Frequenzerkennung in digitalen Audiodaten am Beispiel eines Stimmgeräts

18. Dezember 2020

Zusammenfassung

Ein Stimmgerät das wirklich gut funktioniert ist so intuitiv, dass es dem Benutzer die Schritte zur Stimmung seines Instruments anschaulich erklärt. Aus der konkreten Anwendung ergeben sich Anforderungen auf der technischen Seite, die hier implementiert werden sollen. Auf dem Weg dahin werden einige Grundlagen gelegt; es ist nicht hinderlich wenn man schon mal ein „hallo welt“ Programm geschrieben hat.

„[...] da schreiben wir mal eben ein Programm [...]“

„[...] it's not just the fft [...]“

Teil I
Fork

```
// initialize portaudio
err = Pa_Initialize();
if( err != paNoError ) goto error;
```

1 Warum?

In meiner etwa 30 Jahre andauernden Beschäftigung mit der Gitarre habe ich viele Verfahren kennengelernt, das Instrument in Stimmung zu bringen. Im Nachhinein wüßte ich gern, wie lange ich mich damals mit schlecht gestimmten Instrumenten begnügt habe und es nicht besser wußte... denn Stimmen ist das täglich Brot und nicht eben leicht erlernt. Das erste Werkzeug hierfür war eine Stimmpfeife, welche die sechs Grundtöne der Gitarre erzeugen konnte. Nur das Spektrum dieser Töne war dem der sechs Saiten so verschieden, dass es äußerst schwer war dies aus dem Gehör heraus auf einen Nenner zu bringen. Mit viel Leidensfähigkeit und Experimentiergeist ließ sich in die Nähe

Abbildung 1: Stimmpfeife

kommen. Das nächste - die fünfter Bund Methode. Ein motorisch anspruchsvoller Vorgang, die Hände entgegen dem beim Gitarre Spielen verwendeten Sinn einzusetzen und zwei Töne gleichzeitig zum Klingen zu bringen während der Stimmwirbel verdreht wird. Hierbei ist die Ähnlichkeit der Töne zueinander größer, aber ein entwickeltes Gehör ist erforderlich um tatsächlich eine annähernde Gleichheit herzustellen. Je nach Gitarre ist hier außerdem ein Fehler wegen der Bundunreinheit wahrscheinlich. Mit fortgeschrittener Spieltechnik kommen die Obertöne als Werkzeug zum Gitarre Stimmen in Reichweite, endlich ein Lichtblick. Jahrelang meine Lieblingsmethode. Dann noch die Stimmgabel mit 440 Hertz dazu, und das Drehen an der Mechanik macht schon etwas mehr Spaß. Schließlich fand ich irgendwann heraus, daß keine Gitarre jemals perfekt gestimmt sein kann - das Zusammenspiel von Saitenspannung, Saitenlänge, Bundpositionen, Steg- und Sattelpositionen ist ein immer anders ausgestalteter Kompromiß.

In der Vermittlung von Gitarrenfähigkeiten habe ich festgestellt, daß Stimmgeräte oftmals bei den tieferen Tönen Probleme haben. An der tiefen E-Saite scheiden sich die Geister, oder die Spreu vom Weizen was die Stimmgeräte angeht. Viele billige Varianten haben die Krankheit der Wackeligkeit - besonders einige Digitalgeräte. Sie springen zwischen Werten herum, und selbst wenn der Notenname korrekt erkannt wird kann sich das Gerät manchmal nicht für „die Mitte“ entscheiden.

2 Evolution der digitalen Stimmgeräte

Von der Stimmpfeife ging es zunächst zu Stimmgeräten als Bodeneffektgerät, Rackgerät und Clip-On Tuner. Die Anklemmgeräte markierten schon einen Meilenstein in der Praktikabilität, aber nicht immer funktioniert hier die Tonerkennung so sauber wie gewünscht. Je nach Gitarre ist hier auch ein erfahrener Benutzer notwendig; die Benutzung erfordert ein wenig Einübungszeit. Auch die Schwierigkeit der tiefen E-Saite bleibt vielen Geräten erhalten. Mit dem Erscheinen der digitalen Endgeräte in fast jedermanns Hand - Smartphones - beginnt nun eine neue Zeitrechnung der Stimmgeräte. Auf einmal ist ein Stimmgerät in der Lage, auf traumwandlerisch ruhige und intuitiv verständliche Art und Weise zu funktionieren. Was ist hier passiert, worin liegt nun der

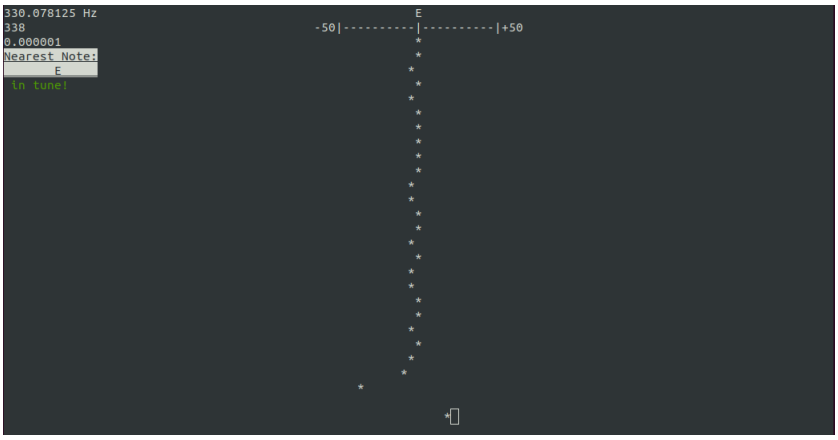

Abbildung 2: ccgt im Terminal

geniale Streich dieser neuen Generation von Stimmgeräten?

3 Not necessarily from scratch

Es finden sich im Internet unzählige Tuner-Algorithmen für diverse Plattformen. Definitiv muss hier das Rad nicht neu erfunden werden, nur sollte der Anfang nicht allzu umfangreich sein. Um ein grundlegendes Verständnis der digitalen Signalverarbeitung herzustellen und einen Rahmen vorzugeben, wird hier daher Bjørn Roches Code[1] verwendet. Dabei handelt es sich um einen in der Programmiersprache c geschriebenen Tuner mit dem Schwerpunkt auf Verständlichkeit des Codes. Die Grundzüge der Verarbeitung sind:

- Lese Audio-Daten für die Fourier-Transformation
- Verwende Tiefpass-Filter auf den Daten
- Verwende eine Fensterfunktion auf den Daten
- Wende Fourier-Transformation an
- Finde einen Maximalwert in den Daten
- Finde die Hauptfrequenz anhand des erzeugten Maximalwerts

Anhand dieses Aufbaus wird im Folgenden unter Berücksichtigung einiger Voraussetzungen der Signalweg nachvollzogen.

[1] http://blog.bjornroche.com/2012/07/frequency-detection-using-fft-aka-pitch.html

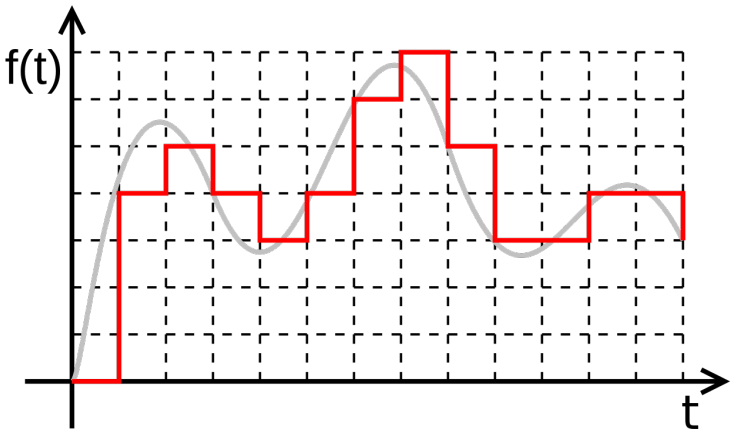

Abbildung 3: Digitale Abtastung

Teil II
Durchführung

```
applyfft( fft, data, datai, false );
```

4 Digitale[2] Audiosignale

Alle Klänge der realen Welt, die ihren Weg in den Rechner finden, leben dort nur als Zahlen. Während die Schwingungen der Luftteilchen kontinuierlich stattfinden, so wird von einem Rechner nur ein Zeitpunkt dieser Schwingung erfasst und durch eine Zahl dargestellt.

Wenn also unser Programm Klänge aufnimmt, werden in irgend einer Form Luftschwingungen in Zahlen umgewandelt. Hierbei gibt es definierte Zeitpunkte, auch genannt die Samplerate. Ein Sample ist eine Abtastung, und die Frequenz der Abtastung gibt an wie häufig pro Sekunde abgetastet wird. Dementsprechend wird die Samplerate in Hertz angegeben.

4.1 Wahrnehmung von Geräuschen und Klängen

Nehmen wir an ich erzeuge ein beliebiges Geräusch; zum Beispiel tippe ich mit meinem Bleistift gegen den Tisch. Als einzelnes Geräusch höre ich eine Art

[2]Lateinisch „digitus“ ist das Wort für Finger, also bedeutet digital so in etwa alles (an den Fingern) abzählbare. Auch wenn heutige Rechner viele „Finger“ haben.

„Tick“. Würde ich schneller gegen den Tisch tippen, gäbe es eine schnellere Folge von Ticks. Ab einer gewissen Schnelligkeit würde das Ticken jedoch umkippen in einen Ton mit einer Frequenz. Ab einer Frequenz von 20 Luftdruckschwingungen pro Sekunde und aufwärts würden wir das wahrnehmen, was vielleicht Ton genannt werden kann - zumindest kann von einer periodischen Schwingung geredet werden. In Musikinstrumenten schwingen Rohrblätter, Saiten und andere Dinge und erzeugen eben diese mehr oder weniger periodischen Schwingungen. Wer das Phänomen einmal klangsinnlich erleben möchte, dem sei der Anfang des Stücks „Gamma Ray“ der Krautrocker Birthcontrol[3] empfohlen. Hier übernimmt ein Synthesizer die Aufgabe, von schnellen Impulsen zu einem wahrnehmbaren Ton überzugehen.

Das menschliche Gehör ist ein für uns überlebenswichtiges Organ, das aus einer langen Evolutionsgeschichte hervorgegangen ist. Mit anderen Worten, wir sind im Idealfall zu einer sehr feinsinnigen Abtastung unserer akustischen Umwelt befähigt. Wie steht es hingegen mit dem Rechner? Auf welcher Grundlage arbeitet er? Die eine Grundbedingung der Samplerate wurde oben schon genannt. Es gibt eine feste Anzahl Abtastungen pro Sekunde. Wenn der Mensch ab einer Frequenz von 20 Hz in der Lage ist, einen Ton wahrzunehmen... wie oft müßte der Computer dann die Luftschwingung abtasten um ebensolches zu leisten?

4.2 Wie hört der Rechner

Das Abtasttheorem erklärt, wie oft ein Signal abgetastet werden muss um eine gewisse Frequenz darstellen zu können. Die Abbildung auf der folgenden Seite[4] versucht optisch zu verdeutlichen, wie sich Frequenzen oberhalb der halben Abtastrate verhalten. Die halbe Abtastrate spielt für die im Rechner darstellbaren Frequenzen eine wichtige Rolle. Wie kommt es dazu, dass die Grenze dessen was an Frequenzen darstellbar ist ausgerechnet bei der Hälfte der Rate liegen?

Würde der Rechner das Brummen bei 20 Hz hören? Ja, wenn er oft genug pro Sekunde lauscht. Nehmen wir an ich wohne direkt neben einer Straße, auf der im Abstand von genau einer Minute jeweils genau ein Auto an mir vorbeirauscht. Würde ich die ganze Zeit aufmerksam lauschen, dann würde das Fahrgeräusch zunächst immer lauter werden. Schließlich erreicht der Krach seinen Höhepunkt und ebbt wieder gemächlich ab, bis sich irgendwann das nächste Auto durch lauter werdendes Rauschen ankündigt. Allerdings sitze ich zu allem Übel mit Kopfhörern vor dem Rechner und kann die Geräusche der Außenwelt nicht wahrnehmen. Würde ich alle 60 Sekunden genau im Moment der Stille die Kopfhörer abnehmen um zu lauschen, könnte ich zu der Auffassung gelangen es führen keine Autos auf der Straße. Es könnte auch passieren, dass ich den Eindruck bekomme es führen die ganze Zeit Autos vorbei - wenn ich immer im Höhepunkt der Fahrgeräusche lausche.

[3] https://www.youtube.com/watch?v=Vzlv7LFmLMg

[4] freundlicherweise von Peterpall - Eigenes Werk, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=8730560 zur Verfügung gestellt.

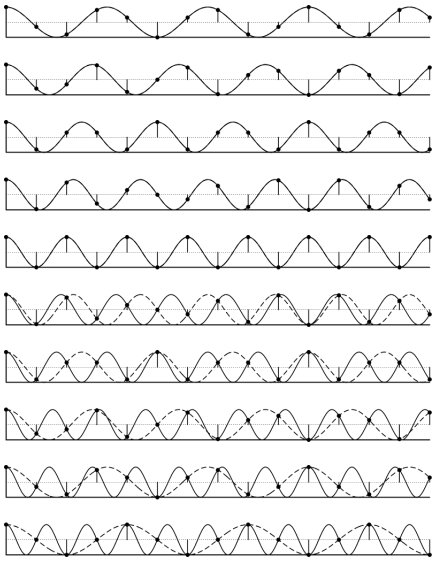

Abbildung 4: Abtasttheorem

Der Computer macht es wie ich: abgetastete Tonsignale stellen keine kontinuierliche Welle dar, sondern eine Ansammlung von Punkten zwischen denen nichts ist. Abbildung 5 verdeutlicht dies - die hohen Frequenzen sind sehr schnelle „Autos“, die einfach zwischen den Punkten durchfahren und eigentlich nicht bemerkt werden. In einem periodischen Tonsignal sind manchmal unglaublich viele „Autos“ unterschiedlicher Geschwindigkeit unterwegs; und es kann bei falschen Voraussetzungen passieren, dass man einen falschen Eindruck vom „Verkehr“ bekommt. Da das menschliche Gehör im Idealfall Schwingungen bis etwa 20000 Hz wahrnehmen kann, würde eben eine Abtastrate von 20000 nicht ausreichen um all das abzubilden. In der Praxis findet sich daher der Wert 44100, das Doppelte der hörbaren Schwingungen plus ein wenig „headroom“.

Zusätzlich zur zeitlichen Diskretisierung wird auch noch die Amplitude in Zahlen umgewandelt. Auch die Lautstärke wird nicht kontinuierlich dargestellt, sondern auf einer lückenhaften Skala. Also je lauter ein Signal ist, desto höher der zugewiesene Wert in der resultierenden Serie von Zahlen. Hierbei kommen meist 8 oder 16 Bit pro Abtastwert zum Einsatz. Je mehr Bit das Audioformat zuläßt, umso größer können die Werte der Zahlenreihe werden.

```
recorder = new AudioRecord(AUDIO_SOURCE,
SAMPLE_RATE, CHANNEL, AUDIO_FORMAT, BUFFER_SIZE);
```

Wenn man die Verarbeitung von Eingabesignalen als das „Hören“ des Computers bezeichnet, so ist selbst dieser Prozeß diskret. Der Rechner kann nicht ohne Weiteres wie das menschliche Gehör eine Einheit von Wahrnehmung

und Verarbeitung zur gleichen Zeit darstellen. Es wird immer eine definierte - endliche - Zahlenmenge bearbeitet und das Ergebnis zurückgegeben; natürlich als Zahl. Dann kann eine nachfolgende Zahlenmenge bearbeitet werden und so weiter. Natürlich sind Rechner bisweilen in der Lage so schnell zu rechnen, dass der menschlichen Wahrnehmung vorgegaukelt wird es wäre annähernd kontinuierlich was da passiert. Die Änderungen sind im Millisekundenbereich. Dennoch muß es für die Bearbeitung der Audiodaten eine Grenze geben, anhand derer einzelne Bearbeitungsschritte definiert werden können. Hier kommt der Puffer ins Spiel.

4.3 Puffer: Immer schön hinten anstellen

Die Größe des Puffers bestimmt die Menge an Audiodaten, auf denen die Berechnungen durchgeführt werden. Nehmen wir an ich bin in der Küche eines Restaurants angestellt, und soll Kartoffeln schälen. Die Kartoffeln liegen im Keller. Ich gehe nicht für jede Kartoffel in den Keller, schäle sie dann und lege sie in den Topf - ich hole einen Topf voll. Diese Menge an Kartoffeln verarbeite ich als eine Untermenge der gesamten Kartoffeln. Wenn ein Topf kocht - sich also im nächsten Bearbeitungsschritt befindet - könnte ich losgehen und die nächste Ladung Kartoffeln holen. Es ist eigentlich unwahrscheinlich, dass an einem Tag so viele Kartoffeln gegessen werden. Aber egal wie schlecht das Beispiel ist, die Größe meines Topfes entspricht der Puffergröße für die Verarbeitung der Audiodaten. Der Rechner bekommt die ganze Zeit Daten, aber er kann sie nur in definierten Portionen verwerten. Das Audiosystem benötigt eine feine zeitliche Auflösung um die gewünschten Frequenzen erkennen zu können; und auch die Puffergröße kann einen gewissen Wert weder übernoch unterschreiten. Vielleicht hat der eine oder andere schon mal das Problem der Latenzzeit am Rechner gehabt und infolge dessen an seiner Puffergröße herumgestellt. Dabei stellt man fest, dass als Größen oft Potenzen von 2 vorkommen.

```
err = Pa_ReadStream( stream, data, FFT_SIZE );
```

Im einfachsten Fall ist die Puffergröße gleich der Größe des fft-Arrays. Es gibt natürlich ausgefeiltere Pufferkonzepte[5] um das Ergebnis zu verbessern. Doch zunächst einmal vereinfachen wir dahingehend, dass pro Berechnungsperiode einmal Audiodaten gelesen werden und dann auf diesem „Screenshot“ der Klänge herumanalysiert wird.

[5] https://de.wikipedia.org/wiki/First_In_%E2%80%93_First_Out

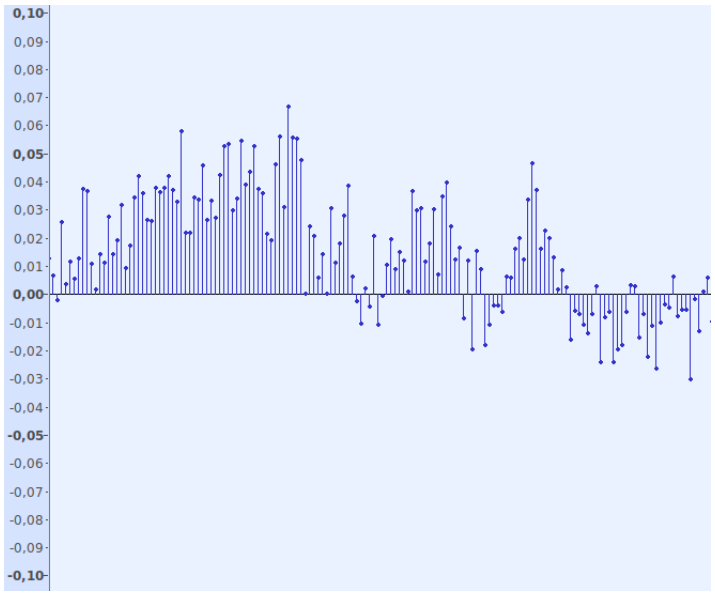

Abbildung 5: Abtastwerte auf der Zeitachse

5 Fouriertransformation

5.1 Dimensionswechsel

Wenn das Programm Audiodaten eingelesen hat, liegen diese als zeitdiskrete Werte im Speicher. Sie repräsentieren die Auslenkung der Luftteilchen - die Amplitude zu einem gegebenen Zeitpunkt. Die Daten sind also auf der Zeitachse angeordnet und haben jeweils eine gewisse „Höhe“, wenn die Vorstellung als Wellenform hilft. Es liegt ein Vektor von Zahlen vor, dessen Länge der Puffergröße n entspricht.

$$v = [v_{0,} v_1, ..., v_{n-1}]^T \; v \, \epsilon \, \mathbb{R}$$

In unserem Code sind die n Elemente dieses Vektors als Fließkommazahlen abgebildet, so daß Werte von -1 bis 1 vorkommen. Der vorliegende Vektor v wird durch die Fouriertransformation in einen Vektor c der gleichen Länge umgewandelt:

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{pmatrix} = ft \begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ \vdots \\ v_{n-1} \end{pmatrix}$$

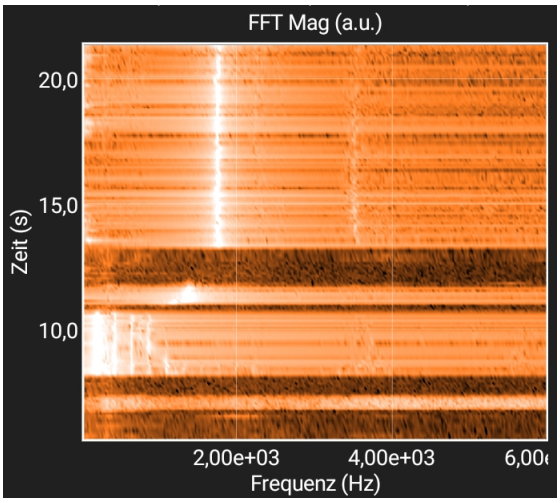

Abbildung 6: Wasserfall-Display

Es entsteht also wiederum eine Zahlenfolge der entsprechenden Länge, nur dass diese nicht mehr die Amplituden sondern Frequenzen repräsentieren. Der Vektor c

$$c = [c_0, c_1, ..., c_{n-1}]$$

enthält nach Durchführung der Fouriertransformation das errechnete Frequenzspektrum des Eingangssignals. Hatte man vorher den gesamten Cocktail im Glas, so ist nun jeder Bestandteil des Gemischs im Glas getrennt voneinander vorhanden - zumindest in einer idealisierten Betrachtung. Vorher war der Kuchen fertig gebacken, nun ist er aufgeteilt in die Bestandteile seiner Herstellung. Die Fouriertransformation führt vom fertigen Produkt zum Rezept. Die Transformation macht aus Amplituden auf der Zeitachse - Amplituden auf der Frequenzachse. Anschaulich läßt sich dies beispielsweise an einem Wasserfall-Display[6] erleben, in dem die spektralen Anteile abhängig von der Zeit farblich abgesetzt angezeigt werden.

Wie aber funktioniert die Fouriertransformation wirklich? Was passiert mit den Eingangsdaten, das sie nachher so bedeutungsvoll macht? Nehmen wir an, unser Audiopuffer hätte die Länge 8, und in ihm liegen einige Werte:

$$v = [1, 2, 3, 4, 5, 6, 7, 8]$$

dann würde die Fouriertransformation diese Werte als Eingangsdaten bekommen:

[6]Besitzer eines Android Smartphones können zum Beispiel unter https://phyphox.org/de/home-de/ die Software finden, um solche Experimente auf dem eigenen Telefon durchzuführen.

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \end{pmatrix} = ft \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{pmatrix}$$

Die Fouriertransformation berechnet die Koeffizienten von Sinus- und Kosinusschwingungen. Fourier ging von der Annahme aus, dass sich jede periodische Schwingung als Summe von Sinusschwingungen und deren ganzzahligen Vielfachen darstellen lässt[7]. Dieser Prozess wird auch Fouriersynthese genannt. Für eine periodische Funktion f(t) mit der Periode T lautet die theoretische Definition der Fouriersynthese:

$$f(t) = \sum_{n=-\infty}^{\infty} c_n \, e^{2\pi i \, nt/T}$$

Die Funktion wird hier aus unendlich vielen Teilfunktionen zusammengebaut, die alle jeweils für sich von der Zahl n abhängen. Im Exponenten von e findet sich die Grundfrequenz als t/T, und n als ihr ganzzahliges Vielfaches. Der Computer wird mit seiner Berechnung allerdings nicht bis ins Unendliche kommen - oder wir können und wollen nicht so lange warten. Die Frequenzen des Eingangssignals gehen auch nicht bis ins Unendliche, da es ja aus Samples besteht und deswegen eine begrenzte Frequenzauflösung hat. Daher treten wir in der Formel etwas kürzer und reduzieren uns auf die Länge des Puffers. Außerdem wollen wir keine periodische Schwingung aus Einzelwellen zusammenbauen, sondern vielmehr umgekehrt von einem beliebigen Audiopuffer die Frequenzanteile bestimmen. Zur Anwendung kommt also das Gegenteil der Synthese, die Fourieranalyse. Hierbei rechnen wir nicht kontinuierlich, sondern verwenden die diskrete Fouriertransformation. Der Vektor c wird also unter diesen Bedingungen wie folgt berechnet:

$$c_k = \frac{1}{n} \sum_{j=0}^{n-1} v_j \, e^{-2\pi i \frac{kj}{n}}$$

Jeder Sample in unseren Eingangsdaten stellt nun ein Element v mit dem Index j innerhalb der Summe dar. Das heißt: Um ein einziges Element im transformierten Vektor c zu erhalten, wird eine Berechnung über den gesamten Eingangsdaten durchgeführt. Schreibt man die Berechnung für das Element im Vektor c als Zeile auf, und das für jedes Element in c, so erhält man die Formel der Fouriertransformation in Matrixschreibweise.

[7] https://www.youtube.com/watch?v=7bi5_KC08Kg

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \end{pmatrix} = \begin{pmatrix} \odot^0 & \odot^0 & \odot^0 & \odot^0 & \odot^0 & \odot^0 & \odot^0 & \odot^0 \\ \odot^0 & \odot^1 & \odot^2 & \odot^3 & \odot^4 & \odot^5 & \odot^6 & \odot^7 \\ \odot^0 & \odot^2 & \odot^4 & \odot^6 & \odot^8 & \odot^{10} & \odot^{12} & \odot^{14} \\ \odot^0 & \odot^3 & \odot^6 & \odot^9 & \odot^{12} & \odot^{15} & \odot^{18} & \odot^{21} \\ \odot^0 & \odot^4 & \odot^8 & \odot^{12} & \odot^{16} & \odot^{20} & \odot^{24} & \odot^{28} \\ \odot^0 & \odot^5 & \odot^{10} & \odot^{15} & \odot^{20} & \odot^{25} & \odot^{30} & \odot^{35} \\ \odot^0 & \odot^6 & \odot^{12} & \odot^{18} & \odot^{24} & \odot^{30} & \odot^{36} & \odot^{42} \\ \odot^0 & \odot^7 & \odot^{14} & \odot^{21} & \odot^{28} & \odot^{35} & \odot^{42} & \odot^{49} \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{pmatrix}$$

$$\odot = e^{-i\frac{2\pi}{n}}$$

Für jede Zeile der Fourier-Matrix bleibt der Index k konstant, während j sich pro Spalte um eins erhöht. So ergeben sich in der ersten Zeile Nullen, in der zweiten Zeile ganzzahlige Vielfache von eins, in der dritten Zeile ganzzahlige Vielfache von 2 und so fort. Die erste Spalte ist eine Nullspalte, da der Index j immer bei null beginnt. Die sich ergebende Matrix besitzt eine Symmetrie, was Algorithmen wie die „fast fourier transform“ ausnutzen. Aufgrund des Erfolges dieser Optimierung wird eigentlich oft nur noch „fft“ gesagt, und so heißen dann auch meistens die entsprechenden code-Schnipsel.

Je weiter unten und je weiter hinten in der Matrix, um so höher wird der Exponent. Um so höher werden auch die Frequenzen, deren Vorhandensein geprüft wird. Noch sinnfälliger wird dies, wenn wir die Gleichung der Koeffizienten umformen und statt der Exponentialschreibweise cosinus und sinus verwenden.

$$c_k = \frac{1}{n} \sum_{j=0}^{n-1} v_j \left[cos\left(\frac{2\pi \cdot k \cdot j}{n}\right) - i\, sin\left(\frac{2\pi \cdot k \cdot j}{n}\right) \right]$$

Das Produkt aus k und j können wir direkt oben in der Matrix ablesen. Es wird also eine Summe über einer Menge von Kreisbewegungen gebildet, und je nachdem wie das Signal (der Wert im Vektor der Eingangsdaten) aussieht werden manche Frequenzen stärker gewichtet als andere. Die Fourieranalyse findet heraus, welche Frequenzen[8] man mit welcher Amplitude und Phase addieren muss, um etwas wie das Eingangssignal zu erhalten.

5.2 Frequenzverteilung im Ergebnis

Wie bestimmt sich nun über die Größe des Audiopuffers bzw. des fft's die Frequenzverteilung, wie kann ich dem user sagen wie viel Hz der gespielte Ton hat? Es gibt ja nun einen Ausgabevektor mit den von der fft gelieferten Daten. Was bedeuten dessen Werte? Um das herauszufinden, müssen wir unserem Programm erst einmal ein paar Parameter geben:

[8] Hier könnte auch das Wort „Kreisbewegungen“ stehen.

```
#define FFT_SIZE (8192)
#define SAMPLERATE (8000)
```

Das Signal wird 8000 mal pro Sekunde abgetastet, Frequenzen bis etwa 4000 Hz können dargestellt werden. Der Audiopuffer und damit der Vektor der Eingangsdaten hat in unserem Beispiel die Länge 8192. Somit haben wir jetzt ebensoviele komplexe Zahlen als Ergebnisvektor der fft. Die zugehörige Frequenz f eines Wertes aus c errechnet sich näherungsweise aus den vorgegebenen Parametern:

$$N = 8192$$

$$sr = 8000$$

$$f\left(c_k\right) = \frac{sr}{N}$$

Der Ausdruck kann offensichtlich eine Größe von 0-4000 annehmen, was unseren Erwartungen entspricht. Ein wichtiges Detail ist hierbei, daß jeder Wert im Ergebnisvektor eine „Zentralfrequenz“ beschreibt - für jedes k verändert sich der Frequenzwert bei unserer Konfiguration um

$$\Delta f = \frac{sr}{N} = \frac{8000}{8192} \approx 1Hz.$$

Zum Vergleich sei ein Alternativszenario beschrieben:

$$N = 1024$$

$$sr = 48000$$

$$\Delta f = \frac{48000}{1024} \approx 48Hz$$

Eine relativ grob auflösende Samplerate zusammen mit dem größeren Puffer minimiert den Fehler, während die feinere Abtastung und gleichzeitige Verkleinerung des Puffers das Frequenzraster[9] drastisch vergrößert.

Es gibt mehrere Arrays von der Länge des fft, jeweils einen für Frequenz, Notenname und pitch. Der Frequenzarray wird über die gesamte Länge initialisiert mit dem Wert

[9] Die Breite jedes Frequenzabschnitts wird auch „fft bin“ genannt.

$$f_i = \frac{sr \cdot i}{N}$$

Die Samplerate geteilt durch die Größe des fft-Fensters ergibt die Bandbreite eines einzelnen Frequenzabschnitts. Über den Index i wird die Zentralfrequenz des jeweiligen Abschnitts bestimmt.

```
for ( int i = 0; i < FFT_SIZE; ++i ) {
freqTable[i] =
(SAMPLE_RATE * i) / (double)(FFT_SIZE);
}
```

5.3 Tonhöhen und Namen

Das Programm braucht eine interne Repräsentation dessen, wie wir die Tonhöhen benennen. Gegeben zu einer im Eingangssignal ermittelten Frequenz muss es wissen, welcher Notenname zu der Frequenz gehört. Die zwölf Notennamen werden also über einen Index zugeordnet. Aber über welche Frequenzen reden wir hier eigentlich? Oftmals wird das a4 bei 440 Hz als Referenzton genannt. Von da aus kann ich - mit dem Wissen, dass eine Oktave die Frequenz verdoppelt - 880 Hz, 220 Hz oder 110 Hz ebenso als Töne mit dem Namen a identifizieren. Was aber passiert dazwischen? Vielleicht ist die Saite der Gitarre einen Halbton zu hoch gestimmt, oder die Saiten werden komplett gewechselt und haben gar keinen Ausgangswert.

Dadurch, dass sich die Frequenz über eine Oktave hinweg verdoppelt, wachsen die Oktaven nach oben hin exponentiell. Daher eignet sich zur Darstellung von Frequenzabständen die logarithmische Skala, welche wieder eine Äquidistanz zwischen den Halbtönen herstellt - während der Unterschied zwischen den eigentlichen Frequenzwerten immer ein anderer sein kann. Und nun wollen wir wissen, welche Frequenz der Halbton über a4 hat. Die Oktave - hier mit p=2 als Verdopplung angesehen - wird eingeteilt in zwölf Schritte gleichen Abstands:

$$p = 2$$

$$n = 12$$

$$r = \sqrt[n]{p} = \sqrt[12]{2} = 2^{\frac{1}{12}}$$

Wenn ich eine Frequenz mit 2 multipliziere, erhalte ich ihre Oktave. Wenn ich nur um einen Halbton erhöhen möchte, sollte der Faktor sehr viel geringer sein. Um wie viel genau lässt sich errechnen, wenn die Ausgangsfrequenz und das Verhältnis r bekannt ist.

Note	r	Hz
A	$2^{\frac{0}{12}}$	440.00
A#/Bb	$2^{\frac{1}{12}}$	466.16
B/H	$2^{\frac{2}{12}}$	493.88
C	$2^{\frac{3}{12}}$	523.25
C#/Db	$2^{\frac{4}{12}}$	554.37
D	$2^{\frac{5}{12}}$	587.33
D#/Eb	$2^{\frac{6}{12}}$	622.25
E	$2^{\frac{7}{12}}$	659.26
F	$2^{\frac{8}{12}}$	698.46
F#/Gb	$2^{\frac{9}{12}}$	739.99
G	$2^{\frac{10}{12}}$	783.99
G#/Ab	$2^{\frac{11}{12}}$	830.61
A	$2^{\frac{12}{12}}$	880.00

Ein Programm könnte nun die zu den Notennamen gehörenden Frequenzen rechnerisch erzeugen. In welcher Datenstruktur dies geschieht, ist wieder der Implementation überlassen.

5.4 Peak

```
// distance formula
v = this.ar[j]*this.ar[j] + this.ai[j]*this.ai[j];
```

Wie findet das Programm die Fundamentalfrequenz? Der gezeigte Code bestimmt die Energie der einzelnen Frequenzbänder[10], indem die von der fft ausgegebene komplexe Zahl als Distanz interpretiert wird. Im einfachsten Fall wird über den Array der Ausgabewerte iteriert und so der höchste Wert ermittelt.

Die Fundamentalfrequenz der tiefen E-Saite der Gitarre liegt bei 82.5 Hz. Aber diese Frequenz ist in der fft magnitude bei den meisten Gitarren um ein Vielfaches leiser als der erste Oberton. Für eine zuverlässige Erkennung der Grundfrequenz ist die fft magnitude hier nicht aussagekräftig genug. Es kann

[10] Auf stackoverflow „fft magnitude“

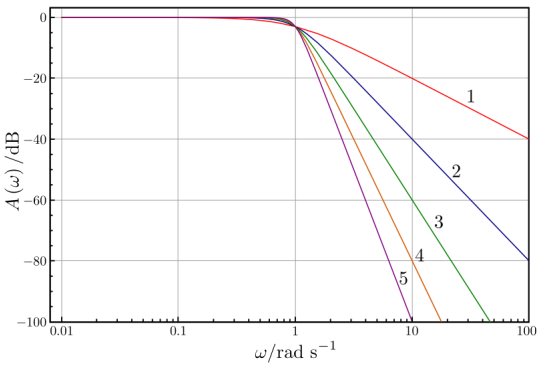

Abbildung 7: Tiefpassfilter

sein, dass eine wahrgenommene Grundfrequenz im Spektrum nur sehr wenig zu sehen ist. Tonhöhe ist auch ein psychoakustisches Phänomen, das sich nicht notwendigerweise mit den von der fft generierten Daten deckt. Ob die Fourieranalyse mittels fft ein zuverlässiges Verfahren zur Tonhöhenerkennung sein kann, hängt also auch vom Einsatzbereich ab. Will man seine Gitarre in Stimmung bringen, und ist die fft der einzige verwendete Algorithmus, so bedarf er der Optimierung und Verbesserung.

6 Filter

Um das Problem der fehlenden Fundamentalfrequenzen zu minimieren, wird im Eingang des Signals vor der fft ein Tiefpassfilter eingesetzt. Es werden also nur Frequenzen bis zu einem gewissen Wert durchgelassen. Die ganz hohen Frequenzen können ja ohnehin nicht dargestellt werden. Und um hoch qualitatives Audio geht es bei einer Tuner-Anwendung auch nicht. Somit reduziert sich die Fehleranfälligkeit der fft, wenn höhere Frequenzen außer Acht gelassen werden. Auch nicht zum Instrumentenklang gehörende hochfrequente Störgeräusche fallen dadurch weg.

Zusätzlich werden Fensterfunktionen eingesetzt, die den Audio-Screenshot an den Seiten „abrunden“. Die Fouriertransformation „tut so“, als käme das Eingangssignal unendlich oft hintereinander. Daher reduziert ein weicher Einstieg auch hier den Fehler.

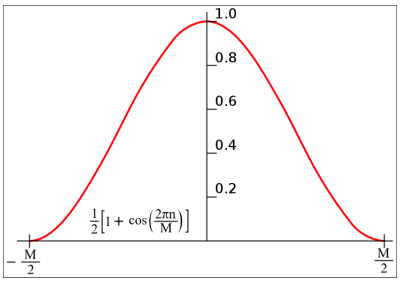

Abbildung 8: Hanning-Fenster

Teil III
Synthesis

Während Aspekte der grafischen Interaktion mit dem Benutzer in dieser Auseinandersetzung nicht thematisiert wurden, ist die Implementation der zugrunde liegenden Funktionalität entfaltet worden. Die simple Verwendung einer fft-Bibliothek führt noch nicht zwingend zum richtigen Ergebnis. Allerdings stellt dies den ersten Schritt in Richtung einer Lösung der realen Anforderungen dar. Die Bühne des Problemfelds ist aufgebaut, auf welcher nun weitergehende Verfeinerungen und Verbesserungen stattfinden können.

Das Stimmen einer Gitarre erscheint hier als ein Sonderfall des umfassenderen Problemfeldes der Tonhöhenerkennung[11]. Seit digitale elektronische Geräte klein genug werden konnten, werden sie als Stimmgeräte eingesetzt. Mit dem Einzug der Smartphones in unser Leben begann eine weitere kleine Revolution, was die Rechenleistung angeht. So ist die fft auf den meisten Geräten von der Rechenleistung her machbar. Allerdings ist der Prozeß dort nicht stehen geblieben und es gibt unzählige Algorithmen zur pitch detection. Die fft ist dort Teil einer größeren Kette von Berechnungen.

[11] Im Internet „pitch detection“