

Sven Eric Panitz

**Lehrbriefe**

**Python für Java-Programmierer**

*$\pi$*

Audioverarbeitung



# Audio Verarbeitung

Sven Eric Panitz

8. Juni 2024

## Inhaltsverzeichnis

<b>1 Die Physik von Audiodaten</b>	<b>1</b>
<b>2 Die Diskretisierung von Audiodaten</b>	<b>3</b>
<b>3 Audiodaten</b>	<b>5</b>
<b>4 Audio Synthese</b>	<b>5</b>
4.1 Klänge . . . . .	6
4.2 Tonfolgen . . . . .	7
4.3 Akkorde . . . . .	8
<b>5 Audio Analyse</b>	<b>8</b>
5.1 Einlesen von WAV-Dateien . . . . .	8
5.2 Frequenzanalyse . . . . .	10
<b>6 Lernzuwachs</b>	<b>14</b>

## 1 Die Physik von Audiodaten

Audiodaten, Geräusche, Klänge, Musik sind für einen Informatiker eine ganz besondere und faszinierende Datenstruktur. Diese Daten existieren nur in der Zeit. Ein Musikstück kann nur in einer fest vorgegebenen Zeitspanne von einem Anwender rezipiert werden. Das gilt für keine andere Art von Daten. Wie lange man ein Bild betrachtet, bleibt dem Betrachter überlassen. Wie schnell ein Text gelesen wird, ist individuell für jeden Leser eine eigene Entscheidung.

Musik gibt uns die Zeit vor. Wir können Musik nicht schneller hören, ohne dass sie nicht mehr dieselbe Musik ist. Dieses unterscheidet Audiodaten fundamental von den meisten anderen Daten.

Und was sind Geräusche und Klänge? Sie sind physikalisch nichts anderes als die Veränderung des Luftdrucks über die Zeit. Stellen wir uns hierzu eine feine Membran vor, die wenn auf einer

Seite ein Überdruck existiert, sich zur anderen Seite ausbeult, wenn dort ein Unterdruck herrscht, sich einknüllt. Die Membran wackelt also auf Grund von Luftschwankungen hin und her. Und dieses Wackeln der Membran nehmen wir als Geräusche wahr.

Die Membran, die in unserem Ohr durch die Druckschwankungen in Schwingung gebracht wird, ist das Trommelfell.

Die Stärke der Luftdruckschwankungen nennen wir Lautstärke. Wenn also Druck und Unterdruck sehr hoch sind, empfinden wir das als laut, wenn sie gering sind als leise.

Wenn der Wechsel zwischen Unterdruck und Überdruck sich regelmäßig wiederholt, dann wird diese regelmäßige Schwingung als Ton wahrgenommen.

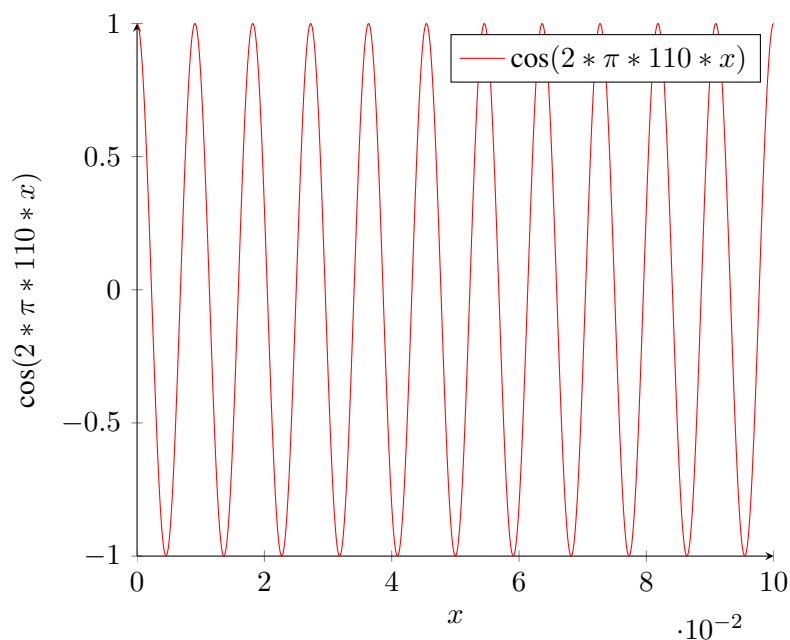
Die Anzahl der Schwingungen pro Sekunde wird Frequenz genannt und mit der Einheit Hertz bezeichnet.

Das menschliche Ohr ist etwa in der Lage Schwingungen zwischen 20 und 20000 Hertz wahrzunehmen.

Eine Schwingung von 440 Hertz ist in der Musik für den Kammerton A als Referenz festgelegt.<sup>1</sup>

Der Ausschlag der Schwingungen, die wir als Lautstärke wahrnehmen, wird als Amplitude bezeichnet.

Mathematisch lassen sich regelmäßige Schwingungen durch eine trigonometrische Funktion darstellen. Eine Schwingung von 110 Hertz, also 110 Eindrücken und Ausbuchtungen einer Membran in der Sekunde kann durch die Funktion  $\lambda x \cdot \cos(2 * \pi * 110 * x)$  dargestellt werden. Dieses ist zum Beispiel die Schwingung der A-Saite einer Gitarre. In folgender Darstellung ist der Graph einer 110 Hertz Schwingung für eine Zehntelsekunde dargestellt:

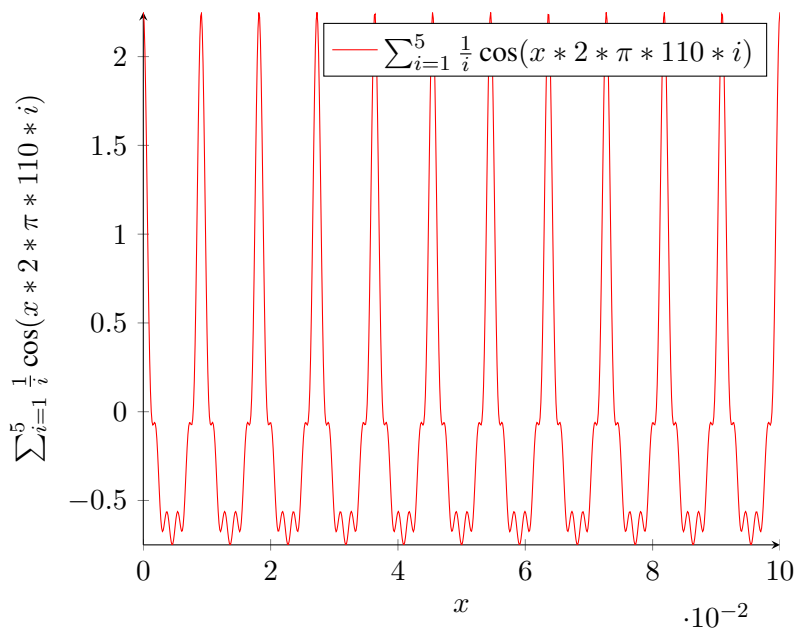


---

<sup>1</sup>Wobei die meisten klassischen Ensembles heute eine etwas höhere Stimmung nutzen.

Solche reinen Schwingungen finden sich in der Natur nicht. Auch wenn man eine Gitarrensaite zupft, dann wird nicht nur eine Schwingung mit der Saite erzeugt, die dann über die Holzdecke der Gitarre den wechselnden Luftdruck erzeugt, sondern die Addition vieler Schwingungen. Besonders harmonisch sind Schwingungen, die ein Vielfaches einer Grundschwingung sind. Diese werden in der Musik als Obertöne bezeichnet.

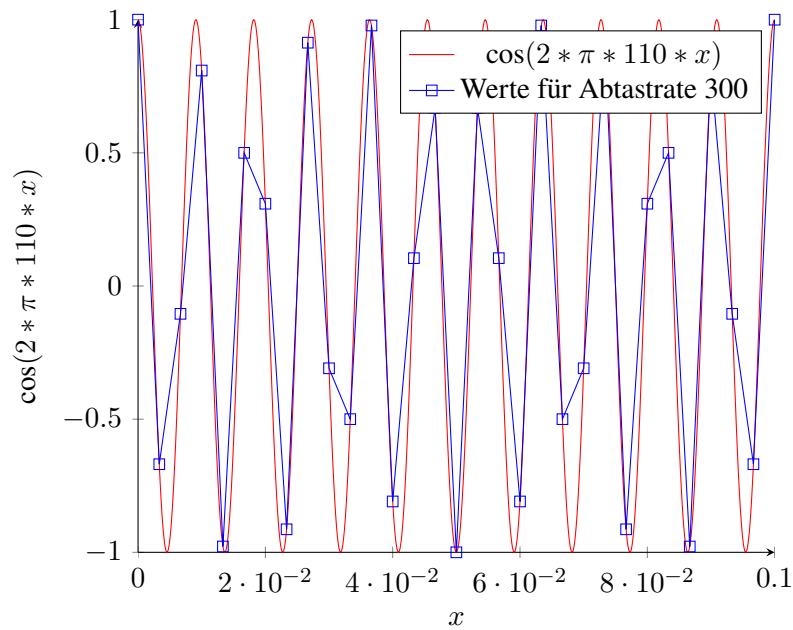
Folgender Graph zeigt die Schwingung von 110Hertz in Addition mit den Obertönen in 220, 440 und 550 Hertz.



## 2 Die Diskretisierung von Audiodaten

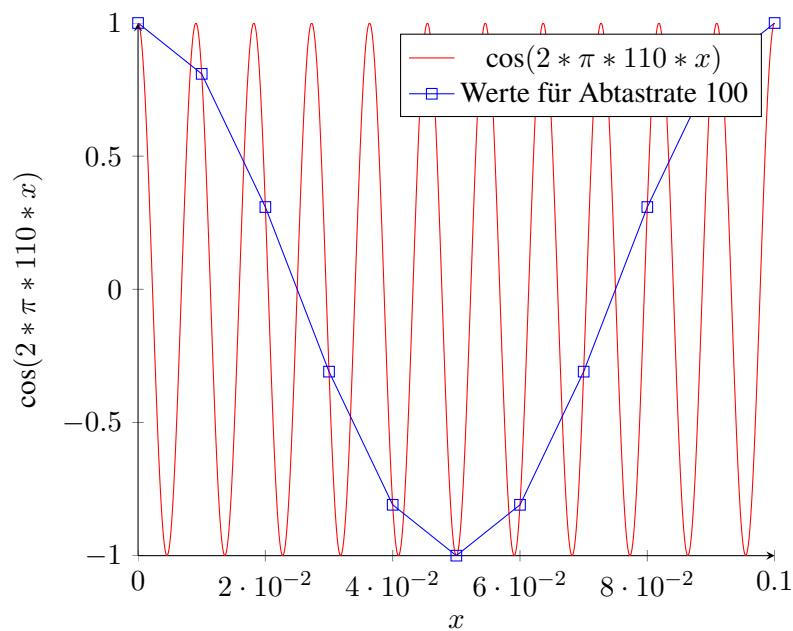
Wir Informatiker kennen nur eine diskrete Welt, d.h. im Prinzip nur Folgen von Zahlen. Daher werden wir Geräusche nur als eine Folge von Messwerten des Luftdrucks darstellen. So werden Geräusche durch eine Folge von Messpunkten des Luftdrucks auf der Membran dargestellt. Wie oft in einer Sekunde Messwerte aufgezeichnet werden, wird als Abtastrate bezeichnet.

Folgender Graph zeigt nochmal die Schwingung von 110 Hertz. Zusätzlich sind mit einer Abtastrate von 300 die Messpunkte eingetragen und mit der blauen Linie verbunden.



Schwingungen, die höher sind als die Abtastrate, lassen sich nicht messen. Es passiert einfach zu viel zwischen zwei Messpunkten. Stattdessen bekommt man scheinbar eine Schwingung angezeigt, die überhaupt nicht existiert.

Im folgenden Graphen scheint man durch die zu geringe Abtastrate eine Frequenz von 10 Hertz zu messen.



Für Audiodaten hat sich eine Abtastrate von 44200 etabliert und wird für CD-Aufnahmen angewandt. Damit lässt sich das Hörspektrum des menschlichen Ohr gut abtasten. Bei einer gerin-

geren Abtastrate könnten hohe Töne verloren gehen, bei einer höheren Abtastrate würde man unnötig mehr Daten sammeln, nur um Frequenzen zu speichern, die niemand mehr hört.

### 3 Audiodaten

Wie wir gesehen haben, sind Audiodaten nichts anderes als eine Folge von Messwerten. Somit lässt sich recht einfach in Python ein kleines Modul schreiben, das Audiodaten verarbeitet.

Wir werden dabei zur Hilfe einige Module importieren. Das Modul `scipy.io` kann uns helfen, WAV-Dateien zu lesen und zu schreiben. Die übrigen Pakete unterstützen unterschiedliche Zahlentypen und Arrays von Zahlenwerten.

#### Audio.py

```
import math
import numpy
from scipy.io import wavfile
```

Um einen Ton von 10 Sekunden zu erzeugen, brauchen wir also 4410000 Zahlenwerte auf einer Schwingung. Um den Kammerton A darzustellen nehmen wir eine Schwingung von 440 Hertz, die erzeugt wird durch die Funktion:  $\lambda x. \sin(2 * \pi * 440 * x)$ . Als Amplitude setzen wir den Wert 10000.

Über eine List Comprehension lässt sich die Liste der Messwerte einfach erzeugen.

#### Audio.py

```
kammertonA = [ 10000*math.sin(2*440*math.pi*x/44100) for x in range(0,5*44100)]
```

### 4 Audio Synthese

Audiodaten sind nichts weiter als eine Liste von Zahlenwerten. Wir brauchen nur ein paar Hilfsmittel, um diese Liste als WAV-Datei zu speichern.

Hierzu stellt das Modul `Data.Audio` einen Konstruktor bereit, der 3 Parameter hat:

- Die Abtastrate in der die Daten vorliegen. Wir nehmen stets 44100.
- Die Anzahl der Kanäle. Wir arbeiten stets mit Monodaten und dem Wert 1.
- Die eigentlichen Daten, die in einem Array übergeben werden. Diese werden wir als 16-Bit Zahlen des Typ `int16` bereit stellen.

Wir müssen also unsere Liste von Fließkommazahlen in einem Array von 16-Bit Zahlen umwandeln. Die einzelnen Wert können mit der Funktion `round` umgewandelt werden.

Im Modul `scipy.io` ist eine Funktion definiert, mit deren Hilfe sich so erzeugte Audiodaten in eine Wav-Datei speichern lässt.

#### Audio.py

```
def writeWav( fn, ds):  
    wavfile.write(fn,44100,numpy.array(list(map(lambda x:numpy.int16(round(x)),ds))))
```

Jetzt können wir unseren ersten Klang als Sounddatei speichern und anhören.

#### Audio.py

```
def w1(): writeWav("kammertonA.wav", kammertonA)
```

Die Datei kammertonA.wav, die so erzeugt wird, enthält einen Signalton, wie wir ihn als Freizeichen im Telefon kennen. Er klingt sehr steril.

## 4.1 Klänge

Die Klänge vom Musikinstrumenten beinhalten nicht eine Frequenz, sondern Anteile vieler Frequenzen, deren Amplitude sich auch über die Zeit ändern. Genau in diesem Frequenzgemisch unterscheiden sich die Klänge der unterschiedlichen Instrumenten. Eine Gitarre klingt anders als ein Klavier, ein Banjo oder ein Ukulele. So wie auch Oboe, Klarinette, Fagott oder Flöte sehr unterschiedlich klingen.

Unserem Kammerton können wir in einem ersten Schritt die doppelte Schwingung als Oberton mit halber Amplitude der Grundschwingung hinzufügen:

#### Audio.py

```
def kammertonA2():  
    return [10000*math.sin(2*440*math.pi*x/44100)+5000*math.sin(2*880*math.pi*x/44100)  
            for x in range(0,10*44100)]
```

Zum Anhören, wird dieser Ton in eine Datei gespeichert.

#### Audio.py

```
def w2():writeWav("kammertonA2.wav", kammertonA2())
```

**Aufgabe 1** Schreiben Sie eine Funktion pluggedTime die einen Ton erzeugt, der halbwegs den Klang eines Zupfinstrument simuliert. Das erste Argument gibt an, wie viel Abtastwerte erzeugt werden sollen. Das zweite Argument soll die Grundschwingung des erzeugten Klangs sein.

Die Funktion der Schwingung sei dort folgende Formel bestimmt.

$$f(x) = \sum_{i=1}^{10} \frac{1}{i} \sin(2 * \pi * x * i)$$



Der erzeugte Ton soll mit einer Amplitude von 10000 beginnen und seine Amplitude jeweils nach 5000 halbieren.

Audio.py

```
def pluggedTime( t, wv): return []
```

Mit dieser Funktion können wir jetzt kürzere und längere Töne einer beliebigen Wellenlänge erzeugen, die ein wenig nach einem Zupfinstrument klingen

Audio.py

```
def pluggedH(x):return pluggedTime(44100//2,x)
def plugged(x): return pluggedTime(44100,x)
def pluggedD(x): return pluggedTime(2*44100,x)

def kammertonAHarmonics(): return plugged(110)

def w3(): writeWav( "pluggedA.wav", kammertonAHarmonics())
```

Hören Sie sich ihren erzeugten Ton an.

## 4.2 Tonfolgen

Eine Tonfolge wird als Melodie erzeugt. Melodien werden aus einer Tonleiter gebildet. Diese sind Töne bestimmter Wellenlängen ausgehend von einem Grundton. Die A-Dur Tonleiter besteht aus folgenden Wellenlängen.

Audio.py

```
a = 440
b = 493.88
cs= 554.37
d = 587.33
e = 659.25
fs= 739.99
gs= 830.61
aP= 880.00
```

**Aufgabe 2** Erzeugen Sie für den Zupfklang eine Sequenz der Töne der A-Dur Tonleiter. Jeder Ton soll eine halbe Sekunde lang erklingen.

Audio.py

```
def scale(): return []
```

Hören Sie sich die erzeugte Datei an.

Audio.py

```
def w4(): writeWav( "scale.wav", scale())
```

### 4.3 Akkorde

Töne, die gleichzeitig erklingen, werden in der Musik als Akkord bezeichnet.

**Aufgabe 3** Erzeugen Sie einen Akkord von 2 Sekunden Länge mit dem Zupfklang, der aus dem ersten, dritten, fünften und siebten Ton der A-Dur Tonleiter besteht.

Dabei sollen die höheren Töne jeweils leicht Zeitversetzt nach jeweils 2000 Abtastwerten erst einsetzen. Musiker nennen das Arpeggio.

Audio.py

```
def maj7(): return []
```

Hören Sie sich den Akkord an. Er müsste nach Jazz-Musik klingen.

Audio.py

```
def w5(): writeWav( "maj7.wav", maj7())
```

#### Projektidee 1

Sie haben in diesem Abschnitt gesehen, wie man aus einzelnen Wellen, Klänge synthetisieren kann. Schreiben Sie mit Hilfe einer GUI-Bibliothek eine Anwendung, die es erlaubt über Regler für einzelne Frequenzen Klänge zu kreieren.

## 5 Audio Analyse

Nachdem wir im letzten Abschnitt gesehen haben, wie man Klänge aus einfachen Wellenformeln synthetisieren kann, wollen wir in diesem Abschnitt Klänge analysieren. Für Audiodaten soll analysiert werden, welche Frequenzen an dem Klang beteiligt sind. Insbesondere für Töne soll so die Tonhöhe bestimmt werden.

### 5.1 Einlesen von WAV-Dateien

Da wir von nun an nicht neue Klänge erzeugen wollen, die wir in Audiodateien speichern, sondern bestehende Klänge analysieren wollen, beginnen wir mit einer kleinen Hilfsfunktion, die es

erlaubt, WAV-Dateien aus dem Dateisystem einzulesen und die Rohdaten, d.h. die Abtastwerte als Liste bereitzustellen.

Wir gehen davon aus, dass die gelesenen Daten in der Abtastrate 44100 vorliegen und nur ein Kanal (Mono) in der Datei gespeichert ist.

#### Audio.py

```
def getWavFromFile( fn ) : return list(wavfile.read(fn)[1])
```

Damit haben wir Zugriff auf die Abtastdaten einer Wav-Datei.

**Aufgabe 4** Das  $\text{\LaTeX}$ -Paket pgfplots[Feu14] erlaubt es auf einfache Weise Funktionsgraphen oder einzelne Messpunkte zu visualisieren. Hierzu sind die Messpunkte als Paare anzugeben. Schreiben Sie eine Funktion, die eine Messreihe der Abtastrate 44100 als Punkte darstellt, die in pgfplots verwendet werden können

#### Audio.py

```
def abtast():
    x = 0
    while (True):
        yield x
        x += 1/44100
```

#### Audio.py

```
def toPgplot( ws ): return ""
```

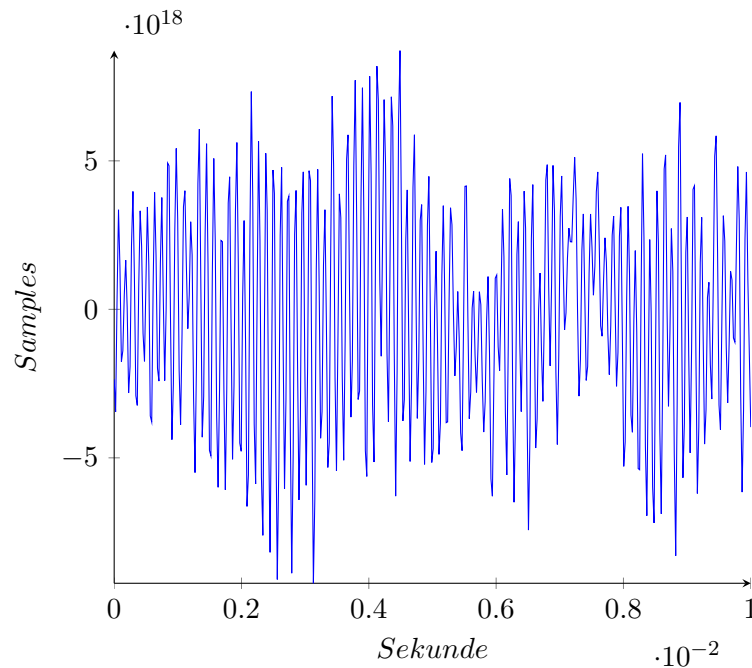
Diese Funktionen können Sie nun verwenden um  $\text{\LaTeX}$ -Quelltext für den Frequenzverlauf zu erzeugen.

#### Audio.py

```
def writeForLaTeX(resultFileName, ws):
    start = "\\begin{tikzpicture}\n" \
            "\\begin{axis}\n" \
            "[axis lines = left,xlabel = $Sekunde$,ylabel = {Samples},]\n" \
            "\\addplot[color=blue,mark=dot,]\n" \
            "\\coordinates {\n"
    end= "};\n" \
        "\\end{axis}\n" \
        "\\end{tikzpicture}"
    f = open(resultFileName, "w")
    f.write(start)
    f.write(toPgplot(ws[:441]))
    f.write(end)
    f.close()
```

Erzeugen Sie eigene Graphen für kurze Klangbeispiele.

Folgender Graph wurde mit dieser Funktion aus einer Audio-Datei für die erste Hundertstelssekunde erzeugt.



## 5.2 Frequenzanalyse

Wenn Musik in einer Audio-Datei vorliegt, sind nur einzelne Abtastwerte gespeichert. Die eigentliche musikalische Information, die wir hören, ist nicht gespeichert, nämlich die Tonhöhe der einzelnen Töne. Diese hören wir als Melodie. Es wäre schön, diese auch automatisch zu hören, also die Abtastwerte so zu interpretieren, dass wir wissen, welche Frequenzen den Ton erzeugen.

Dass dieses überhaupt möglich ist, hat im frühen 19. Jahrhundert der Mathematiker Joseph Fourier mit dem fundamentalen Satz gezeigt, der besagt:

Jede periodische Schwingung lässt sich als eine Summe von potentiell unendlich vielen gewichteten trigonometrischen Funktionen darstellen. Da die Sinusfunktion nur die phasenverschobene Cosinusfunktion ist, mündet der Satz in folgende Formel<sup>2</sup>:

Ein periodisches Signal  $x(t)$  mit einer Periode  $T$  kann repräsentiert werden als:

$$x(t) = C_0 + \sum_{n=1}^{\infty} C_n \cos\left(\frac{2\pi}{T}nt + \phi_n\right)$$

Damit wurde gezeigt, dass Signale in einzelne Schwingungen zerlegen lassen. Jede Frequenz hat dabei eine Amplitude  $C_n$  und eine Phasenverschiebung.

<sup>2</sup>Wir halten uns dabei an die Darstellung aus [HQ18].

Aber es kommt noch besser. Es gibt einen Algorithmus, der diese Zerlegung im Diskreten für uns durchführt: die diskrete Fourier Transformation.

Wenn wir  $N$  Abtastwerte in einer Sekunde haben und die Funktion  $x[n]$  uns jeweils den  $n$ -ten Abtastwert als komplexe Zahl liefert, dann sei die diskrete Fourier Transformation die Funktion  $\hat{x}[k]$ , die wie folgt definiert ist<sup>3</sup>:

$$\hat{x}[k] = \frac{1}{N} \sum_{n=0}^{N-1} x[n] e^{-i \frac{2\pi kn}{N}}$$

Die komplexen Zahlen der  $\hat{x}[k]$  beschreiben für  $k \leq \frac{N}{2}$  Phase und Amplitude der Frequenz  $k$ . Die Amplitude, an der wir vorerst interessiert sind, ist der Betrag der komplexen Zahl.

Für  $k \geq \frac{N}{2}$  wird das Spektrum nur noch einmal im Negativen gespiegelt. Diese Werte haben für uns keine neue Information.

In der Folge sollen Sie die diskrete Fourier Transformation implementieren. Python unterstützt komplexe Zahlen. Eine komplexe Zahl kann mit Real- und Imaginärteil über die Funktion `complex` erzeugt werden:

```
complex(17,4)
```

```
(17+4j)
```

Oder als Literal durch das Anfügen eines `j` an den Imaginärteil:

```
17+4j
```

```
(17+4j)
```

Der Typ `Complex` implementiert alle numerischen Typklassen, so dass für ihn alle arithmetischen Infix-Operatoren zur Verfügung stehen. Weitere Funktionen befinden sich im Modul `cmath`.

### Aufgabe 5 Implementieren Sie die diskrete Fourier Transformation.

Audio.py

```
import cmath
```

```
def dft(xs): return []
```

Testen wir die Fourier Transformation jetzt einmal an einem Beispiel, indem wir eine Datei mit der Mikrophonaufnahme eines Tons eines Instruments einlesen und die Fourier Transformation für die erste Zehntelsekunde durchführen.

<sup>3</sup>In Textbüchern findet sich statt des Buchstaben  $i$  für den Imaginärteil komplexer Zahlen, der Buchstabe  $j$ . Dieses liegt daran, dass Physiker den Buchstaben  $i$  für elektrische Stromstärke verwenden und daher das  $j$  für den Imaginärteil gewählt haben.

Da wir nur eine Zehntelsekunde analysieren, sind die Frequenzen, also die Indizes mit 10 zu Multiplizieren, um die eigentliche Frequenz zu bekommen.

Der Betrag der einzelnen komplexen Zahlenwerte stellt jeweils die Amplitude der Frequenz an einem Index dar.

Hier die komplette Funktion. 4410 Abtastwerte werden zu Analyse genommen, dann als komplexe Zahlen dargestellt, die diskrete Fourier Transformation durchgeführt, davon die ersten 200 Frequenzwerte genommen, von diesen der Betrag errechnet und schließlich die Werte mit der Frequenz gepaart.

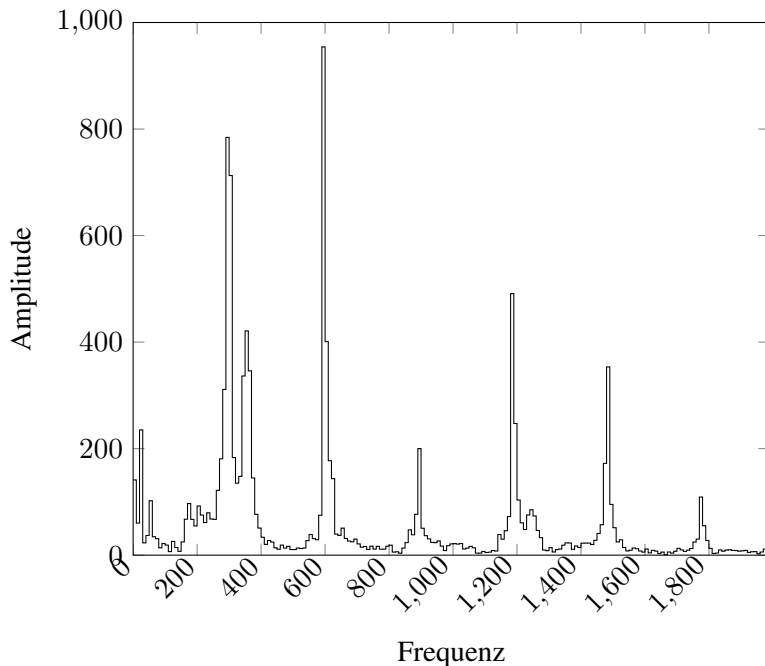
#### Audio.py

```
def stepFrom0(n):  
    x = 0  
    while (True):  
        yield x  
        x += n
```

#### Audio.py

```
def analyseFileStart( fn ):  
    ws = getWavFromFile(fn)  
    return zip(stepFrom0(10)  
              ,map (lambda x:abs(x).real,dft(list(map(lambda x: complex(0,x),ws[:4410])))[:200]))
```

Das Ergebnis der Analyse können wir mit der Funktion `' '.join(map(str,xs))` wieder für das L<sup>A</sup>T<sub>E</sub>X-Paket pgfplots aufarbeiten. Folgendes Diagramm zeigt die Frequenzen der ersten Zehntelsekunde der Tonaufnahme einer Datei.



Man erkennt sehr gut, dass die maximalen Werten sich bei 290 und Vielfachen von 290 befinden. Die Frequenz 293.66 ist der Ton D, wenn der Kammerton A auf 440 Hertz gestimmt ist. Offensichtlich ist auf der Tonaufnahme ein D gespielt. Wir sehen im Frequenzspektrum deutliche die Grundschiwingung des Tons und 5 Obertöne die mitschwingen.

Zusätzlich sehen wir noch eine starke Amplitude bei 350 Hertz. Das entspricht dem Ton F, also eine kleine Terz über den Hauptton schwingt da noch mit.

#### Projektidee 2

Sie haben in diesem Abschnitt gesehen, wie man einen Klang nach seinen Einzelfrequenzen analysieren kann. Musikinstrumente müssen auf bestimmte Grundsequenzen gestimmt werden. Ein hilfreiches Tool ist dabei ein Stimmgerät, das die aktuelle Frequenz der Hauptschiwingung eines Klangs anzeigt. Entwickeln Sie ein solches Stimmgerät. Wenn Sie Bastler sind und einen kleinen Motor steuern können, der die Wirbel eines Saiteninstrumentes dreht, können Sie sogar ein automatisches Stimmgerät entwickeln.

#### Projektidee 3

Wir haben gesehen, dass sich Töne klassischer Musikinstrumente leicht im Spektrum identifizieren lassen. Denkbar wäre, eine Software zu entwickeln, die musikalische Aufnahme in Notenschrift umsetzt. Hierzu können Sie das in Haskell entwickelte Notensatzprogramm Tinte[Pan18] verwenden.

## 6 Lernzuwachs

Folgende Erkenntnisse sollte sich nach Studium dieses Kurses gesammelt haben.

- List-Comprehensions sind ein gute Vehikel für Summenformeln
- Diskrete Anwendungen der Analysis
- Komplexe Zahlen in Python

Wer mehr im Detail in funktionaler Programmierung mit Klängen und Musik arbeiten will, dem sei das entsprechende Lehrbuch von Paul Hudak empfohlen[HQ18].

## Literatur


- [Don03] Matt Donadio. dsp: Haskell digital signal processing. <https://hackage.haskell.org/package/dsp-0.2.4.1>, 2003. [Online; accessed 11-April-2019].
- [Feu14] Christian Feuersänger. Manual for package pgfplots. <https://sourceforge.net/projects/pgfplots/>, 2014.
- [Gio07] George Giorgidze. Hcodecs: A library to read, write and manipulate midi, wave, and soundfont2 files. <https://hackage.haskell.org/package/HCodecs-0.5.1>, 2007. [Online; accessed 11-April-2019].
- [HQ18] P. Hudak and D. Quick. *The Haskell School of Music: From Signals to Symphonies*. Cambridge University Press, 2018.
- [Pan18] Panitz, Sven Eric. Tinte user guide, typesetting sheet music. <http://www.tinte-music.org/tinteUserGuide.pdf>, 2018.










$$\pi$$

Lesen und Schreiben von WAV Dateien  
Diskrete Fourier-Transformation  
Audio Generierung  
Audio Analyse