jugend forscht

Von Schallwellen zu Lichtwellen

Ein parametrisierbares “Electronic LED Orchestra”

Marius De Kuthy Meurers

Kepler Gymnasium Tübingen

1. März 2019

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Die Idee

Musik hört man bekanntlich mit den Ohren. Sie wird mittels Schallwellen an das Ohr übertragen. Aber Schallwellen sind natürlich nicht die einzigen Wellen, die wir wahrnehmen können. Für Licht, also elektromagnetische Wellen, sind unsere Augen zuständig. In meinem Projekt untersuche ich, ob und wie man die eine Art von Wellen in die andere übersetzen kann. Ich entwickle einen flexiblen Übersetzer, der Schallwellen genau analysiert und frei parametrisierbar in Lichtwellen umsetzt. Das in C auf einem Raspberry Pi implementierte Projekt, das ich Open Source auf GitHub veröffentlicht habe, verbindet hierzu eine effiziente, GPU-basierte Fourier-Transformation mit einer frei parametrisierbaren Umsetzung zur Ansteuerung von LEDs (Light Emitting Diodes), die ein breites Spektrum von Farben und Helligkeiten darstellen können.

In der Praxis sind sogenannte Lichtorgeln schon seit vielen Jahren in Diskotheken im Einsatz, wo man beim Tanzen sowohl die Musik hört als auch unterschiedliche Lichter im Takt blinken sieht. Aber lässt sich diese Übersetzung von Schallwellen in Lichtwellen mit aktueller digitaler Technologie so umsetzten, dass die Schallwellen erst genau analysiert werden, um die Informationen dann ganz flexibel in unterschiedliche, frei programmierbare Farbdarstellungen umsetzen zu können? Wir sind also auf der Suche nach Methoden für die Beantwortung der folgenden zwei Fragen:

1. Wie können die Schallwellen analysiert werden, so dass man herausfindet, welche Frequenzen zu einem bestimmten Zeitpunkt im Signal enthalten sind?
2. Wie lassen sich diese Frequenzbestandteile in eine farbliche Darstellung übertragen?

Für die zweite Frage müssen wir einerseits die technische Seite klären, wie wir eine Kette von LEDs ansteuern, so dass sie unterschiedliche Farben darstellen. In Bezug auf die erste Frage interessiert uns insbesondere, wie wir den Klang in Farbe, Helligkeit, und Position der einzelnen LEDs übersetzen. Eine solche detaillierte Analyse des Klangs bietet dann eine Vielzahl von Möglichkeiten für ein parametrisierbares elektronisches LED-Orchester aus dem Titel dieser Arbeit, für die ich vier Beispiele vorstelle.

Bei der technischen Umsetzung und dem Aufbau des von mir entwickelten Programms habe ich mich entschieden, das Programm in C für den Raspberry Pi zu schreiben, da es sich um ein relativ hardwarenahes Projekt handelt. Als Input, also Quelle der Musik, dient eine Audio Datei oder ein angeschlossenes Mikrofon. Bevor wir in Abschnitt 3 zunächst die Verarbeitung des Audiosignals, also die Analyse des Klangs besprechen, führe ich in Abschnitt 2 den Stand der Technik ein. In Abschnitt 4 sehen wir uns dann die Ansteuerung des LED-Streifens an, bevor wir in Abschnitt 5 dann verschiedene Übersetzungsmöglicheiten von Klängen in Farben auf den LED-Streifen beschreiben.

2 Stand der Technik

Die grundsätzliche Idee einer Lichtorgel ist wohlbekannt – fast jeder hat schon die eine oder andere in Diskotheken oder bei Feiern gesehen. Zunächst gibt es die wohl noch immer häufigsten, analoge Lichtorgeln. Analoge Lichtorgeln sind mit 3–6 farbigen Lampen sehr günstig, können aber auch mit mehr Lampen erweitert werden. Diese analogen Lichtorgeln funktionieren so, dass die Frequenzen des Musiksignals über Filter aufgespaltet werden und dann direkt über Transistoren in größere Stromschwankungen umgesetzt werden, die Glühbirnen unterschiedlich hell zum Leuchten bringen. Dies hat den Nachteil, dass die Musik

nicht im Detail analysiert wird, sondern je nach Anzahl der Filter in eine kleine Anzahl von Frequenzbereichen unterteilt wird. Die Aufteilung der Musikfrequenzen in die Lichtorgel ist also durch Hardware festgelegt und die Möglichkeiten durch diese eng begrenzt.

Um eine genauere Analyse des Klangs umzusetzen, werden digitale Lichtorgeln verwendet. Bei diesen kann das Frequenzspektrum sehr viel genauer analysiert werden, um eine größere Vielzahl an Effekten zu erzeugen. Auch können diese die Musik so analysieren, dass z.B. nur die Melodie visualisiert wird oder der Bass mehr hervorgehoben wird. Das Problem solcher digitaler Lichtorgeln ist aber der Preis. Ein Controller, der die Lichter steuert, kostet meist schon mehrere hundert oder tausend Euro. Bei einfacheren digitalen Modellen kann man selber keine Effekte hinzufügen, dies geht dann erst bei professionellem Bühnen-Equipment. Aber wie funktioniert die digitale Frequenzanalyse von Musik, und lässt sich mit einem Raspberry Pi vielleicht ein günstiges und flexibles digitales LED Orchesters realisieren?

3 Analyse des Klangs

Gehen wir zunächst die Frage an, wie sich Schallwellen analysieren lassen. Schallwellen sind Luftmoleküle, die sich bewegen und dabei für Druck- und Dichteschwankungen in der Luft sorgen. Um diese Schallwellen auf dem Computer verarbeiten zu können, müssen sie zunächst mit einem Mikrofon aufgenommen werden. Ein Mikrofon kann man sich als eine Spule auf einer Membran vorstellen, die durchs Hin- und Herbewegen entsteht per elektromagnetische Induktion ein analoges elektrisches Signal. Die Spannung ist umso höher, je lauter die Schallwelle ist, die auf das Mikrofon triff. Damit ein Computer dieses analoge Signal verarbeiten kann, muss es zunächst digitalisiert werden, also in Folgen von 0 und 1 kodiert werden. Hierzu wird in kurzen Zeitabständen regelmäßig die analoge Spannung gemessen und der digitale Wert gespeichert. Man spricht dabei von einem *Sample* und die Zeitabstände zwischen den Messungen wird entsprechend *Samplingfrequenz* genannt. Bei einer CD sind das normalerweise 44.100 Hz – es wird also ein Sample alle 0.022 Millisekunden bzw. 0,000022 Sekunden aufgezeichnet.

Wenn wir uns die 1000 Hz Sinuskurve in Abbildung 1 ansehen, so stellen die Punkte mögliche Samples dar. So eine einfache Sinuskurve ist auch mit relativ wenig Punkten schon klar zu erkennen. Komplexere Klänge

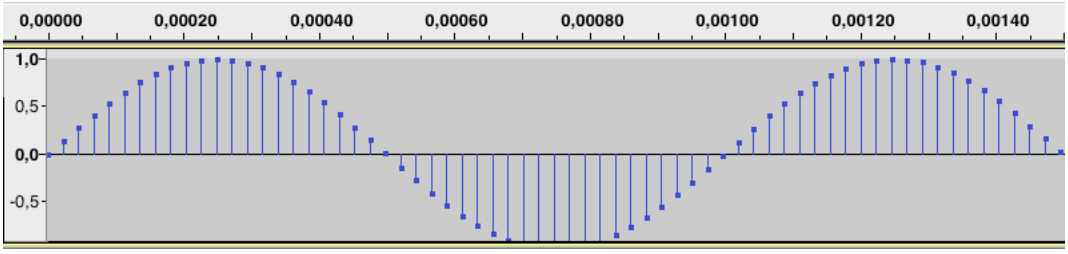

Abbildung 1: Samples von einer 1000Hz Sinus Kurve

enthalten aber eine Vielzahl von überlagerten Sinuskurven. Um herauszufinden, welche Eigenschaften die einzelnen Sinuskurven, aus denen sich der Klang zusammensetzt, haben, brauchen wir eine hohe Samplingfrequenz. Um aus diesen Klangsamples die im Klang enthaltenen Sinuskurven, die die Tonhöhe bestimmen, herauszufinden, benutzt man die sogenannte Fourier-Transformation, deren Grundlagen bis in

babylonische Zeiten zurückgeht[1], die aber entscheidend durch Fourier (1808) vorangebracht wurde.

Stellen sie sich vor, sie haben einen Smoothie und möchten gerne wissen, was der Smoothie enthält. Sie suchen also eine Methode, die einzelnen Zutaten des Smoothies herauszufinden. Genau das macht die Fourier-Transformation – nur eben mit Audiosignalen. Sie nimmt ein Audiosignal und zerlegt das in die Bestandteile, wobei die Bestandteile jeweils Sinuskurven bestimmter Frequenzen sind, die zusammengenommen das Audiosignal ergeben.

Wenn wir alles zusammen nehmen, so können wir also ein Audiosignal aufnehmen, die Spannung (in Volt) regelmäßig in bestimmten Zeitabständen ablesen, die Werte digitalisieren, und schließlich die Fourier-Transformation anwenden, die einem eine Abfolge der Energie pro Frequenz zurück gibt. Da dies eine zentrale Komponente unserer Analyse und Übersetzung von Schallwellen in Lichtsignale ist, sehen wir uns das im nächsten Abschnitt genauer an.

3.1 Den Klang in die Bestandteile zerlegen: Fourier-Transformation

Die Fourier-Transformation ist ein Algorithmus, der ein Audiosignal in seine Frequenzbestandteile zerlegt. Die Formel der diskreten Fourier-Transformation ist hier in (1) zu sehen:

$$F(f) = \int_{-\infty}^{\infty} g(t) e^{2\pi i f t} dt \tag{1}$$

Hier ist $g(t)$ der Zeit-Amplitude Graph des Audiosignals und t ist der Zeitpunkt an dem wir einen Wert aus von dem Graphen ablesen. Der Teil $e^{2\pi i f t}$ erstellt einen Einheitskreis um den Ursprung. Wenn dieser mit $f(t)$ multipliziert wird, so legt sich das Audiosignal um den Ursprung. Jetzt können wir mit der Frequenz f, die wir auslesen wollen, das Signal mehr oder weniger drehen. Wenn dabei eine herzförmige Form, eine sogenannte Kardioide, herauskommt, so ist diese Frequenz im Audiosignal vorhanden.

Die Formel legt also ein Reihe von Amplitudenwerten des Audiosignals um einen Einheitskreis und dreht das Signal, wobei das Integral umgangssprachlich den “Schwerpunkt des umgelegten Signals” berechnet. Wenn eine bestimmte Frequenz im Audiosignal vorhanden ist, dann bildet sich eine herzförmige Kurve. In der Visualisierung von Sanderson (2018) in Abbildung 2 ist diese Kurve unten links zu sehen (sein in der Literaturangabe verlinktes Video veranschaulicht auch sehr schön den Prozess).

In der Formel können wir natürlich nicht das herzförmige Bild sehen. Also berechnet das Integral $\int_{-\infty}^{\infty}$ in der Formel (1) für uns, wo sich der Schwerpunkt des um den Ursprung gelegten Kreises befindet. Je weiter entfernt der Schwerpunkt vom Ursprung ist, desto mehr von der Frequenz f enthält das Audiosignal. Wie viel von dieser Frequenz letztendlich im Audiosignal vorhanden ist, gibt dann die Funktion $F(f)$ zurück.

3.2 Fourier-Transformation auf dem Raspberry PI umsetzen

Den Wert für jede Frequenz in der gerade beschriebenen Weise zu berechnen, würde sehr viel Zeit beanspruchen. In der Praxis gibt es eine Reihe von optimierten Methoden, diese Berechnung effizienter zu machen, die zusammen unter Fast (= schnell) Fourier Transforms (FFT) bekannt sind.

[1] https://en.wikipedia.org/wiki/Fourier_analysis#History

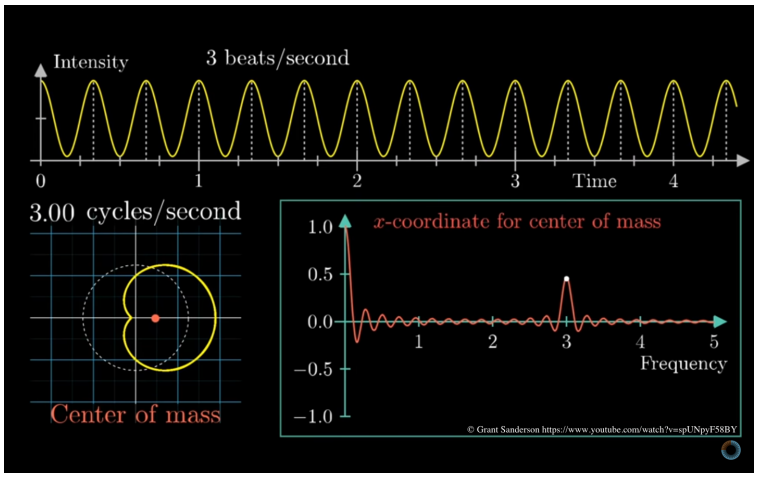

Abbildung 2: Wenn eine Frequenz identifiziert wird, erhalten wir eine Kardioide (Sanderson 2018)

Meine Umsetzung auf dem Raspberry Pi basiert auf der FFT Implementierung von Andrew Holmes, die einer Variante des Cooley-Tukey FFT Algorithmus mit Hilfe des GPU (Graphical Processing Unit) des Raspberry PIs besonders effizient berechnet (Holmes 2014). Dadurch, dass die Library auf dem GPU läuft, ist sie bis zu 12 mal schneller als auf dem CPU (Central Processing Unit) des Raspberry PIs. Das liegt daran, dass GPUs dafür optimiert sind, große Mengen von ähnlichen Berechnungen sehr effizient zu berechnen, die zum Beispiel für die schnelle Darstellung von Oberflächen in Computerspielen benötigt werden. Durch die geschickte Kodierung der Fourier-Transformation als eine solche wiederkehrende Operation wird die Effizienz der GPU durch die GPU_FFT Library ausgenutzt.

Auf der praktischen Seite gibt es zur Nutzung der GPU_FFT Library zwei Grundfunktionen, `gpu_fft_prepare()` und `gpu_fft_execute()`.

```
int gpu_fft_prepare(
   int mb,        // mailbox file_descriptor Rueckgabe von mbox_open()
   int log2_N,    // FFT Groesse 2^n = N
   int direction, // FFT Richtung: vorw. GPU_FFT_FWD, rueckw. GPU_FFT_REV
   int jobs,      // Anzahl Berechnungen im Batch Mode, ansonsten auf 1.
   struct GPU_FFT **fft    // Pointer zu einem Pointer zum FFT struct
   )

unsigned gpu_fft_execute(
   struct GPU_FFT *info    // Pointer zum FFT struct
   )

struct GPU_FFT {
   struct GPU_FFT_BASE base;
   struct GPU_FFT_COMPLEX *in, *out;
   int x, y, step;
};
```

Die erste Funktion muss nur einmal am Programmstart aufgerufen werden, denn sie richtet die Transformation ein. Interessanter ist der Aufruf von `gup_fft_execute()`, denn als Input und Output für die FFT wird hier eine Datenstruktur mit komplexen Zahlen benötigt. Diese Zahlen haben einen imaginären und einen reellen Teil. Wir werden das gleich und später auch bei der Diskussion eines Beispiels in Abschnitt 3.3 näher illustrieren. Für die Spezifikation des Inputs ist der imaginäre Teil aber noch nicht relevant. Wir schreiben also einfach das digitalisierte Audiosignal als reelle Zahlen in den Buffer und rufen `gup_fft_execute()` auf.

Im Anschluss haben wir das Ergebnis der FFT als komplexe Zahlen im Buffer. Komplexe und reelle Zahlen werden meist in einem zweidimensionalen Koordinatensystem dargestellt, die reellen Zahlen auf der X-Achse und die komplexen auf der Y-Achse. Um einen reellen Wert herauszubekommen, wird meist der Satz des Pythagoras verwendet, um eine Gerade zwischen Ursprung (0,0) und einem Wert auszurechnen. Dies muss aber für alle Werte gemacht werden, so dass wir dies in der folgenden for-Schleife kodiert haben:

```
for(i=0;i<(N/2);i++)
    {
    // record the amplitude of this frequency bin
    OutData[i]= (2*sqrt((base[i].re * base[i].re) +
                        (base[i].im * base[i].im)))/N;
    }
```

Zu beachten ist, dass hier nur die Hälfte der Output Bins mit Anzahl N ausgelesen werden. Dies liegt daran, dass ich nur einen reellen Teil als Input gegeben habe, so dass die zweite Hälfte, der Imaginäre Teil leer bzw. 0 ist. Die zweite Hälfte der Ausgabe enthält daher nur eine Spiegelung der ersten Hälfte.

3.3 Ein Beispiel

Sehen wir uns einen konkreten Fall etwas näher an. In Abbildung 3 ist als möglichst einfaches Beispiel eine Sinuskurve mit einer Frequenz von 600Hz und einer Amplitude von 1 zu sehen.[2]

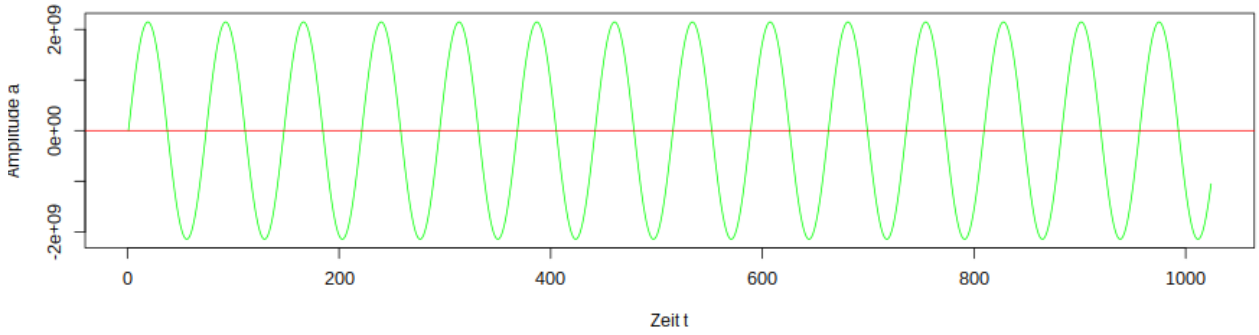

Abbildung 3: 600hz Sinus Kurve

Wenn man dieses Signal jetzt durch die Fast Fourier-Transformation laufen lässt, so bekommt bekommt man wie besprochen komplexe Zahlen heraus. Diese sind in Abbildung 4 dargestellt. Der einzelne Punkt oben rechts entspricht dem Ausschlag bei 600 Hz.

[2] Alle hier gezeigten Diagramme habe ich in R mit der plot() Funktion erstellt.

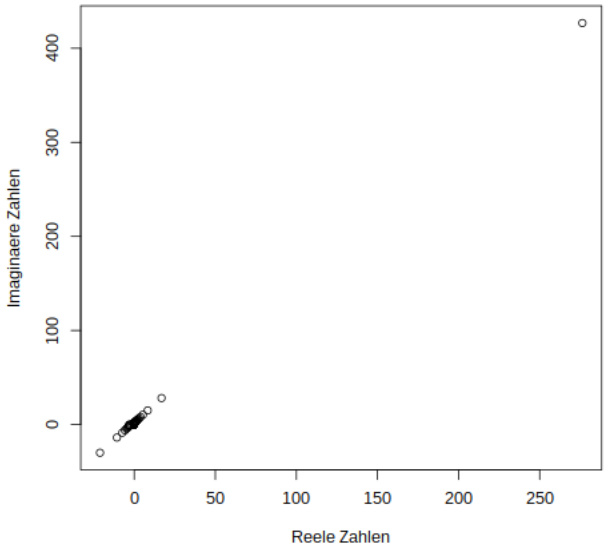

Abbildung 4: Reelle und imaginäre Zahlen nach der Fast Fourier-Transformation

Wie kommen wir nun zu diesem Frequenzwert? Wenn man den Satz des Pythagoras anwendet, so bekommt man ein schönes Frequenzspektrum, das in Abbildung 5 gezeigt ist und das eine klare Frequenzspitze bei 600Hz zeigt.

4 Berechnung der LEDs

Als Output der Fourier-Transformation haben wir jetzt $N/2$ sogenannte *Bins*, die unsere Frequenzdaten beinhalten. Solche "Schnappschüsse" des aktuellen Frequenzspektrums bekommen wir in schneller Folge, jeweils für ein kurzes Fenster des Audiosignals. Mit diesen Daten wollen wir nun LEDs anzusteuern. Dabei gibt es unendlich viele Möglichkeiten, da wir so gut wie das gesamte hörbare Schallwellenspektrum in unseren Daten haben. Das Spektrum welches wir nun lesen können ist definiert im Bereich $\frac{\text{sampling rate}}{N} * \frac{N}{2}$. Hier ist $\frac{N}{\text{sampling rate}}$ die Auflösung in den einzelnen Bins. Das heißt, wenn wir $N = 1024$ samples haben und unser Audiosignal eine Samplingrate von sampling rate $= 44.100Hz$ hat, so ist unsere Auflösung $\frac{44100Hz}{1024samples} \approx 43.07$. Das bedeutet, dass jede unserer Bins $43.07Hz$ des Frequenzspektrums beinhaltet. Zu einer weiteren Erhöhung der Auflösung könnte wir noch die Anzahl der Samples erhöhen, z.B. auf 2048. Dann hätte man $\approx 21.53Hz$ pro Bin. Der Nachteil der Erhöhung der Auflösung durch die Hinzunahme von mehr Samples in das zu analysierende Fenster ist, dass die Samples dann eine längere Zeitspanne auf einmal analysieren. Wenn wir, zum Beispiel, $2^{15} = 32.768$Samples nehmen, so sind dies $\frac{32.760}{44.100} \approx 0,74s$ in den Samples enthalten. Änderungen des Klangs innerhalb dieser $0,74$ Sekunden können also nicht unterschieden werden. Eine genauere Auflösung innerhalb des Frequenzspektrums ginge also klar auf Kosten der zeitlichen Auflösung.

Jetzt wollen wir diese Informationen auf LEDs verteilen. Im typischen Fall wird sich die Anzahl der LEDs aber von der Anzahl der Bins, in die wir das Frequenzspektrum unterteilt haben, unterscheiden. In meinem

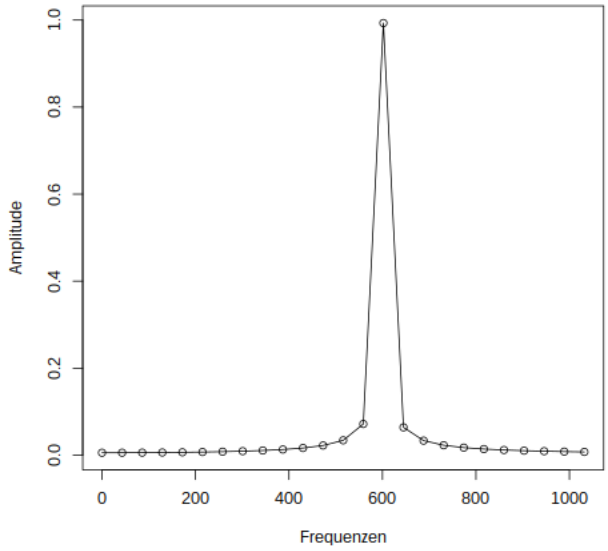

Abbildung 5: Von der FFT berechnetes Frequenzspektrum der 600Hz Sinuskurve

Beispiel arbeiten wir mit $N = 1024$ also $N/2 = 512$ Bins und 28 LEDs. Daher stellt sich die Frage, wie man diese Bins sinnvoll aufteilt. Im einfachsten Falle könnte man sie linear aufteilen, so dass jede LED die Information aus $512/28 = 18,29$ Bins bezieht. Jedoch können wir natürlich keine halben oder viertel Bins auf LEDs verteilen. Also müssen wir den Rest der Division berücksichtigen. In unserem Fall haben wir $512 - (18 * 28) = 512 - 504 = 8$; wir müssen also diese 8 Bins gleichmäßig auf die LEDs aufteilen. Wir erhalten $28/8 = 3.5 \approx 4$. Jede vierte LED erhält also einen Bin zusätzlich. Im C Code sieht dies wie folgt aus:

```
if((current_led % overflow) == 0)
        num_bins = scaling + 1;
else
        num_bins = scaling;
```

Hier ist *current_led* die aktuelle LED, von 0-27, für die wir die Anzahl der Bins berechnen wollen und *overflow* ist die Anzahl der Bins die wir nicht zuteilen konnten. Hierbei ist das % nichts anderes als eine Division mit Rest, wo nur der Rest zurückgegeben wird. Wenn kein Rest zurückgegeben wird, dann ist die Zahl genau teilbar. Wenn wir zum Beispiel die vierte LED betrachten, so bekommen wir heraus, dass diese Zahl genau teilbar ist $(4\%4 = 0)$; wenn wir 5 einsetzen, so bekommen wir $5\%4 = 1$, da diese nicht genau teilbar sind. Wenn die Zahl ohne Rest teilbar ist, erhalten wir also 0. In diesem Fall nehmen wir also die *scaling* Variable, die kodiert wie viele Bins auf eine LED zugeteilt sind, und addieren 1. Im anderen Fall, bei einer Division mit Rest, bleibt die Anzahl der Bins *num_bins* unverändert.

Aber für das menschliche Gehör sind Frequenzen über 1000Hz sehr hoch und wir können sie längst nicht mehr so gut unterscheiden wie Frequenzunterschiede im tieferen Frequezbereich. Eine einfache Möglichkeit wäre es, Frequenzen ab einer bestimmten Höhe schlicht nicht mehr anzuzeigen. Eine attraktivere Lö-

sung ist jedoch die Verwendung einer logarithmischen Skala. Die Skala muss allerdings gewährleisten, dass sie aus X LEDs $N/2$ Bins aufteilt und am Ende alle Bins, die auf die LEDs aufgeteilt sind, addiert wieder $N/2$ ergeben. Basierend auf einer Berechnung zur Basis 2 dachte ich mir zunächst, dies wäre eine einfache Sache, da alle Zweierpotenzen bis $n-1$ addiert die Potenz von 2^n-1 ergeben, z.B., $2^0+2^1+2^2+2^3 = 2^4-1 = 15$. Doch wenn man dies jetzt mit 3 ausprobiert, so erhält man $3^0+3^1+3^2+3^3 = 40$ und nicht $3^4-1 = 80$. Eine logarithmische Skala die auf einer Summenformel basiert ist also ziemlich kompliziert. Daher habe ich eine Funktion geschrieben, welche am Anfang einmal die Basis für die logarithmische Skalierung approximiert, die dann immer genutzt werden kann.

```
static float log_scale(int width, int range){
  static int st, num, i;
  float log = 1.0, min = range*1.0, sum;
  int min_rnd = range, sum_rnd;
  for(st = 1; st < 7;st++){
    for(num = 0; num<10; num++){
      sum = 0.0, sum_rnd = 0;
      log += pow(10, -1*st);
      for(i = 0; i < width; i++){
        sum += pow(log, i);
        sum_rnd += round(pow(log, i)); }
      if(abs(range-sum) <= min && abs(range-sum_rnd) <= min_rnd){
        min = abs(range-sum);
        min_rnd = abs(range - sum_rnd);
      }else{
        log-=1.5*pow(10, -1*st);
        sum = 0.0, sum_rnd = 0;
        for(i = 0; i < width; i++){
          sum += pow(log, i);
          sum_rnd += round(pow(log, i)); }
        min = abs(range*1.0-sum);
        min_rnd = abs(range - sum_rnd);
        break; }}}
  return log;}
```

Dies funktioniert so, dass die Funktion für 6 Nachkommastellen schaut ob diese als Basis für die addierten Potenzen geeignet ist. Das Programm geht diese mit *for(st = 1; st < 7;st++)* durch. Zuerst wird bei jeder ausprobierten Basis die Summe der gesamten Bins ausgerechnet. Wenn diese im Vergleich mit der Vorherigen Summe besser ist, wird bei der gleichen Nachkommastelle eine höhere Zahl ausprobiert. Ist jedoch die Summe der aktuellen Nachkommastelle schlechter als die vorherige, so wird mit der vorherigen Nachkommastelle in der nächsten Stelle nach der optimalen Basis gesucht.
In einem Beispiel sähe dies so aus:
$1,1 \rightarrow 1,2 \rightarrow 1,3 \rightarrow 1,21 \rightarrow 1,22 \rightarrow 1,211$ usw.
Jetzt hat sich aber herausgestellt, dass die vorherige Nachkommastelle immer eine gerundete Zahl von der

darauffolgenden Nachkommastelle ist. Also wenn $1,3$ die erste optimale Nachkommastelle wäre, könnte die in Wirklichkeit optimale Basis zwischen $1,25$ oder $1,34$ liegen. Also geht der Alorithmus statt auf die optimale Nachkommastelle auf die Zahl, die um 0,5 darunter liegt. In einem Beispiel sähe dies so aus:
$1,1 \rightarrow 1,2 \rightarrow 1,3 \rightarrow 1,15 \rightarrow 1,16 \rightarrow 1,17 \rightarrow 1,155$ usw.
Die Funktion gibt in der Variable *log* die optimale Basis zurück.

4.1 Ansteuerung einzelner LEDs

Nachdem wir die FFT Bins nun gleichmäßig oder logarithmisch auf eine beliebige Anzahl von LEDs verteilen können, wollen wir nun die LEDs zur Musik blinken lassen oder Farbe wechseln lassen. Zum Einsatz kommen die in Abbildung 6 gezeigten WS2812B RGB-LEDs (RGB steht für Red, Green, Blue), die über ein einadriges, serielles Protokoll angesteuert werden.

Abbildung 6: WS2812B LED

Abbildung 7 zeigt das Blockschaltbild und die Anschlussbelegung einer WS2812B. Um mehrere LEDs ansteuern zu können, werden VDD (Eingangsspannung) und GND (Erdung), sowie Dout (Datenausgang) von der einen LED mit Din (Dateneingang) mit der nächsten LED verbunden (siehe Abbildung 8). Über ein spezielles Protokoll werden der ersten LED 24bit Daten (8bit für rot, 8bit für grün und 8bit für blau) geschickt,

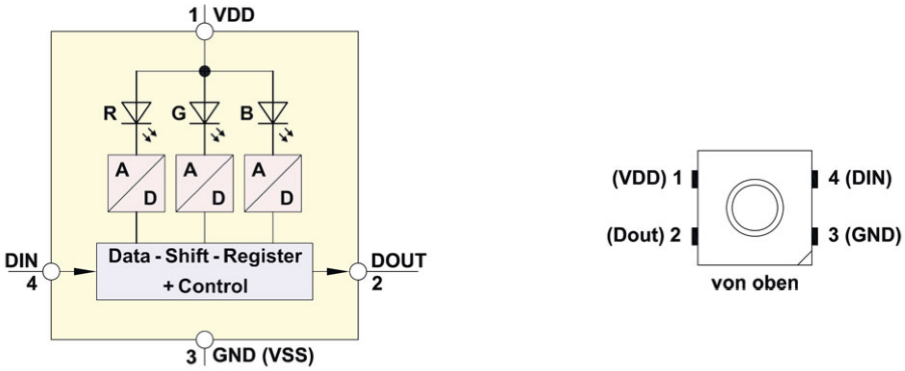

Abbildung 7: Aufbau von WS2812B LEDs

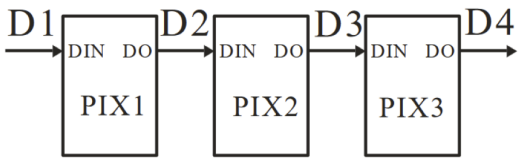

Abbildung 8: LEDs in Serie

das im WS2812B Datenblatt näher spezifiziert ist.[3]. Danach hat die erste LED ihre Daten vollständig empfangen und leitet ab sofort alle Daten an die nächste LED weiter. Diese empfängt nun ebenfalls 24bit, leitet danach dann weiter, u.s.w. So lassen sich mit einem einzelnen Pin des Raspberry Pi nahezu beliebig viele LEDs ansteuern, wohingegen der Aufwand bei normalen RGB-LEDs unermesslich gewesen wär.

Um eine Farbe anzuzeigen, muss für jeden der RGB Werte ein Byte, bzw. 8 Bit geschrieben werden, also immer ein Wert von 0-255. Als Beispiel hat Rot den RGB Wert (255,0,0) und Grün (0,255,0). Da die Lichtwellen addiert werden, hat Weiß den Wert (255,255,255). Jetzt ist es möglich, hiermit sehr sehr viele verschieden Farben anzuzeigen. Um die Helligkeit zu regulieren, müssen nur alle Werte verkleinert werden. (255,255,255) ist Weiß in seiner maximalen Helligkeit, (127,127,127) ist dementsprechen 50% der Helligkeit. Genauso ist es bei allen Farben, z.B. ist der Wert von Rot (127,0,0) bei 50% Helligkeit. Jetzt müssen aber diese LEDs irgendwie ihre Daten bekommen. Ursprünglich dachte ich mir, ich würde die LEDs über einen Mikrocontroller, einen Arduino, ansteuern, doch dies würde eine weitere Ebene der Komplexität mit sich bringen: die Kommunikation zwischen Raspberry Pi und Mikrocontroller. Jetzt verwende ich aber die SPI Schnittstelle vom Raspberry Pi, um die LEDs zu Steuern. Serial Peripheral Interface kurz SPI ist eine Schnittstelle, die normalerweise für die Kommunikation zwischen einem Master und einem Slave genutzt wird.

Abbildung 9: SPI Beispiel

In meinem Fall haben die LEDs nur einen Dateneingang also benutze ich nur den “Master Output Slave Input” Port kurz MOSI. Die anderen Ports brauche ich nicht. Da die LED-Streifen keinen Ausgang haben, brauche ich den “Master Input Slave Output” kurz MISO Port nicht. Den SCLK “Serial Clock” Port muss ich auch nicht verwenden, da die LEDs ein eingebautes Timing haben, wo kein zusätzliches Timing mehr gebraucht wird. Da ich die Taktfrequenz/Timing frei wählen kann, konnte ich diese auf meinen LED-Streifen anpassen, sodass dieser einwandfrei funktioniert. Hier haben 4 MHz (Megaherz) sich als passend erwiesen. Das Ansteuern der LEDs funktioniert so, dass immer einzelne Bytes gesendet werden und wie schon erwähnt besteht ein Byte aus 8 Bits.

[3] https://cdn-shop.adafruit.com/datasheets/WS2812B.pdf

Für die WS2812b LEDs wird eine 1 als 1100 kodiert und eine 0 als 1000. Um eine Farbe zu senden, muss man demnach 32 SPI Bits senden, also 4 SPI Bytes. Also wenn man jetzt die Taktfrequenz betrachtet, mit der die einzelnen Bits bei den LEDs ankommen, ist dies nur noch 1 MHz oder 1000 kHz. Dies ist nahe genug an den 800 kHz, die als Standard auf dem Datenblatt der WS2812b LEDs angegeben sind. Mit dieser Geschwindigkeit kann man jetzt ausrechnen, wie lange es dauert, bis der gesamte LED-Streifen beschrieben wird. $1\mu * 24$bit $*300$LEDs $= 7200\mu \approx 0.007s$ ist also die Zeit, die für einen LED-Streifen mit 300 LEDs gebraucht wird, bis dieser beschrieben wurde. Doch wie sich nach einigen Testversuchen herausstellte, zeigten die LEDs falsche Farben an. Nach Betrachten des Datanblattes stellte sich heraus, dass die LEDs die Farben in der Reihenfolge Grün, Rot, Blau, statt wie normalerweise Rot, Grün, Blau, kurz RGB, anzeigen. Dies ließ sich glücklicherweise einfach beheben bei der Ausgabe der LEDs. Zur Steuerung der LEDs habe ich das Programm von penfold42 `https://github.com/penfold42/stuff/blob/master/ws2812_spi.c` modifiziert und eingebunden.

5 Vom Klang zum Farbenspiel: eine parametrisierbare Übersetzung

Was möchten wir in was übersetzen? Ausganspunkte sind Schallwellen, d.h. eine Sequenz von Amplituden über Zeit. Für die Ansteuerung der LEDs haben wir also $N/2$ Bins und diese beinhalten Daten, genauer gesagt Amplituden über das Frequenzspektrum.

Jetzt wissen wir zwar, wie die Bins auf die LEDs aufgeteilt werden, doch die LEDs haben noch gar keine Daten und wissen nicht, was sie anzeigen sollen. Hier ist mein Programm sehr modular und es lässt sich sehr leicht parametrisieren. Als Beispiele haben wir vier Parametrisierungen eingebaut. D.h. es gibt verschiedene Möglichkeiten die Schallwellen, die in FFT Bins vorhanden sind, darzustellen. Ich habe in meinem Programm bis zu diesem Zeitpunkt vier verschiedene Arten implementiert, die FFT Ergebnisse darzustellen. Bei der ersten Parametrisierung wird das Lichtspektrum auf die LEDs geschrieben und die höchste Amplitude der Bins, die auf eine LED zugewiesen sind, ist die Helligkeit dieser LED. Also immer, wenn eine bestimme Frequenz vorhanden ist, leuchtet die zugehörige LED auf. In Abbildung 10 sieht man einen C-Dur Dreiklang, der mit Hilfe der ersten Parametrisierung angezeigt wird.

Abbildung 10: Modus 1

Die zweite Parametrisierung zeigt die Amplitude nicht als Helligkeit, sondern als Farbe an: je höher die Amplitude desto höher ist die Lichtfrequenz der LED. Da Licht eine elektromagnetische Wellenform ist, gibt es hierfür auch ein Spectrum: das Licht, welches für den Menschen sichtbar ist liegt zwischen 400 und 700 Nanometern Wellenlänge. Um die Lichtfrequenz herauszubekommen oder das Lichtspektrum, habe ich eine Funktion eingefügt und modizfiziert, die mir die Wellenlänge des Lichtes in RGB Werte umrechnet.

In der Abbildung 11 sieht man, wie die RGB Werte zum Lichtspektrum aussehen. Um die RGB Werte herauszubekommen, habe ich eine Funktion geschrieben. In die Funktion gibt man die Wellenlänge des Lichtes von 400-700 und die Helligkeit und bekommt ein *led_data* Struct zurück. Das Struct wird dann in das Array geschrieben, welches für alle LEDs ein sogenanntes *led_data* Struct beinhaltet. Dieses *led_data*

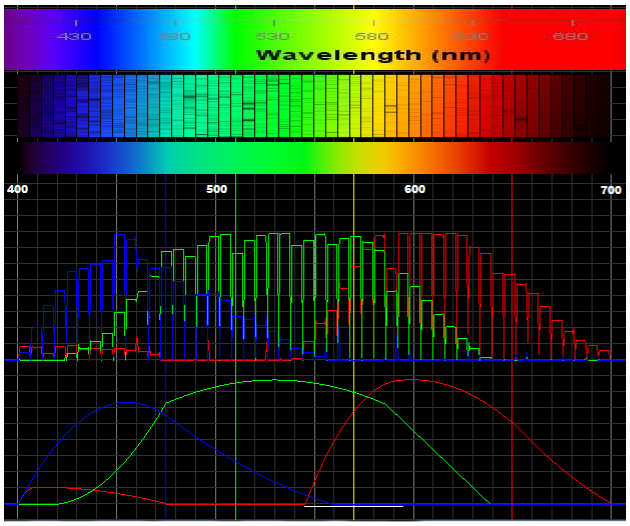

Abbildung 11: Analyse des Lichtspektrum zur Umsetzung in RGB Werte

Struct beinhaltet für jeden RGB Wert eine Variable und zwar im 8 Bit Format.

```
struct led_data {
        uint8_t r;
        uint8_t g;
        uint8_t b;
};
```

Am Ende wird dann noch die Helligkeit mit einberechnet, indem durch sie geteilt wird. Erst müssen die Helligkeitswerte, die von 1, ganz dunkel, bis 100, maximale Helligkeit, einberechnet werden. In Abbildung 12 wird ein C-Dur Dreiklang mit Modus 2 angezeigt.

Abbildung 12: Modus 2

Nun zur dritten Parametrisierung, welche ich etwas genauer erläutern werde. Diese Parametrisierung ist sehr ähnlich wie die zweite, doch es gibt auch noch eine Bass Analyse. Dieser Bass erscheint dann als Helligkeit des Gesamten LED-Streifens. Der Bass ist so definiert, dass die Frequenzen von 0-200Hz herausgefiltert werden und eigens untersucht werden. Dies funktioniert so, dass alle Ampltiuden die im Bass enthalten sind addiert werden und dann mit 200 multipliziert werden, sodass sie auf eine Skala von 0-200. Eigentlich rechne ich bei der Helligkeitseinrechnung nur mit einer Helligkeit von 0-100 aber wenn der Wert über diese 100 hinaus ist, so wird extra Helligkeit hinzugefügt, welche den Effekt erzeugt, dass immer wenn der Bass pulsiert der gesamte LED-Streifen dies auch macht.

```
for (current_bin=1;current_bin<low_bins;current_bin++)
{
        max += Data[bin];
}
//set brightness by * multipling with the low_bins max
brightness = round(max * 200)
```

Wie viele Bins als Bass definiert sind, steht in *low_bins*, so kann die for-Schleife durch alle Bass Bins durchgehen. Mittels *+=* werden alle Bass-Amplituden summiert. Am Ende wird, wie schon erwähnt, die maximale Amplitude in *max* noch auf 200 skaliert und dann in die *brightness* Variable geschrieben.

Die Auslese der Mittelfrequenzen, die angezeigt werden, da die Höhen schon herrausgefiltert wurden, funktioniert genau so, wie bei der den Bässen, nur dass die maximalen Amplituden in Farben umgerechnet werden. Bevor die Amplituden in Farben umgerechnet werden filtere ich noch kaum hörbare Amplituden raus, das diese LEDs zum leuchten bringen würden, obwohl diese nicht hörbar sind.

```
if (max < 0.05) max = 0.0;
amplitude = max*250.0+400;
```

Durch das Multiplizieren mit 200 wird *max* auf 0-250 skaliert und durch das addieren von 400 haben wir eine Zahl zwischen 400 und 650 im Lichtspektum. Eigentlich ist das Lichtspectrum zwischen 400-700 wie in Abbildung 11 zu sehen ist, aber ich habe herausgefunden, dass die letzten 50nm kaum einen Unterschied machen. Die ersten 50nm haben zwar auch kaum einen Unterschied in Farbe, jedoch in Helligkeit, so werden Amplituden die sehr gering sind auch nur sehr dunkel angezeigt.

```
(*ledstrip_data)[current_led] = spectral_color(amplitude, brightness);
```

Letztendlich werden die Lichtspectrums Werte in *amplitude* und die Helligkeitswerte *brightness* mit der Funktion *spectral_color* in RGB Werte umgeschrieben und in den Pointer zum aktuellen *led_data* Struct im Array geschrieben.

6 Laufzeitanalyse und Optimierung

Mein Programm war zunächst so aufgebaut, dass erst Samples vom Mikrofon gesammelt wurden und danach die Fourier Transformation, LED-Berechnung und LED-Ansteuerung durchgeführt wurden. Das Problem dabei ist, dass, während die anderen Schritte durchgeführt werden, die direkt darauffolgenden Samples nicht aufgezeichnet werden und so verloren gehen. Und diese verlorenen Samples können dann nicht visualisiert werden.

Aber der Raspberry Pi hat einen Quad-Core, also vier separate Kerne, die sich unterschiedlichen Aufgaben widmen können. In der Programmierspreache C können mit POSIX Threads (pthread) mehrere parallele Ausführungsstränge erstellt werden, welche auf die Prozessorkerne aufgeteilt werden. Das habe ich mir zu Nutzen gemacht. Ich habe das Auslesen der Samples in einen separaten Thread gepackt und so dafür gesorgt, dass immer alle Samples aufgezeichent werden. Die gesamte Berechnung der LEDs mit Fourier

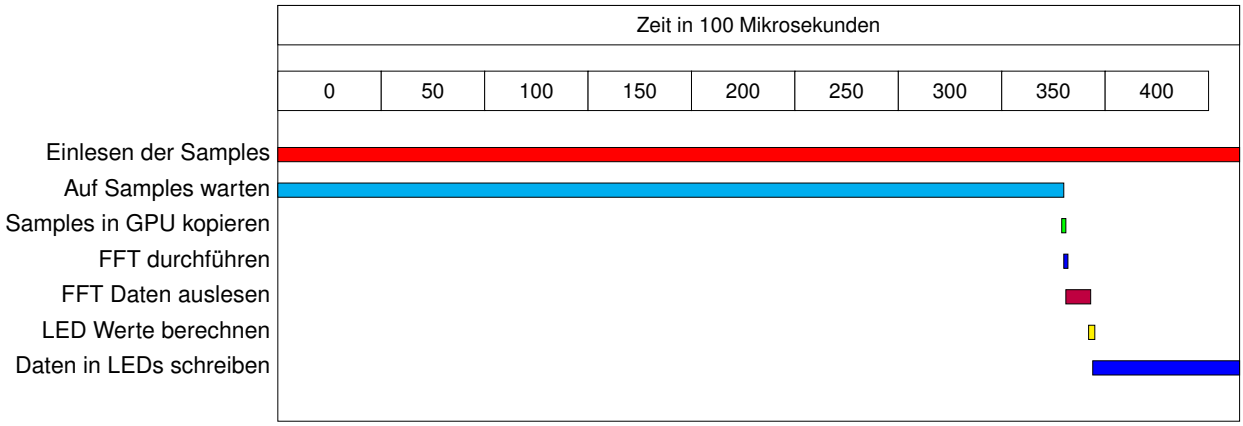

Abbildung 13: Überblick über Laufzeit des Programms

Transformation und Ansteuerung dauert kürzer als das Auslesen der Samples. Also wartet das Programm darauf, bis es vom Auslese-Thread ein Signal bekommt, dass der Buffer mit den Samples voll ist. In Abbildung 13 sind die Timings des Programmes zu sehen. Ein Programmdurchlauf dauert insgesammt 46400 Mikrosekunden oder 0,0464 Sekunden, bei $N = 2048$, da mit einer Samplingfrequenz von 44100 die Zeit für das Lesen der Samples sich folgendermaßen berechnet: $\frac{1s}{44100Hz} * 2048 = 0,0464s$. Die Timings, welche abgebildet sind, gibt es erst nach dem ersten Durchlauf, da beim ersten Mal noch überhaut keine Samples vorhanden sind.

7 Zusammenfassung

Das Ziel meines Projektes war es, eine frei verfügbare Methode zu entwickeln, die Schallwellen in Lichtsignale umwandelt und damit LEDs ansteuert. Diese flexible Methode habe ich mit Hilfe der Fourier-Transformationen umgesetzt. Außerdem habe ich nicht wie bei traditionellen Lichtorgeln nur 1 oder 3 Kanäle, sondern eine präzise Analyse vom gesamten Frequenzspectrum implementiert. Um es möglichst vielen Menschen möglich zu machen, das Programm selber laufen zu lassen, habe ich das Projekt auf GitHub veröffentlicht. Es gibt aber noch einige Dinge, welche noch erweitert werden können. Eines der größten Projekte ist es, noch mehr Modi zu einwickeln. Durch die Parametrisierung ist hier sehr viel Spielraum noch offen. Im Moment kann das Programm nur von einem Mikrofon oder einer WAV-Datei Daten bekommen. Ich würde gerne das Programm im Hintergrund laufen lassen, sodass das Programm sich die Daten aus dem internen Audiospeicher holt, welcher die Daten für den Kopfhöreranschluss beinhaltet und damit die Fourier-Transformation betreibt. Wieder auch ist es möglich das Programm effizienter zu machen, sodass es mehr Zeit gibt, komplizierte Lichteffekte zu berechnen. Auch die moderne Technik mit Künstlichen Intelligenzen möchte ich mit einfließen lassen und so z.B. den Rhythmus einer Melodie herauszufinden und vielleicht auch die Melodie zu separieren. Auch den Google Assistant möchte ich einbinden, sodass es möglich ist, die Einstellung der LED per Sprachbefehle zu steuern.

Ich habe das Projekt als freie Software entwickelt und unter der GNU Lizenz auf GitHub frei verfügbar gemacht unter `https://github.com/mariusdkm/rpi-fft-led/`

Literatur

Fourier, J. (1808). Mémoire sur la propagation de la chaleur dans les corps solides. *Nouveau Bulletin des Sciences de la Société Philomathique de Paris*, 6:112–116.

Holmes, A. (2014). The GPU-FFT Library. `http://www.aholme.co.uk/GPU_FFT/Main.htm`.

Sanderson, G. (2018). But what is the Fourier Transform? A visual introduction. `https://www.youtube.com/watch?v=spUNpyF58BY`.