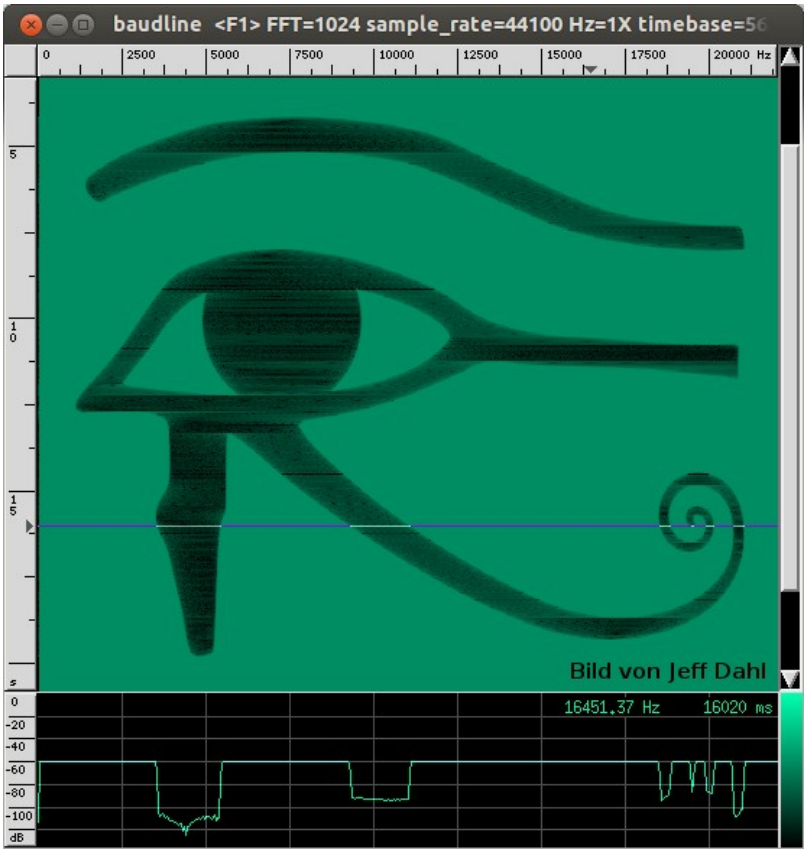
GYMNASIUM PENZBERG
Oberstufenjahrgang 2011/2013

Seminarfach Informatik

Seminararbeit



Umwandlung von Bildern in Audio-Signale

von Florian Märkl

Kursleiter: StD Karl Steiner
Bewertung: ..... Punkte
Unterschrift des Kursleiters: .....

INHALTSVERZEICHNIS

1. EINLEITUNG

Ein Bild besteht aus Pixeln mit Farbwerten, ein Audiosignal hingegen aus mehreren überlagerten Schwingungen. Was wäre, wenn man ein Bild in eine Audiodatei umwandeln könnte? Prinzipiell müsste man lediglich einen Weg finden, die gegebenen Bildinformationen so in Audioinformationen zu konvertieren, dass es möglich ist, nur aus diesen das ursprüngliche Bild zumindest so wiederherzustellen, dass es immer noch klar erkennbar ist. In der folgenden Arbeit wird ein solches Verfahren vorgestellt.

Zunächst werden Grundlagen zur Akustik und der digitalen Darstellung von Bildern geklärt. Danach wird das Verfahren und die dazu nötige Mathematik erläutert. Zuletzt wird es durch ein Programm in die Praxis umgesetzt und getestet.

2. GRUNDLEGENDES

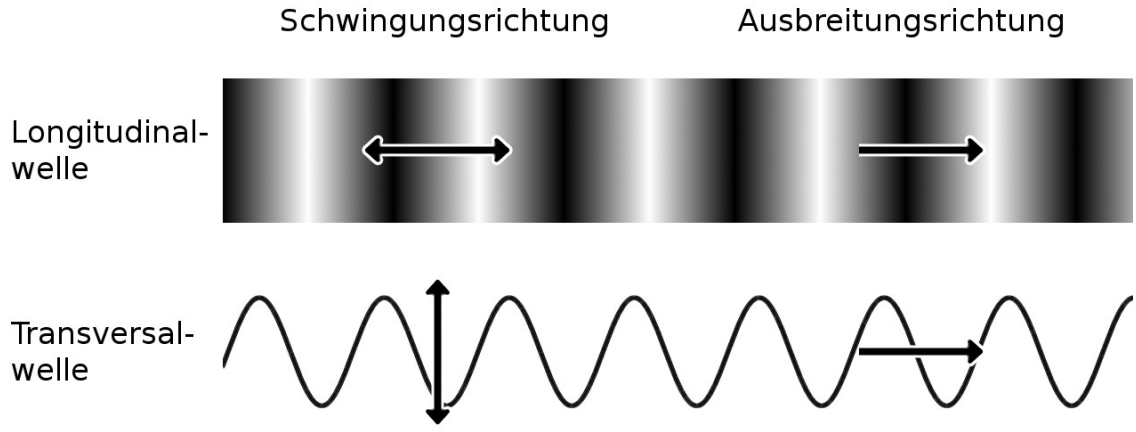
2.1 Akustik

Akustik ist der Zweig der Physik, welcher sich mit Schall und dessen Ausbreitung beschäftigt. Dabei werden auch zusammenhängende Aspekte wie die Entstehung, die Analyse oder die Wahrnehmung des Schalls betrachtet.

Grundsätzlich bezeichnet „Schall“ mechanische Schwingungen in einem elastischen Material (Druckveränderungen der Raumluft). Entstehen können diese Schwingungen z.B. durch mechanische Anregungen wie dem Auf- und Abschwingen einer Gitarrensaite.

2.1.1 Wellen

Bei der Betrachtung von Schwingungen ist es notwendig die verschiedenen Arten von Wellen zu kennen: Bei Longitudinalwellen ist die Schwingungsrichtung gleich der Ausbreitungsrichtung während bei Transversalwellen die Richtungen senkrecht aufeinander stehen.

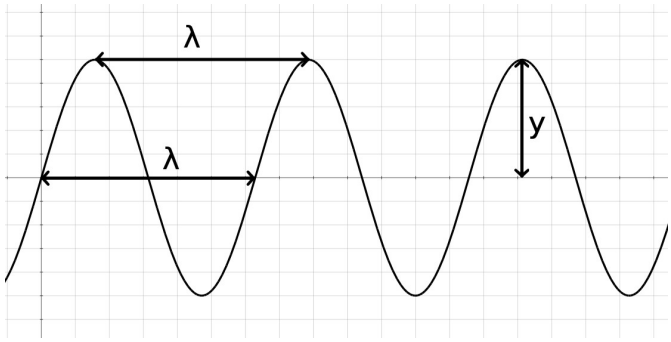


In der Realität sind Schallwellen Longitudinalwellen, da die Druckveränderungen der Luft in der gleichen Richtung wie die Ausbreitung auftreten. Hat man Audiosignale als elektrisches Signal (z.B. aus einem Mikrofon) vorliegen, verwendet man zu deren Dar-

stellung die Form einer Transversalwelle, deren momentane Auslenkung in einem Koordinatensystem senkrecht und die Zeitachse waagrecht aufgetragen wird, wie es z.B. bei einem Oszilloskopen passiert.

2.1.2 Begriffsklärungen

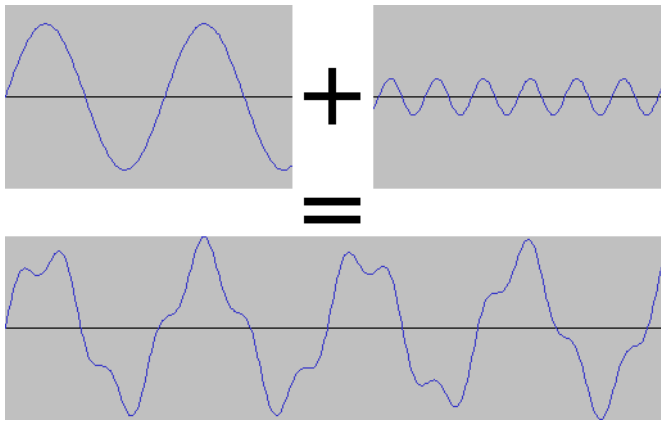
Bei der Betrachtung von Schwingungen verwendet man folgende Begriffe:



- **Phase π:** Ein bestimmter Schwingungszustand mit bestimmter Auslenkung und Bewegungsrichtung
- **Wellenlänge λ (Lambda)**: Minimaler Abstand zwischen zwei Punkten mit gleicher Phase
- **Amplitude y:** Maximale Auslenkung der Schwingung
- **Frequenz f:** Anzahl der vollen Schwingungen pro Sekunde. Einheit Hz = 1/s
- **Schwingungsdauer T:** Dauer einer vollen Schwingung. Zusammenhang zur Frequenz: f=1/t

2.1.3 Überlagerung von Schwingungen

In der Praxis bestehen Sprache, Musik, Geräusche etc. aus vielen verschiedenen Schwingungen mit unterschiedlicher Frequenz. Diese Frequenzen werden durch einfache Addition der Auslenkungen gemischt. Das Ergebnis klingt, als würden die Signale gleichzeitig klingen.



Addition von Frequenzen

2.1.4 Digitale Darstellung

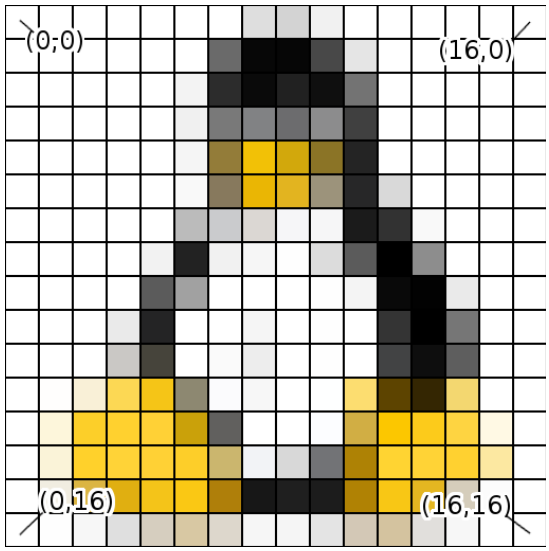
Digital erfasst werden diese Signale, indem kontinuierlich Messwerte, genannt „Samples“, der positiven oder negativen Auslenkung abgespeichert werden. Die Frequenz, in der die Samples aufgenommen werden, wird als Abtastrate, Abtastfrequenz oder Samplerate bezeichnet.

Sehr häufig (z.B. bei Audio-CDs oder Musikdateien) werden 44,1kHz als Abtastrate verwendet, also 44100 Messwerte pro Sekunde. In professionellen Studios wird häufig mit höheren Abtastraten, wie z.B. 96kHz, gearbeitet.

2.1.5 Nyquist-Frequenz

Um alle Frequenzanteile einer Audio-Datei rekonstruieren zu können, ist eine Abtastfrequenz notwendig, die mindestens doppelt so groß ist wie die höchste vorkommende Frequenz. Umgekehrt können also halb so hohe Frequenzen wie die Abtastfrequenz dargestellt werden. Die halbe Abtastrate wird als „Nyquist-Frequenz“ bezeichnet.

2.2 Bilder



2.2.1 Allgemein

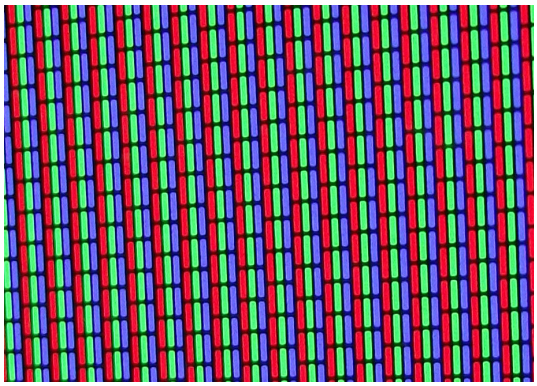
Wie bereits genannt, bestehen Bilder grundsätzlich aus Pixeln, die horizontal und vertikal angeordnet sind. Will man auf ein bestimmtes Pixel referenzieren, gibt man dessen x- und y-Koordinate an. Der Ursprung liegt in den meisten Fällen links oben (OpenGL ist beispielsweise eine Ausnahme).

2.2.2 Farbräume

Jedem Pixel ist ein Farbwert zugeordnet. Wie dieser Wert angegeben wird, bestimmt der sogenannte Farbraum. Von diesen gibt es mehrere:

2.2.2.1 RGB

Der RGB-Raum ist der bekannteste und für digitale Bildverarbeitung am meisten genutzte Farbraum. Dabei wird jeweils der Anteil der Farben Rot, Grün und Blau, meist in dieser Reihenfolge, angegeben. Die Farben ergeben sich durch additive Mischung der drei Anteile. (0,0,0) ist also schwarz, (1,1,1) ist weiß, (1,0,0) ist rot usw.



Nahaufnahme einer weiß aussehenden Fläche auf einem CRT-TV

Dieses Prinzip entspricht der Mischung von Licht. Beleuchtet man eine Fläche mit rotem, grünem und blauem Licht zu gleichen Teilen, ergibt die Mischung weißes Licht. Dies machen sich u. A. auch Röhrenfernseher zu nutze. Da diese eine relativ niedrige Auflösung haben, kann man die Farbanteile der Bildpunkte mit bloßem Auge erkennen (siehe Bild).

2.2.2.2 CMYK

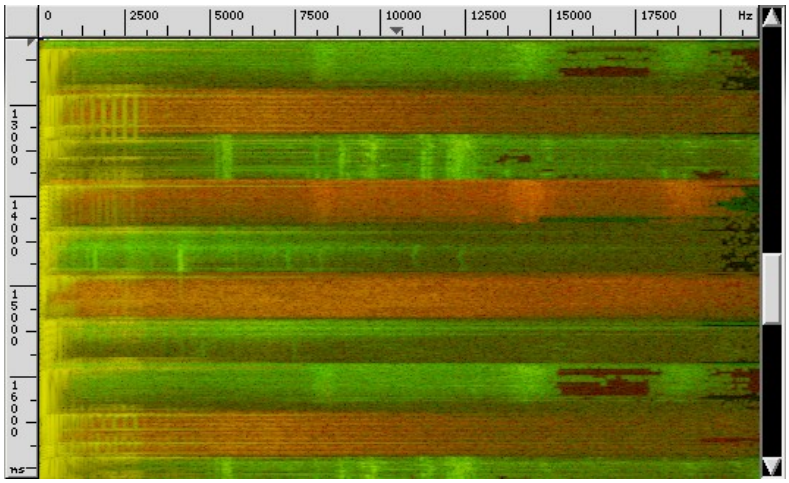
Analog zum additiven RGB gibt es den CMYK-Farbraum, der dem Prinzip der subtraktiven Farbmischung folgt. Es werden Werte für die Farben Cyan, Magenta und Gelb (Yellow) angegeben. (1,1,1) ist demnach schwarz und (0,0,0) weiß.

In der Realität entspricht dieses Modell der Mischung von Farbe auf Papier. Deshalb wird es auch häufig bei Farbdruckern genutzt. Die Farben der Tinte in Tintenstrahldruckern sind deshalb ebenfalls Cyan, Magenta und Gelb.

Im Folgenden der Arbeit wird ausschließlich ein 24 Bit RGB-Farbraum verwendet. Das bedeutet pro Kanal 8 Bit, also Werte von 0 bis 255.

3. Die Brücke zwischen Bild und Ton

3.1 Überlegungen



Spektralanalyse eines Ausschnittes aus dem Intro des Liedes "Money" von Pink Floyd. Grün = Links; Rot = Rechts

Eine häufig verwendete Methode zur Analyse von Audio ist die der Spektralanalyse. Dabei werden die Anteile aller vorkommenden Frequenzen betrachtet. Man kann nun beispielsweise die Zeitachse vertikal, die Frequenz-Werte horizontal auftragen und jedem Punkt in diesem System einen Helligkeitswert entsprechend dem Frequenzanteil zuweisen. Somit hat man bereits ein Graustufen-Bild. Bei Stereo-Signalen liegen zwei Audio-Signale vor, also kann man hierfür z.B. zwei verschiedene Grundfarben benutzen, die addiert werden (siehe Bild).

Nun soll eine Audio-Datei so erzeugt werden, dass die Spektralanalyse ein vorgegebenes Bild ergibt. Um dies zu erreichen, ist es notwendig zuerst die Methode zu betrachten, mit der das Frequenzspektrum analysiert wird. Diese Methode wird im nächsten Abschnitt beschrieben.

3.2 Die Fouriertransformation

Im Bereich der Signalverarbeitung, z.B. für Audiotechnik, ist die Fouriertransformation essentiell. Sie bietet die Möglichkeit, Signale aller Art, die als Abtastwerte in Abhängigkeit von der Zeit angegeben sind, in die Darstellung {(Frequenzanteil, Amplitude, Phase)} umzuwandeln. Durch die inverse Fouriertransformation lassen sich die errechneten Werte wieder in die Ausgangsform konvertieren.

Ein digitaler Equalizer beispielsweise bekommt ein Audiosignal, welches er zunächst mithilfe der Fouriertransformation analysiert, dann die Frequenzanteile je nach Einstellung hebt oder senkt und zum Schluss wieder durch die inverse Transformation in ein kontinuierliches Signal überführt.

Im Folgenden wird die Fouriertransformation angelehnt an die Herleitung in Quelle 2 im Literaturverzeichnis erklärt. Der Beweis, wieso die Transformation funktioniert, würde den Umfang dieser Arbeit jedoch sprengen.

3.2.1 Mathematische Grundlagen

Definition: $w \in \mathbb{C}$ ist eine komplexe Zahl. Ist $w^n = 1$, so wird *w* als ***n*-te Einheitswurzel** bezeichnet. Gilt gleichzeitig $w^k \neq 1$ für alle $k \in \{1, \ldots, n-1\}$, so ist *w* **primitive *n*-te Einheitswurzel**.

Beispiel: *i* ist primitive 4-te Einheitswurzel, da $i^4 = 1$ und dies gleichzeitig für keine kleinere ganze Zahl als 4 gilt, die mindestens 1 ist.

Man kann sich *n*-te Einheitswurzeln folgendermaßen errechnen:

$w = \cos(k \cdot 2\pi / n) + i \sin(k \cdot 2\pi / n)$ mit $k \in \{0, \ldots n-1\}$

w ist primitive *n*-te Einheitswurzel wenn *k* teilerfremd zu *n* ist.

Eigenschafen:

1. Aus einer primitiven *n*-ten Einheitswurzel *w* lassen sich alle anderen Einheitswurzeln durch die Potenzen von *w* berechnen. Es gibt genau n verschiedene davon:

   $w^0, w^1, w^{2,} \ldots, w^{n-1}$

2. w^k mit $k \in \mathbb{Z}$ ist immer ebenfalls *n*-te Einheitswurzel:

   $(w^k)^n = w^{k \cdot n} = (w^n)^k = 1^k = 1$

3. Für gerade *n* und primitive *n*-te Einheitswurzeln *w* ist: $w^{n/2} = -1$, denn $(w^{n/2})^2 = w^n = 1$, also muss $w^{n/2}$ 2-te Einheitswurzel sein (entweder 1 oder -1).

   Da *w* allerdings primitiv ist, gilt $w^{n/2} \neq 1$. Deshalb muss $w^{n/2} = -1$ sein.

4. Ebenfalls für gerade *n* ist das Quadrat w^2 einer primitiven *n*-ten Einheitswurzel *w* primitive *n*/2-te Einheitswurzel, da

   $(w^2)^{n/2} = w^n = 1$ (es ist also sicher *n*/2-te Einheitswurzel)

   Wäre w^2 nicht primitiv, so würde ein $k \in \{1, \ldots, n/2 - 1\}$ existieren, für welches $(w^2)^k = w^{2k} = 1$, da *w* aber primitiv ist und 2*k* immer kleiner als *n* ist, gibt es kein solches k, also muss w^2 ebenfalls primitiv sein.

5. Wenn *w* eine primitive *n*-te Einheitswurzel ist, dann ist w^{-1} auch primitive *n*-te Einheitswurzel, denn

   $$(w^{-1})^n = w^{-n} = \frac{1}{w^n} = \frac{1}{1} = 1$$

   Wäre w^{-1} nicht primitiv, gäbe es ein $k \in \{1, \ldots, n-1\}$, für das gilt $(w^{-1})^k = w^{-k} = 1$.

   Allerdings müsste ebenfalls $w^k = 1$ sein, denn $w^k = \frac{1}{w^{-k}} = 1/1$. dies kann jedoch nicht sein, da *w* primitiv ist.

3.2.2 Diskrete Fouriertransformation (DFT)

Um die Fouriertransformation anzuwenden, muss zunächst eine sogenannte Fouriermatrix erstellt werden.

Dazu wird ein $n \in \mathbb{N}$ gewählt und mit der primitiven *n*-ten Einheitswurzel eine $n \times n$ -Matrix *F* gebildet:

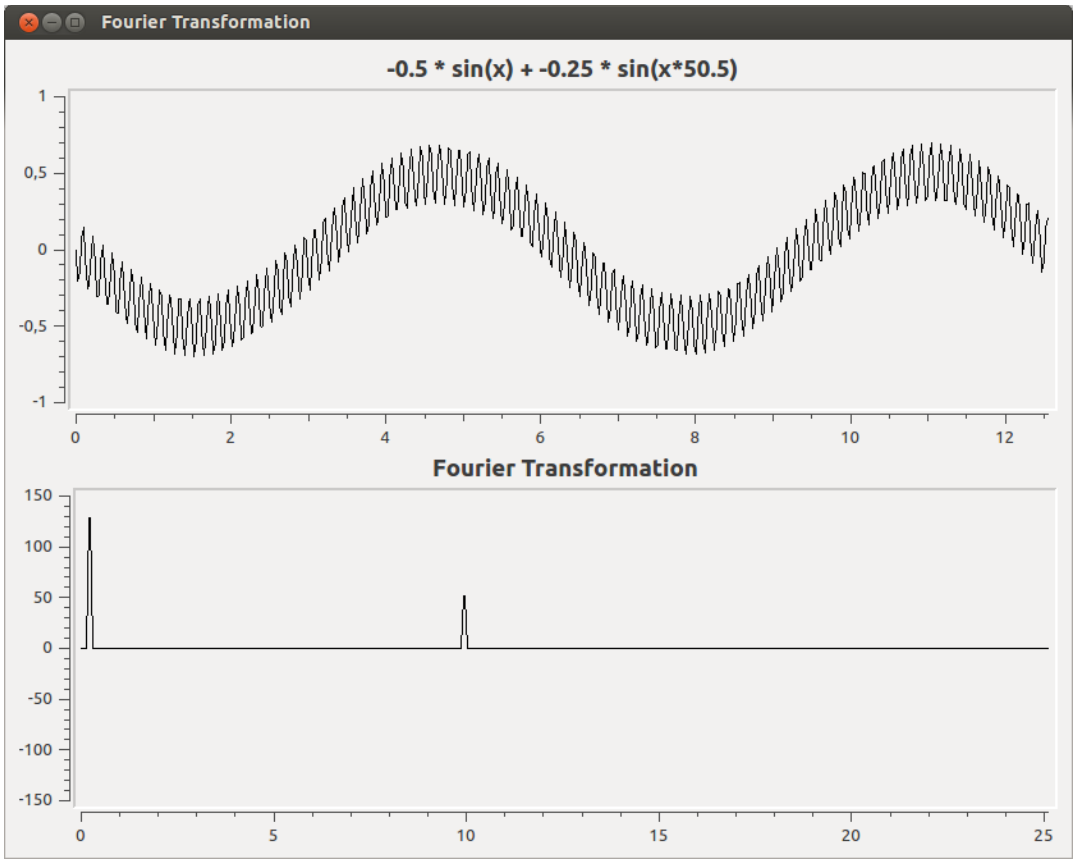
$F_{ij} = w^{ij}$ für alle $i, j \in \{0, \ldots, n-1\}$

Die Funktion $f : \mathbb{C}^n \to \mathbb{C}^n$ festgelegt als $f(a) = a \cdot F$ mit $a \in \mathbb{C}^n$, also einem Vektor mit *n* Komponenten aus $\mathbb{C}$, nennt man diskrete Fouriertransformation.

Die inverse Fouriermatrix ist definiert als $F_{ij}^{-1} = \frac{1}{n} \cdot w^{-i \cdot j}$. Folglich ist die Umkehrfunktion zur DFT: $f^{-1}(a) = a \cdot F^{-1}$. Dies wird als inverse Fouriertransformation bezeichnet.

Um also die DFT auf ein Signal anzuwenden, nimmt man n aufeinander folgende Abtastwerte daraus, erstellt die Fouriermatrix der Größe $n \times n$ und multipliziert den Vektor *a*, der die Ausgangswerte enthält, mit der Matrix. Das Ergebnis ist ein Vektor *b*, der die Frequenzanteile erhält. Um aus *b* wieder *a* zu erhalten, multipliziert man diesen mit der inversen Fouriermatrix.

3.2.2.1 Beispiel einer angewandten Fouriertransformation



Der obere Graph beschreibt zwei addierte negative Sinus Frequenzen. Der zweite Sinus-Teil hat eine höhere Frequenz (wegen x*50.5) und eine halb so große Amplitude (*-0.25) wie der erste.

Im unteren Graphen sind die beiden Frequenzen sehr schön als Ausschläge zu erkennen, von denen der zweite, der sich bei einem höheren x-Wert (Frequenz) befindet, genau halb so hoch ist wie die erste (Anteil der Frequenz, Amplitude).

3.2.3 Schnelle Fouriertransformation (FFT)

Ist *n* eine Potenz von 2, lässt sich die Fouriertransformation schneller durchführen als mit der DFT. Hierzu führt man die Multiplikationen mit der Fouriermatrix in einer bestimmten Reihenfolge aus und greift dabei auf bereits berechnete Ergebnisse unter Ausnutzung der oben genannten Eigenschafen von *n*-ten Einheitswurzeln zurück.

Es ist also der Vektor $a=[a_0 \dots a_{n-1}]$ gegeben. Aus ihm soll der Vektor $y=[y_0 \dots y_{n-1}]$ berechnet werden. Dieser wird zunächst in zwei ineinander verschränkte Vektoren y' und y'' der Länge $n/2$ aufgeteilt. y' enthält also alle Komponenten von y mit geradem Index und y'' alle mit ungeradem:

$$y=[y'_0 \; y''_1 \; \dots \; y'_{n/2-1} \; y''_{n/2-1}]$$

Für die folgenden Berechnungen gilt: $m=n/2$

Erst werden die Werte für y' berechnet:

$$y'_k=y_{2k}=\sum_{i=0}^{n-1} a_i w^{i\cdot 2k}$$

Diese Summe zerlegt man zunächst in zwei Teile:

$$y'_k=\sum_{i=0}^{n-1} a_i w^{i\cdot 2k}=\sum_{i=0}^{m-1} a_i w^{i\cdot 2k}+\sum_{i=0}^{m-1} a_{i+m} w^{(i+m)\cdot 2k}$$

Da $w^{nk}=1$ ist, kann folgende Vereinfachung in der zweiten Summe vorgenommen werden:

$$w^{(i+m)\cdot 2k}=w^{i\cdot 2k+m\cdot 2k}=w^{i\cdot 2k}\cdot w^{m\cdot 2k}=w^{i\cdot 2k}\cdot w^{n\cdot k}=w^{i\cdot 2k}$$

Somit können die beiden Summen wieder zusammen gefasst werden:

$$y'_k=\sum_{i=0}^{m-1} (a_i+a_{i+m}) w^{i\cdot 2k}$$

Mit der primitiven *m*-ten Einheitswurzel $v=w^2$ also:

$$y'_k=\sum_{i=0}^{m-1} (a_i+a_{i+m}) v^{i\cdot k}$$

Also ist y' die Fouriertransformation des Vektors $[a_i+a_{i+m}]_{i=0,\dots,m-1}$, der nur halb so lang ist wie *n*.

Die Berechnung von y'' verläuft ähnlich:

$$y''_k=y_{2k+1}=\sum_{i=0}^{n-1} a_i w^{i\cdot(2k+1)}$$

Wieder wird die Summe zerlegt:

$$y''_k=\sum_{i=0}^{n-1} a_i w^{i\cdot(2k+1)}=\sum_{i=0}^{m-1} a_i w^{i\cdot(2k+1)}+\sum_{i=0}^{m-1} a_{i+m} w^{(i+m)\cdot(2k+1)}$$

Da $w^{nk}=1$ und $w^{n/2}=-1$ ist, lässt sich hier folgende Vereinfachung durchführen:

$$w^{(i+m)(2k+1)}=w^{i2k+m2k+i+m}=w^{i\cdot 2k}\cdot w^{nk}\cdot w^{i}\cdot w^{n/2}=-w^{i}\cdot w^{i\cdot 2k}$$

Also in den Summen (mit $v=w^2$):

$$y''_k=\sum_{i=0}^{m-1} a_i w^i\cdot w^{i\cdot 2k}+\sum_{i=0}^{m-1} -a_{i+m} w^i\cdot w^{i\cdot 2k}=\sum_{i=0}^{m-1} w^i\cdot(a_i-a_{i+m})\cdot w^{i\cdot 2k}$$
$$=\sum_{i=0}^{m-1} w^i\cdot(a_i-a_{i+m})\cdot v^{i\cdot k}$$

Folglich ist y'' die Fouriertransformation des Vektors $w^i\cdot[a_i-a_{i+m}]_{i=0,\ldots,m-1}$, der ebenfalls nur halb so lang ist wie n.

Zusammengefasst teilt die FFT die Berechnung also in zwei verschiedene Fouriertransformationen von Vektoren halber Länge des ursprünglichen auf. Diese beiden Transformationen werden ebenfalls mit der FFT berechnet, so dass sich eine Rekursion ergibt. Hat der Vektor irgendwann nur noch eine Komponente, wird er unverändert zurückgegeben, da die 1×1 Fouriermatrix nur eine 1 enthält.

Damit dieses rekursive Verfahren funktionieren kann, muss also die Länge des Vektors immer wieder durch 2 teilbar sein, bis er schließlich nur noch eine Komponente besitzt. Es kommen für n also nur Potenzen von 2 in Frage.

Will man mit der FFT eine inverse Fouriertransformation ausführen, teilt man zuerst alle Komponenten des Ausgangsvektors a durch n und benutzt dann statt der primitiven n-ten Einheitswurzel w deren Potenz w^{-1}.

4. Umsetzung in ein Programm

Nach so viel Theorie nun zur Praxis: Es soll also ein Programm geschrieben werden, das ein Bild aus einer Datei liest, von oben nach unten auf alle Pixelreihen die inverse FFT anwendet und das Ergebnis dann in eine Audio-Datei schreibt.

Das Programm in dieser Arbeit ist ein reines Konsolenprogramm (also ohne grafische Oberfläche), das primär auf Linux-Plattformen laufen soll. Als Programmiersprache wird C++ mit den Libraries FreeImage (FreeImagePlus für C++) und libsndfile verwendet. Die Entwicklungsumgebung ist ein Ubuntu 12.10 64-Bit x86 Rechner mit g++ als Compiler, VIM als Editor und make. Es würde jedoch kein Problem darstellen, das Programm auch für Windows und Mac OS zu kompilieren, da die Libraries diese Plattformen ebenfalls unterstützen.

4.1 Implementierung

Die folgenden Codeausschnitte sind auf das Wesentliche gekürzt und dienen nur zur Veranschaulichung. Es werden keine Fehler abgefangen. Der komplette Programmcode befindet sich im Anhang.

4.1.1 FFT

Zunächst wird die Schnelle Fouriertransformation implementiert. Die Funktion `fft` bekommt als Parameter ein Array `y` der Länge `n` von komplexen Zahlen, die transformiert werden sollen und eine primitive `n`-te Einheitswurzel `w`.

```
void fft(complex<float> *y, int n, complex<float> w)
{
	if(n <= 1)	// wenn nur noch ein Wert bei der Rekursion übrig ist;
				kleiner als 1 macht einfach nur keinen Sinn
		return;

	int m = n / 2; // halbe Länge vorberechnet
	complex<float> *a, *b; // a ist y'; b ist y''
	complex<float> v = w*w;  // v = w^2 als primitive m-te Einheitswurzel
	int i; // Zählvariable

	// Reservieren des Speichers für y' und y''
	a = new complex<float>[m];
	b = new complex<float>[m];

	// Berechnung der Ausgangsvektoren, durch die y' und y'' berechnet
		werden
	for(i=0; i<m; i++)
	{
		a[i] = y[i] + y[i+m];
			// Für die Berechnung von y': a[i] + a[i+m]
		b[i] = pow(w, i) * (y[i] - y[i+m]);
			// Für die Berechnung von y'':	w^i * (a[i] - a[i-m])
	}

	fft(a, n/2, v); // => y'
	fft(b, n/2, v); // => y''

	// Verschränken von y' und y''
	for(i=0; i<m; i++)
	{
		y[2*i+0] = a[i];
		y[2*i+1] = b[i];
	}

	// Freigeben des Speichers
	delete[] a;
	delete[] b;
}
```

Diese Implementierung der FFT ist definitiv nicht die effizienteste. Es wäre beispielsweise möglich, anstatt die beiden Arrays `a` und `b` zu erstellen, direkt in dem bereits mitgegebenen Array `y` zu arbeiten, allerdings wäre der Code dann schwerer nachzuvollziehen.

Damit die Funktion in der Praxis einfach verwendet werden kann, wird noch eine Funktion erstellt, die statt der konkreten Länge `n` den Logarithmus zur Basis 2 von `n` als zweiten Parameter `ldn` bekommt und `n` dann daraus berechnet. Es ist also ausgeschlossen, dass `n` keine Potenz von 2 ist. Zudem wird die erste `n`-te Einheitswurzel der Rekursion automatisch berechnet. Der Parameter `inv` gibt an, ob eine inverse FFT durchgeführt werden soll.

```
void fft(complex<float> *v, unsigned int ldn, bool inv)
{
	int n = pow(2, (int)ldn); // n = 2^ldn
	complex<float> w(cos(2*M_PI/n), sin(2*M_PI/n));
			// w = primitive n-te Einheitswurzel
	int i;

	if(inv) // für inverse Transformation
	{
```

```
                w = pow(w, -1); // w^-1
                for(i=0; i<n; i++) // alle Werte durch n teilen
                        v[i] = v[i] * (float)(1.0 / n);
        }
        fft(v, n, w); // FFT ausführen
}
```

4.1.2 Umwandlung eines Bildes

Der Rest des Programms ist vergleichsweise simpel. Die Dateinamen werden als Parameter auf der Konsole mitgegeben, stehen also in argv[].

Als Erstes wird die Bild-Datei mit FreeImage geladen und eine WAV-Datei mit libsndfile zum Schreiben geöffnet.

```
int main(int argc, char *argv[])
{
        fipImage image; // Bild
        image.load(argv[1]); // Laden des Bildes


        int ldn = 10; // n ist also 2^10. Dieser Wert kann beliebig angepasst
                werden
        int n = pow(2, ldn); // n = 2^ldn
        complex<float> *v; // Array von komplexen Zahlen für FFT
        float *samples; // Array, das später in die Audio-Datei geschrieben
                wird
        RGBQUAD color; // zum auslesen von Pixelfarben
        SF_INFO out_info; // gibt diverse Parameter für die zu erstellende
                Datei an
        SNDFILE* out; // Ausgabedatei
        int x, y; // Zählervariablen
        int i;

        image.rescale(n / 2, n, FILTER_BICUBIC) // Skalieren des Bildes. Damit
                        die Nyquist-Frequenz nicht überschritten wird, nur
                        halb so breit wie n.

        out_info.frames = n*n; // Anzahl der Samples = n^2
        out_info.samplerate = 44100; // Gebräuchliche 44,1kHz
        out_info.channels = 1; // Mono
        out_info.sections = 0;
        out_info.seekable = 0;
        out_info.format = SF_ENDIAN_FILE | SF_FORMAT_PCM_16 | SF_FORMAT_WAV;
                        // Es soll eine WAV-Datei erstellt werden

        out = sf_open(argv[2], SFM_WRITE, &out_info); // Öffnen der Audio-Datei
```

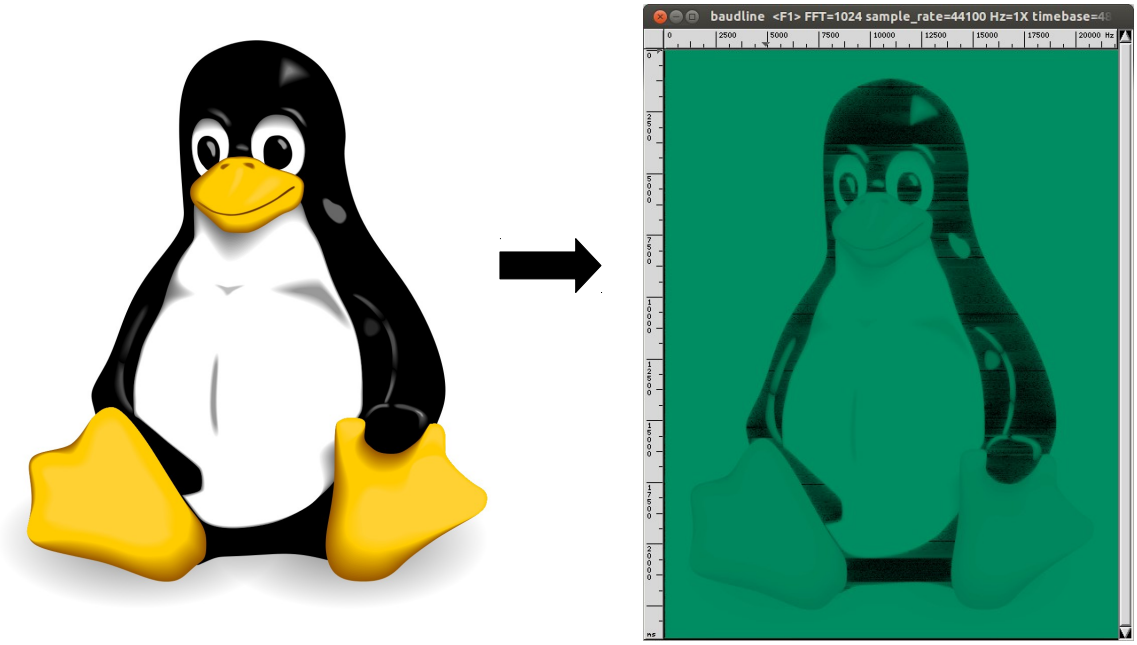
Die Vorbereitungen sind nun getroffen. Als Nächstes werden die Arrays `v` und `samples` initialisiert. Danach werden alle Pixelzeilen von oben bis unten (bei FreeImage ist y=0 am unteren Rand, deshalb wird `y` herab gezählt) abgearbeitet. Für jede Zeile wird `v` gefüllt und die FFT darauf angewandt. Zum Schluss werden die berechneten Werte in das `float`-Array `samples` und dieses in die Datei geschrieben. Sind alle Daten geschrieben, wird die Datei geschlossen.

```
        // Erstellen der Arrays
        v = new complex<float>[n];
        samples = new float[n];

        for(y=image.getHeight()-1; y>=0; y--) // Alle Pixelzeilen abarbeiten
        {
                for(i=0; i<n; i++) // Leeren des Arrays
                        v[i] = complex<float>(0.0, 0.0);

                for(x=0; x<image.getWidth(); x++) // Füllen des Arrays mit
                                                  Werten aus dem Bild
                {
                        image.getPixelColor(x, y, &color);
```

```
                    v[x] = complex<float>(0.0, ((float)(color.rgbRed +
                    color.rgbGreen + color.rgbBlue) / (255.0 * 3.0)));
                            // Durchschnitt der Grundfarben
            }

            fft(v, ldn, true); // Anwendung der FFT


            for(i=0; i<n; i++) // Schreiben der errechneten Werte in das
                                    float-Array
                    samples[i] = v[i].real();

            sf_writef_float(out, samples, n) // Schreiben in die Datei
    }

    sf_close(out); // Schließen der Datei

    return 0;
}
```

4.2 Test

Jetzt ist es an der Zeit, das Programm mit einem Bild zu testen, um zu sehen, ob das Verfahren wirklich funktioniert. Die erstellte Audio-Datei wird dann mit dem Programm „baudline“ analysiert.



Es werden zwar nur Helligkeitswerte statt der richtigen Farben übermittelt, trotzdem ist das Bild noch gut zu erkennen.

5. Schlusswort

Ein Kryptographie-Algorithmus arbeitet ähnlich wie das hier vorgestellte Verfahren. Dabei werden die zu verschlüsselnden Informationen in eine andere Form übergeführt, so dass sie später wieder (oftmals nur mit einem speziellen Schlüssel) rekonstruierbar sind. In der Mathematik gibt es ebenfalls derartige Fälle: Eine mathematische Funktion, die vollständig umkehrbar ist kann dazu verwendet werden, beliebige Fließkommawerte in vollkommen andere Werte umzuwandeln, aus denen die ursprünglichen Werte wieder errechenbar sind.

Es ist also möglich, jegliche Art von Information in einer beliebigen anderen Form darzustellen und aus dieser wieder zu rekonstruieren, sofern diese Form genügend Daten aufnehmen kann und die Methode, mit der man die Daten konvertiert auch umkehrbar ist.

6. LITERATURVERZEICHNIS

1. Downer, Jason: Quellcode des Programms „Enscribe“ http://coppercloudmusic.-com/enscribe/ (Stand 01.11.2012)
2. Lang, H.W. (2011): Schnelle Fouriertransformation (FFT) http://www.iti.fh-flensburg.de/lang/algorithmen/fft/fft.htm (Stand: 01.11.2012)
3. Pietzsch, Sylvia: Farbe und Farbräume http://www9.in.tum.de/seminare/hs.SS04.cvision/Vortragsthemen.html (Stand 01.11.2012)
4. Prof. Dr. Dr. Kollmeier, Birger: Physikalische, technische und medizinische Akustik http://medi.uni-oldenburg.de/html/teaching_download.html (Stand 01.11.2012)
5. Weisstein, Eric W.: Fourier Matrix aus „MathWorld--A Wolfram Web Resource.“ http://mathworld.wolfram.com/FourierMatrix.html (Stand 01.11.2012)
6. Weisstein, Eric W.: Nyquist Frequency aus „MathWorld--A Wolfram Web Resource.“ http://mathworld.wolfram.com/NyquistFrequency.html (Stand 01.11.2012)

7. Anhang

7.1 Inhalt der CD

Auf der beiliegenden CD befinden sich folgende Inhalte:

- Die komplette Seminararbeit als PDF
- PDF-Versionen der Internetquellen
- Das Programm zu dieser Arbeit als Sourcecode und kompilierte Executables

7.2 Quellcode des Programms

```
/*
 * audiocrypt 1.0
 * Teil der Seminararbeit "Umwandlung von Bildern in Audio-Signale"
 * von Florian Märkl
 *
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see <http://www.gnu.org/licenses/>.
 *
 */


#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <complex>

#include <FreeImagePlus.h>
#include <sndfile.h>

using namespace std;


void fft(complex<float> *v, unsigned int ldn, bool inv = false);

int main(int argc, char *argv[])
{
	if(argc != 3)
	{
		printf("usage: audiocrypt [input file] [output file]\n");
		exit(0);
	}

	fipImage image; // Bild
	if(image.load(argv[1]) == FALSE) // Laden des Bildes
	{
		printf("Failed to load input image\n");
		exit(1);
	}

	printf("Loaded Image %s (%dx%d)\n", argv[1], image.getWidth(),
			image.getHeight());
```

```
int ldn = 10; // n ist also 2^10. Dieser Wert kann beliebig angepasst werden
int n = pow(2, ldn); // n = 2^ldn
complex<float> *v; // Array von komplexen Zahlen für FFT
float *samples; // Array, das später in die Audio-Datei geschrieben wird
RGBQUAD color; // zum Auslesen von Pixelfarben
SF_INFO out_info; // gibt diverse Parameter für die zu erstellende Datei an
SNDFILE* out; // Ausgabedatei
int x, y; // Zählervariablen
int i;

if(image.rescale(n / 2, n, FILTER_BICUBIC) == FALSE) // Skalieren des Bildes.
        Damit die Nyquist-Frequenz nicht überschritten wird, nur halb so breit
        wie n.
{
        printf("Failed to scale image.\n");
        exit(1);
}

out_info.frames = n*n; // Anzahl der Samples = n^2
out_info.samplerate = 44100; // Gebräuchliche 44,1kHz
out_info.channels = 1; // Mono
out_info.sections = 0;
out_info.seekable = 0;
out_info.format       = SF_ENDIAN_FILE | SF_FORMAT_PCM_16 | SF_FORMAT_WAV;
                // Es soll eine WAV-Datei erstellt werden

out = sf_open(argv[2], SFM_WRITE, &out_info); // Öffnen der Audio-Datei

if(!out)
{
        printf("Failed to open output file %s for writing\n", argv[2]);
        exit(1);
}

printf("Opened %s for writing\n", argv[2]);

printf("Beginning to convert data (n=%d)\n", n);

// Erstellen der Arrays
v = new complex<float>[n];
samples = new float[n];

for(y=image.getHeight()-1; y>=0; y--) // Alle Pixelzeilen abarbeiten
{
        printf("\r%d%%", (int)((1.0 - ((float)y / (float)image.getHeight())) *
                100));

        for(i=0; i<n; i++) // Leeren des Arrays
                v[i] = complex<float>(0.0, 0.0);

        for(x=0; x<image.getWidth(); x++) // Füllen des Arrays mit Werten aus
                                                dem Bild
        {
                image.getPixelColor(x, y, &color);
                v[x] = complex<float>(0.0, ((float)(color.rgbRed + color.rgbGreen
                        + color.rgbBlue) / (255.0 * 3.0)));
                        // Durchschnitt der Grundfarben
        }

        fft(v, ldn, true); // Anwendung der FFT

        for(i=0; i<n; i++) // Schreiben der errechneten Werte in das float-Array
                samples[i] = v[i].real();

        if(sf_writef_float(out, samples, n) != n) // Schreiben in die Datei
        {
                printf("\nWriting failed.\n");
                exit(1);
        }
}
printf("\rFinished.\n");

sf_close(out); // Schließen der Datei
```

```
	return 0;
}

void fft(complex<float> *y, int n, complex<float> w)
{
	if(n <= 1) // wenn nur noch ein Wert bei der Rekursion übrig ist; kleiner als 1
			macht einfach nur keinen Sinn
		return;

	int m = n / 2; // halbe Länge vorberechnet
	complex<float> *a, *b; // a ist y'; b ist y''
	complex<float> v = w*w;  // v = w^2 als primitive m-te Einheitswurzel
	int i; // Zählvariable

	// Reservieren des Speichers für y' und y''
	a = new complex<float>[m];
	b = new complex<float>[m];

	// Berechnung der Ausgangsvektoren, durch die y' und y'' berechnet werden
	for(i=0; i<m; i++)
	{
		a[i] = y[i] + y[i+m]; // Für die Berechnung von y': a[i] + a[i+m]
		b[i] = pow(w, i) * (y[i] - y[i+m]);  // Für die Berechnung von y'':
							w^i * (a[i] - a[i-m])
	}

	fft(a, n/2, v); // => y'
	fft(b, n/2, v); // => y''

	// Verschränken von y' und y''
	for(i=0; i<m; i++)
	{
		y[2*i+0] = a[i];
		y[2*i+1] = b[i];
	}

	// Freigeben des Speichers
	delete[] a;
	delete[] b;
}

void fft(complex<float> *v, unsigned int ldn, bool inv)
{
	int n = pow(2, (int)ldn); // n = 2^ldn
	complex<float> w(cos(2*M_PI/n), sin(2*M_PI/n)); // w = primitive n-te
							Einheitswurzel
	int i;

	if(inv) // für inverse Transformation
	{
		w = pow(w, -1); // w^-1
		for(i=0; i<n; i++) // alle Werte durch n teilen
			v[i] = v[i] * (float)(1.0 / n);
	}

	fft(v, n, w); // FFT ausführen
}
```

7.3 Anleitung zum Kompilieren und Benutzen des Programms

Um das Programm auf einer Linux-Plattform zu kompilieren werden die Header und Shared Libraries der Bibliotheken FreeImage und libsndfile benötigt. Zudem werden der Compiler g++ und make vorausgesetzt.

Sind diese Bedingungen erfüllt, lässt sich das Programm mit folgendem Befehl im entsprechenden Verzeichnis kompilieren:

```
make
```

Es wird das Executable „audiocrypt“ erstellt, welches sich folgendermaßen auf der Konsole benutzen lässt:

```
./audiocrypt [Bilddatei] [Ausgabedatei]
```

Auf der CD befinden sich bereits fertige Executables für Linux (64 und 32Bit) und Windows (32Bit)

Ich erkläre hiermit, dass ich die Seminararbeit ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benützt habe.

…....., den ….....
Ort Datum

….....
Unterschrift des Verfassers