

**Projektarbeit Elektrotechnik**

Real-Time Android Spectrum-Analyzer

|  |  |  |
| --- | --- | --- |
| **Autor** |  | Manuel König, koenigma |
| **Hauptbetreuung** |  | Dr. Martin Loeser, loma |
| **Nebenbetreuung** |  | Dr. Sigisbert Wyrsch, wyrs |
| **Datum** |  | 21.Dezember 2015 |



**Erklärung betreffend das selbständige Verfassen einer Projektarbeit an der School of Engineering**

Mit der Abgabe dieser Projektarbeit versichert der/die Studierende, dass er/sie die Arbeit selbständig und ohne fremde Hilfe verfasst hat. (Bei Gruppenarbeiten gelten die Leistungen der übrigen Gruppenmitglieder nicht als fremde Hilfe.)

Der/die unterzeichnende Studierende erklärt, dass alle zitierten Quellen (auch Internetseiten) im Text oder Anhang korrekt nachgewiesen sind, d.h. dass die Projektarbeit keine Plagiate enthält, also keine Teile, die teilweise oder vollständig aus einem fremden Text oder einer fremden Arbeit unter Vorgabe der eigenen Urheberschaft bzw. ohne Quellenangabe übernommen worden sind.

Bei Verfehlungen aller Art treten die Paragraphen 39 und 40 (Unredlichkeit und Verfahren bei Unredlichkeit) der ZHAW Prüfungsordnung sowie die Bestimmungen der Disziplinarmassnahmen der Hochschulordnung in Kraft.

Ort, Datum: Unterschrift:

………………………………………. …………………………………………………….

Manuel König

**Zusammenfassung**

In der Signalanalyse werden immer wieder Spektren anhand eines Spektrogramms analysiert. Das kann helfen, wenn man Störungen oder freie Kanäle finden will. Bisher wurden dafür immer teure Hard- und Software verwendet. Für eine grobe Analyse würde auf eine geringe Genauigkeit ausreichen.

Das Ziel dieser Projektarbeit ist es eine Applikation zu entwickeln, welche auf einem mobilen Endgerät, einem Samsung Galaxy S5, ein Sprektrogramm von einem Audiofile generieren kann. Mit Hilfe von MATLAB soll ein Nachbau des Befehls spectrogram erstellt werden. Dieser Nachbau wird zunächst in Java implementiert. Mittels bestehenden FFT-Bibliotheken soll in Java eine Abschätzung gemacht werden, welche Bibliotheken sich eignen. Am Schluss wird eine Applikation in Android entwickelt, welche die Audiodaten einliest, verarbeitet und als Spektrogramm ausgibt.

Diese Projektarbeit zeigt auf, dass eine Real-Time Android Spectrum-Analyser Applikation machbar ist und ein lesbares Spektrogramm dargestellt werden kann. Der Vergleich mit dem MATLAB-Befehl spectrogram zeigt, dass die Applikation sehr genaue Daten liefert.

**Abstract**

During signal analyses spectrographs are often used to analyze spectrums. This can help, if you want to find faults or free channels. Up until now you needed expensive equipment to do this. For a first impression or a rough estimate the equipment wouldn’t need to be so accurate and expensive.

The Goal of this Project is to design an Application, which can be run on a Mobile device and generates a spectrograph from an audio file. With MATLAB an analyst of an audio file is done, afterwards the existing FFT libraries of Java are tested to see which generates the best results. At the end an application for android developed, which takes an audio file, processes it and shows a spectrograph.

This Project shows that a real-time Android spectrum analyzer application is feasible and an understandable spectrograph can be shown. The comparison of the MATLAB version and the Java version shows that the Java libraries generate accurate Data.

**Vorwort**

In dieser Projektarbeit konnten verschiedenste Themengebiete, von „Digitale Signalverarbeitung“ über „Informatik“ bis hin zu „Mathematik“ angewendet werden. Durch die Wahl des Studiengangschwerpunktes Informationstechnologie haben die Arbeitsinhalten dieser Projektarbeit von Anfang an mein Interesse geweckt.

Bei der Erarbeitung der notwendigen Grundlagen wie auch beim gesamten Projektverlauf wurde ich tatkräftig durch Herrn Dr. Martin Loeser und Herrn Dr. Sigisbert Wyrsch unterstützt. An dieser Stelle bedanke ich mich bei Herrn Dr. Martin Loeser und Herrn Dr. Sigisbert Wyrsch für die Unterstützung.

**Inhalt**

[1 Einleitung 6](#_Toc438422836)

[1.1 Ausgangslage 6](#_Toc438422837)

[1.2 Aufgabenstellung 6](#_Toc438422838)

[2 Der MATLAB-Befehl spectrogram 8](#_Toc438422839)

[2.1 Nachbau des Befehls in MATLAB 8](#_Toc438422840)

[2.2 Vergleich der Methoden 9](#_Toc438422841)

[3 Implementation in Java 11](#_Toc438422842)

[3.1 Die einzelnen Bibliotheken 11](#_Toc438422843)

[3.2 Die Auswertung 12](#_Toc438422844)

[4 Android Applikationsentwicklung 13](#_Toc438422845)

[4.1 Einlesen der .wav-Datei 13](#_Toc438422846)

[4.2 Fast Fourier Transformation 14](#_Toc438422847)

[4.3 Anzeige des Spektrums 14](#_Toc438422848)

[5 Endresultat 15](#_Toc438422849)

[6 Reflexion / Ausblick 19](#_Toc438422850)

[7 Verzeichnisse 20](#_Toc438422851)

[7.1 Literaturverzeichnis 20](#_Toc438422852)

[7.2 Abbildungsverzeichnis 20](#_Toc438422853)

[8 Anhang 21](#_Toc438422854)

[8.1 Offizielle Aufgabenstellung 21](#_Toc438422855)

[8.2 Projektplanung 23](#_Toc438422856)

[8.3 MATLAB-Sourcecode 24](#_Toc438422857)

[8.3.1 Nachbau und Vergleich mit dem MATLAB-Befehl spectrogram 24](#_Toc438422858)

[8.3.2 Export der Audiodaten in ein Textfile 26](#_Toc438422859)

[8.3.3 Import der Java-Resultate und deren Auswertung 26](#_Toc438422860)

[8.3.4 Generierung der WAVE-Files für die Android-Tests 27](#_Toc438422861)

[8.4 Java-Sourcecode 27](#_Toc438422862)

[8.5 Android Sourcecode 31](#_Toc438422863)

# Einleitung

Die Analyse von Spektren ist in der Nachrichtentechnik essentiell und sehr aufschlussreich. Bisher konnten nur mit komplexen Programmen und spezieller Hardware Spektren erfasst und dargestellt werden. Jedoch werden Spektren in der Signalanalyse gebraucht und nicht immer steht die spezielle Hardware zur Verfügung. Somit muss eine Alternative geschaffen werden, womit man Signalanalyse betreiben kann ohne im Besitz spezieller Hardware zu sein.

Ein grosser Fortschritt wäre es, wenn man mit seinem mobilen Endgerät, also seinem Handy oder seinem Tablet, Daten erfassen und auswerten könnte. Somit würden die Kosten für die komplexen Programme und die spezielle Hardware entfallen und man könnte mit vorhandenen Geräten eine grobe Analyse der Spektren machen. Für erste Analysen wäre das eine grosse Hilfe und man könnte die komplexen Programme und die spezielle Hardware nur noch in speziellen Fällen nutzen.

Bisher existieren einfache Applikationen für die mobilen Endgeräte, welche jedoch nur das aktuelle Spektrum ausgeben, nicht jedoch ein Spektrogramm. Es ist also bisher nicht möglich den zeitlichen Verlauf eines Spektrums aufzuzeigen.

## Ausgangslage

Diese Projektarbeit ist eine reine Softwareaufgabe und kann somit ohne jegliche Hardware auskommen. Zur Verwendung stehen folgende Programme : MATLAB, Eclipse und Android Studio. Ausserdem wurde das Samsung Galaxy S5 verwendet um die App zu testen.

## Aufgabenstellung

Auszug aus der offiziellen Aufgabenstellung:

„Das Ziel der Arbeit besteht darin, beliebige Audiosignale mit Hilfe einer selbst entwickelten App auf Android-Endgeräten zu analysieren. Die App soll dabei insbesondere in der Lage sein, das Spektrogramm eines Audiosignales mit geeigneten Algorithmen zu berechnen und das Resultat graphisch auf dem Endgerät anzuzeigen. Als Orientierungshilfe kann dabei der MATLAB-Befehl ‚spectrogram’ dienen, der eine ähnliche, wenn auch umfangreichere Funktionalität zur Verfügung stellt. „

Das Endresultat dieser Arbeit soll eine App sein, die ein Spektrogramm aus einer Audiodatei, genau einer .wav-File, generiert. Der Ablauf wurde bereits vor dem Start der Projektarbeit festgelegt. Zuerst soll ein MATLAB-Programm entworfen werden, welches die gleiche Ausgabe erzeugt, wie der Befehl spectrogram. Der so entwickelte Algorithmus soll dann in Java konvertiert werden, zusätzlich soll eine Bibliothek in Java gefunden werden, welche die Fast Fourier Transformation machen kann. Dafür soll die Ausgabe der gefundenen Bibliotheken mit der Ausgabe von der MATLAB-FFT verglichen werden, sollten mehrere Bibliotheken die gleiche Abweichungen haben, soll der Laufzeitunterschied beachtete werden. Die evaluierte Bibliothek und der Algorithmus sollten zum Schluss auf Android programmiert werden und mit einer Oberfläche leicht bedienbar sein.

Optionale Ziele wären zum Beispiel die Verwendung von anderen Audiodateien, ausser der .wav-Datei, das MP3-Format ist weit verbreitet, es wäre von Vorteil, wenn ein bekanntes Format eingelesen werden könnte. Weiter könnte man die Abstufung des Spektrogramms verbessern, so dass nicht nur Sekundenabschnitte gezeigt werden können, sondern auch Abschnitte im Millisekunden Bereich. Ebenfalls könnte man die Overlapp & Add Methode implementieren, um das Spektrum weiter zu verbessern. Sollte nach diesen optionalen Zielen noch Zeit übrig sein, wäre ein weiteres Ziel nicht nur Audiodateien einzulesen, sondern die Audiospur vom Mikrophon oder einem Stream, wie Spotify zu analysieren.

# Der MATLAB-Befehl spectrogram

Das Programm MATLAB stellt den Befehl spectrogram zur Verfügung, welcher ein Spektrogramm anhand von Daten erstellen kann. Das Spektrogramm zeigt ein Diagramm, indem Frequenz, Zeit und Stärke der Frequenz, meist in dB, angezeigt wird. Die Stärke der Frequenz wird anhand von verschieden Farben dargestellt, wie in Abbildung 1 gezeigt. Die Abbildung 1 zeigt das Spektrogramm eines etwa 30 Sekunden langen Songs, es sind vor allem die Frequenzen von 0Hz bis etwa 16kHz vertreten, was ungefähr dem Bereich des menschlichen Gehörs entspricht.

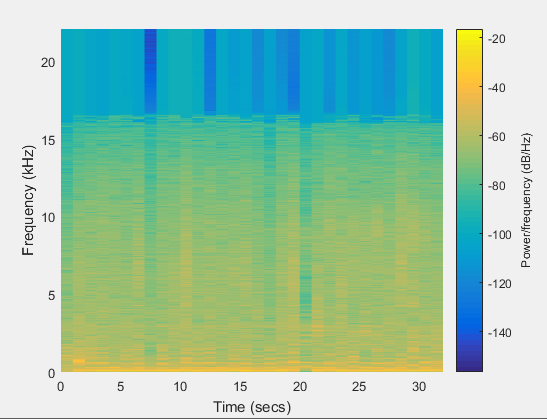


Abbildung 1: MATLAB Ausgabe des Befehls spectrogram für die Audiodatei Cant\_Feel\_My\_Face.wav

## Nachbau des Befehls in MATLAB

Der Nachbau des Befehls spectrogram wurde vor allem mit Hilfe des MATLAB-Befehls fft und imagesc bewerkstelligt. Der Befehl fft wandelt die Audiodaten um und der Befehl imagesc erzeugt ein Bild, was das Spektrum enthält.

Als Erstes werden die Audiodaten eingelesen, das geschieht mit dem Befehl audioread, der ebenfalls zur Werkzeugpallete von MATLAB gehört. Mit Hilfe von audioread können diverse Audioformate eingelesen werden. Jedoch hat man sich auf die Formate MP3 und WAVE begrenzt, da MP3 das populärste ist und das WAVE-Format die rohen Audiodaten enthält. Die Audiodaten im MP3-Format sind komprimiert, also nicht direkt verwendbar, sondern müssen erst dekomprimiert werden.

Nach dem Einlesen werden die Daten in Segmente aufgeteilt. Jedes Segment enthält Audiodaten von einer Sekunde. Was bei einer Samplefrequenz von 44.1kHz, wie bei einer CD üblich, genau 44100 Werten entspricht.

Die einzelnen Segmente werden mit der FFT von MATLAB umgewandelt. Daraus entstehen komplexe Zahlenpaare, welche für eine Darstellung ungeeignet sind. Die Darstellung von komplexen Zahlenpaaren ist nur in einer Ebene möglich, es soll jedoch eine Ebene mit Zeit, Frequenz und Stärke der Frequenz dargestellt werden, mit komplexen Zahlen ist das nicht möglich.

Für eine lesbare Darstellung werden folglich die Beträge der komplexen Zahlenpaare berechnet. Diese Zahlen müssen jedoch noch normiert werden, damit wie im Bild vom Befehl spectrogram in Dezibel dargestellt werden können.

Jedes Segment enthält nun die gewünschten Informationen, also Zeit, Frequenz und Pegel der dazugehörigen Frequenz. Diese Daten können in einem Bild, wie beim Befehl spectrogram, dargestellt werden, das heisst mit Hilfe des Befehls imagesc wird ein Bild generiert und ausgegeben.

## Vergleich der Methoden

Zum Vergleich wurden die generierten Bilder als auch die Daten kontrolliert.

Der Vergleich der Bilder ergab das Unterschiede vorhanden sind. Das Ergebnis ergab, dass die Bilder, wie in Abbildung 2 veranschaulicht, nicht komplett gleich aussehen. Man kann sehen, dass der Befehl spectrogram von MATLAB einen Filter hat, das heisst die Daten werden weichgezeichnet. Die Daten vom MATLAB-Befehl spectrogram sind nicht klar begrenzt, sondern gehen ineinander über, was beim Nachbau nicht der Fall ist. Ebenfalls weicht die Skalierung der Farbanzeige vom Original ab, das heisst die Skala vom MATLAB spectrogram fängt bei -20dB an und geht bis ca. -150dB, der Nachbau fängt schon bei 0dB an und geht bis -150dB. Somit sind die dunkleren Gelbtöne des Nachbaus ungefähr gleich zu setzen mit den helleren Gelbtönen vom MATLAB spectrogram. Trotz der Farbunterschiede sind die Bilder als nahezu gleich zu bewerten, da die entsprechenden Stellen des Nachbaus wesentlich dunkler sind.

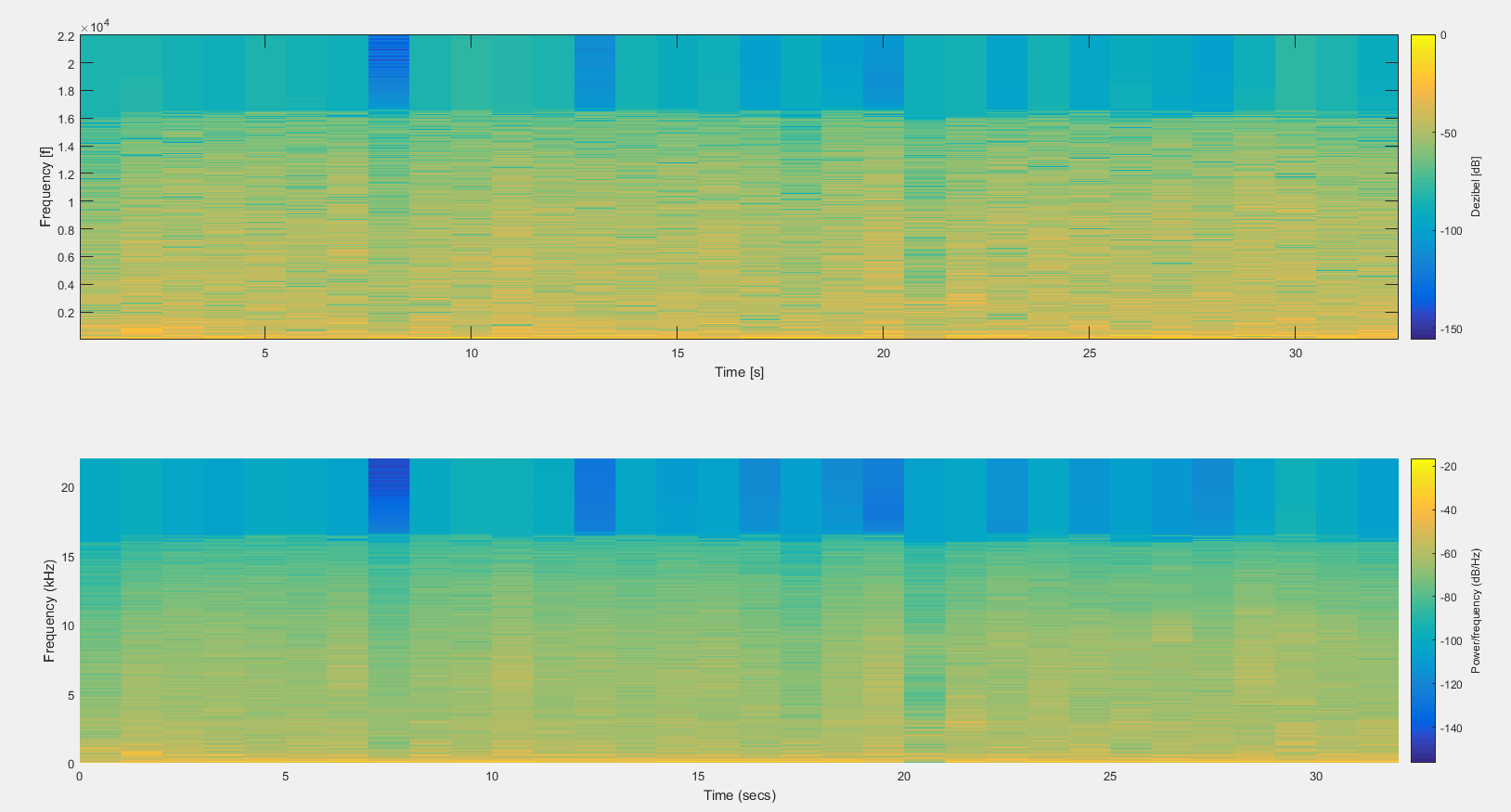


Abbildung 2: Nachgebautes Spektrogramm (oben) vs. MATLAB spectrogram (unten)

Da die Bilder etwas unterschiedlich sind, wurden die Daten dafür umso genauer verglichen. Die Daten vom MATLAB-Befehl spectrogram und die Daten, welche selbst generiert wurden sind komplett identisch. Die Daten, welche vom Befehl spectrogram zurückgegeben werden, enthalten die FFT-Daten, diese werden mit den FFT-Daten vom MATLAB-Befehl fft verglichen.

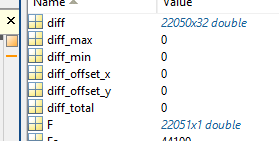


Abbildung 3: Vergleich der Daten Nachbau vs. MATLAB spectrogram

In Abbildung 3 wird ersichtlich, dass die Differenz zwischen den Daten im Minimum und im Maximum genau null ist. Ebenfalls wurde der Offset der beiden Maxima berechnet, also die Position des Maximums von beiden Verfahren. Sowohl die maximale, sowie die minimale Abweichung sind null und auch die Position der Maxima stimmt zu 100% überein.

# Implementation in Java

Nachdem die Methode spectrogram nachgebildet wurde, müssen nun bestehende FFT-Klassen in Java analysiert werden. Der Zeitaufwand, der von Nöten wäre um eine komplett neue FFT-Adaption in Java zu programmieren, wäre sehr gross und andere Personen haben sich schon intensiver damit beschäftigt. Es gibt einige bestehende Klassen, welche bereits optimiert wurden und somit keine oder nur wenige Ressourcen verschwenden, was Laufzeit spart.

Es gibt zwei Kriterien, welche essentiell für die Bewertung der bestehenden Klassen sind. Erstens die Genauigkeit der Werte, welche die Methoden zurückgeben. Die Abweichung darf dabei nicht zu gross sein, gemäss Angabe von Herr Kr. Loeser dürfen die Werte nicht mehr als 0.1‰ von den Werten vom MATLAB-FFT abweichen. Grössere Abweichungen sind nicht akzeptabel, somit wären Klassen mit einer grösseren Abweichung nicht zu gebrauchen.

Der zweite Punkt, welcher betrachtet wird ist die Laufzeit, denn es sollte verhindert werden, dass unnötig Rechenkapazität verschwendet wird, da auf einem mobilen Endgerät nicht übermässig Leistung zur Verfügung steht.

## Die einzelnen Bibliotheken

Insgesamt wurden vier Klassen analysiert. Um die Klassen zu analysieren musste ein Grundgerüst programmiert werden. Um die Daten einfach einlesen zu können, wurden die Daten mit Hilfe von MATLAB in ein .txt-File geschrieben. Repräsentativ wurde von einer Audiodatei, die Daten von einer Sekunde in ein File geschrieben, was bei der verwendeten Audiodatei bedeutet, dass 44100 Werte in das File geschrieben wurden.

Die erste Klasse, namens JTransforms[[1]](#footnote-1), konnte nur komplexe Zahlenpaare verarbeiten, was unnötiger Aufwand wäre, um die vorliegenden Daten überhaupt aufzubereiten. Aus diesem Grund wurde die erste Klasse JTransforms von der Analyse ausgeschlossen. Das würde eine zusätzliche Schleife bedeuten, welche jeden zweiten Wert auf null setzen müsste, da der imaginäre Teil der Audiodatei immer null wäre, denn es handelt sich bei den Audiodaten immer um reale Zahlen. So würde unnötig Speicherplatz verschwenden und die Anwendung verlangsamen, das Ziel ist jedoch eine möglichst schnelle und genaue Anwendung zu generieren.

Die zweite Klasse, namens FFTbase[[2]](#footnote-2), ist nicht optimiert, jedoch kann sie einfach mit realen Zahlenwerten versorgt werden. Es wird ausdrücklich darauf hingewiesen, dass diese Klasse nicht optimiert ist und noch Potenzial besteht die Methoden in der Klasse zu optimieren.

Die dritte Klasse wurde von der Columbia University[[3]](#footnote-3) entwickelt. Genau wie die Klasse FFTbase, ist es sehr einfach die Methode von der Columbia University mit realen Zahlenwerten zu füllen. Die Erwartungen an eine Klasse von einer bekannten Universität sind relativ hoch, selbst wenn die Universität keine Garantie gibt, dass die Klasse einwandfrei funktioniert.

Die vierte und letzte Klasse JFFT[[4]](#footnote-4), welche zu analysieren ist wurde mir von Herr Dr. Loeser vorgeschlagen, da er selbst schon mit ihr gearbeitet hat. Genau wie FFTbase und die Klasse von der Columbia University kann man die Methode einfach mit realen Werten füllen. Der einzige Unterschied zu den anderen Klassen ist, dass man keine Zweierpotenz an Werten zur Verfügung stellen muss, da die Methode auch mit einer anderen Anzahl an Werten Ergebnisse liefert.

Für alle Klassen werden die Anzahl der Werte von 44100 auf 65536 erhöht, dies 216 entspricht und wird von der Klasse von der Columbia University und FFTbase verlangt. Die Werte nach den 44100 echten Werten werden mit Nullen aufgefüllt. Das resultiert in einer besseren Laufzeit und einer besseren Auflösung der Frequenzdaten. Die Klasse JFFT kann auch mit 44100 Werten arbeiten, es wird jedoch empfohlen, dass die Anzahl der Werte eine Zweierpotenz ist, um die Laufzeit zu optimieren. Die Auflösung wurde verbessert, von 1Hz-Schritten zu ungefähr 0.67-Hz Schritten. Somit wurde die Auflösung um ca. 33% verbessert.

## Die Auswertung

Die Genauigkeit ist bei allen drei getesteten Klassen genau gleich, die maximale Abweichung beträgt nur 0.000156 % oder 0.0015631 ‰. Da die Genauigkeit bei allen drei Klassen gleich ist, sind alle drei Klassen in dem Punkt gleich gut. Die Vermutung liegt nahe, dass bei der Ausgabe in ein Textfile die Genauigkeit der einzelnen Klassen beeinflusst wird. Denn es ist sehr unwahrscheinlich, dass alle drei Klassen genau die gleichen Werte liefern.

|  |  |
| --- | --- |
| Differenz FFTbase | 0.00015631% |
| Differenz Columbia FFT | 0.00015631% |
| Differenz JFFT | 0.00015631% |

Tabelle 1: Abweichung der einzelnen Klassen in %

Die Genauigkeit wird bei allen drei Klassen als gleich gut eingestuft. Einen Unterschied findet man bei den Klassen in den Laufzeiten. Die nicht optimierte Klasse FFTbase braucht für 100 FFT-Durchgänge mit je 44100 echten Werten am längsten. Am schnellsten ist die Klasse JFFT, die nur ca. 10 Sekunden benötigt um 100 Durchläufe mit je 44100 echten Werten. Dem Testprogramm werden 44100 Werte übergeben, welches die Anzahl Werte dann auf 65536 erhöht und schlussendlich den Klassen übergibt.

|  |  |
| --- | --- |
| Laufzeit der Klasse JFFT | 10222 ms |
| Laufzeit der Klasse Columbia FFT | 18459 ms |
| Laufzeit der Klasse FFTbase | 28646 ms |

Tabelle 2: Laufzeiten der einzelnen Klassen für 44100 Werte

Anhand der Laufzeiten wurde die Klasse JFFT ausgewählt, um diese in Android zu benutzen. Die Klasse von der Columbia University braucht ca. 80% länger als die Klasse JFFT. Die Klasse FFTbase braucht fast dreimal solange, wie die Klasse JFFT. Dies ist auf die fehlende Optimierung zurückzuführen, die Klasse JFFT wurde sehr stark optimiert.

# Android Applikationsentwicklung

Das Endprodukt ist eine Applikation auf einem mobilen Endgerät. Als Endgerät wird ein Samsung Galaxy S5 verwendet. Nach der Evaluierung der optimalen FFT-Klasse in Java, wird diese Klasse in Android implementiert und eine graphische Bedienoberfläche erstellt. Am Ende kann die Applikation Audiodateien einlesen, mit Hilfe der FFT-Klasse die Daten umwandeln und schlussendlich ein Bild generieren, welches das gewünschte Spektrum enthält. Vorerst sind die Audiodateien auf das WAVE-Format beschränkt, da dieses die einzelnen Audiodaten als Rohdaten enthält. Diese Daten sind nicht komprimiert.

## Einlesen der .wav-Datei

Ein wichtiger Teil der Applikation, wenn auch nicht sichtbar, ist das korrekte Einlesen der Audiodatei. Wichtig ist, dass jedes WAVE-File einen Header besitzt, der essentielle Informationen enthält. Die ersten 44 Byte des Files gehören zum Header und enthalten keine Audiodaten. Hier werden Informationen wie die Samplefrequenz, Bitrate, Bit pro Sample, Anzahl der Kanäle und Bytes pro Sekunde extrahiert. In Abbildung 4 sind alle Informationen eines WAVE-Headers aufgelistet.

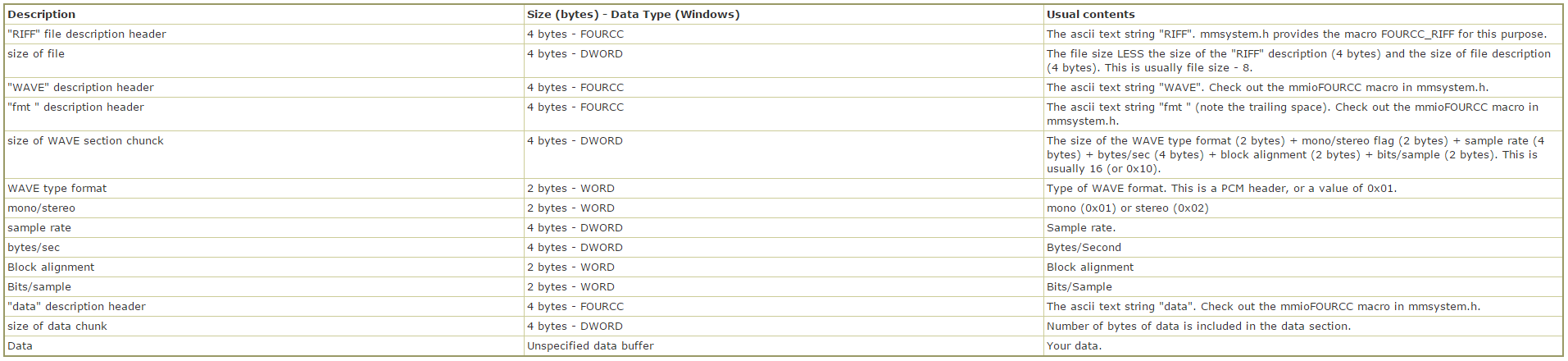


Abbildung 4: Aufbau des Headers im WAVE-Format[[5]](#footnote-5)

Die Audiodaten im data buffer sind je nach Gegebenheiten ebenfalls speziell aufgebaut. Besteht ein Sample aus 8 Bit oder weniger ist der Wert als ein unsigned byte vorhanden. Diese Werte decken einen Bereich von 0 bis 255 ab. Werden jedoch mehr als 8 Bit für ein Sample verwendet, sind die ersten 8 Bit als unsigned byte vorhanden und es muss noch das zweite Byte als signed byte um 8 Bit nach links verschoben addiert werden, wie in Formel 1 ersichtlich. Bei mehr als 8 Bit pro Sample, werden Werte von -32768 bis +32767 abgedeckt.

Formel 1: Berechnung des Wertes eines Audio-Samples mit mehr als 8 Bit

Bei mehr als einem Kanal ist zu beachten, dass nach dem Sample für den linken Kanal, zuerst das Sample für den rechten Kanal ansteht, bevor das nächste Sample für den linken Kanal folgt. Bei mehr als einem Kanal muss also nur jeder zweite bzw. n-te Wert tatsächlich verwendet werden. Theoretisch wird die Anzahl der Bytes im data buffer im Header ausgelesen, jedoch wurde festgestellt, dass diese Zahl nicht immer stimmt. Also muss während dem Einlesen der Daten noch gezählt werden wie viele Werte bereits eingelesen wurden.

## Fast Fourier Transformation

Die eingelesenen Daten werden zu Paketen à einer Sekunde verarbeitet. Zuvor wird die Anzahl Werte pro Sekunde mit Hilfe des Zeropadding auf die nächste Zweierpotenz erhöht. Die Methode JFFT gibt die FFT-Daten im gleichen Array zurück, in dem sie der Methode übergeben wurden. Die Ergebnisse liegen nun als komplexe Zahlen paare vor, es müssen die Beträge berechnet werden. Durch das berechnen der Beträge halbiert sich die Arraygrösse. Nach dem Errechnen der Beträge werden die Werte normiert mit Formel 2. Da ein Grundgerüst bereits in Java programmiert ist, ist die Implementierung in Android nicht sehr kompliziert.

Formel 2: Normierung der Werte auf Dezibel

## Anzeige des Spektrums

Die einzelnen Werte sind nun als Dezibel-Zahlen vorhanden und müssen noch grafisch dargestellt werden. Die Dezibel-Werte müssen skaliert werden, damit sie einer Farbe entsprechen, dafür werden die Werte nochmals normiert und mit einem Farbwert multipliziert.

Eine fordernde Aufgabe war das mehr Datenpunkte vorhanden sind, als Pixel zur Darstellung zur Verfügung stehen, dies bedeutet dass nicht alle Punkte angezeigt werden können. Die Y-Achse oder auch Frequenzachse kann mit ca. 1250 Pixel dargestellt werden, bei Audiodaten mit CD-Qualität, sprich einer Samplefrequenz von 44.1kHz, stehen 32768 Werte zur Verfügung. Da ca. 26 Mal mehr Werte vorhanden sind als effektiv dargestellt werden können, wird jeweils das Maximum von 26 Werten bestimmt und angezeigt.

In der X-Achse oder auch Zeitachse besteht das gegenteilige Problem, denn es stehen mehr Pixel zur Verfügung, als Werte vorhanden sind. Es stehen ungefähr 700 Pixel auf der X-Achse zur Verfügung, jedoch müssen nur wenige Sekunden bis einige Minuten dargestellt werden. Eine Sekunde entspricht dabei einem Punkt auf der X-Achse, das heisst es stehen mehr Pixel auf der X-Achse zur Verfügungen als benötigt werden. Darum müssen die gleichen Werte bei mehreren X-Koordinaten dargestellt werden.

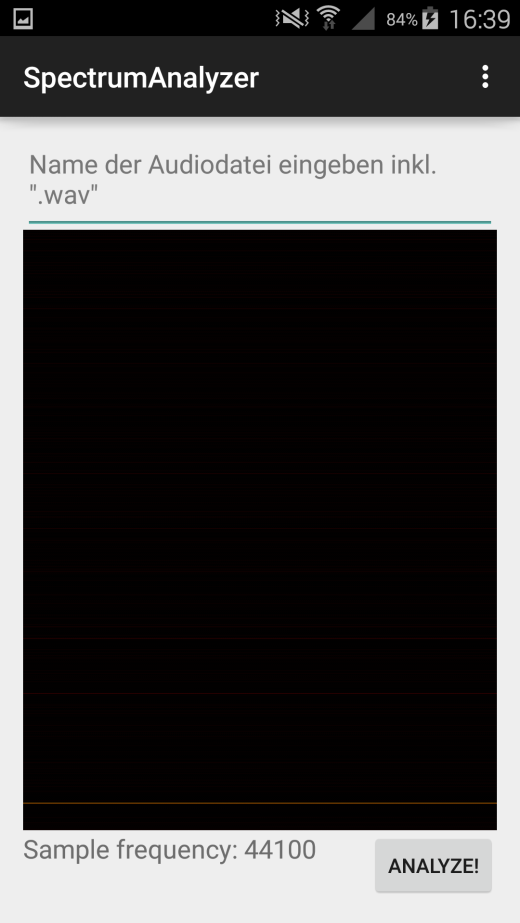
Der Farbverlauf geht von schwarz über rot zu gelb zu weiss, schwarz sind die niedrigsten Werte und weiss werden die höchsten Zahlen dargestellt. Der Verlauf wird immer auf das Maximum und auf das Minimum angepasst, somit ist der höchste dB-Wert immer weiss und der niedrigste Wert ist immer schwarz. Das Bild wird als Bitmap generiert und könnte theoretisch auch als Datei ausgegeben werden, momentan ist diese Funktion jedoch nicht eingebaut.

# Endresultat

Als Endprodukt der Arbeit entstand eine Applikation die auf dem Zielgerät, dem Samsung Galaxy S5, läuft. Momentan braucht die Applikation sehr lange bis ein Bild angezeigt wird, das liegt daran, dass die Programmierung mit vielen Schleifen arbeitet, was eher unvorteilhaft für die Performance ist. Oft wird eine Warnung ausgegeben, das Programm sei abgestürzt, obwohl es noch einwandfrei arbeitet. Die Linien sind sehr dünn, es sollten mehr Werte der Y-Achse zusammengefasst werden, um die Leserlichkeit zu verbessern.

Für Testzwecke wurde ein .wav-File mit Sinuswellen von verschiedenen Frequenzen erzeugt. Der Test mit einem selbst generierten .wav-File war erfolgreich, die Frequenzen wurden erfolgreich aus dem File ausgelesen und wurden gut ersichtlich dargestellt.

In Abbildung 5 wird ersichtlich, wie die Applikation aufgebaut ist. Ganz oben kann man den Namen der .wav-Datei eingeben, welche analysiert werden soll. Mittig findet man das Spektrogramm, mit allen Frequenzen von 0Hz bis zur halben Samplefrequenz. Der untere Teil des Bildes zeigt die tieferen Frequenzen und der obere Teil die hohen Frequenzen. Unter dem Bild befindet sich eine kleine Textausgabe, wo die aktuelle Samplefrequenz angezeigt wird und daneben der Knopf um die Analyse zu starten.



**Textfeld und Button**

Hier wird die aktuelle Samplefrequenz angezeigt. Mit einem Druck auf den Button kann die Analyse der Audiodatei gestartet werden.

**Eingabefeld**

Hier kann der Name der zu analysierenden Datei eingegeben werden.

**Spektrum**

Hier wird das Spektrum angezeigt. Oben sind die hohen Frequenzen bis zur halben Samplefrequenz. Unten befinden sich die niedrigen Frequenzen. Die Stärke der Frequenz wird mit verschiedenen Farben angezeigt. Weiss ist die stärkste Frequenz, gelb ist eine starke Frequenz, rot ist eine schwache Frequenz und schwarz ist die schwächste Frequenz.

Abbildung 5: Test mit 1kHz Sinuswelle

Das Ergebnis ist klar ersichtlich und die Oberfläche ist sehr aufgeräumt. Es befinden sich nur wenige Elemente auf der Oberfläche, was zur Übersichtlichkeit beiträgt. Momentan kann das Spektrum nicht perfekt abgelesen werden, jedoch kann ein grober Überblick verschafft werden.

Wichtig ist der Vergleich mit dem MATLAB Befehl spectrogram, da dieser als Massstab gilt und nachgebaut werden sollte. In Abbildung 6 wurde die gleiche Audiodatei analysiert, wie in Abbildung 5. Es ist ersichtlich, dass der MATLAB Befehl seine Probleme hat und die Linie von 1kHz nicht klar darstellen kann.

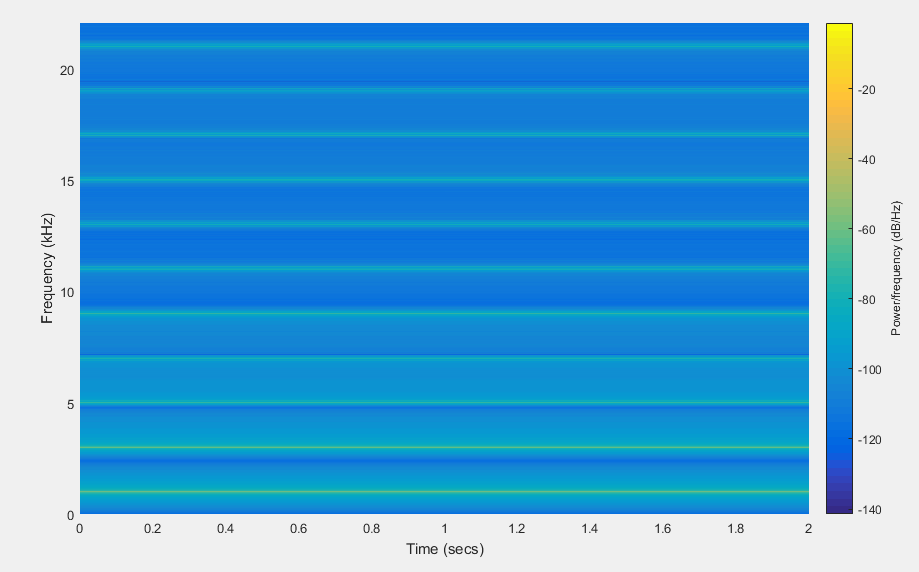


Abbildung 6: MATLAB spectrogram einer Sinuswelle mit 1kHz

In Abbildung 7 wurde die Applikation mit Sinuswellen der Frequenzen 1kHz, 8kHz und 16kHz ausgetestet. Wieder wird ein klar leserliches Bild generiert und die einzelnen Frequenzen sind klar ersichtlich. Die genaue Frequenz kann nicht abgelesen werden, jedoch kann mit der Angabe der Samplefrequenz abgeschätzt werden, welche Frequenzen vertreten sind.

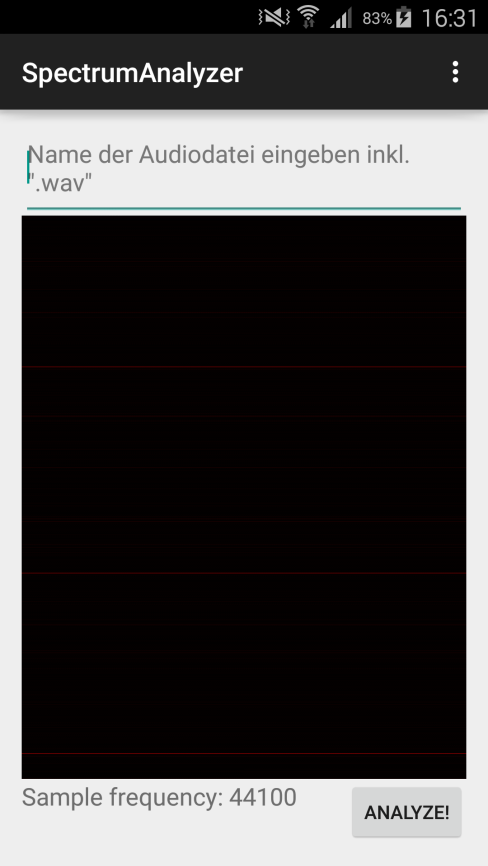


Abbildung 7: Test mit 1kHz, 8kHz und 16kHz Sinuswellen

Erneut kann der MATLAB Befehl nicht überzeugen, denn die einzelnen Frequenzen sind nicht klar ersichtlich, wie in Abbildung 8 dargestellt. Die Linie bei 16kHz und 8kHz ist erkennbar, jedoch kann man die 1kHz Welle nicht klar erkennen. Für die Generierung der Abbildung 8 wurden exakt dieselben Audiodaten verwendet, wie für die Abbildung 7. Um sicher zu gehen, dass sich auf dem Audiofile wirklich Daten befinden wurde es angehört. Man konnte ganz klar einen Pfeifton hören, bereits bei den Audiodaten für die Abbildung 5 und 6 wurde das Audiofile angehört. Der 1kHz-Sinuston ist deutlich hörbar, somit wird ausgeschlossen, dass sich auf dem Audiofile keine Daten befinden. Der akustische Test wurde gemacht, da die Daten auf den ersten Blick unverständlich scheinen. Akustisch konnte aber festgestellt werden, dass sich durchaus Daten im Audiofile befinden.

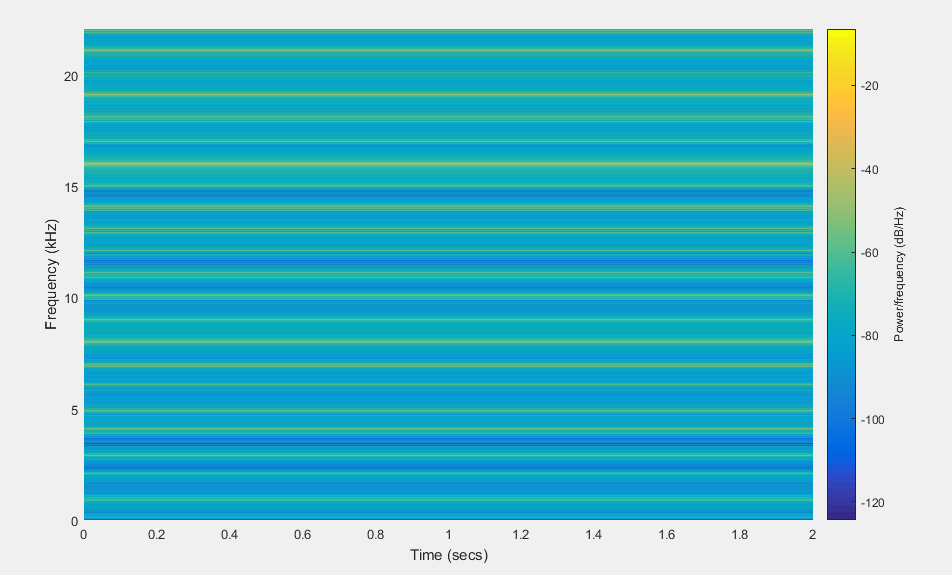


Abbildung 8: MATLAB spectrogram einer Sinuswelle mit 1kHz, 8kHz und 16kHz

Zum Abschluss wurde ein WAVE-File erzeugt, dass die Frequenzen 1kHz, 2kHz, 4kHz, 8 kHz und 16kHz enthält. Die Applikation liefert erneut klare Ergebnisse, es sind deutlich die einzelnen Frequenzen zu erkennen. In Abbildung 9 kann man die einzelnen Frequenzen erkennen und man erkennt auch, dass sich der Abstand zwischen den Frequenzen immer verdoppelt. Genau aus diesem Grund wurden die Frequenzen so gewählt, um zu zeigen wie sich der Abstand zwischen den Frequenzen entwickelt.

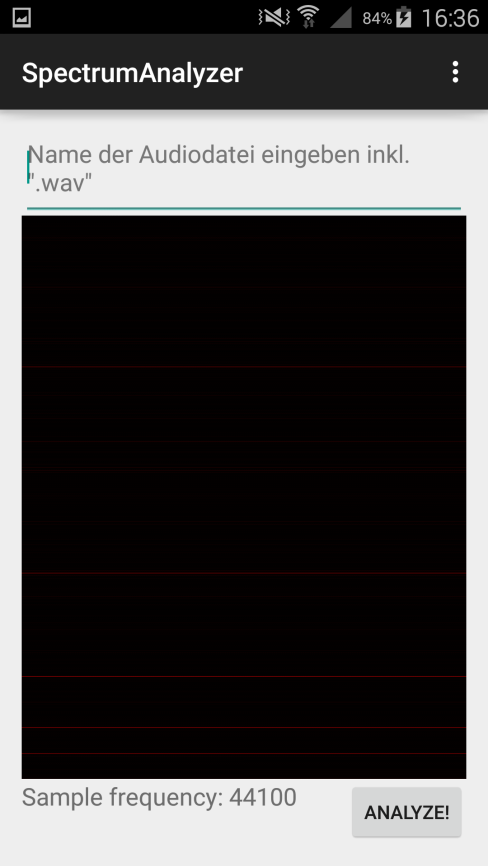


Abbildung 9: Test mit 1kHz, 2kHz, 4kHz, 8kHz und 16kHz Sinuswellen

In Abbildung 10 kann man erneute sehen, dass nicht alle Frequenzen klar ersichtlich sind. Die Frequenz 4kHz kann man knapp erkennen, die Frequenz 8kHz ist deutlich sichtbar und die Frequenz 16kHz ist am deutlichsten erkennbar. Wo sich die Frequenzen 2kHz bzw. 1kHz kann man jedoch nur erahnen.

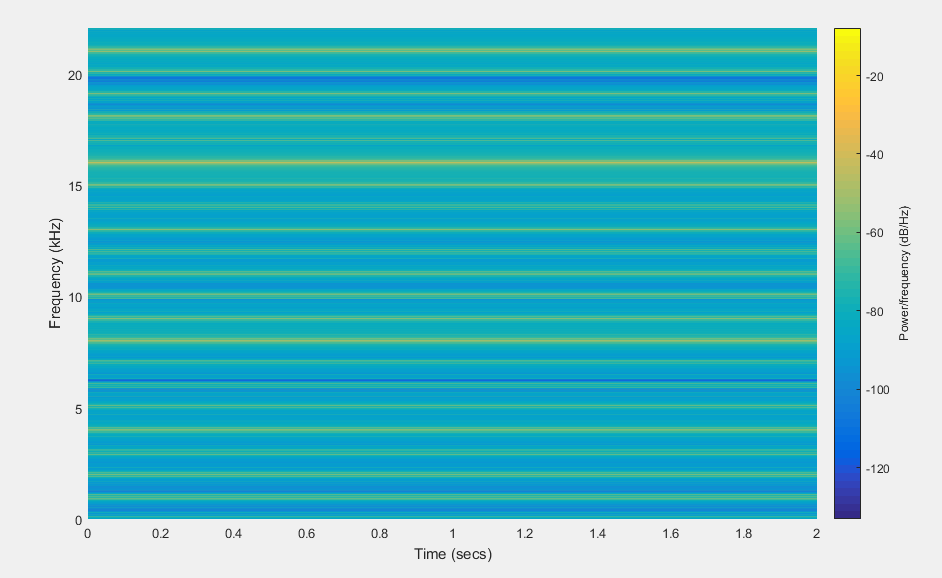


Abbildung 10: MATLAB spectrogram einer Sinuswelle mit 1kHz, 2kHz, 4kHz, 8kHz und 16kHz

# Reflexion / Ausblick

Die Arbeit war sehr interessant und ich habe sehr viel gelernt. Es wurde Wissen von verschiedenen Gebieten verlangt und es war erfreulich, das über Jahre angeeignete Wissen einzusetzen und teilweise zu ergänzen. Trotz der fortgeschrittenen Ausbildung musste viel nachgeschlagen werden und gewisse Gebiete wieder aufgefrischt werden.

Die Applikation in Android ist noch lange nicht perfekt und kann noch weiter ausgebaut werden. Im Rahmen einer Bachelorarbeit könnte man die ganze Applikation verbessern und benutzerfreundlicher, als auch ressourcenschonender gestalten. Momentan werden sehr viele Schleifen zur Verarbeitung der Daten gebraucht, es wäre bestimmt möglich diese Vorgänge effektiver zu gestalten. Ebenfalls könnte man die Qualität der Ausgabe noch verbessern indem man weitere Methoden, wie die Überlappung einfügen würde. Ebenfalls ist die Eingabe von Audiodateien auf .wav-Files beschränkt, es wäre von Vorteil wenn wenigstens das bekannte MP3-Format unterstützt werden würde.

Die Benutzerfreundlichkeit könnte man sicherlich noch erhöhen, wenn der Name der Audiodatei nicht von Hand eingegeben werden müsste, sondern über ein Datenverzeichnis ausgewählt werden könnte. Das Design könnte ebenfalls noch verbessert werden, bis anhin ist die Applikation sehr rudimentär, sie tut was sie machen soll, jedoch nicht mehr. Man könnte auch über weiter Eingabemöglichkeiten nachdenken, es wäre bestimmt interessant die Daten vom Mikrophon zu analysieren oder nach Möglichkeit auch einen Musikstream, wie zum Beispiel von Spotify.

Rückblickend war das Projekt relativ zeitintensiv und ich habe meinen eigenen Zeitplan nicht eingehalten, was gegen das Ende der Projektarbeit sehr stressig wurde. Jedoch wurde diese Zeit mit disziplinierter Vorgehensweise gut überbrückt und ich bin stolz auf mein Ergebnis.

# Verzeichnisse

## Literaturverzeichnis

|  |  |
| --- | --- |
| [1] | Piotr Wendykier, «https://github.com/wendykierp» 8. September 2015. [Online] Available: https://github.com/wendykierp/Jtransforms {Zugriff am 20. Oktober 2015] |
| [2] | Orlando Selenu, « http://www.wikijava.org» 23. Dezember 2009. [Online] Available: http://www.wikijava.org/wiki/The\_Fast\_Fourier\_Transform\_in\_Java\_%28part\_1%29  {Zugriff am 22. Oktober 2015] |
| [3] | Columbia University, «https://www.ee.columbia.edu» 6. Februar 2007. [Online] Available: https://www.ee.columbia.edu/~ronw/code/MEAPsoft/doc/html/FFT\_8java-source.html  {Zugriff am 22. Oktober 2015] |
| [4] | zhangbs, «http://sourceforge.net/» 16. November 2005. [Online] Available: http://sourceforge.net/projects/jfftpack/ {Zugriff am 20. Oktober 2015] |
| [5] | Ring This, « http://www.ringthis.com » [Online] Available: http://www.ringthis.com/dev/wave\_format.htm {Zugriff am 8. Dezember 2015] |

## Abbildungsverzeichnis

[Abbildung 1: MATLAB Ausgabe des Befehls spectrogram für die Audiodatei Cant\_Feel\_My\_Face.wav 8](#_Toc438414745)

[Abbildung 2: Nachgebautes Spektrogram (oben) vs. MATLAB spectrogram (unten) 9](#_Toc438414746)

[Abbildung 3: Vergleich der Daten Nachbau vs. MATLAB spectrogram 10](#_Toc438414747)

[Abbildung 4: Aufbau des Headers im WAVE-Format 13](#_Toc438414748)

[Abbildung 5: Test mit 1kHz Sinuswelle 15](#_Toc438414749)

[Abbildung 6: MATLAB spectrogram einer Sinuswelle mit 1kHz 16](#_Toc438414750)

[Abbildung 7: Test mit 1kHz, 8kHz und 16kHz Sinuswellen 16](#_Toc438414751)

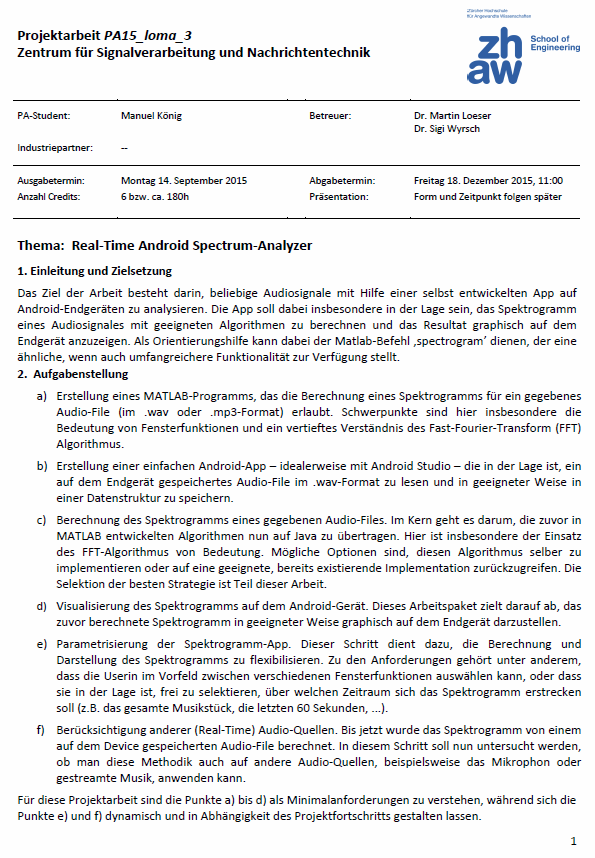
[Abbildung 8: MATLAB spectrogram einer Sinuswelle mit 1kHz, 8kHz und 16kHz 17](#_Toc438414752)

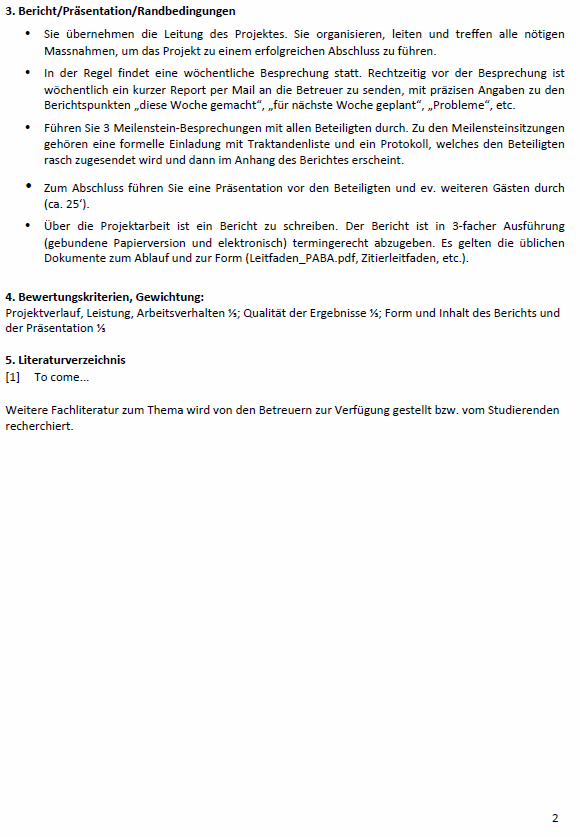
[Abbildung 9: Test mit 1kHz, 2kHz, 4kHz, 8kHz und 16kHz Sinuswellen 18](#_Toc438414753)

[Abbildung 10: MATLAB spectrogram einer Sinuswelle mit 1kHz, 2kHz, 4kHz, 8kHz und 16kHz 18](#_Toc438414754)

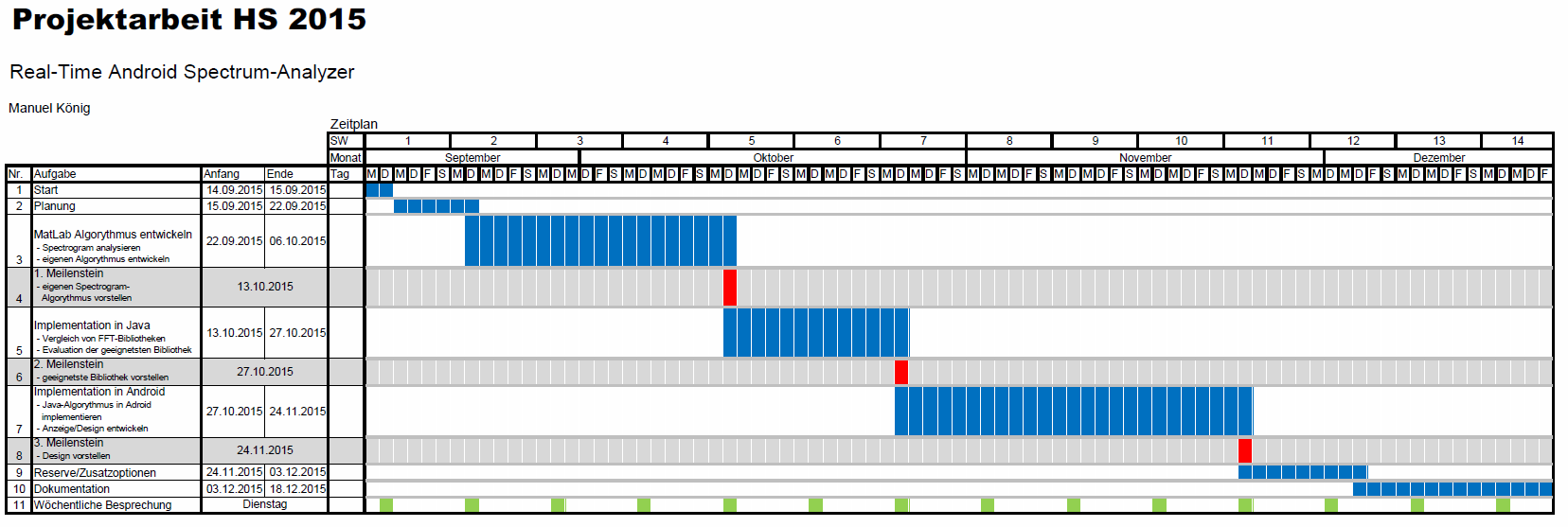
# Anhang

## Offizielle Aufgabenstellung





## Projektplanung



## MATLAB-Sourcecode

### Nachbau und Vergleich mit dem MATLAB-Befehl spectrogram

clc; clear all;

%[y,Fs] = audioread('Bit\_Rush.mp3');

%[y,Fs] = audioread('Bit\_Rush.wav');

%[y,Fs] = audioread('Cant\_Feel\_My\_Face.mp3');

[y,Fs] = audioread('Cant\_Feel\_My\_Face.wav');

%[y,Fs] = audioread('Throught\_it\_all.mp3');

%[y,Fs] = audioread('Throught\_it\_all.wav');

%[y, Fs] = audioread('test.wav');

%sound(y,Fs);

%% linker und rechter Kanal abspeichern

left = y(:,1);

%right = y(:,2);

%% Konstanten definieren

l = length(left);

window1s = cast(l/Fs-.5,'int32'); % cast für Ganzzahlen; -0.5 damit immer abgerundet wird

window\_divi = 1; % 1 = 1 Sekunde; 10 = 0.1 Sekunde

%% Audiosignal in Segmente aufteilen

Z = zeros(Fs/window\_divi,window1s\*window\_divi); % Array definieren

for R = 1:window1s\*window\_divi

if R == 1

Z(:,R) = left(1:R\*Fs/window\_divi,1);

else

Z(:,R) = left((R-1)\*Fs/window\_divi+1:R\*Fs/window\_divi,1);

end

end

%% FFT linker Kanal (unnötig)

% left = fft(left);

% left = abs(left);

% left\_max = max(left);

% left = left/left\_max;

%% FFT rechter Kanal (unnötig)

% right = fft(right);

% right = abs(right);

% right = right/max(right);

%% Audiosegmente mit FFT umwandeln --> Position = Frequenz, Wert = Stärke (Farbe)

Z = fft(Z,Fs);

Z = abs(Z);

Z = 20.\*log10(Z/max(max(Z)));

%% Array auf Fs/2 begrenzen

z = Z(1:Fs/2,:);

% for R = 1:Fs/2

% for H = 1:window1s\*window\_divi

% z(R,H) = Z(Fs/2-R+1,H);

% end

% end

z\_min = min(min(z));

z\_max = max(max(z));

%% Spectrogram in Matrix speichern

[S,F,T] = spectrogram(left,rectwin(Fs/window\_divi),0,Fs/window\_divi,'yaxis');

% Daten in Dezibel umwandeln

S = abs(S);

S\_max = max(max(S));

S = S/S\_max;

S = 20.\*log10(S);

% Maximum und Minimum bestimmen

S\_min = min(min(S));

S\_max = max(max(S));

% Differenzen aus eigenem FFT und Spectrogrammdaten

diff\_max = S\_max - z\_max;

diff\_min = S\_min - z\_min;

% Position vom Maximum der Spectrogramdaten bestimmen und mit Positon vom

% Maximum der eigenen Daten bestimmen

[num, idx] = max(S(:));

[S\_offset\_x, S\_offset\_y] = ind2sub(size(S), idx);

[num, idx] = max(z(:));

[Z\_offset\_x, Z\_offset\_y] = ind2sub(size(z), idx);

diff\_offset\_x = S\_offset\_x - Z\_offset\_x;

diff\_offset\_y = S\_offset\_y - Z\_offset\_y;

%Differenz von eigenen Daten und Spectrogrammdaten bestimmen

diff = S(1:Fs/2,:) - z;

diff\_total = min(min(diff));

%% Ausgabe

figure(1)

% my own Spectrogram

subplot(2,1,1);

imagesc(z)

set(gca,'YDir','normal')

h = colorbar;

h.Label.String = 'Dezibel [dB]';

xlabel('Time [s]');

ylabel('Frequency [f]');

% title('My own Spectrogram');

% Spectrogram Matlab

% figure(2)

subplot(2,1,2);

spectrogram(left,rectwin(Fs/window\_divi),0,Fs/window\_divi,Fs,'yaxis');

### Export der Audiodaten in ein Textfile

clc, clear all;

%[y,Fs] = audioread('Bit\_Rush.mp3');

%[y,Fs] = audioread('Bit\_Rush.wav');

%[y,Fs] = audioread('Cant\_Feel\_My\_Face.mp3');

%[y,Fs] = audioread('Cant\_Feel\_My\_Face.wav');

%[y,Fs] = audioread('Throught\_it\_all.mp3');

[y,Fs] = audioread('Throught\_it\_all.wav');

x = y(:,1);

csvwrite('raw.dat', x');

% x = y(1:Fs,2);

% csvwrite('right.dat',x');

% Fs

### Import der Java-Resultate und deren Auswertung

clc, clear all;

%% Audioimport

[s,Fs] = audioread('Throught\_it\_all.mp3');

left = s(1:Fs,1);

right = s(:,2);

ndim = 2^15;

%% MatLab FFT

Z = fft(left, 2^nextpow2(Fs));

Z = abs(Z);

%% Import

x = importdata('result\_fftbase.dat');

d = x'-Z;

d\_max\_fftbase = max(d)/max(Z)\*100

x = importdata('result\_columbia.dat');

d = x'-Z;

d\_max\_columbia = max(d)/max(Z)\*100

x = importdata('result\_jfft.dat');

Z = Z(1:ndim);

d = x'-Z;

d\_max\_jfft = max(d)/max(Z)\*100

### Generierung der WAVE-Files für die Android-Tests

%% Zeitachse generieren mit 2 Sekunden

x = linspace(0, 2, 88200);

%% Einzelne Sinuswellen genereieren

a = sin(1000\*2\*pi\*x);

b = sin(16000\*2\*pi\*x);

c = sin(2000\*2\*pi\*x);

d = sin(4000\*2\*pi\*x);

e = sin(8000\*2\*pi\*x);

%% Sinuswellen addieren

y = a + b + c + d + e;

%% WAVE-File generieren

audiowrite('test.wav', y, 44100);

## Java-Sourcecode

**import** java.io.BufferedReader;

**import** java.io.FileNotFoundException;

**import** java.io.FileReader;

**import** java.io.FileWriter;

**import** java.io.IOException;

**import** java.util.Arrays;

**import** java.io.\*;

**public** **class** Main {

**private** **static** **double**[] *wave\_values*;

**private** **static** **int** *ndim*;

**public** **static** **void** main(String[] args) **throws** IOException

{

**long** time;

**int** test = 100;

time = System.*currentTimeMillis*(); // Systemzeit in Milisekunden erfassen

// JFFT mehrmals testen

**for**(**int** i = 0; i < test; i++)

{

*test\_jfft*();

}

time = System.*currentTimeMillis*() - time; // Zeitunterschied berechnen

System.***out***.println("Laufzeit für JFFT:\t\t" + time);

time = System.*currentTimeMillis*();

**for**(**int** i = 0; i < test; i++)

{

*test\_columbia\_fft*();

}

time = System.*currentTimeMillis*() - time;

System.***out***.println("Laufzeit für Columbia FFT:\t" + time);

time = System.*currentTimeMillis*();

**for**(**int** i = 0; i < test; i++)

{

*test\_FFTbase*();

}

time = System.*currentTimeMillis*() - time;

System.***out***.println("Laufzeit für FFTbase:\t\t" + time);

System.***out***.println("DONE!");

}

**public** **static** **void** test\_FFTbase() **throws** IOException

{

*generate\_wave*();

FFTbase fftb = **new** FFTbase();

**double**[] img = **new** **double**[*wave\_values*.length]; // Array mit imaginären Werten generieren

Arrays.*fill*(img, 0); // Array mit Nullen füllen

**double**[] result = **new** **double**[*wave\_values*.length\*2];

result = fftb.*fft*(*wave\_values*, img, **true**);

// Beträge berechnen

**for**(**int** i = 0; i < *wave\_values*.length; i++)

{

*wave\_values*[i] = Math.*sqrt*(Math.*pow*(result[i\*2], 2) + Math.*pow*(result[2\*i+1], 2)) \* 256; // \*256 cause of reasons

}

// Werte in result\_fft\_base.dat speichern

FileWriter writer = **new** FileWriter("result\_fftbase.dat", **false**);

**for**(**double** i : *wave\_values*)

{

writer.write(Double.*toString*(i )+ ",");

}

writer.close();

}

// Testroutine für die Klassen von der Columbia University

**public** **static** **void** test\_columbia\_fft() **throws** IOException

{

*generate\_wave*();

Columbia\_FFT cfft = **new** Columbia\_FFT(*nextpow2*(*ndim*));

**double**[] img = **new** **double**[*wave\_values*.length]; // Array mit imaginären Werten generieren

Arrays.*fill*(img, 0); // Array mit Nullen füllen

cfft.fft(*wave\_values*, img);

// Beträge berechnen

**for**(**int** i = 0; i < *wave\_values*.length; i++)

{

*wave\_values*[i] = Math.*sqrt*(Math.*pow*(*wave\_values*[i], 2) + Math.*pow*(img[i], 2));

}

// Ergebnis in result\_columbia.dat schreiben.

FileWriter writer = **new** FileWriter("result\_columbia.dat", **false**);

**for**(**double** i : *wave\_values*)

{

writer.write(Double.*toString*(i )+ ",");

}

writer.close();

}

// Testroutine für die Klasse JFFT

**public** **static** **void** test\_jfft() **throws** IOException

{

*generate\_wave*();

RealDoubleFFT jfft = **new** RealDoubleFFT(*wave\_values*.length);

jfft.ft(*wave\_values*);

*decode\_wave*();

}

**private** **static** **void** decode\_wave() **throws** IOException

{

**double**[] output = **new** **double**[*wave\_values*.length/2];

**int** i;

FileWriter writer = **new** FileWriter("result\_jfft.dat", **false**);

output[0] = *wave\_values*[0];

// Beträge berechnen

**for**(i = 1; i < output.length ; i++)

{

output[i] = Math.*sqrt*(Math.*pow*(*wave\_values*[2\*i], 2) + Math.*pow*(*wave\_values*[2\*i-1], 2));

}

// Werte in result\_jfft.dat schreiben

**for**(i = 0; i < output.length; i++)

{

writer.write(Double.*toString*(output[i]) + ",");

}

writer.close();

}

**public** **static** **void** generate\_wave() **throws** IOException

{

**int** i;

String temp;

String[] parts;

temp = *in\_read*("left.dat");

parts = temp.split(",");

*ndim* = parts.length;

*wave\_values* = **new** **double**[*nextpow2*(parts.length)];

// Werte in Array schreiben

**for**(i = 0; i < parts.length; i++)

{

*wave\_values*[i] = Double.*parseDouble*(parts[i]);

}

// Zeropadding

**for**(i = parts.length; i < *nextpow2*(parts.length); i++)

{

*wave\_values*[i] = 0;

}

}

// Nächste 2-er Potenz zurückgeben

**public** **static** **int** nextpow2(**int** length)

{

**int** value = 1;

**while**(value<length)

{

value = value \* 2;

}

**return** value;

}

// Daten aus Textfile lesen

**public** **static** String in\_read(String path) **throws** IOException

{

String temp;

FileReader reader = **new** FileReader(path);

BufferedReader breader = **new** BufferedReader(reader);

temp = breader.readLine();

reader.close();

**return** temp;

}

}

## Android Sourcecode

package com.example.manuel.spectrumanalyzer;  
  
import android.content.pm.ActivityInfo;  
import android.graphics.Bitmap;  
import android.graphics.Color;  
import android.media.AudioFormat;  
import android.media.AudioManager;  
import android.media.AudioTrack;  
import android.media.MediaPlayer;  
import android.media.\*;  
import android.os.Environment;  
import android.support.v7.app.AppCompatActivity;  
import android.os.Bundle;  
import android.util.Log;  
import android.view.Menu;  
import android.view.MenuItem;  
import android.view.View;  
import android.widget.Button;  
import android.widget.EditText;  
import android.widget.ImageView;  
import android.widget.TextView;  
import android.widget.Toast;  
  
import java.io.BufferedReader;  
import java.io.BufferedWriter;  
import java.io.File;  
import java.io.FileInputStream;  
import java.io.FileNotFoundException;  
import java.io.FileWriter;  
import java.io.IOException;  
import java.io.InputStream;  
import java.io.InputStreamReader;  
import java.io.\*;  
import java.nio.ByteBuffer;  
  
public class MainActivity extends AppCompatActivity {  
  
 double[] data, output;  
 int num\_channels = 0, bytes\_per\_second, bits\_per\_sample, num\_bytes, length\_data, format;  
 double frames;  
 public String path = Environment.*getExternalStorageDirectory*().getAbsolutePath() + "/PA\_Android";  
 int sample\_F = 44100;  
 ImageView bild;  
 Bitmap spectrum;  
 EditText file\_name;  
  
 public TextView display;  
 public Button bsp\_button;

protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 setContentView(R.layout.*activity\_main*);  
  
 display = (TextView) findViewById(R.id.*display*);  
 bsp\_button = (Button) findViewById(R.id.*bsp\_button*);  
 bild = (ImageView) findViewById(R.id.*imageView*);  
 file\_name = (EditText) findViewById(R.id.*path\_editText*);  
 file\_name.setHint("Name der Audiodatei eingeben inkl. \".wav\"");  
 this.setRequestedOrientation(ActivityInfo.*SCREEN\_ORIENTATION\_PORTRAIT*);  
  
 File dir = new File(path);  
 dir.mkdir();  
 }  
  
 @Override  
 public boolean onCreateOptionsMenu(Menu menu) {  
 // Inflate the menu; this adds items to the action bar if it is present.  
 getMenuInflater().inflate(R.menu.*menu\_main*, menu);  
 return true;  
 }  
  
 @Override  
 public boolean onOptionsItemSelected(MenuItem item) {  
 // Handle action bar item clicks here. The action bar will  
 // automatically handle clicks on the Home/Up button, so long  
 // as you specify a parent activity in AndroidManifest.xml.  
 int id = item.getItemId();  
  
 //noinspection SimplifiableIfStatement  
 if (id == R.id.*action\_settings*) {  
 return true;  
 }  
  
 return super.onOptionsItemSelected(item);  
 }  
  
 public void generate\_image(View v){  
 // Höhe und Breite des Bildes ermitteln  
 int width = bild.getWidth();  
 int height = bild.getHeight();  
  
 normalize\_data();  
  
 // leeres Bild erstellen  
 spectrum = Bitmap.*createBitmap*(width, height, Bitmap.Config.*RGB\_565*);  
  
 // Multiplikatoren berechnen  
 double height\_multiplikator = ((double)nextpow2(sample\_F / 2)) / (double)height;  
 double width\_multiplikator = frames / width;  
  
 int i = 0, j = 0;  
 double mean = 0;  
  
 // Farbe für jedes Pixel berechnen  
 while (i < width){  
 j = 0;  
  
 while (j < height){  
 // Position im Array berechnen  
 int x = (int)(i\*width\_multiplikator);  
 x = x\*(nextpow2(sample\_F / 2));  
 x = x + (j \* (int)(height\_multiplikator));  
 mean = 0;

// Maximum der nächsten Werte ermitteln  
 for (int z = 0; z < (int)height\_multiplikator; z++){  
 if (mean < output[x + z]){  
 mean = output[x + z];  
 }  
 }  
  
 // Pixel einfärben  
 if (x < output.length){  
  
 spectrum.setPixel(i , height - j - 1, get\_color(mean));  
 }  
 else {  
 spectrum.setPixel(i, j, Color.*BLACK*);  
 }  
  
 j++;  
 }  
  
 i++;  
 }  
  
 // generiertes Bild ausgeben  
 bild.setImageBitmap(spectrum);  
 display.setText("Sample frequency: " + String.*valueOf*(sample\_F));  
  
  
 }  
  
  
 public void normalize\_data(){  
  
 double max = get\_max(output, output.length);  
  
 // dB Wert berechnen  
 for (int i = 0; i < output.length; i++){  
 output[i] = 20\*Math.*log10*(output[i]/max);  
 }  
  
 // Farbwert berechnen  
 for (int i = 0; i < output.length; i++){  
 output[i] = output[i] / max \* 767;  
 }  
 }  
  
 // Farbe zurückgeben  
 public int get\_color(double value){  
 int color = 0;  
 int red = 0x00FF0000;  
 int green = 0x0000FF00;  
  
 if (value < 256){  
 color = (((int)value) << 16);  
 }  
 else if (value < 512){  
 color = (red + ((int)(value-256) << 8));  
 }  
 else if (value < 768){  
 color = (red + green + ((int)(value - 512)));  
 }  
 else {  
 color = 0;  
 }  
  
 return color;  
 }

// Maximum eines Arrays bestimmen  
 public double get\_max(double[] temp, int length){  
 double maximum = temp[0];  
  
 for (int i = 1; i < length; i++){  
 if (temp[i] > maximum)  
 {  
 maximum = temp[i];  
 }  
 }  
  
 return maximum;  
 }  
  
 public void fft\_audio(){  
 RealDoubleFFT jfft = new RealDoubleFFT(nextpow2(sample\_F));  
 double[] temp = new double[nextpow2(sample\_F)];  
  
 frames = Math.*round*(length\_data / sample\_F); // Anzahl Frames berechnen, sprich Anzahl Sekunden  
  
 int length = (int)(frames) \* ((nextpow2(sample\_F) / 2));  
  
 output = new double[length];  
  
 int count = 0;  
  
  
 for (int i = 0; i < length\_data; i = i + sample\_F){  
 // Zeropadding  
 for(int j = 0; j < nextpow2(sample\_F); j++){  
  
 if((j < sample\_F) && (length\_data > (i + j))){  
 temp[j] = data[i + j];  
 }  
 else{  
 temp[j] = 0;  
 }  
 }  
  
 // FFT  
 jfft.ft(temp);  
  
 // Resultat in Array schreiben  
 if(count<output.length){  
 output[count] = temp[0];  
 count++;  
 }  
  
 int j = 1;  
  
 //Beträge berechnen  
 while(j < temp.length / 2 && count < output.length){  
 output[count] = Math.*sqrt*(Math.*pow*(temp[2 \* j], 2) + Math.*pow*(temp[2 \* j - 1], 2));  
 count++;  
 j++;  
 }  
 }  
 }

// nächste 2-er Potenz berechnen  
 public int nextpow2(int number){  
 int value = 1;  
 while (value < number){  
 value = value \* 2;  
 }  
 return value;  
 }  
  
 // Werte in File schreiben  
 public void write\_data(){  
  
 File file = new File(path + "/result.txt");  
  
 try{  
 FileWriter fw = new FileWriter(file);  
 BufferedWriter bw = new BufferedWriter(fw);  
  
 for (int i = 0; i < output.length; i++){  
 bw.write(String.*valueOf*(output[i]) + ",");  
 }  
 bw.close();  
  
 }catch (Exception e){e.printStackTrace();}  
 }  
  
  
 public void audio\_read(){  
  
 File file = new File(path + "/" + file\_name.getText());  
 //File file = new File(path + "/test.wav");  
  
 byte[] header = new byte[44];  
 int[] uns\_header = new int[44];  
  
 try{  
 InputStream inputStream = new FileInputStream(file);  
 BufferedInputStream bis = new BufferedInputStream(inputStream);  
 DataInputStream dis = new DataInputStream(bis);  
  
 // Header einlesen  
 int i = 0;  
 while (i < 44 && dis.available() > 0){  
 header[i] = dis.readByte();  
 uns\_header[i] = header[i] & 0xFF;  
 i++;  
 }  
  
 // wichtige Informationen abspeichern  
 sample\_F = uns\_header[24] + (uns\_header[25]<<8) + (uns\_header[26]<<16) + (uns\_header[27]<<24);  
 num\_channels = uns\_header[22] + (uns\_header[23]<<8);  
 bytes\_per\_second = uns\_header[28] + (uns\_header[29]<<8) + (uns\_header[30]<<16) + (uns\_header[31]<<24);  
 bits\_per\_sample = uns\_header[34] + (uns\_header[35]<<8);  
 num\_bytes = uns\_header[40] + (uns\_header[41]<<8) + (uns\_header[42]<<16) + (uns\_header[43]<<24) + 44;  
 format = uns\_header[20] + (uns\_header[21]<<8);

// Audiodaten einlesen  
 i = 0;  
 byte black\_hole;  
 data = new double[6000000];  
 while (dis.available() > 0){  
  
 if (num\_channels > 1){  
 data[i] = (dis.readByte() & 0xFF);  
  
 if ((dis.available() > 0) && (bits\_per\_sample / 8 > 1)){  
 data[i] = ((int)(data[i]) + (dis.readByte() << 8));  
 }  
  
 if ((dis.available() > 0) && (bits\_per\_sample / 8 > 1)){  
 black\_hole = dis.readByte();  
 }  
  
 if (dis.available() > 0){  
 black\_hole = dis.readByte();  
 }  
 }  
 else {  
 data[i] = (dis.readByte() & 0xFF);  
  
 if ((dis.available() > 0) && (bits\_per\_sample / 8 > 1)){  
 black\_hole = dis.readByte();  
 }  
 }  
  
 i++;  
 }  
  
 length\_data = i;  
  
 dis.close();  
 }catch(Exception e){  
 e.printStackTrace();  
 display.setText(e.toString());  
 }  
 }  
  
 // Methode für den Button  
 public void buttonOnClick(View v){  
 audio\_read(); // Audiofile einlesen  
 if (num\_channels > 0){ // wenn die Audiodaten korrekt eingelesen wurden, weiter machen  
 fft\_audio();  
 write\_data();  
 generate\_image(v);  
 }  
 }  
}

1. Bestehende FFT-Klasse JTransforms für Java [↑](#footnote-ref-1)
2. Bestehende FFT-Klasse FFTbase für Java [↑](#footnote-ref-2)
3. Bestehende FFT-Klasse der Columbia University für Java [↑](#footnote-ref-3)
4. Bestehende FFT-Klasse JFFT für Java [↑](#footnote-ref-4)
5. Header des WAVE-Formats [↑](#footnote-ref-5)