait until 'Corrected signal written to .mat files' message is displayed in bottom right corner

1.6 Click 'Write wav from mat files'

1.7 Wait until 'Wav files ready' message is displayed in bottom right corner

1.8 Select file for playback, click 'Stop' if swapping selection before playback ends

1.9 Exit application by File -> Close

1.10 Restart GUI after each session (song) to avoid MATLAB memory errors.

1.11 Restart MATLAB if 'Out of memory' error occurs

2. Running the console application

2.1 Run <install dir>\supplementCD\3 - MATLAB project\2 - console version\bd014_v5_for_figures.m

2.2 If MATLAB returns 'Out of memory error' before line 404:

2.2.1 Close MATLAB

2.2.2 Open <install dir>\supplementCD\3 - MATLAB project\2 - console version\bd014_v5_for_figures.m

2.2.3 Type in MATLAB console: load workspace_for_time_stretch

2.2.4 Run line 404 separately by selecting the line, then pressing F9

3. Listening to demos

All corrected songs along with their originals can be found in:

<install dir>\supplementCD\2 - demo songs for listening\

The folders contain five waveform files:

3.1 original_signal.wav : the original signal

3.2 sola_output_st_compromise.wav: the corrected signal using the compromise model stereo SOLA

3.3 sola_cmprms+metronome.wav: the corrected signal using the compromise model stereo SOLA mixed with metronome beeps

3.4 raw+metronome.wav: the original signal mixed with metronome beeps

3.5 metronome_beeps.wav: metronome beeps

4. Documentation

The MSc Thesis documentation can be found in format .doc and .pdf in:

<install dir>\supplementCD\1 - documentation\

11 List of references

- [1] 2013 DJ Mixing Software Comparisons and Reviews
<http://dj-mixing-software-review.toptenreviews.com/>
- [2] Best DJ software: 10 apps we recommend 2013
<http://www.techradar.com/news/software/applications/best-dj-software-10-apps-we-recommend-1134266>
- [3] Eric D. Scheirer - Tempo and beat analysis of acoustic musical signals, 1996
<http://mykal.free.fr/synthesis/detectrhythm.pdf>
- [4] Beat this - A Beat Synchronization Project, 2001
http://www.clear.rice.edu/elec301/Projects01/beat_sync/index.html
- [5] MP3WRITE and MP3READ by Alfredo Fernandez, 2004
<http://www.mathworks.com/matlabcentral/fileexchange/6152-MP3WRITE%20and%20MP3READ>
- [6] Mixed In Key
<http://www.mixedinkey.com/>
- [7] Frédéric Patin : Beat detection algorithms, 2003
<http://www.flipcode.com/misc/BeatDetectionAlgorithms.pdf>
- [8] MR2 Petőfi Rádió Official Youtube Channel
<http://www.youtube.com/user/MR2Music>
- [9] Lanczos filtering in one and two dimensions, Claude E Duchon, 1979, Journal of applied meteorology
- [10] Julius Smith: Theory of Ideal Bandlimited Interpolation
https://ccrma.stanford.edu/~jos/resample/Theory_Ideal_Bandlimited_Interpolation.html
- [11] Handbook of Signal Processing in Acoustics, Volume 1, edited by David Havelock, Sonoko Kuwano, Michael Vorländer
<http://ebookey.org/David-Havelock-Sonoko-Kuwano-Michael-Vorlander-quot-Handbook-of-Signal-Processing-in-Acoustics-quot-2-vol-set-repost-1084009.html>

- [12] U. Zölzer: Digital Audio Signal Processing, J. Wiley & Sons, 1997., pp. 211
- [13] Jont B. Allen (June 1977). "Short Time Spectral Analysis, Synthesis, and Modification by Discrete Fourier Transform". *IEEE Transactions on Acoustics, Speech, and Signal Processing*
- [14] Time-Domain Algorithms for Harmonic Bandwidth, Reduction and Time Scaling of Speech Signals, IEEE Transactions on Acoustics, Speech, and Signal Processing VOL. ASSP-27, NO.2, APRIL 1979
- [15] Elastique Efficient Time Stretching SDK 2.2.5
<http://www.zplane.de/uploads/files/elastiqueEffSdkDocu.pdf>
- [16] Microsoft Visual Studio Products
<http://www.microsoft.com/visualstudio/eng>
- [17] Compile MEX-function from C/C++ or Fortran source code
<http://www.mathworks.com/help/matlab/ref/mex.html>
- [18] Creating Graphical User Interfaces in MATLAB
<http://www.mathworks.com/discovery/matlab-gui.html>
- [19] Soundcloud: Bélamühely-Tekno 2
https://soundcloud.com/belamuhely/tekno_2
- [20] YouTube: Skiller the fastest Beatboxer from Europe
<http://www.youtube.com/watch?v=8qxTo5PLw9Y>

12 Referred figures

- (1) Figure 2: Beat This Beat Detection Block Diagram:
http://www.clear.rice.edu/elec301/Projects01/beat_sync/images/FinalBlockDiag2.gif
- (2) Figure 13: PEAK measurement system
U. Zölzer: Digital Audio Signal Processing, J. Wiley & Sons, 1997., page. 107
- (3) Figure 23: Lanczos-r01-filtering.svg
<http://en.wikipedia.org/wiki/File:Lanczos-r01-filtering.svg>
- (4) Figure 24: U. Zölzer: Digital Audio Signal Processing, J. Wiley & Sons, 1997., page. 235

List of figures

Figure 1: The track deck of Traktor Scratch	8
Figure 2: Block diagram of the Beat This algorithm (Figure taken from [4])	12
Figure 3: Differentiated envelope	14
Figure 4: Differentiated envelope, closer look	15
Figure 5: BPM prediction of Will Smith – Gettin’ Jiggy With It	16
Figure 6: BPM prediction of Csík Zenekar – Csillag Vagy Fecske	16
Figure 7: Fine BPM prediction over time for Will Smith – Gettin’ Jiggy With It	17
Figure 8: Fine BPM prediction over time for Csík Zenekar – Csillag Vagy Fecske	17
Figure 9: Fine BPM prediction over time for Green Day – Longview	21
Figure 10: Envelope $e[n]$ of a track and its corresponding comb filter	24
Figure 11: Sharpened envelope $s[n]$ (black) and envelope $e[n]$ (red) signals	25
Figure 12: The beat grid on track	26
Figure 13: Peak measurement system (Figure taken from [12])	28
Figure 14: Envelope $e_{\text{peak}}[n]$ signal, using the peak measurement system	28
Figure 15: Envelope signal $e_{\text{peak}}[n]$ (black) and derivative of the envelope $d[n]$ signal (blue)	29
Figure 16: Window $w[n]$ signal (black), on a bar of the original file $y[n]$ (red), at the peak of the diffd envelope $d[n]$ local maximum (blue)	30
Figure 17: Comparing the envelopes of the original and the modified signals	31
Figure 18: Cleaned up derivative of envelope $c[n]$ (black) and threshold signal $th[n]$ (cyan)	33
Figure 19: Beat grid $g[n]$ (red) in the cleaned envelope $c[n]$ (black)	35
Figure 20: Beat grid $g[n]$ in the cleaned envelope $c[n]$, a closer look	35
Figure 21: Beeps mixed on beats (song: Bélaműhely – Tekno2)	36
Figure 22: Beat distances over time (exchanged to BPM)	37
Figure 23: Resampling with sinc functions (Figure taken from (3))	40
Figure 24: Steps of performing phase vocoding (Figure taken from [12])	42
Figure 25: Adjacent sections of a sinusoid signal after windowing	44
Figure 26: Adjacent sections mixed with OLA algorithm	44
Figure 27: Adjacent sections selected by the SOLA algorithm	45
Figure 28: Adjacent sections selected by the SOLA algorithm mixed together	45
Figure 29: Outputs of MATLAB and C/C++ decimate6() functions	50
Figure 30: Difference of the MATLAB and C/C++ decimate6() functions	51
Figure 31: Comparison of MATLAB FFT model and MATLAB shift model tempo predictions	52
Figure 32: Comparison of the MATLAB and C/C++ shift model tempo detection models	53
Figure 33: Difference of the MATLAB and C/C++ shift model tempo detection outputs	53
Figure 34: GUI window after startup	55
Figure 35: Opening the input file	56
Figure 36: Estimated tempo plot	56
Figure 37: Fine tempo estimation over time	57
Figure 38: Beat grid plot for the song	57
Figure 39: Beat grid plot for the song	58
Figure 40: Closing the GUI	59

List of tables

Table 1: Comparison of beat detection algorithms	19
Table 2: Beat detection module vs Traktor	20
Table 3: Determined ideal SOLA parameter values	46
Table 4: Computation times [s] for MATLAB functions and their C/C++ equivalents	54