Chord Analysis App

Bachelor's Thesis

Rafael Dätwyler

darafael@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Manuel Eichelberger
Prof. Dr. Roger Wattenhofer

September 23, 2018

Acknowledgements

I want to thank my supervisor Manuel for his valuable inputs and for helping me whenever I was not able to get ahead. I also want to thank my friends and family for supporting me and my idea during this thesis.

Abstract

This thesis is concerned with the development of a smartphone application which can extract chords from music. The implemented algorithm is largely based on work by Mauch [1]. The app is written in Java and can simultaneously record and analyze the recorded music.

Contents

Acknowledgements **i**

Abstract **ii**

1 Introduction **1**

1.1 Chord Estimation 1

1.2 Related Work 1

2 Theory **3**

2.1 Harmonics 3

2.2 Intervals and Chords 4

3 Algorithm **6**

3.1 Note Profiles 6

3.2 Processing Steps 7

3.2.1 Recording and Downsampling 7

3.2.2 Fourier Transform 7

3.2.3 Bucketization 7

3.2.4 Tuning 8

3.2.5 Noise Reduction 9

3.2.6 Non-Negative Least Squares 9

3.2.7 Chromagrams 9

3.2.8 Dynamic Bayesian Network 10

4 Implementation **14**

4.1 Python 14

4.1.1 Note Profile Generator 14

4.1.2 Chromagram Generator 15

4.2 Android Application 15

4.2.1 Structure of the App 16
4.2.2 MainActivity 17
4.2.3 ChordDetection 17
4.2.4 JavaAPI 18
4.2.5 BayesNetStructure 18
4.2.6 Concurrency 20

5 Evaluation **21**
5.1 Accuracy 21
5.2 Performance 22

6 Conclusion **24**

Bibliography **25**

Chapter 1

Introduction

Composing a piece of music is a creative process which requires talent and training. Writing down a melody one has heard is not easy either, but in contrast, the process is rather a mechanical one than a creative one. Naturally, people tried to develop techniques to automate this process and nowadays a whole field of research called *Music Information Retrieval* (MIR) exists, dedicated to the task of extracting information from music. This can range from the determination of the key of a song and genre classification to the extraction of a melody or lyrics. Some research is focused on the transcription of chords from music, a task called *Chord Estimation*. Other common terms for the same are *Chord Transcription*, *Chord Detection* and *Chord Extraction*.

1.1 Chord Estimation

Audio chord estimation is the task to "extract or transcribe a sequence of chords from an audio music recording" as defined by MIREX (Music Information Retrieval Evaluation eXchange), the largest contest for evaluating MIR algorithms. It has scientific uses such as the semantic analysis of a piece or segmentation of a song into characteristic segments for further analysis. But it has also everyday uses, like when trying to play the guitar along with a song. Chord estimation can then be used to determine the right chords to play.

The goal of this thesis is to develop a smartphone application that can perform chord estimation in real time, which is useful mainly for everyday uses like the one mentioned above.

1.2 Related Work

There has already been a lot of research about the chord estimation of music in the field of Music Information Retrieval. In a lot of approaches, a representation of the audio called chromagram [2] is used for further processing. Chroma refers

to the specific "position" of a tone inside an octave. The chromagram discards all octave information and aggregates the intensity of every chroma over all octaves. For practical purposes, the chromas are partitioned into buckets for every semitone or pitch class, as seen in Figure 1.1.

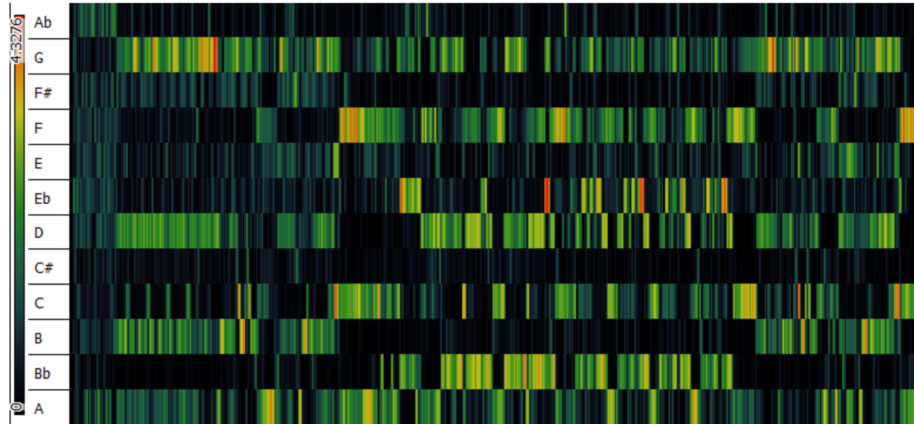

Figure 1.1: Chromagram of the beginning of the song *Rosanna* by *Toto*.

These chomagrams are then used as features for machine learning algorithms or as observed variables in a *Hidden Markov Model* (HMM) [3][4] or a *Dynamic Bayesian Network* (DBN) [5][6]. Our implementation also uses a DBN. Other approaches are based on high-order HMMs [7], Conditional Random Fields [8] or use a Support Vector Machine and incorporate information from future frames [9]. There are approaches for chord estimation in a real-time setting [4][10], but none of them are implemented as an app. The only open source Android chord estimation app found is one by De Santana Neto[1] which is based on a Fourier transform. Two apps called *Chord detector*[2] and *ZAX Chords*[3] are available at the Google Play store at the time of writing (21.09.2018) but none of them delivers satisfactory results.

[1] https://github.com/josepedro/ChordsDetector
[2] https://play.google.com/store/apps/details?id=com.xssemble.chordnamefinder
[3] https://play.google.com/store/apps/details?id=com.finestandroid.chorddetector

Chapter 2

Theory

2.1 Harmonics

Every time a note is played by a string or wind instrument, additional frequencies are audible, namely integer multiples of the fundamental frequency. They are called *harmonics* (if the fundamental tone is included) or *overtones* (if it is not included) and their existence is a consequence of the physical behaviour of resonant systems. The amplitudes of the individual harmonics define the characteristic sound of an instrument and these harmonics can often not be heard separately by an untrained ear.

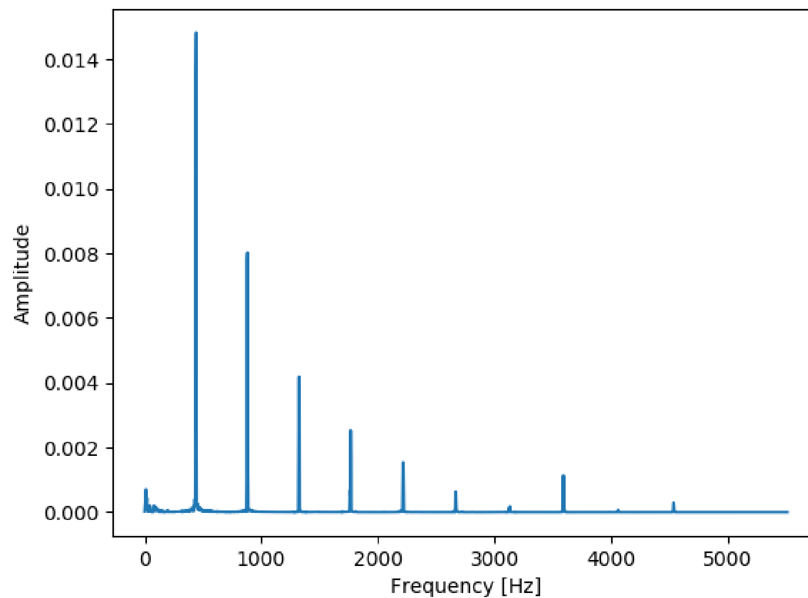

Figure 2.1: Frequency spectrum of an A4 note played on a piano.

Harmonics make the automated detection of individual notes complicated because the frequency spectrum of a sound contains, additionally to the fundamental frequency, a lot of frequencies which do not originate from a distinct note. Additionally, since the strings of real instruments are not infinitely long and infinitely thin, the frequencies of the overtones are not perfect integer multiples of the fundamental frequency, but slightly higher. This phenomenon is called *inharmonicity* and complicates the automated detection of tones further.

2.2 Intervals and Chords

Western music is typically written down in notes on staves. The distance between two notes is referred to as *interval* and the smallest commonly used interval is a *semitone*. Twelve semitones build up an *octave* and notes that are an octave apart sound the same. This is also reflected by the ratio of their frequencies: it is 2:1.

All notes have names, but since notes that are an ocvate apart sound the same, they have the same name and are distinguished by a number. On a piano which has 88 keys, the notes range from A0 (deepest) to C8 (highest). The notes in an octave, starting with C, are: C, C#, D, D#, E, F, F#, G, G#, A, A#, B. The reference note to tune an instrument is A4 with a fundamental frequency of typically 440 Hz.

A chord is a collection of notes with well-defined intervals in between them. Chords with three notes are most common and are called *triads*. For example, a major (maj) chord in root position consists of a *root*, a *third* which is four semitones over the root and a *fifth* which is seven semitones over the root. For a C major chord, this are the notes C, E and G as seen in Figure 2.2. There exist other triads like the minor (min) chord where the third is a semitone lower than in a major chord.

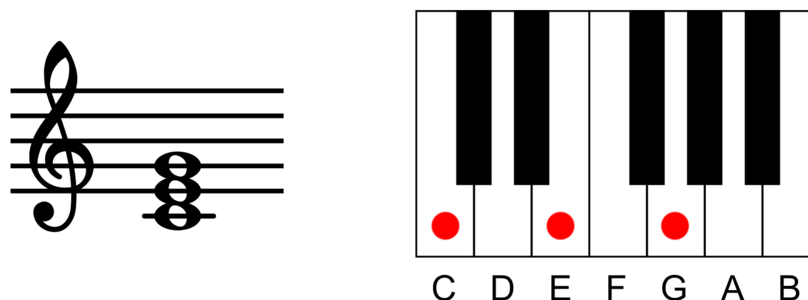

Figure 2.2: C major chord noted on a staff and played on a piano.

A chord does not have to consist of only three notes. Common types of chords with four notes are the *seventh chords* which have an additional note

called the *seventh* placed three or four semitones over the fifth. Also, the root note does not have to be at the lowest position of a chord. When playing a chord in a *inversion*, the note played by the bass can be the third or the fifth (or the seventh, for seventh chords).

MIREX proposes different levels of complexity for evaluating the accuracy of chord estimation algorithms, the most complex of which consists of “no chord”, maj, min, maj7, min7 and 7 chords and all inversions. In this thesis, only major and minor triads are considered.

Chapter 3

Algorithm

In a first step, the probabilities of individual notes currently being present are estimated. The note detection is performed by means of a Fourier transform and *note profiles* as described below. The note probabilities are then used to create treble and bass chromagrams which serve as evidence for a dynamic Bayesian network to determine the most likely chord that is being played. The individual steps are explained in this chapter.

The implemented algorithm is largely based on the one described by Mauch in his PhD thesis [1]. The non-negative least squares part is taken from a subsequent publication [11]. The main difference is that our implementation analyzes the recorded music in real time whereas Mauch's algorithm, escpecially the tuning part, depends on the whole recording being available at the start of the analysis. The tuning part is adjusted to work in a real time setting. Another difference is that our implementation does not rely on beat tracking, because the app is intended to also work on music with no beat, for example when trying different chords on a guitar.

3.1 Note Profiles

For the detection of individual notes, the algorithm tries to "rebuild" the recorded music from predefined note profiles. We consider 84 such note profiles, one for every note from C1 to B7. This way we get seven full octaves. Lower or higher notes are rarely present in music and cannot be properly recorded by most devices. Each note profile is generated in advance and is supposed to represent a most general tone which can match as many instruments as possible. For this, the overtone series is of great importance. Mauch uses a geometrically decreasing overtone series, so the k-th overtone has amplitude

$$a_k = s^k \tag{3.1}$$

with the factor s being linearly spaced from 0.9 for the deepest note to 0.6 for the highest note, which means that the series decreases faster for higher notes.

This makes sense intuitively as, for example on a piano, a vibrating string of a low note causes a lot of the higher strings to vibrate as well, whereas a vibrating string of a high note influences only few other (higher) strings. Testing different parameters shows that this model indeed works better than using a constant value for s.

3.2 Processing Steps

3.2.1 Recording and Downsampling

In a first step, a chunk of sound is recorded. A frame length of 4096 is chosen because it delivers the best results. The sampling rate is 11025 Hz. This has performance advantages over prevalent sampling rates like 44.1 kHz or 48 kHz, because fewer samples have to be considered later in the Fourier transform. If this sampling rate is not available on a device, the music is recorded with 44100 Hz and downsampled to 11025 Hz. According to the Nyquist-Shannon theorem, a maximal frequency of about 5512 Hz can be described with this sampling rate. Since harmonic information is mostly present below this frequency, we can safely discard the higher frequencies (for comparison: C8, the highest note on a piano, has a frequency of 4186 Hz).

3.2.2 Fourier Transform

For further processing, we want our audio to be in the frequency domain. For this, we perform a discrete Fourier transform on the recorded chunk. To smoothen the edges in time domain, the recorded chunk is first windowed by a Hamming window. The frame length being a power of two allows us to perform the efficient Fast Fourier Transform (FFT).

3.2.3 Bucketization

Because the amount of different frequencies that result from the Fourier transform, namely 2049, is still too large for least squares solving, similar frequencies are summed up and considered as one value. More concretely, the whole frequency spectrum gets divided into buckets spaced a third of a semitone apart. Hence we get 252 buckets and every bucket corresponds to a range of frequencies. The note profiles mentioned above are also divided into 252 buckets. This will allow us to solve for the probabilities of the 84 notes, see Subsection 3.2.6.

3.2.4 Tuning

Because not all instruments are tuned to 440 Hz and some instruments might not be tuned at all, the data is tuned to 440 Hz after bucketization. This highly improves the note detection accuracy, see Figure 3.1. For every three buckets corresponding to a semitone, only the middle bucket should contain information if the instrument is perfectly tuned, while the information spills to the left or right bucket if the instrument is tuned too low or too high, respectively. This is reflected in the phase angle φ of the normalized frequency $\pi/3$ when interpreting the vector of buckets as a time series and calculating the Fourier transform on that series. This phase angle can be used to tune the data.

In order to use as much information as possible, buckets from previously recorded chunks are also considered. More concretely, a “tuning vector” z_t is maintained and at every time step, the new vector

$$z_t = \frac{b_t + s \cdot z_{t-1}}{1 + s} \tag{3.2}$$

is calculated where $s \in [0, 1)$ is a parameter and b_t is the vector of the newest buckets. This way, the previous buckets are considered with exponentionally decreasing importance to account for possible tuning changes. The Fourier transform is then performed on the vector z_t. The tuning factor $\delta \in [-0.5, 0.5)$ describes which fraction of a semitone the piece is off-tune and is calculated by the following formula:

$$\delta = \frac{\mathrm{wrap}(-\varphi - \frac{2\pi}{3})}{2\pi} \tag{3.3}$$

where wrap() phase wraps each number to the interval $[-\pi, \pi)$. Using this factor, the tuning is performed by linear interpolation between the buckets.

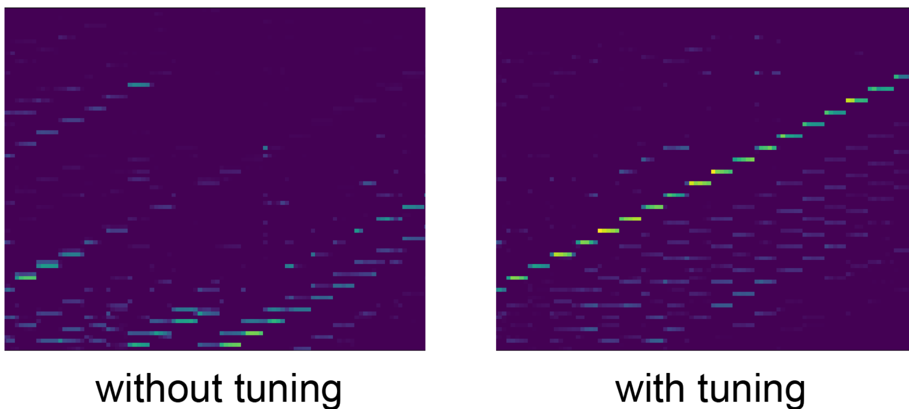

Figure 3.1: Estimated notes of an ascending series played on a piano tuned to 450 Hz with and without the tuning step applied.

3.2.5 Noise Reduction

In order to reduce noise and emphasize peaks, the running mean and running standard deviation are subtracted from the buckets. More concretly, for every bucket b^i, the buckets from b^{i-18} to b^{i+18} (index taken over frequency) are considered, where the bucket vector is padded at the top and the bottom using the nearest value. Then the mean is calculated as weighted average using the values of a discretized Hamming window as weights.

$$\mu^i = \frac{\sum\limits_{k=-18}^{18} w^k \cdot b^{i+k}}{\sum\limits_{k=-18}^{18} w^k} \tag{3.4}$$

The running standard deviation is computed similarly, again using a Hamming window as weights:

$$\sigma^i = \sqrt{\frac{\sum\limits_{k=-18}^{18} w^k \cdot (b^{i+k} - \mu^i)^2}{\sum\limits_{k=-18}^{18} w^k}} \tag{3.5}$$

The denoised buckets are defined as $\bar{b}^i = \max\{0, b^i - \mu^i - k \cdot \sigma^i\}$ where k is a factor experimentally set to 1.

3.2.6 Non-Negative Least Squares

After tuning and noise reduction, the crucial step of determining the played notes is performed by solving a non-negative least squares (NNLS) problem. NNLS is defined as

$$\min_x \|Ax - b\|_2 \qquad \text{s.t. } x \geq 0 \tag{3.6}$$

In our case, $A \in \mathbb{R}_+^{252\times 84}$ is the dictionary matrix consisting of the note profiles, $x \in \mathbb{R}_+^{84}$ is the sought vector of note probabilities and $b \in \mathbb{R}_+^{252}$ is the vector of frequency buckets. The non-negativity constraint is added because it does not make sense for a note to contribute a negative amount.

Solving this problem is equivalent to finding the amplitudes of the 84 notes, the sum of which minimizes the Euclidean distance to the recorded, bucketized sound. So, if the recorded sound consists of music, we can assume to have a confident measure of the intensity with which each note was played.

3.2.7 Chromagrams

Until now, octave information was preserved. Without this, solving NNLS using the note profiles would not have been possible. In this step, octave information is

discarded and the same tones are aggregated to create chromagrams as described in Section 1.2. Two different kinds of chromagrams are generated: a *treble* and a *bass* chromagram. The treble chromagram only considers notes in mid and high frequency ranges which contain most of the harmonic information. The bass chromagram deals with the bass. Bass information is important for chord estimation as it narrows down the set of possible chords and is necessary to detect chord inversions.

The vector of notes is windowed before summing up over all octaves. The window is shown in Figure 3.2.

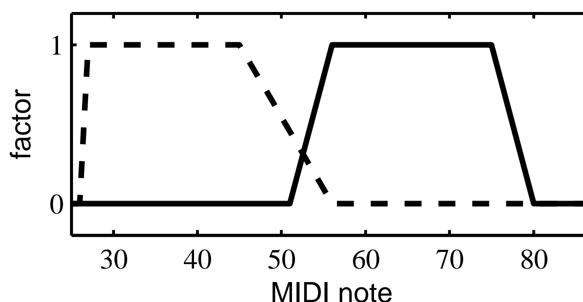

Figure 3.2: Window which is applied to the note vector before summing up to treble chromagram (solid line) and bass chromagram (dashed). (from [5])

3.2.8 Dynamic Bayesian Network

A Bayesian network is a probabilistic graphical model that connects random variables in a directed acyclic graph (DAG). In the DAG, the nodes are random variables and a node that is connected to a parent is conditionally dependent on that parent. Some of the variables might be known ("observed") and serve as *evidence* in order to infer the probabilities of the hidden variables by *Bayesian inference*.

A *dynamic* Bayesian network (DBN) connects nodes over adjacent time steps and is used to model structures that change over time, such as speech or music. We are interested in inferring the probabilities for different chords in a model that consists of the song key, chord and bass as hidden variables and treble chromagram and bass chromagram as observed variables, as depicted in Figure 3.3. Using this model, the algorithm infers the chord probabilities and outputs the chord with the highest probability.

The DBN is an adapted version of [11], but without the metric position. We give a quick overview over the different nodes, the details are explained in [1].

The song key is assumed to be dependent on the previous key estimation and

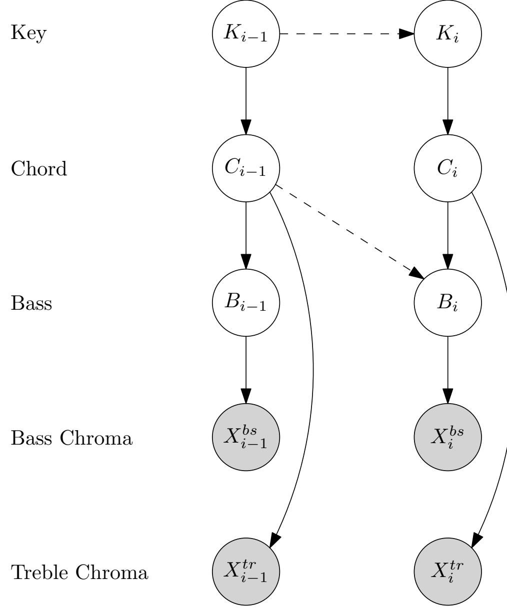

Figure 3.3: Model of the dynamic Bayesian network. (adapted from [11])

unlikely to change. Therefore:

$$P(K_i|K_{i-1}) = \begin{cases} 0.98 & \text{if } K_{i-1} = K_i \\ (1-0.98)/23 & \text{otherwise.} \end{cases} \tag{3.7}$$

The chord is only dependent on the key. We use an adapted version of the perceptual chord ratings by Krumhansl [12] (page 171). Table 3.1 shows the probabilities for the different chords in a C major and C minor key.

The bass is dependent on whether or not a new chord is played, because the bass usually plays the root note after a chord change. This is reflected in the conditional probability:

$$P(B_i|C_{i-1} \neq C_i) = \begin{cases} 0.8 & \text{if the bass is the root note of } C_i \\ 0.2/11 & \text{otherwise.} \end{cases} \tag{3.8}$$

If the chord has not changed, the bass is still likely to play the root note, but also other notes of the chord are possible. A small probability is left to the cases where the bass plays any other note.

$$P(B_i|C_{i-1} = C_i) = \begin{cases} 0.4 & \text{if the bass is the root note of } C_i \\ 0.2 & \text{if the bass is one of the other notes of } C_i \\ 0.2/9 & \text{otherwise.} \end{cases} \tag{3.9}$$

	maj	min	maj	min
C	7.28%	5.84%	6.94%	7.42%
C#	4.13%	3.31%	4.55%	2.68%
D	3.95%	4.47%	3.99%	2.96%
D#	3.48%	2.42%	4.61%	3.38%
E	4.02%	3.68%	4.31%	3.10%
F	5.55%	4.62%	5.15%	5.23%
F#	3.56%	2.31%	4.17%	2.51%
F	5.13%	3.51%	5.09%	3.35%
G#	4.62%	3.36%	5.23%	3.43%
A	4.02%	5.55%	3.71%	3.85%
A#	4.16%	3.25%	4.77%	2.76%
B	4.07%	3.68%	4.03%	2.78%

Table 3.1: Chord probabilities dependent on the song key. Here shown for the C major key (left) and C minor key (right).

Finally, the treble chromagram is dependent on the key and the bass chromagram is dependent on the bass. Each semitone of the treble and bass chromagram is modelled as a Gaussian variable. If a note is played, it is likely that the corresponding value in the treble chromagram has a high value. And the value should be low if the note is not played. Therefore, for each entry of the chromagram is the mean set to 1, if the entry is part of the chord and to 0, if it is not (see Figure 3.4). The variance is set to 0.2. For the bass chromagram, the mean for each entry is set to 1 if and only if the corresponding bass note is played. The variance is reduced to 0.1.

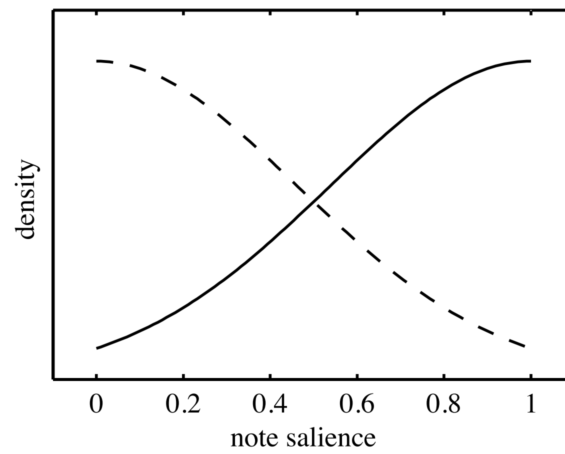

Figure 3.4: The entries of the treble chromagram are modelled as Gaussian distributions which differ in whether the entry is a note of the chord (solid line) or not (dashed). (from [1])

Chapter 4

Implementation

Two implementations of our algorithm exist. The first is written in Python because the development is more lightweight and testing can be performed quicker. The second is written in Java and is the core of the Android application.

4.1 Python

For development in Python, the libraries NumPy and SciPy are used. Because many of the operations are performed on vectors, the vectorized methods provided by these libraries are useful. A lot of plots and graphics are created by means of the library Matplotlib. Few methods are also used from other libraries. In contrast to the Android implementation, no real time analysis is performed with Python, but whole WAVE files are read and processed. Two main programs are explained in the following subsections.

4.1.1 Note Profile Generator

This program is used to generate the note profiles which are saved in the matrix for the NNLS problem. Two different approaches are tested. The first generates the note profiles directly and saves the amplitudes of the harmonics into the corresponding buckets. The other approach first generates a waveform from the note profile, then uses discrete Fourier transform and bucketization to fill the buckets, similar to the steps described in Sections 3.2.2 and 3.2.3. Due to numerical inaccuracies, this does not lead to the exact same results as the first approach. Especially lower tones show distortions as can be seen in Figure 4.1. But because recorded music runs through the same process, the latter approach results in better note detection.

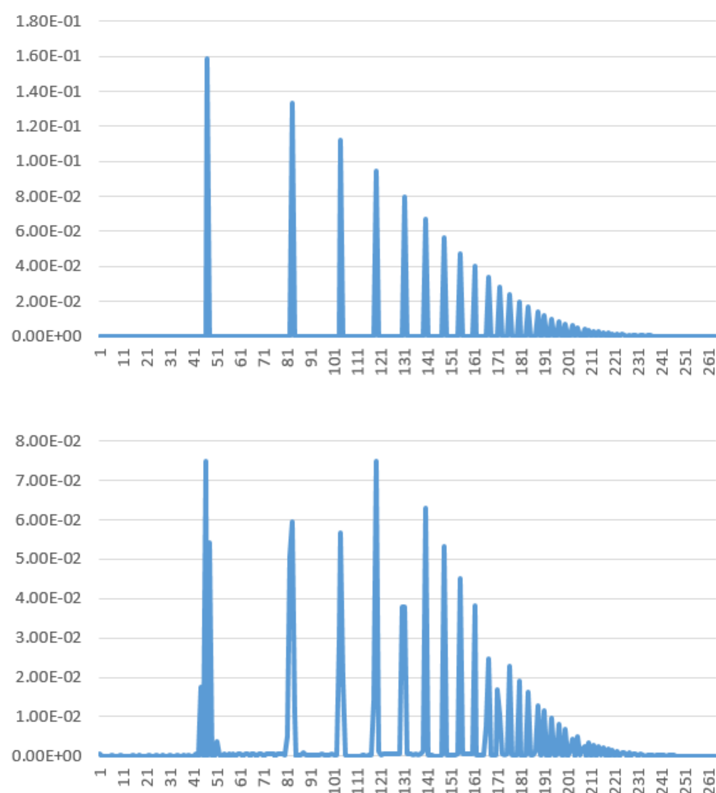

Figure 4.1: Note profile of the C2 note, generated directly (upper) and via DFT (lower).

4.1.2 Chromagram Generator

The chromagram generator implements the algorithm described in Chapter 3. This program is also used to generate graphics like Figure 3.1, mainly to visually confirm that the implementation of the algorithm produces the intended results. The graphics include Fourier transformations of music, graphs of buckets, semitones and chromagrams and are used to analyze the frequency spectrum of recorded music and to experiment with the parameters in the noise reduction step.

4.2 Android Application

The Android application is developed in Android Studio using the Java programming language. In contrast to Python, Java is strongly typed and does not have an “industry standard” numerical library which provides native support for vector operations and unites all the funcionality required for this purpose. An-

droid allows to write code in C++ which is linked to the application through the Java Native Interface (JNI) provided by the Android Native Development Kit (NDK). This allows for better performance because the code does not have to be compiled by a just-in-time compiler while executing. However, Java and C++ store certain types differently in memory and the structure of the app gets more complicated when using NDK. So, all numerical operations are implemented directly in Java which turns out to perform well enough. But the app is built in a way that would make switching to a native implementation easy.

4.2.1 Structure of the App

The app essentially consists of the MainActivity and three additional classes: the ChordDetection class, the JavaAPI class and the BayesNetStructure class. The MainActivity is the starting point of the application and manages the user interface (UI). The ChordDetection is responsible for the creation and initialization of the native AudioRecord class, for managing the audio buffer and for invoking the calculation for chord detection. The JavaAPI class contains all methods that are necessary to compute treble and bass chromagrams from a chunk of recorded audio. It implements the interface “API” and could be replaced by a native (C++) implementation. The BayesNetStructure defines the Bayesian network which is used to calculate the most likely chord.

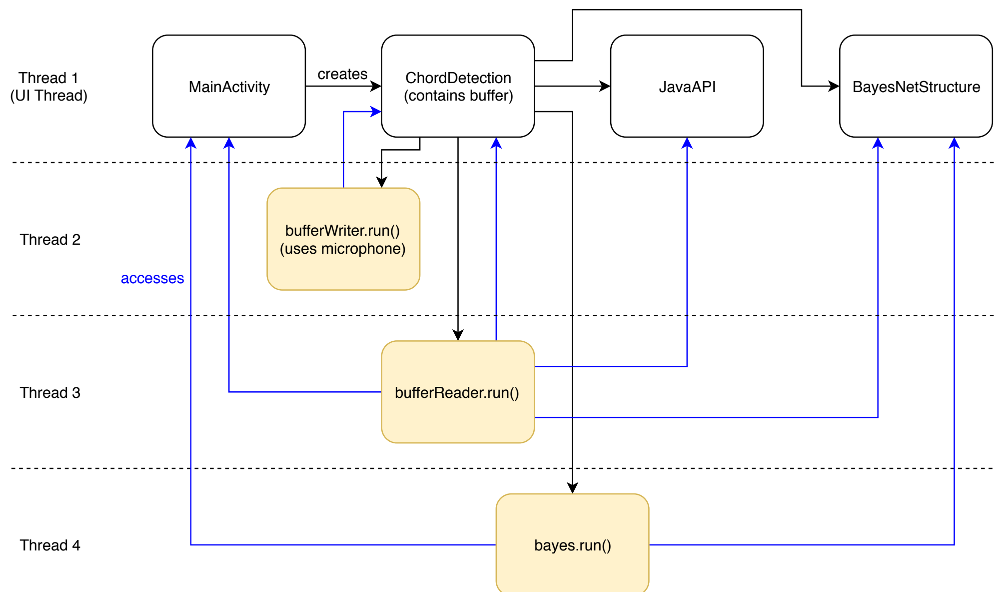

Figure 4.2: Flow chart of the app structure.

4.2.2 MainActivity

The MainActivity, as seen in Figure 4.3, consists of a button to start/stop the chord detection, a set of progress bars to show the treble and bass chromagram, a "song mode" switch and a TextView which shows the estimated chord. The chromagrams are used for debugging and to give the user an intuition of how the chord was estimated. The "song mode" switch allows the user to switch between *song* and *no-song* mode: Song mode is intended to be used on a song which has a harmonic structure over time and a song key. No-song mode does not require these properties and is better for detecting randomly played chords. The technical details are explained in Subsection 4.2.5. The MainActivity is also responsible for checking that the audio recording permission is granted and eventually creates an instance of the ChordDetection class.

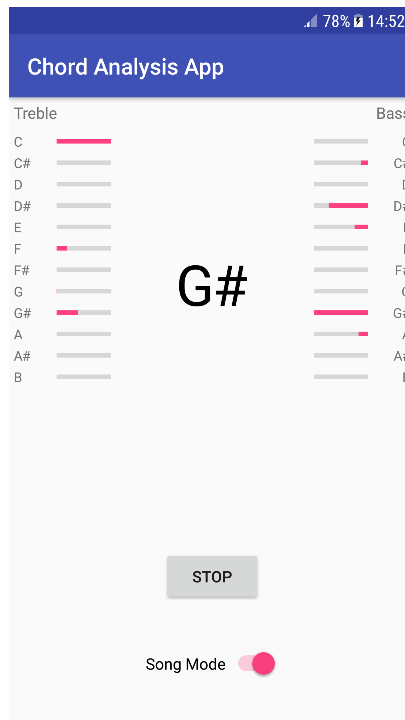

Figure 4.3: Screenshot of the MainActivity.

4.2.3 ChordDetection

The ChordDetection class takes care of initializing the AudioRecord class, which is provided by the Android API. This class reads audio from a microphone input and writes it into an internal buffer. From this buffer, chunks of arbitrary size (as long as they are shorter than the internal buffer) can be read into a working buffer

in a blocking or non-blocking way. For this application, three working buffers were chosen which are filled circularly by a blocking read. This way, when the JavaAPI is calculating chord estimation using one buffer, the other two buffers can still be filled alternately and there is always one full buffer available. The ChordDetection class contains the three threads bufferWriter, bufferReader and bayes which are invoked once the chord detection is started. The exact way the buffers are processed and how the threads interact with each other is explained in Section 4.2.6.

4.2.4 JavaAPI

The JavaAPI class contains all methods to calculate the treble and bass chromagram. Upon creation, initialization takes place, such as the creation of Hamming windows and reading the note profile matrix from a CSV file into a two-dimensional array. The main method is called calculate() and is invoked by the bufferReader. This method implements the concurrency mechanisms and calls the procedure() method which then executes the algorithm and saves the treble and bass chromagram into a list, from where it can be processed by the BayesNetStructure. Because no Java library was found which supports all necessary operations, the data is generally stored in a double array and temporarily converted to library-specific data representations if necessary.

For the discrete Fourier transform, the library JTransforms[1] by Piotr Wendykier is used. For solving NNLS, a first approach uses the least squares package of Apache Commons Math[2] with the additional restriction of the parameters being non-negative. This package solves the problem iteratively using a Levenberg-Marquardt method. However, benchmarking shows that this method performs poorly (see Section 5.2), so an alternative was sought. The current approach uses Parallel Java 2 (PJ2)[3] by Alan Kaminsky which is approximately eight times faster. For reading CSV files, the library opencsv[4] is used.

4.2.5 BayesNetStructure

The BayesNetStructure class, when instantiated, creates a Bayesian network using the library Jayes[5]. No free Java library could be found that supports dynamic Bayesian networks, so the structure of a DNB is emulated by a normal Bayesian network with multiple layers created manually. The number of layers n can be passed to the constructor as a parameter. This class maintains a list to which pairs of treble and bass chromagrams can be appended and offers the

[1] https://sites.google.com/site/piotrwendykier/software/jtransforms
[2] http://commons.apache.org/proper/commons-math/userguide/leastsquares.html
[3] https://www.cs.rit.edu/~ark/pj2.shtml
[4] http://opencsv.sourceforge.net/
[5] http://www.eclipse.org/recommenders/jayes/

method update(). Every time this method is called, the class starts Bayesian inference using the latest n chromagrams of this list (if available) as observed variables by means of the junction tree algorithm. update() returns an array of chord probabilities which the bayes thread uses to update the UI with the most likely chord.

Two different Bayesian networks are created (see Figure 4.4): One (used by song mode) which is multi-layered to use previous chromagrams and one (used by no-song mode) which consists of only one layer and does not incorporate the key information. The former is an adapted version of the one described in 3.2.8. More concretely, the temporal connection from the previous chord to the current bass is left out and only the probabilities from Equation 3.9 are used. This is because the large amount of conditional probabilities that arises when considering the current *and* previous chord slows down the inference significantly and renders the app unusable.

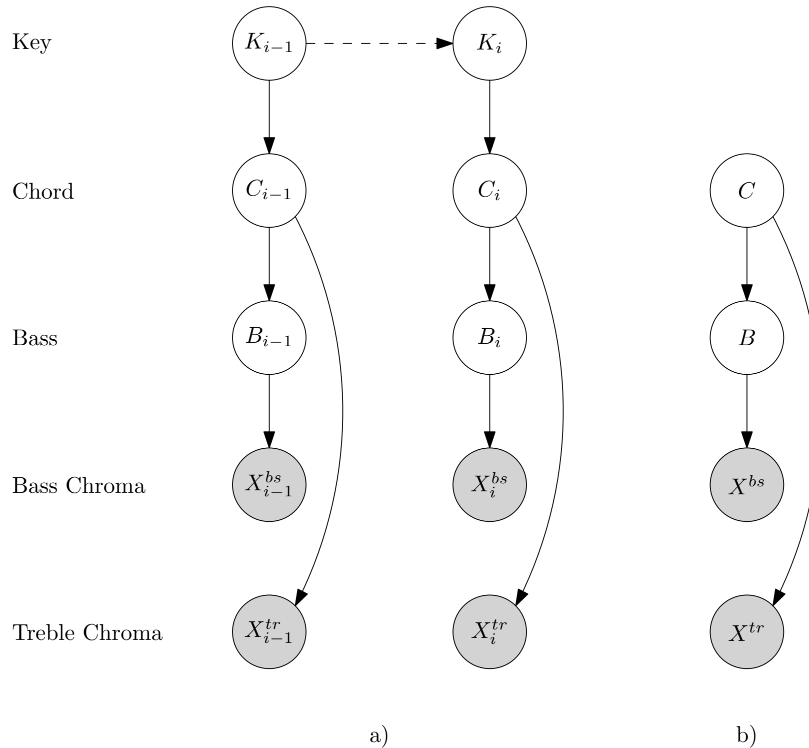

Figure 4.4: Model of the implemented dynamic Bayesian network. a) The only temporal connection is the key. Used by song mode. b) A model which does not depend on previous information and does not incorporate a key. Used by no-song mode. (adapted from [11])

4.2.6 Concurrency

This subsection describes how the different threads interact with each other. Four threads are maintained (see Figure 4.2): The main thread which takes care of all initialization steps and invokes the execution of the other threads once the start button is pressed. This thread is also the UI thread which updates all UI elements. The bayes thread is explained below. The writer thread records the audio and fills the buffers and the reader thread reads from the buffers and is responsible for all the computation. The buffers act as a producer-consumer data structure, but in contrast to the usual implementation, the consumer (bufferReader) is only concerned with the most recent chunk of audio data. It is therefore possible that a chunk of audio is dropped if the computation is not fast enough (see Figure 4.5). However, the goal is to keep the number of dropped chunks as low as possible.

The information of which buffer was most recently filled and which is being processed by the JavaAPI is stored in variables, the write-access to which is mediated by a lock. The consumer tries to read the newest chunk of data and if there is no data, the thread sleeps. Meanwhile, the producer fills a buffer and upon completion, notifies the consumer thread which can then process this buffer. During computation, the producer only fills the other two buckets, alternately. Once the consumer finishes computation, it causes the MainActivity to update the progress bars and appends the treble and bass chromagrams to the list which is maintained by the BayesNetStructure. Then, if there is new data available in a new buffer, it can directly continue computation on that buffer.

The bayes thread operates independently of the producer and consumer. Once started, it repeatedly invokes the update() method defined by the class BayesNetStructure. The producer, consumer and bayes thread operate in a while loop whose condition is an AtomicBoolean which can be switched by the button in the MainActivity.

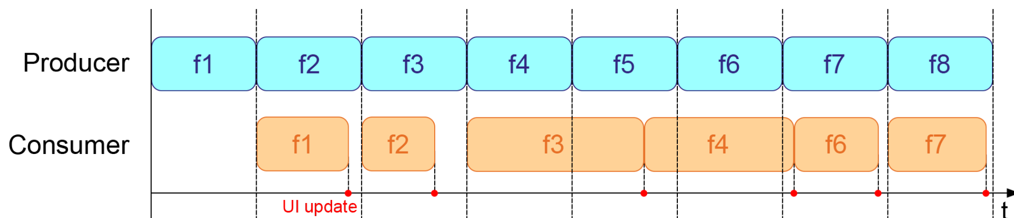

Figure 4.5: Schema of how the producer (bufferWriter) and consumer (bufferReader) work with the data. In this example, the block f5 is dropped because the calculation of f3 and f4 took too long.

Chapter 5

Evaluation

This chapted is concerned with the evaluation of the app in terms of accuracy and performance. For testing, a Samsung Galaxy S6 phone is used. It was released in 2015 and has an eight core 64bit processor and 3GB RAM.

5.1 Accuracy

To test the accuracy of the app under real-world conditions, audio is played on a portable speaker and recorded by the phone about 0.5m away. The series of calculated chords is saved and compared against ground truth from the Isophonics data set[1]. We measure the fraction of the song at which the app shows the correct chord. Because the song and the recording are started manually, the data points are shifted in time before comparison to account for these inaccuracies. The amount of displacement is determined by seeking the local maximum of correct overlap time. The results for three songs are shown in Table 5.1 for different amounts of Bayesian network layers and for the no-song mode.

	The Beatles - I Saw Her Standing There	Queen - We Are The Champions	Zweieck - Andersrum
No-song mode	26.60%	25.26%	30.96%
5 layers	26.54%	26.28%	42.28%
10 layers	34.24%	27.04%	36.90%
20 layers	34.48%	31.34%	37.84%
50 layers	40.04%	32.07%	34.20%

Table 5.1: The fraction of a song at which the app shows the correct chord. Tested with three different songs and different amounts of Bayesian network layers.

The fraction of correctly displayed chords peaks at about 40%. State of the art algorithms achieve up to 80% but they operate on the original audio data and

[1]http://www.isophonics.net/datasets

not on audio recorded through a smartphone microphone. Some algorithms rely on the whole song being available at the start of analysis, which is not possible in a real-time setting.

In some cases, the app performs better with fewer layers. This could be the case if the harmony of a song does not behave like the model defined by the Bayesian network. Also, the accuracy of the app can vary between different recordings of the same song because the alignment of the recorded chunks can be shifted in time, resulting in a different frequency spectrum and eventually in different chord probabilities.

5.2 Performance

When recording with 11025 Hz, a chunk of 4096 samples is 371'519 μs long. So, if the processing from recorded audio to chromagrams is longer than this, frames have to be dropped. On the Galaxy S6, both the implementation using Apache Commons Math and the one using PJ2 as NNLS solver are able to stay under this time limit. The calculation of one chunk using Apache Commons Math takes on average 101'666 μs whereas using PJ2 takes only 18'269 μs (see Table 5.2). The phone supports a sampling rate of 11025 Hz, so no downsampling is needed. Downsampling is tested manually and also shown in the benchmark, but not added to the total running time. The data is divided by its norm after the Fourier transform, which is also shown in the benchmark.

Step	Time
Downsampling	1'661 μs
Fourier transform	1'764 μs
Division by norm	197 μs
Bucketization	23 μs
Tuning	281 μs
Noise subtraction	4'222 μs
NNLS (Apache)	94'920 μs
NNLS (PJ2)	11'523 μs
Chroma generation	259 μs

Table 5.2: Benchmark for the different processing steps, averaged over ten iterations.

The performance of the Bayesian network calculation is tested for different amount of layers (see Table 5.3). Even a Bayesian network with 50 layers has feasible runtime, but startup time already amounts to 2.92 seconds when using

20 layers and rises to 13.41 seconds when using 50 layers. When trying 100 layers, the app crashes because it runs out of memory. This is due to the large amount of conditional probabilities which are loaded by the nodes of the network.

Relating to the results regarding accuracy, we choose 10 layers for the Bayesian network as a tradeoff between accuracy and startup time.

1 layer (no-song mode)	452 μs
5 layers	2'616 μs
10 layers	6'247 μs
20 layers	16'068 μs
50 layers	46'454 μs

Table 5.3: Benchmark for the inference of chord probabilities using different amounts of layers in the Bayesian network, averaged over ten iterations.

Chapter 6
Conclusion

Estimating the chords from a piece of music is not easy, and the accuracy of the app reflects that. As soon as the harmony does not follow the structure of the Bayesian network, chords start to change quickly or the melody becomes predominant, the app fails to deliver accurate results. This makes it difficult to use it as a general chord recognition tool like intended. However, when playing chords over a longer period of time or with little noise, the prediction become more accurate. Adding a beat tracking algorithm to the song mode would certainly help to deliver a clearer result, as the app currently cannot determine whether a complex chromagram belongs to a complex chord or is just the result of a chunk which happens to overlap a chord change.

The app leaves a lot of options for future work: Additionally to the chord display, the tuning factor δ from Subsection 3.2.4 could be displayed such that the app can be used as a tuner. The chords could be saved on the phone or online in a database which could be used as a library to look up songs. One could also add a music theoretical analysis such as determining the key of the song, the groundwork for which is already laid by the Bayesian network. Based on the key, one could perform Roman numeral analysis which can be used to transpose a song to a different key.

Bibliography

[1] Mauch, M.: Automatic Chord Transcription from Audio Using Computational Models of Musical Context. PhD thesis, Queen Mary University of London (2010)

[2] Wakefield, G.H.: Mathematical representation of joint time-chroma distributions. In: Proc. Int. Symp. Opt. Sci., Eng. Instrum. Volume 99. (1999) 18–23

[3] Sheh, A., Ellis, D.: Chord segmentation and recognition using em-trained hidden markov models. In: Proc. 4th Int. Soc. Music Inf. Retrieval. (2009) 183–189

[4] Cho, T., Bello, J.: Real-time implementation of hmm-based chord estimation in musical audio,. In: Proc. Int. Comput. Music Conf. (2009) 16–21

[5] Mauch, M., Dixon, S.: Simultaneous estimation of chords and musical context from audio. **18** (09 2010) 1280 – 1289

[6] Ni, Y., Mcvicar, M., Santos-Rodriguez, R., De Bie, T.: An end-to-end machine learning system for harmonic analysis of music. **20** (07 2011)

[7] Scholz, R., Vincent, E., Bimbot, F.: Robust modeling of musical chord sequences using probabilistic n-grams. In: Proc. IEEE Int. Conf. Acoust., Speech, Signal Process. (04 2009) 53–56

[8] Burgoyne, J., Pugin, L., Kereliuk, C., Fujinaga, I.: A cross-validated study of modelling strategies for automatic chord recognition in audio. In: Proc. 8th Int. Conf. Music Inf. Retrieval. (01 2007) 251–254

[9] Weller, A., P. W. Ellis, D., Jebara, T.: Structured prediction models for chord transcription of music audio. In: Proc. Int. Conf. Mach. Learn. Appl. (12 2009) 590–595

[10] M Stark, A., Plumbley, M.: Real-time chord recognition for live performance. In: Proc. Int. Comput. Music Conf. (08 2009) 85–88

[11] Mauch, M., Dixon, S.: Approximate note transcription for the improved identification of difficult chords. In: ISMIR. (2010)

[12] Krumhansl, C.: Cognitive Foundations of Musical Pitch. Volume 25. (01 1990)