Wave-Datei-Analyse via Fast Fourier Transformation

Grundlagen der Wave-Datei-Analyse und Implementation der entsprechenden Algorithmen in Java

Inhaltsverzeichnis

1. Einführung

Wave-Dateien enthalten gesampelte Daten, die in bestimmten Zeitabständen gespeichert wurden. Eine Fourier-Transformation über diesen Daten verrät das Frequenz-Spektrum der zugrunde liegenden Wellen. Dies kann man effektiv über Fast Fourier Transformation (FFT) algorithmisieren...

Dieses Ziel soll mit der Programmiersprache Java[1] erreicht werden, da sie unabhängig von der Plattform, flexibel und weit verbreitet ist. Außerdem ist so eine Integration in das Musitech-Projekt möglich.

1 Java™ 2 SDK, Standard Edition (1.3.1_01) (http://www.java.sun.com/)

Diese Ausarbeitung ist in drei Teile eingeteilt: Vorüberlegungen und Fourier-Transformation, Fast Fourier Transformation und Wave-Datei-Analyse.

In dem Kapitel über Vorüberlegungen und Fourier-Transformation sollen die grundlegenden Voraussetzungen mathematischen Überlegungen dargelegt werden.

In dem Kapitel Fast Fourier Transformation soll die Fast Fourier Transformation und ihre algorithmische Realisierung dargelegt und erläutert werden.

In dem Kapitel Wave-Datei-Analyse soll die Fast Fourier Transformation konkret auf Wave-Dateien angewendet werden. Höhepunkt dieses Kapitel ist ein fertiges Stand-Alone-Programm, das eine Wave-Datei einliest und die Analyse graphisch auf dem Bildschirm darstellt.

2. Vorüberlegungen und Fourier-Transformation

2.1. Format und Wesen von Wave-Dateien

Wave-Dateien enthalten gesampelte Daten, wobei die einzelnen Werte zwischen -128 und 127 (bei 8 Bit) bzw. -32768 und 32767 (bei 16 Bit) liegen. Zwischen den einzelnen Werten liegen konstante Abstände, die nach folgender Formel errechenbar sind:

$$\Delta t = \frac{1}{SamplingRate} \tag{1}$$

Die gesampelten Daten liegen pro Kanal je in sortierter Reihenfolge in einem `int[] data` vor. Dazu gibt es ein `float samplingRate`. Die Klasse `AudioData.java` bietet diese Daten praktisch an.

Letztlich bleibt noch zu bemerken, dass eine endliche Anzahl von Daten, jedoch in großen Mengen, vorhanden sind. Pro aufgezeichneter Sekunde sind in der Regel zwischen 8000 und 44000 Werte gespeichert.

Diese Vorüberlegungen werden zu späterer Zeit gebraucht.

2.2. Grundlagen der Fourier-Transformation

Wie Fourier 1807 gezeigt hat[2], läßt sich jede mögliche periodische Schwingung auf einer Periode T[3] als eine unendliche Summe von Sinus- und Kosinus-Funktionen darstellen:

$$\begin{aligned} f(t) &= \sum_{k=0}^{\infty} \left(A_k \cos \omega_k t + B_k \sin \omega_k t\right) \\ \text{mit } \omega_k &= \frac{2\pi k}{T} \\ \text{und } B_0 &= 0 \end{aligned} \tag{2}$$

Das Wichtigste ist es nun, die Koeffizienten A_k und B_k von (2) zu berechnen, da sie die Funktion $f(t)$ vollständig festlegt. Dies geht, wie man durch arithmetische Operationen zeigen kann[4], so:

2 Sydney VisLab: *„1. The Fourier Series: Background“*, http://www.vislab.usyd.edu.au/CP3/Four1/node2.html, 30. Juni 2002.

3 Eine Periode ist der Zeitraum, in der sich eine Schwingung komplett wiederholt. Wenn z.B. zwei Schwingungen mit 50 Hz und 51 Hz sich überlagern ist diese Periode nicht etwa bei $\frac{1}{50}$ Min. oder $\frac{1}{51}$ Min., sondern bei 1 Min.

4 Tilman Butz: *„Fouriertransformation für Fußgänger“*, Stuttgart und Leipzig 2000, S. 18-20.

$$A_k = \frac{2}{T} \int_{-T/2}^{+T/2} f(t) \cos \omega_k t \, dt \text{ für } k \neq 0$$

$$A_0 = \frac{1}{T} \int_{-T/2}^{+T/2} f(t) \, dt \tag{3}$$

$$B_k = \frac{2}{T} \int_{-T/2}^{+T/2} f(t) \sin \omega_k t \, dt \text{ für alle } k$$

Mit der Formel (3) können wir nun prinzipiell die einzelnen Koeffizienten A_k und B_k berechnen. Diese sind schon das Frequenz-Spektrum der Schwingung, die durch $f(t)$ beschrieben wird.

Die Euler'sche Identität[5] läßt uns Sinus und Kosinus auch im komplexen Fall fassen[6]:

$$e^{i \alpha t} = \cos \alpha t + i \sin \alpha t \tag{4}$$

Wir können nun unsere Formel (2) komplex notieren:

$$f(t) = A_0 + \sum_{k=-\infty}^{\infty} C_k e^{i \omega_k t}$$

$$\text{mit } \omega_k = \frac{2 \pi k}{T} \tag{5}$$

$$\text{und } C_0 = A_0, C_k = \frac{A_k - i B_k}{2}, C_{-k} = \frac{A_k + i B_k}{2}$$

Natürlich lassen sich dann auch die Koeffizienten C_k komplex formulieren:

$$C_k = \frac{1}{T} \int_{-T/2}^{+T/2} f(t) e^{-i \omega_k t} \, dt \text{ für } -\infty \leq k \leq +\infty \tag{6}$$

Mit diesen Formeln haben wir jetzt also die Möglichkeit, die Frequenz-Spektren reeller Schwingungen (Formel (3)) sowie komplexer Schwingungen (Formel (6)) zu errechnen. Diese Formeln sind jedoch auf kontinuierlichen Funktionen definiert und zudem mit simplen Algorithmen nicht so einfach zu berechnen, es braucht schon Computer Algebra Systeme wie Mathematica[7], um diese Integrale im allgemeinen Fall berechnen zu können. Von zeitlicher Effizienz ist man hier natürlich noch weit entfernt.

2.3. Grundlagen der Fourier-Transformation auf gesampelten Daten

Die bisher betrachteten Formeln (3) und (6) beziehen sich auf kontinuierliche Funktionen. Bei Wave-Dateien haben wir jedoch nur „Stichproben“, die Samples, die gespeichert sind. Im Folgenden sollen Sie als $\{f_k\}$ bezeichnet werden. Das Frequenz-Spektrum soll mit $\{F_j\}$ bezeichnet werden. N ist dabei die Anzahl der Daten. Umformuliert lautet die Formel (6) für das Spektrum $\{F_j\}$ so[8]:

5 Wird in so ziemlich jedem Analysis I-Buch behandelt, exemplarisch sei hier auf Peter Meyer-Nieberg: *„Analysis – Vorlesung SS 2000“*, Uni Osnabrück, 2000 verwiesen.

6 Wem komplexe Zahlen unbekannt sind, dem sei die vorherige Lektüre des Anhangs über komplexe Zahlen empfohlen.

7 http://www.wolfram.com/

8 Tilman Butz: *„Fouriertransformation für Fußgänger“*, Stuttgart und Leipzig 2000, S. 103.

$$F_j = \frac{1}{N} \sum_{k=0}^{N-1} f_k W_N^{-k\,j} \qquad \text{mit } W_N = e^{\frac{2\pi i}{N}} \tag{7}$$

Diese endliche Summe kann man nun einfach mit simplen Algorithmen errechnen, indem man eine `for`-Schleife für jedes *j* macht, die von 0 bis *N*-1 läuft. In jedem Durchlauf der Schleife müsste dann nur ein Sample-Wert mit $e^{\frac{-2\pi i k j}{N}}$ multipliziert werden. Diese einzelnen Werte können in konstanter Zeit errechnet werden.

Würde man diese Formel also direkt in einen Algorithmus umformulieren, würde man feststellen, dass er eine Laufzeit[9] von O(N^2) hat (für jedes der *N j*'s müssen wir *N k*'s durchlaufen).

Daten	***Laufzeit***
256	65536
512	262144
1024	1048576
2048	4194304

Überlegungen von Cooley und Tuckey besagen, dass bei einer Fourier-Reihe der Länge 1 das Frequenzspektrum $F_0 = f_0$ ist, und dass man aus zwei Transformationen gleichlanger Teil-Fourier-Reihen in einem Arbeitsgang (linearer Zeit) eine Gesamt-Transformation machen kann. Wenn man also Daten hat, deren Anzahl *N* eine Zweier-Potenz ist, kann man einen Algorithmus der Laufzeit O($N \log_2 N$) schreiben.

Daten	***Laufzeit***
256	2048
512	4608
1024	10240
2048	22528

Wie man sieht, ist die Laufzeit ungleich kürzer als bei dem oben nahe gelegten O(N^2)-Algorithmus. Diese Idee führt zur Fast Fourier Transformation.

3. Fast Fourier Transformation

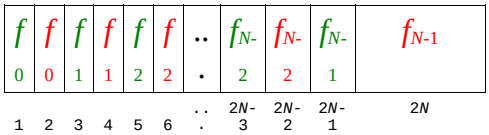
3.1. `public static void four1(double data[], int nn, int isign)`

Der zentrale Algorithmus einer Fast Fourier Transformation ist der, in dem die einzelnen eingegebenen Sample-Daten in Gruppen von schon transformierten Daten überführt werden. Dieser Algorithmus ist in `FastFourierTransformation.java` die Methode `public static void four1(double data[], int nn, int isign)`, die aus der Sammlung Numerical Recipies stammt[10] und von mir nach Java übersetzt wurde. Er ist in zwei Teile unterteilt, der erste sortiert die Daten um („Bit reversal“), während der zweite die eigentliche Fourier-Transformation übernimmt.

Die Daten werden wie folgt in den Algorithmus übergeben:

9 Das sogenannte O-Kalkül ist eine Methode, Laufzeiten von Programmen unabhängig von dem konkreten Computer-System abzuschätzen. Hierbei wird die Abhängigkeit der Laufzeit von den eingegebenen Daten deutlich. Siehe auch: Oliver Vornberger, Olaf Müller und Ralf Kunze: *„Algorithmen – Vorlesung im WS 2001/2002“*, Osnabrück, 2001, S. 53ff.

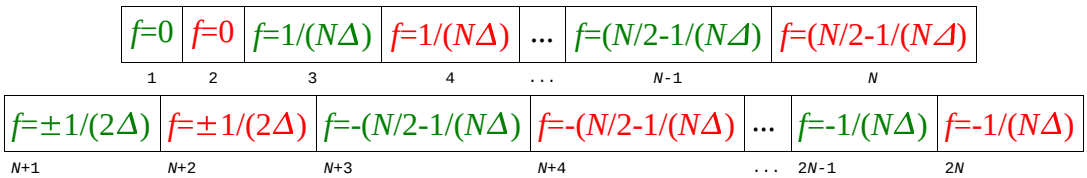
10 Numerical Recipies Software: *„NUMERICAL RECIPIES IN C: THE ART OF SCIENTIFIC COMPUTING“*, Cambridge, 1992, http://www.nr.com/.



Hierbei ist f_0 der reelle Teil und f_0 der imaginäre Teil der komplexen Zahl f_0.

Die Parameter der Methode `four1` sind: `nn` ist die Anzahl der Daten *N*, `isign=1` bedeutet, dass eine Fast Fourier Transformation gemacht werden soll und `isign=-1` bedeutet, dass eine inverse Fast Fourier Transformation gemacht werden soll, die Umkehrfunktion also.

Die Daten werden im Feld `data[]` zurückgegeben, und zwar in folgendem Format:

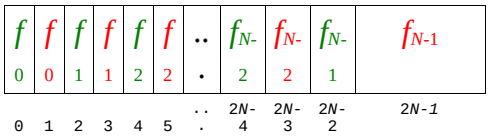


Hierbei ist wieder f_0 der reelle Teil und f_0 der imaginäre Teil der komplexen Zahl f_0.

Für die Methode `four1` selber, siehe Anhang.

3.2. `public static void complexFFT(double data[])` und `public static void complexFFTinv(double data[])`

Aufmerksame Zeitgenossen haben sicherlich schon bemerkt, dass der Aufruf von `four1` noch sehr an C-Gepflogenheiten erinnert, indem z.B. zusätzlich die Datengröße angegeben muss, die man in Java ja durch `data.length` zuverlässig herausfinden kann, und dass man zusätzlich mit einer Variablen auswählen muss, ob es sich um einer Fast Fourier Transformation oder um eine inverse Fast Fourier Transformation handelt. Zuletzt ist es noch unüblich in Java, dass Felder bei dem Index 1 beginnen, in der Regel beginnen sie hier schon beim Index 0. Hierzu habe ich zwei Methoden geschrieben, die die Methode `four1` ummanteln. Die Daten beginnen jetzt beim Index 0 und die anderen Parameter werden automatisch gesetzt. `comlexFFT` führt eine komplexe Fast Fourier Transformation durch und `complexFFTinv` eine komplexe inverse Fast Fourier Transformation. Die Daten werden im Prinzip wie bei `four1` übergeben, nur dass hier der Index mit 0 beginnt:



Hierbei ist f_0 der reelle Teil und f_0 der imaginäre Teil der komplexen Zahl f_0.

Die zurückgegebenen Daten sind analog formatiert.

Für die Methode `complexFFT` und `complexFFTinv` selber, siehe Anhang.

3.3. Reelle Fast Fourier Transformation

Eine reelle Fast Fourier Transformation ist nichts anderes als eine komplexe Fast Fourier Transformation, wo alle imaginären Teile der f_n=0. Demnach ist der Algorithmus für die komplexe Fast Fourier Transformation ausreichend für die Behandlung reeller Fast Fourier Transformationen. Die Ergebnisse sind in diesem Fall jedoch komplex, wobei hier noch eine Phasenverschiebung mit eingebaut ist. Diese kann man aber wieder raus rechnen, indem man den Betrag der komplexen Zahlen nimmt:

$$|z| = \sqrt{\Re(z)^2 + \Im(z)^2} \tag{8}$$

Zudem gilt noch: $F_n=F_{N-n}$* (komplex konjugiert)[11]. Deshalb braucht nur das positive Frequenzspektrum gespeichert zu werden. Somit können die komplexen F_n in demselben Arbeitsspeicher gespeichert werden, in dem die reellen f_n eingegeben wurden. Zuletzt ist es noch möglich, zwei reelle FFT in dem Algorithmus für komplexe FFT gleichzeitig durchzuführen (`twofft()`). Somit kann man also genauso effektiv reelle Fast Fourier Transformationen durchführen wie komplexe.

3.4. `public static void realFFT(double data[])` und `public static void realFFT(double data[])`

Auch für die reelle Fast Fourier Transformation gibt es ummantelnde Methoden. Das Prinzip ist dasselbe wie bei `complexFFT` und `complexFFTinv`, wichtig wäre nur noch, darauf hinzuweisen, wie die Daten formatiert sein müssen. Der Input ist folgendermaßen:

f_0	f_1	...	f_{N-2}	f_{N-1}
0	1	...	N-2	N-1

Der Output ist wie folgt formatiert (für f=0 und $f=\pm 1/(2\Delta)$ ist der imaginäre Teil gleich 0):

f=0	$f=\pm 1/(2\Delta)$	$f=1/(N\Delta)$	$f=1/(N\Delta)$	...	$f=(N/2-1)/(N\Delta)$	$f=(N/2-1)/(N\Delta)$
0	1	2	3	...	N-2	N-1

Hierbei ist f_0 der reelle Teil und f_0 der imaginäre Teil der komplexen Zahl f_0.

3.5. Zusammenfassende Dokumentation der Klasse `FastFourierTransformation.java`

3.5.1. `realFFT`

```
public static void realFFT(double[]data)
```

Performs a Discrete Fast Fourier Transformation of the real `data[]`. Please note that `data[]` must have the length of a power of 2, otherwise a `IllegalArgumentException` is thrown.

The returned data is organizes as follows:
`data[0]`: real part at f=0 (imaginary part=0)
`data[1]`: real part at f=+/- 1/(2*Delta) (combined, imaginary part=0)
`data[2]`: real part at f=1/(N*Delta)
`data[3]`: imaginary part at f=1/(N*Delta)
...
`data[2*i]`: real part at f=i/(N*Delta)
`data[2*i+1]`: imaginary part at f=i/(N*Delta)
...
`data[N-2]`: real part at f=(N/2-1)/(N*Delta)
`data[N-1]`: imaginary part at f=(N/2-1)/(N*Delta)

The other values (f=-(N/2-1)/(N*Delta) .. -1/(N*Delta)) can be calculated using the following symmetry: `F[N] = F[N-n]*` (complex conjugation).

Please note that the data is not normalized - you need to multiply each value with 1/N. For further information of this bug in the "Numerical Recipes" algorithm see *Tilman Butz: "Fouriertransformation für Fußgänger", Stuttgart, Leipzig, 2000, p. 103.*

Parameters:
`data[]` - the real data array

Throws:
`java.lang.IllegalArgumentException` - if `data[]` does not have the length of a power of 2.

3.5.2. realFFTinv

```
public static void realFFTinv(double[]data)
```

Performs an inverse Discrete Fast Fourier Transformation of the real `data[]`. Please note that `data[]` must have the

11 Tilman Butz: *„Fouriertransformation für Fußgänger“*, Stuttgart und Leipzig 2000, S. 26-27.

length of a power of 2, otherwise a `IllegalArgumentException` is thrown.

Parameters:
`data[]` - the real data array

Throws:
`java.lang.IllegalArgumentException` - if `data[]` does not have the length of a power of 2.

3.5.3. complexFFT

`public static void` **`complexFFT`**`(double[]data)`

Performs a Discrete Fast Fourier Transformation of the complex `data[]`. Please note that `data[]` must have the length of a power of 2, otherwise a `IllegalArgumentException` is thrown.

The returned data is organized as follows:
`data[0]`: real part at f=0
`data[1]`: imaginary part at f=0
`data[2]`: real part at f=1/(N*Delta)
`data[3]`: imaginary part at f=1/(N*Delta)
...
`data[2*i]`: real part at f=i/(N*Delta)
`data[2*i+1]`: imaginary part at f=i/(N*Delta)
...
`data[N-2]`: real part at f=(N/2-1)/(N*Delta)
`data[N-1]`: imaginary part at f=(N/2-1)/(N*Delta)
`data[N]`: real part at f=+/- 1/(2*Delta) (combined)
`data[N+1]`: imaginary part at f=+/- 1/(2*Delta)
`data[N+2]`: real part at f=-(N/2-1)/(N*Delta)
`data[N+3]`: imaginary part at f=-(N/2-1)/(N*Delta)
...
`data[N+2*i]`: real part at f=-(N/2-i)/(N*Delta)
`data[N+2*i+1]`: imaginary part at f=-(N/2-i)/(N*Delta)
...
`data[2*N-2]`: real part at f=-1/(N*Delta)
`data[2*N-1]`: imaginary part at f=-1/(N*Delta) .

Please note that the data is not normalized - you need to multiply each value with 1/N. For further information of this bug in the "Numerical Recipes" algorithm see *Tilman Butz: "Fouriertransformation für Fußgänger", Stuttgart, Leipzig, 2000, p. 103*.

Parameters:
`data[]` - the complex data array - the complex part of each number follows directly after the real part.

Throws:
`java.lang.IllegalArgumentException` - if `data[]` does not have the length of a power of 2.

3.5.4. complexFFTinv

`public static void` **`complexFFTinv`**`(double[]data)`

Performs an inverse Discrete Fast Fourier Transformation of the complex `data[]`. Please note that `data[]` must have the length of a power of 2, otherwise a `IllegalArgumentException` is thrown. This data is not normalized, you need to multiply it with `1/N`.

The returned data is organized as follows:
`data[0]`: real part at t=0
`data[1]`: imaginary part at t=0
...
`data[2*i]`: real part at t=i*Delta
`data[2*i+1]`: imaginary part at t=i*Delta
...
`data[2*N-2]`: real part at t=(N-1)*Delta
`data[2*N-1]`: imaginary part at t=-(N*1)*Delta .

Parameters:
`data[]` - the complex data array - the complex part of each number follows directly after the real part.

Throws:
`java.lang.IllegalArgumentException` - if `data[]` does not have the length of a power of 2.

3.5.5. isPowerOf2

```
public static boolean isPowerOf2(longi)
```

Checks recursivly, if i is an integer power of 2.

Parameters:
`i` - the long integer to be checked

3.5.6. four1

```
public static void four1(double[]data,
                         intnn,
                         intisign)
```

Discrete Fourier transformation, as described in *NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5), pp 507 ff.* You should not call this method but the capsulating methods `realFFT`, `realFFTinv`, `complexFFT` and `complexFFTinv`.

Original comment:
Replaces `data[1..2*nn]` by its discrete Fourier transform, if `isign` is input as `1`; or replaces `data[1..2*nn]` by `nn` times its inverse discrete Fourier transform, if `isign` is input as `-1`. `data` is a complex array of length `nn` or, equivalently, a real array of length `2*nn`. `nn` MUST be an integer power of 2 (this is not checked for!).

Please note that the data for `isign = 1` is not normalized - you need to multiply each value with 1/N. For further information of this bug in the "Numerical Recipes" algorithm see *Tilman Butz: "Fouriertransformation für Fußgänger", Stuttgart, Leipzig, 2000, p. 103.*

Parameters:
`data[]` - the data
`nn` - data size
`isign` - `1`: discrete Fourier transformation, `-1`: inverse descrete Fourier transformation.

3.5.7. twofft

```
public static void twofft(double[]data1,
                          double[]data2,
                          double[]fft1,
                          double[]fft2,
                          intn)
```

Discrete Fourier transformation of two real arrays simultaniously, as described in *NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5), pp 511 ff.* You should not call this method but the capsulating methods `realFFT`, `realFFTinv`, `complexFFT` and `complexFFTinv`.

Original comment:
Given two real input arrays `data1[1..n]` and `data2[1..n]`, this routine calls `four1` and returns two complex output arrays, `fft1[1..2n]` and `fft2[1..2n]`, each of complex length `n` (i.e., real length `2*n`), which contain the discrete Fourier transforms of the respective `data` arrays. `n` MUST be an integer power of 2.

Please note that the data for `isign = 1` is not normalized - you need to multiply each value with 1/N. For further information of this bug in the "Numerical Recipes" algorithm see *Tilman Butz: "Fouriertransformation für Fußgänger", Stuttgart, Leipzig, 2000, p. 103.*

Parameters:
`data1[]` - the data of the first function
`data2[]` - the data of the second function
`fft1[]` - the FFT of the first function
`fft2[]` - the FFT of the second function
`n` - data size

3.5.8. realft

```
public static void realft(double[]data,
                          intn,
                          intisign)
```

Discrete Fourier transformation of one real array, as described in *NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5), pp 513 ff.* You should not call this method but the capsulating methods `realFFT`, `realFFTinv`, `complexFFT` and `complexFFTinv`.

Original comment:

Calculates the Fourier transform of a set of `n` real-valued data points. Replaces the data (which is stored in array `data[1..n]` by the positive frequency half of its complex Fourier transform. The real-valued first and last components of the complex transform are returned as elements `data[1]` and `data[2]`, respectively. `n` must be a power of 2. This routine also calculates the inverse transform of a complex data array if it is the transform of real data. (Result in this case must be multiplied by 2/n.)

Please note that the data for `isign = 1` is not normalized - you need to multiply each value with 1/N. For further information of this bug in the "Numerical Recipes" algorithm see *Tilman Butz: "Fouriertransformation für Fußgänger", Stuttgart, Leipzig, 2000, p. 103*.

Parameters:

`data[]` - the data
`n` - data size
`isign` - `1`: discrete Fourier transformation, `-1`: inverse descrete Fourier transformation.

4. Wave-Datei-Analyse

4.1. Daten-Analyse

Jetzt kommen die Vorüberlegungen von oben wieder ins Spiel. Ein `int` in ein `double` zu überführen ist durch implizite Typumwandlung kein Problem. Man braucht jetzt nur noch einen Teil des `int[]`, dessen Länge eine Zweier-Potenz ist, in ein neues double[] zu kopieren, durch den Fast Fourier Transformation Algorithmus schicken, die Ergebnisse interpretieren und zurückgeben. Die Ergebnisse sollen in einem zweidimensionalen Feld gespeichert sein, in dem immer eine Frequenz mit ihrer Intensität ist. Die Formel für die Frequenzen ist, wie schon öfters erwähnt:

$$F_i = \frac{i}{N\,\Delta} \quad \text{mit } \Delta = \frac{1}{SamplingRate} \tag{9}$$

Diese Aufgabe wird von der Methode `public static double[][] analyseWaveData(int[] data, int start, int end, float samplingRate)` in der Datei `AnalyseWaveFile.java` übernommen. Ihr wird ein `int[]` mit Sample-Daten übergeben, ein Index des ersten Samples und der Index nach dem letzten Sample (so dass `end-start` eine Zweierpotenz ist) und die Sampling-Rate der Daten. Zurückgegeben wird ein `double[][]`, das $\frac{N}{2}+1$ Paare von Daten enthält, für jedes Paar im Index 0 die Frequenz und im Index 1 die dazugehörige Intensität. Mit anderen Worten:

`double[i][0]`=Frequenz
`double[i][1]`=Intensität bei Frequenz
mit `i`=0... $\frac{N}{2}+1$

Der Code der Methode `analyseWaveData` ist im Anhang. Die Abfragen am Anfang der Methode dienen dazu, sie idiotensicher zu machen, dass selbst ein GAU sie problemlos benutzen kann.

4.2. Datei-Analyse

Um eine Wave-Datei zu analysieren braucht man nun nur noch die Samples aus einer Wave-Datei auszulesen, sie in die `analyseWaveData`-Methode zu geben und die Analyse-Ergebnisse auszugeben. Hierzu eignet sich die Klasse `AudioData.java`. Wenn man eine neue Instanz dieser Klasse erzeugt, kann man im Konstruktur ein `java.io.File` mitgeben, aus dem die Daten ausgelesen werden.

Das Resultat ist die Methode `public static double[][] analyseWaveFile(String fname, double start_sec, int lengthFFT, boolean leftChannel)`. Hier wird die Startposition in Sekunden anstelle von einem absoluten Index gefordert, was praktischer ist

für die Anwender ist. Da man bei manchen Dateien noch zwischen den Kanälen unterscheiden muss, ist noch der Parameter `leftChannel` dabei, ist er wahr, wird der linke Kanal genommen, ist er unwahr, wird der rechte Kanal genommen.

4.3. Visuelle Darstellung der Ergebnisse

Will man die Ergebnisse auf dem Bildschirm darstellen, bietet sich `java.awt.*` an. Hierzu gibt es die Klasse `FrequencyWindow.java`, die ein simples Fenster ist, auf dem die Ergebnisse wie auf einem Koordinatensystem dargestellt werden. In dieser Klasse ist die Methode `public void paint(Graphics g)` das eigentlich interessante, in ihr wird das eigentliche Zeichnen übernommen.

Bevor man dies machen kann, ist jedoch die Daten anzeigen kann, ist es zu empfehlen, eine logarithmische Skala zu wählen. Während die errechneten Daten die absoluten Intensitäten sind, denken wir Menschen in dB, was eine logarithmische Skala zu den absoluten Intensitäten ist. Zugleich sollte man noch eine Normierung vornehmen. Der dafür notwendige Code ist dieser (`d[][]` ist das Feld mit den berechneten Daten):

```
// logarithmic plot of data gets a better graph
for (int i = 0; i < d.length; i++) {
  d[i][1] = Math.log(d[i][1] / (double)fft_length);
} // for
```

Dies alles ist in der Methode `public static main(String[] argv)` der Klasse `WaveFileAnalysis.java` umgesetzt, man ruft sie auf, indem man

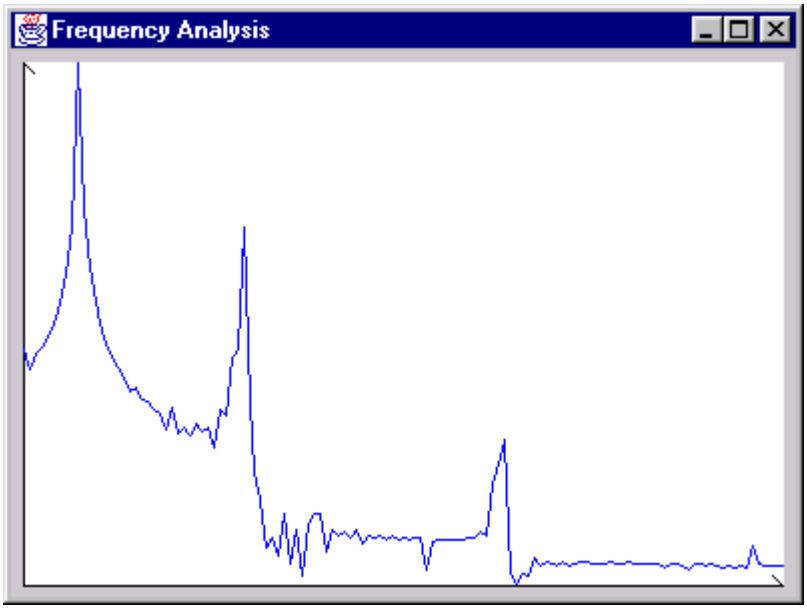
```
java WaveFileAnalysis dateiname start_sekunde länge_der_fft
```

eingibt, also z.B.:

```
java WaveFileAnalysis Ding.wav 0.2 256
```

Wichtig zu beachten: Die Benennung des Dateinamens folgt den Java-Konventionen und nicht den Windows-Konventionen.

Als Ergebnis der oben genannten Befehlszeile mit der `Ding.wav`, die mit Windows mitgeliefert wird, liefert folgendes Fenster:



5. Anhang

5.1. Java-Dateien

5.1.1. `FastFourierTransformation.java`

```
/********************** FastFourierTransformation.java **********************/
/* Datum: 12. Januar 2002
 * Autor: Christian Datzko
 * Copyright: Christian Datzko, 2002
 * E-Mail-Adresse: datzko@t-online.de
 * Programmiersprache und -version: Java(TM) 2 SDK, Standard Edition (1.3.1_01)
 */

/**
 * Basic routines for complex and real Fast Fourier Transformation. It is
 * recommended to use the "safer" routines <CODE>realFFT</CODE>,
 * <CODE>realFFTinv</CODE>, <CODE>complexFFT</CODE> and
 * <CODE>complexFFTinv</CODE>. The basic algorithms come from <I>NUMERICAL
 * RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)</I>.
 * @author Christian Datzko
 * @version 1.0
 */

public class FastFourierTransformation {

  /**
   * Performs a Discrete Fast Fourier Transformation of the real
   * <CODE>data[]</CODE>.
   * Please note that <CODE>data[]</CODE> must have the length of a power of
   * 2, otherwise a <CODE>IllegalArgumentException</CODE> is thrown.<P>
   * The returned data is organizes as follows:<BR>
   * <CODE>data[0]</CODE>: real part at <CODE><I>f</I>=0</CODE> (imaginary
   * part=0)<BR>
   * <CODE>data[1]</CODE>: real part at <CODE><I>f</I>=+/- 1/(2*Delta)</CODE>
   * (combined, imaginary part=0)<BR>
   * <CODE>data[2]</CODE>: real part at <CODE><I>f</I>=1/(N*Delta)</CODE><BR>
   * <CODE>data[3]</CODE>: imaginary part at <CODE><I>f</I>=1/(N*Delta)</CODE>
   * <BR>
   * ...<BR>
   * <CODE>data[2*i]</CODE>: real part at <CODE><I>f</I>=i/(N*Delta)</CODE><BR>
   * <CODE>data[2*i+1]</CODE>: imaginary part at <CODE><I>f</I>=i/(N*Delta)
   * </CODE><BR>
   * ...<BR>
   * <CODE>data[N-2]</CODE>: real part at <CODE><I>f</I>=(N/2-1)/(N*Delta)
   * </CODE><BR>
   * <CODE>data[N-1]</CODE>: imaginary part at <CODE><I>f</I>=(N/2-1)/(N*Delta)
   * </CODE><P>
   * The other values (<CODE><I>f</I>=-(N/2-1)/(N*Delta)</CODE> ..
   * <CODE>-1/(N*Delta)</CODE>) can be calculated using the following
   * symmetry:<BR>
   * <CODE>F[N] = F[N-n]*</CODE> (complex conjugation).<P>
   * Please note that the data is not normalized - you need to multiply each
   * value with 1/N. For further information of this bug in the "Numerical
   * Recipes" algorithm see <I>Tilman Butz: "Fouriertransformation f&uuml;r
   * Fu&szlig;g&auml;nger", Stuttgart, Leipzig, 2000, p. 103</I>.
   * @author Christian Datzko
   * @version 1.0
   * @param data[] the real data array
   * @throws IllegalArgumentException if <CODE>data[]</CODE> does not have the
   * length of a power of 2.
   */
  public static void realFFT(double data[]) {
    if (isPowerOf2(data.length)) {
      double[] d = new double[data.length + 1];
      d[0] = 0.0;
      // copy the array
      for (int i = 0; i < data.length; i++) {
        d[i + 1] = data[i];
      } // for
```

```
    // call the fast fourier transformation algorithm
    realft(d, d.length - 1, 1);
    // copy the array back
    for (int i = 1; i < d.length; i++) {
      data[i - 1] = d[i];
    } // for
  } // if
  else
    throw new IllegalArgumentException("The passed array does not have "
                                       + "the length of a Power of 2.");
} // realFFT(double data[])

/**
 * Performs an inverse Discrete Fast Fourier Transformation of the real
 * <CODE>data[]</CODE>.
 * Please note that <CODE>data[]</CODE> must have the length of a power of
 * 2, otherwise a <CODE>IllegalArgumentException</CODE> is thrown.
 * @author Christian Datzko
 * @version 1.0
 * @param data[] the real data array
 * @throws IllegalArgumentException if <CODE>data[]</CODE> does not have the
 * length of a power of 2.
 */
public static void realFFTinv(double data[]) {
  if (isPowerOf2(data.length)) {
    double[] d = new double[data.length + 1];
    d[0] = 0.0;
    // copy the array
    for (int i = 0; i < data.length; i++) {
      d[i + 1] = data[i];
    } // for
    // call the fast fourier transformation algorithm
    realft(d, d.length - 1, -1);
    // copy the array back
    for (int i = 1; i < d.length; i++) {
      data[i - 1] = d[i];
    } // for
  } // if
  else
    throw new IllegalArgumentException("The passed array does not have "
                                       + "the length of a Power of 2.");
} // realFFTinv(double data[])

/**
 * Performs a Discrete Fast Fourier Transformation of the complex
 * <CODE>data[]</CODE>.
 * Please note that <CODE>data[]</CODE> must have the length of a power of
 * 2, otherwise a <CODE>IllegalArgumentException</CODE> is thrown.<P>
 * The returned data is organized as follows:<BR>
 * <CODE>data[0]</CODE>: real part at <CODE><I>f</I>=0</CODE><BR>
 * <CODE>data[1]</CODE>: imaginary part at <CODE><I>f</I>=0</CODE><BR>
 * <CODE>data[2]</CODE>: real part at <CODE><I>f</I>=1/(N*Delta)</CODE><BR>
 * <CODE>data[3]</CODE>: imaginary part at <CODE><I>f</I>=1/(N*Delta)</CODE>
 * <BR>
 * ...<BR>
 * <CODE>data[2*i]</CODE>: real part at <CODE><I>f</I>=i/(N*Delta)</CODE><BR>
 * <CODE>data[2*i+1]</CODE>: imaginary part at <CODE><I>f</I>=i/(N*Delta)
 * </CODE><BR>
 * ...<BR>
 * <CODE>data[N-2]</CODE>: real part at <CODE><I>f</I>=(N/2-1)/(N*Delta)</CODE>
 * <BR>
 * <CODE>data[N-1]</CODE>: imaginary part at <CODE><I>f</I>=(N/2-1)/(N*Delta)
 * </CODE><BR>
 * <CODE>data[N]</CODE>: real part at <CODE><I>f</I>=+/- 1/(2*Delta)</CODE>
 * (combined)<BR>
 * <CODE>data[N+1]</CODE>: imaginary part at <CODE><I>f</I>=+/- 1/(2*Delta)
 * </CODE><BR>
 * <CODE>data[N+2]</CODE>: real part at <CODE><I>f</I>=-(N/2-1)/(N*Delta)
 * </CODE><BR>
 * <CODE>data[N+3]</CODE>: imaginary part at
 * <CODE><I>f</I>=-(N/2-1)/(N*Delta)</CODE><BR>
 * ...<BR>
```

```
 * <CODE>data[N+2*i]</CODE>: real part at <CODE><I>f</I>=-(N/2-i)/(N*Delta)
 * </CODE><BR>
 * <CODE>data[N+2*i+1]</CODE>: imaginary part at
 * <CODE><I>f</I>=-(N/2-i)/(N*Delta)</CODE><BR>
 * ...<BR>
 * <CODE>data[2*N-2]</CODE>: real part at <CODE><I>f</I>=-1/(N*Delta)</CODE>
 * <BR>
 * <CODE>data[2*N-1]</CODE>: imaginary part at <CODE><I>f</I>=-1/(N*Delta)
 * </CODE>.<P>
 * Please note that the data is not normalized - you need to multiply each
 * value with 1/N. For further information of this bug in the "Numerical
 * Recipes" algorithm see <I>Tilman Butz: "Fouriertransformation f&uuml;r
 * Fu&szlig;g&auml;nger", Stuttgart, Leipzig, 2000, p. 103</I>.
 * @author Christian Datzko
 * @version 1.0
 * @param data[] the complex data array - the complex part of each number
 * follows directly after the real part.
 * @throws IllegalArgumentException if <CODE>data[]</CODE> does not have the
 * length of a power of 2.
 */
public static void complexFFT(double data[]) {
  if (isPowerOf2(data.length)) {
    double[] d = new double[data.length + 1];
    d[0] = 0.0;
    // copy the array
    for (int i = 0; i < data.length; i++) {
      d[i + 1] = data[i];
    } // for
    // call the fast fourier transformation algorithm
    four1(d, (d.length - 1) / 2, 1);
    // copy the array back
    for (int i = 1; i < d.length; i++) {
      data[i - 1] = d[i];
    } // for
  } // if
  else
    throw new IllegalArgumentException("The passed array does not have "
                                       + "the length of a Power of 2.");
} // complexFFT(double data[])

/**
 * Performs an inverse Discrete Fast Fourier Transformation of the complex
 * <CODE>data[]</CODE>.
 * Please note that <CODE>data[]</CODE> must have the length of a power of
 * 2, otherwise a <CODE>IllegalArgumentException</CODE> is thrown.
 * This data is not normalized, you need to multiply it with
 * <CODE>1/N</CODE>.<P>
 * The returned data is organized as follows:<BR>
 * <CODE>data[0]</CODE>: real part at <CODE><I>t</I>=0</CODE><BR>
 * <CODE>data[1]</CODE>: imaginary part at <CODE><I>t</I>=0</CODE><BR>
 * ...<BR>
 * <CODE>data[2*i]</CODE>: real part at <CODE><I>t</I>=i*Delta</CODE><BR>
 * <CODE>data[2*i+1]</CODE>: imaginary part at <CODE><I>t</I>=i*Delta</CODE><BR>
 * ...<BR>
 * <CODE>data[2*N-2]</CODE>: real part at <CODE><I>t</I>=(N-1)*Delta</CODE>
 * <BR>
 * <CODE>data[2*N-1]</CODE>: imaginary part at <CODE><I>t</I>=-(N*1)*Delta
 * </CODE>.<P>
 * @author Christian Datzko
 * @version 1.0
 * @param data[] the complex data array - the complex part of each number
 * follows directly after the real part.
 * @throws IllegalArgumentException if <CODE>data[]</CODE> does not have the
 * length of a power of 2.
 */
public static void complexFFTinv(double data[]) {
  if (isPowerOf2(data.length)) {
    double[] d = new double[data.length + 1];
    d[0] = 0.0;
    // copy the array
    for (int i = 0; i < data.length; i++) {
      d[i + 1] = data[i];
```

```
    } // for
    // call the fast fourier transformation algorithm
    four1(d, (d.length - 1) / 2, -1);
    // copy the array back
    for (int i = 1; i < d.length; i++) {
      data[i - 1] = d[i];
    } // for
  } // if
  else
    throw new IllegalArgumentException("The passed array does not have "
                                       + "the length of a Power of 2.");
} // complexFFTinv(double data[])

/**
 * Checks recursivly, if i is an integer power of 2.
 * @author Christian Datzko
 * @version 1.0
 * @param i the long integer to be checked
 */
public static boolean isPowerOf2(long i) {
  if (i % 2 == 0)                     // the last bit == 0
    return isPowerOf2(i / 2);         // i is power of 2 if i/2 is power of 2
  else                                // the last bit == 1
    if (i == 1)                       // 2^0 = 1
      return true;
    else                              // i == 1+j*2 (j!=0)
      return false;
} // isPowerOf2(long i)

/**
 * Discrete Fourier transformation, as described in <I>NUMERICAL RECIPES IN
 * C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5), pp 507 ff</I>.
 * You should not call this method but the capsulating methods
 * <CODE>realFFT</CODE>, <CODE>realFFTinv</CODE>, <CODE>complexFFT</CODE> and
 * <CODE>complexFFTinv</CODE>.<P>
 * Original comment:<BR>
 * Replaces <CODE>data[1..2*nn]</CODE> by its discrete Fourier transform, if
 * <CODE>isign</CODE> is input as <CODE>1</CODE>; or replaces
 * <CODE>data[1..2*nn]</CODE> by <CODE>nn</CODE> times its inverse discrete
 * Fourier transform, if <CODE>isign</CODE> is input as <CODE>-1</CODE>.
 * <CODE>data</CODE> is a complex array of length <CODE>nn</CODE> or,
 * equivalently, a real array of length <CODE>2*nn</CODE>. <CODE>nn</CODE>
 * MUST be an integer power of 2 (this is not checked for!).<P>
 * Please note that the data for <CODE>isign = 1</CODE> is not normalized -
 * you need to multiply each value with 1/N. For further information of this
 * bug in the "Numerical Recipes" algorithm see <I>Tilman Butz:
 * "Fouriertransformation f&uuml;r Fu&szlig;g&auml;nger", Stuttgart, Leipzig,
 * 2000, p. 103</I>.
 * @author Numerical Recipes Software (adapted by Christian Datzko)
 * @version 1.0
 * @param data[] the data
 * @param nn data size
 * @param isign <CODE>1</CODE>: discrete Fourier transformation,
 * <CODE>-1</CODE>: inverse descrete Fourier transformation.
 */
public static void four1(double data[], int nn, int isign) {
/* C-code:

#define SWAP(a, b) tempr=(a); (a)=(b); (b)=tempr;

void four1(float data[], unsigned long nn, int isign) {
  unsigned long n, mmax, m, j, istep, i;
  double wtemp, wr, wpr, wpi, wi, theta;
                                      // Double precision for the
                                      // trigonometric recurrences.
  float tempr, tempi;

  n = nn << 1;
  j = 1;
  for (i = 1; i < n; i += 2) {        // This is the bit-reversal section
    if (j > i) {                      // of the routine.
      SWAP(data[j], data[i]);         // Exchange the two complex numbers.
```

```
      SWAP(data[j+1], data[i+1]);
    } // if
    m = n >> 1;
    while (m >= 2 && j > m) {
      j -= m;
      m >>= 1;
    } // while
    j += m;
  } // for
  // Here begins the Danielson-Lanczos section of the routine.
  mmax = 2;
  while (n > mmax) {                  // Outer loop executed log2 nn times.
    istep = mmax << 1;
    theta = isign * (6.28318530717959/mmax);
                                      // Initialize the trigonometric
                                      // recurrence.
    wtemp = sin(0.5*theta);
    wpr = -2.0 * wtemp * wtemp;
    wpi = sin(theta);
    wr = 1.0;
    wi = 0.0;
    for (m = 1; m < mmax; m += 2) {   // Here are the two nested inner loops.
      for (i = m; i <= n; i += istep) {
        j = i + mmax;                 // This is the Danielson-Lanczos
                                      // formula:
        tempr = wr * data[j] - wi * data[j + 1];
        tempi = wr * data[j + 1] + wi * data[j];
        data[j] = data[i] - tempr;
        data[j + 1] = data[i + 1] - tempi;
        data[i] += tempr;
        data[i + 1] += tempi;
      } // for
      wr = (wtemp = wr) * wpr - wi * wpi + wr;
      wi = wi * wpr + wtemp * wpi + wi;
    } // for
    mmax = istep;
  } // while
} // four1(float data[], unsigned long nn, int isign)
*/
  int n, mmax, m, j, istep, i;
  double wtemp, wr, wpr, wpi, wi, theta;
                                      // Double precision for the
                                      // trigonometric recurrences.
  double tempr, tempi;

  n = nn << 1;
  j = 1;
  for (i = 1; i < n; i += 2) {        // This is the bit-reversal section
    if (j > i) {                      // of the routine.
      double temp = data[j];          // Exchange the two complex numbers.
      data[j] = data[i];
      data[i] = temp;
      temp = data[j+1];
      data[j+1] = data[i+1];
      data[i+1] = temp;
    } // if
    m = n >> 1;
    while (m >= 2 && j > m) {
      j -= m;
      m >>= 1;
    } // while
    j += m;
  } // for
  // Here begins the Danielson-Lanczos section of the routine.
  mmax = 2;
  while (n > mmax) {                  // Outer loop executed log2 nn times.
    istep = mmax << 1;
    // Initialize the trigonometric recurrence.
    theta = isign * (2.0 * Math.PI/mmax);
    wtemp = Math.sin(0.5*theta);
    wpr = -2.0 * wtemp * wtemp;
    wpi = Math.sin(theta);
```

```
    wr = 1.0;
    wi = 0.0;
    for (m = 1; m < mmax; m += 2) {   // Here are the two nested inner loops.
      for (i = m; i <= n; i += istep) {
        j = i + mmax;                 // This is the Danielson-Lanczos
                                      // formula:
        tempr = wr * data[j] - wi * data[j + 1];
        tempi = wr * data[j + 1] + wi * data[j];
        data[j] = data[i] - tempr;
        data[j + 1] = data[i + 1] - tempi;
        data[i] += tempr;
        data[i + 1] += tempi;
      } // for
      // Trigonometic recurrence.
      wr = (wtemp = wr) * wpr - wi * wpi + wr;
      wi = wi * wpr + wtemp * wpi + wi;
    } // for
    mmax = istep;
  } // while
} // four1(float data[], long nn, int isign)

/**
 * Discrete Fourier transformation of two real arrays simultaniously, as
 * described in <I>NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING
 * (ISBN 0-521-43108-5), pp 511 ff</I>.
 * You should not call this method but the capsulating methods
 * <CODE>realFFT</CODE>, <CODE>realFFTinv</CODE>, <CODE>complexFFT</CODE> and
 * <CODE>complexFFTinv</CODE>.<P>
 * Original comment:<BR>
 * Given two real input arrays <CODE>data1[1..n]</CODE> and
 * <CODE>data2[1..n]</CODE>, this routine calls <CODE>four1</CODE> and
 * returns two complex output arrays, <CODE>fft1[1..2n]</CODE> and
 * <CODE>fft2[1..2n]</CODE>, each of complex length <CODE>n</CODE> (i.e.,
 * real length <CODE>2*n</CODE>), which contain the discrete Fourier
 * transforms of the respective <CODE>data</CODE> arrays. <CODE>n</CODE> MUST
 * be an integer power of 2.<P>
 * Please note that the data for <CODE>isign = 1</CODE> is not normalized -
 * you need to multiply each value with 1/N. For further information of this
 * bug in the "Numerical Recipes" algorithm see <I>Tilman Butz:
 * "Fouriertransformation f&uuml;r Fu&szlig;g&auml;nger", Stuttgart, Leipzig,
 * 2000, p. 103</I>.
 * @author Numerical Recipes Software (adapted by Christian Datzko)
 * @version 1.0
 * @param data1[] the data of the first function
 * @param data2[] the data of the second function
 * @param fft1[] the FFT of the first function
 * @param fft2[] the FFT of the second function
 * @param n data size
 */
public static void twofft(double data1[], double data2[], double fft1[],
                          double fft2[], int n) {
/* C-Code:

  void twofft(float data1[], float data2[], float fft1[], float fft2[],
    unsigned long n)
  unsigned long nn3, nn2, jj, j;
  float rep, rem, aip, aim;

  nn3=1+(nn2=2+n+n);
  for (j=1, jj=2; j<=n; j++, jj+=2) { // Pack the two real arrays into one
    fft1[jj-1]=data1[j];              // complex array.
    fft1[jj]=data2[j];
  }
  four1(fft1, n, 1);                  // Transform the complex array.
  fft2[1]=fft1[2];
  fft1[2]=fft2[2]=0.0;
  for (j=3; j<=n+1; j+=2) {
    rep=0.5*(fft1[j]+fft1[nn2-j]);    // Use symmetries to separate the two
    rem=0.5*(fft1[j]-fft1[nn2-j]);    // transforms.
    aip=0.5*(fft1[j+1]+fft1[nn3-j]);
    aim=0.5*(fft1[j+1]-fft1[nn3-j]);
    fft1[j]=rep;                      // Ship them out in two complex arrays.
```

```
    fft1[j+1]=aim;
    fft1[nn2-j]=rep;
    fft1[nn3-j]= -aim;
    fft2[j]=aip;
    fft2[j+1]= -rem;
    fft2[nn2-j]=aip;
    fft2[nn3-j]=rem;
  }
}
*/
  int nn3, nn2, jj, j;
  double rep, rem, aip, aim;

  nn3 = 1 + (nn2 = 2 + n + n);
  // Pack the two real arrays into one complex array.
  for (j = 1, jj = 2; j <= n; j++, jj += 2) {
    fft1[jj - 1] = data1[j];
    fft1[jj] = data2[j];
  } // for

  four1(fft1, n, 1);                  // Transform the complex array.
  fft2[1] = fft1[2];
  fft1[2] = fft2[2] = 0.0;
  for (j = 3; j <= n + 1; j += 2) {
    // Use symmetries to separate the two transforms.
    rep = 0.5 * (fft1[j] + fft1[nn2 - j]);
    rem = 0.5 * (fft1[j] - fft1[nn2 - j]);
    aip = 0.5 * (fft1[j + 1] + fft1[nn3 - j]);
    aim = 0.5 * (fft1[j + 1] - fft1[nn3 - j]);
    // Ship them out in two complex arrays.
    fft1[j] = rep;
    fft1[j + 1] = aim;
    fft1[nn2 - j] = rep;
    fft1[nn3 - j] = -aim;
    fft2[j] = aip;
    fft2[j + 1] = -rem;
    fft2[nn2 - j] = aip;
    fft2[nn3 - j] = rem;
  } // for
} // twofft(double data1[], double data2[], double fft1[], double fft2[], long n);

/**
 * Discrete Fourier transformation of one real array, as described in
 * <I>NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN
 * 0-521-43108-5), pp 513 ff</I>.
 * You should not call this method but the capsulating methods
 * <CODE>realFFT</CODE>, <CODE>realFFTinv</CODE>, <CODE>complexFFT</CODE> and
 * <CODE>complexFFTinv</CODE>.<P>
 * Original comment:<BR>
 * Calculates the Fourier transform of a set of <CODE>n</CODE>
 * real-valued data points. Replaces the data (which is stored in array
 * <CODE>data[1..n]</CODE> by the positive frequency half of its complex
 * Fourier transform. The real-valued first and last components of the
 * complex transform are returned as elements <CODE>data[1]</CODE> and
 * <CODE>data[2]</CODE>, respectively. <CODE>n</CODE> must be a power of 2.
 * This routine also calculates the inverse transform of a complex data
 * array if it is the transform of real data. (Result in this case must be
 * multiplied by 2/n.)<P>
 * Please note that the data for <CODE>isign = 1</CODE> is not normalized -
 * you need to multiply each value with 1/N. For further information of this
 * bug in the "Numerical Recipes" algorithm see <I>Tilman Butz:
 * "Fouriertransformation f&uuml;r Fu&szlig;g&auml;nger", Stuttgart, Leipzig,
 * 2000, p. 103</I>.
 * @author Numerical Recipes Software (adapted by Christian Datzko)
 * @version 1.0
 * @param data[] the data
 * @param n data size
 * @param isign <CODE>1</CODE>: discrete Fourier transformation,
 * <CODE>-1</CODE>: inverse descrete Fourier transformation.
 */
public static void realft(double data[], int n, int isign) {
/* C-Code:
```

```
#include <math.h>

void realft(float data[], unsigned long n, int isign)
{
  void four1(float data[], unsigned long nn, int isign);
  unsigned long i, i1, i2, i3, i4, np3;
  float c1=0.5,c2,h1r,h1i, h2r, h2i;
  double wr, wi, wpr, wpi, wtemp, theta;
                                       // Double precision for the
                                       // trigonometric recurrences.
  theta=3.141592653589793/(double) (n>>1);
                                       // Initialize the recurrence.
  if (isign == 1) {
    c2 = -0.5;
    four1(data, n>>1, 1);
  } else {
    c2=0.5;
    theta = -theta;
  }
  wtemp = sin(0.5*theta);
  wpr = -2.0*wtemp*wtemp;
  wpi = sin(theta);
  wr = 1.0+wpr;
  wi = wpi;
  np3 = n+3;
  for (i=2; i<=(n>>2); i++) {          // Case i = i done separately below.
    i4 = 1 + (i3=np3-(i2=1+(i1=i+i-1)));
    h1r=c1*(data[i1]+data[i3]);        // The two separate transforms are
    h1i=c1*(data[i2]-data[i4]);        // separated out of data
    h2r= -c2*(data[i2]+data[i4]);
    h2i=c2*(data[i1]-data[i3]);
    data[i1]=h1r+wr*h2r-wi*h2i;        // Here they are recombined to form
    data[i2]=h1i+wr*h2i+wi*h2r;        // the true transform of the original
    data[i3]=h1r-wr*h2r+wi*h2i;        // real data.
    data[i4]= -h1i+wr*h2i+wi*h2r;
    wr = (wtemp=wr)*wpr-wi*wpi+wr;     // The recurrence.
    wi = wi*wpr+wtemp*wpi+wi;
  }
  if (isign == 1) {
    data[1] = (h1r=data[1])+data[2];   // Squeeze the first and last data to-
    data[2] = (h1r-data[2]);           // gether to get them all with the
  }                                    // original array.
  else {
    data[1]=c1*((h1r=data[1])+data[2]);
    data[2]=c1*(h1r-data[2]);
    four1(data,n>>1,-1);               // This is tie hinverse transform for
  }                                    // the case isign = -1.
}
*/
  int i, i1, i2, i3, i4, np3;
  double c1=0.5,c2,h1r,h1i, h2r, h2i;
  double wr, wi, wpr, wpi, wtemp, theta;
                                       // Double precision for the
                                       // trigonometric recurrences.
  theta=Math.PI / (double)(n >> 1);    // Initialize the recurrence.
  if (isign == 1) {
    c2 = -0.5;
    four1(data, n >> 1, 1);
  } // if
  else {
    c2 = 0.5;
    theta = -theta;
  } // else
  wtemp = Math.sin(0.5 * theta);
  wpr = -2.0 * wtemp * wtemp;
  wpi = Math.sin(theta);
  wr = 1.0 + wpr;
  wi = wpi;
  np3 = n + 3;
  for (i = 2; i <= (n >> 2); i++) {    // Case i = i done separately below.
    i4 = 1 + (i3 = np3 - (i2 = 1 + (i1 = i + i - 1)));
```

```
      h1r = c1 * (data[i1] + data[i3]); // The two separate transforms are
      h1i = c1 * (data[i2] - data[i4]); // separated out of data
      h2r = -c2 * (data[i2] + data[i4]);
      h2i = c2 * (data[i1] - data[i3]);
      // Here they are recombined to form the true transform of the original
      // real data.
      data[i1] = h1r + wr * h2r - wi * h2i;
      data[i2] = h1i + wr * h2i + wi * h2r;
      data[i3]= h1r - wr * h2r + wi * h2i;
      data[i4]= -h1i + wr * h2i + wi * h2r;
      // The recurrence.
      wr = (wtemp = wr) * wpr - wi * wpi + wr;
      wi = wi * wpr + wtemp * wpi + wi;
    } // for
    if (isign == 1) {
      // Squeeze the first and last data together to get them all with the
      // original array.
      data[1] = (h1r = data[1]) + data[2];
      data[2] = (h1r - data[2]);
    } // if
    else {
      data[1] = c1 * ((h1r = data[1]) + data[2]);
      data[2] = c1 * (h1r - data[2]);
      four1(data, n >> 1, -1);          // This is the hinverse transform for
    } // else                           // the case isign = -1.
  } // realft(double data[], int n, int isign)
} // FastFourierTransformation
```

5.1.2. WaveFileAnalysis

```
/************************** WaveFileAnalysis.java **************************/
/* Datum: 12. Januar 2002
 * Autor: Christian Datzko
 * Copyright: Christian Datzko, 2002
 * E-Mail-Adresse: datzko@t-online.de
 * Programmiersprache und -version: Java(TM) 2 SDK, Standard Edition (1.3.1_01)
 */

import java.io.File;
import java.awt.event.*;

/**
 * Some useful routines for frequency analysis of wave files.
 */

public class WaveFileAnalysis {

  /**
   * Shows a <CODE>FrequencyWindow</CODE> with the data <CODE>d</CODE>, the
   * widht <CODE>width</CODE> and the height <CODE>height</CODE>.<P>
   * When the window is closed <CODE>System.exit(0);</CODE> is called to quit
   * the VM.
   * @author Christian Datzko
   * @version 1.0
   * @param d[][] an array of pairs of data, where <CODE>d[i][0]</CODE>
   * is a frequency and <CODE>d[i][1]</CODE> is an intensity. Assuming
   * linearity between <CODE>i</CODE> and <CODE>data[i][0]</CODE>.
   * @param width the width of the <CODE>FrequencyWindow</CODE>.
   * @param height the height of the <CODE>FrequencyWindow</CODE>.
   */
  public static void ShowAnalysisWindow(double[][] d, int width, int height) {
    FrequencyWindow frame = new FrequencyWindow(d);
    frame.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent ev) {
        System.exit(0);
      } // windowClosing(WindowEvent ev)
    } // inner class
    );
  frame.setSize(width, height);
  frame.setVisible(true);
  } // void ShowAnalysisWindow(double[][] d)
```

```
/**
 * Simply calls <CODE>ShowAnalysisWindow(double[][] d, int width, int
 * height)</CODE> with <CODE>width</CODE> = 400 and <CODE>height</CODE> = 300.
 * @author Christian Datzko
 * @version 1.0
 * @param d[][] an array of pairs of data, where <CODE>d[i][0]</CODE>
 * is a frequency and <CODE>d[i][1]</CODE> is an intensity. Assuming
 * linearity between <CODE>i</CODE> and <CODE>data[i][0]</CODE>.
 */
public static void ShowAnalysisWindow(double[][] d) {
  ShowAnalysisWindow(d, 400, 300);
} // void ShowAnalysisWindow(double[][] d)

/**
 * Analyses the wave file <CODE>fname</CODE> starting at
 * <CODE>start_sec</CODE>, using <CODE>lengthFFT</CODE> samples.
 * @author Christian Datzko
 * @version 1.0
 * @param fname a valid file name
 * @param start_sec a valid time in seconds within <CODE>fname</CODE>
 * @param lengthFFT length of the FFT array, must be a power of 2
 * @param leftChannel <CODE>true</CODE>: analyse the left channel,
 * <CODE>false</CODE>: analyse the right channel.
 * @throws IllegalArgumentException
 * @return double[][] contains <CODE>lengthFFT/2+1</CODE> pairs of data,
 * <CODE>[i][0]</CODE> is a frequency, <CODE>[i][1]</CODE> the corresponding
 * intensity.
 */
public static double[][] analyseWaveFile(String fname, double start_sec,
                                         int lengthFFT, boolean leftChannel) {
  AudioData d = new AudioData(new File(fname));
  int start = (int)(start_sec * d.getSampleRate());
  if (leftChannel)
    return analyseWaveData(d.getLeftChannel(), start, start + lengthFFT,
                           d.getSampleRate());
  else
    return analyseWaveData(d.getRightChannel(), start, start + lengthFFT,
                           d.getSampleRate());
} // analyseWaveFile(String fname, double start_sec, int lengthFFT,
  // boolean leftChannel)

/**
 * Analyses part (or all) of an array of integers from <CODE>start</CODE> to
 * <CODE>end-1</CODE> with the specified sampling rate.
 * @author Christian Datzko
 * @version 1.0
 * @param data[] array containing sampled data
 * @param start start index
 * @param end end index - 1 (<CODE>end-start</CODE> must be a power of 2)
 * @param samplingRate the sampling rate of the sampled data
 * @throws IllegalArgumentException
 * @return double[][] contains <CODE>lengthFFT/2+1</CODE> pairs of data,
 * <CODE>[i][0]</CODE> is a frequency, <CODE>[i][1]</CODE> the corresponding
 * intensity.
 */
public static double[][] analyseWaveData(int[] data, int start, int end,
                                         float samplingRate) {
  // what can happen...
  if (start >= end)
    throw new IllegalArgumentException("\"start\" must be higher then "
                                       + "\"end\".");
  if (!FastFourierTransformation.isPowerOf2(end - start))
    throw new IllegalArgumentException("Can only analyse data with the "
                                       + "length of a Power of 2.");
  if (data == null)
    throw new IllegalArgumentException("I need some data to analyse.");
  if (end > data.length)
    throw new IllegalArgumentException("\"data\" is too small.");

  // copy the data into a new array
  int N = end - start;
  double[] d = new double[N];
```

```
    for (int i = start, j = 0; i < end; i++, j++) {
      d[j] = data[i];
    } // for

    // Fast Fourier Transformation does the wave file analysis
    FastFourierTransformation.realFFT(d);

    double[][] output = new double[N / 2 + 1][2];

    // get out F[N/2]
    output[N / 2][0] = samplingRate / 2.0;
    output[N / 2][1] = Math.abs(d[1]);
    d[1]=0.0;

    // get out F[0]..F[N/2-1]
    for (int i = 0; i < d.length / 2; i++) {
      output[i][0] = (double)i * samplingRate / N;
      output[i][1] = Math.sqrt(d[i * 2] * d[i * 2] +
                               d[i * 2 + 1] * d[i * 2 + 1]);
    } // for
    return output;
  } // analyseWaveData(int[] data, int start, int end, int samplingRate)

  private static final String syntax_string = "WaveFileAnalysis: java "
                    + "WaveFileAnalysis filename start_sec fft_length";
                                    // syntax for  running this class

  /**
   * Small main method to test <CODE>WaveFileAnalysis</CODE>,
   * <CODE>FastFourierTransformation</CODE> and <CODE>FrequencyWindow</CODE>.
   * Run from command line with <CODE>java WaveFileAnalysis filename start_sec
   * fft_length</CODE>.
   * @author Christian Datzko
   * @version 1.0
   * @param argv[] passed array of parameters
   */
  public static void main(String[] argv) {
    if (argv.length!=3) {
      System.out.println(syntax_string);
    } // if
    else {
      double start_sec = 0.0;
      try {
        start_sec = Double.parseDouble(argv[1]);
      } // try
      catch (NumberFormatException e) {
        System.out.println(e.getMessage());
        System.out.println(syntax_string);
      } // catch
      int fft_length = 256;
      try {
        fft_length = Integer.parseInt(argv[2]);
      } // try
      catch (NumberFormatException e) {
        System.out.println(e.getMessage());
        System.out.println(syntax_string);
      } // catch
      if (!FastFourierTransformation.isPowerOf2(fft_length)) {
        System.out.println("fft_length must be an integer power of 2.");
        System.out.println(syntax_string);
      } // if
      else {
        double[][] d = analyseWaveFile(argv[0], start_sec, fft_length, true);
        // logarithmic plot of data gets a better graph
        for (int i = 0; i < d.length; i++) {
          d[i][1] = Math.log(d[i][1] / (double)fft_length);
        } // for
        ShowAnalysisWindow(d);
      } // else
    } // else
  } // main(String[] argv)
} // WaveFileAnalysis
```

5.1.3. FrequencyWindow.java

```
/************************** FrequencyWindow.java **************************/
/* Datum: 12. Januar 2002
 * Autor: Christian Datzko
 * Copyright: Christian Datzko, 2002
 * E-Mail-Adresse: datzko@t-online.de
 * Programmiersprache und -version: Java(TM) 2 SDK, Standard Edition (1.3.1_01)
 */

import javax.swing.*;
import java.awt.event.*;
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Dimension;

/**
 * This class is an object that is a window plotting frequency data.
 */

public class FrequencyWindow extends JFrame {
  private boolean dataInitialized = false;
                                    // are the data fields initialized?
  private double[][] frequencyTable;    // all frequencies
  private double min;                   // minimum value
  private double max;                   // maximum value

  private int minWidth = 160;           // minimum output width
  private int minHeight = 120;          // minimum output height
  private Color backColor = Color.white;// standard color for background
  private Color graphColor = Color.blue;// standard color for graph
  private Color coordColor = Color.black;
                                    // standard color for coordinate system
  private int spaceLeft = 10;           // space from the left
  private int spaceRight = 10;          // space from the right
  private int spaceTop = 29;            // space from the top
  private int spaceBottom = 10;         // space from the bottom

  /**
   * This constructor gets you a window that plots <CODE>data[][]</CODE>.
   * @author Christian Datzko
   * @version 1.0
   * @param data[][] an array of pairs of data, where <CODE>data[i][0]</CODE>
   * is a frequency and <CODE>data[i][1]</CODE> is an intensity. Assuming
   * linearity between <CODE>i</CODE> and <CODE>data[i][0]</CODE>.
   * @param title is a title string for the window.
   */
  public FrequencyWindow(double[][] data, String title) {
    super(title);
    if (data != null) {
      frequencyTable = data;
      // find min and max
      if (data[0].length >= 2) {
        min = max = data[0][1];
        for (int i = 1; i < data.length; i++) {
          if (data[i][1] < min) {
            min = data[i][1];
          } // if
          else {
            if (data[i][1] > max) {
              max = data[i][1];
            } // if
          } // else
        } // for
        dataInitialized = true;
      } // if
      else
        dataInitialized = false;
    } // if
    else {
      throw new IllegalArgumentException("Need some data to show.");
    } // else
```

```
} // FrequencyWindows(String title)

/**
 * This constructor calls <CODE>FrequencyWindow(double[][] data, String
 * title)</CODE> with <CODE>title</CODE> = "Frequency Analysis".
 * @author Christian Datzko
 * @version 1.0
 * @param data[][] an array of pairs of data, where <CODE>data[i][0]</CODE>
 * is a frequency and <CODE>data[i][1]</CODE> is an intensity. Assuming
 * linearity between <CODE>i</CODE> and <CODE>data[i][0]</CODE>.
 */
public FrequencyWindow (double[][] data) {
  this(data, "Frequency Analysis");
} // FrequencyWindow ()

/**
 * Returns if the frequency data is initialized (this is when the constructor
 * got a correct <CODE>double[][]</CODE> of data).
 * @author Christian Datzko
 * @version 1.0
 * @return boolean data is initialized.
 */
public boolean isDataInitialized() {
  return dataInitialized;
} // boolean isDataInitialized()

/**
 * Sets minimum width of the frequency window
 * @author Christian Datzko
 * @version 1.0
 * @param w must be larger than 0
 */
public void setMinWidth(int w) {
  if (w > 0) {
    minWidth = w;
  } // if
} // void setMinHeight(int h)

/**
 * Gets minimum width of the frequency window
 * @author Christian Datzko
 * @version 1.0
 * @return minimum width
 */
public int getMinWidth() {
  return minWidth;
} // int getMinWidth()

/**
 * Sets minimum height of the frequency window
 * @author Christian Datzko
 * @version 1.0
 * @param h must be larger than 0
 */
public void setMinHeight(int h) {
  if (h > 0) {
    minHeight = h;
  } // if
} // void setMinHeight(int h)

/**
 * Gets minimum height of the frequency window
 * @author Christian Datzko
 * @version 1.0
 * @return minimum height
 */
public int getMinHeight() {
  return minHeight;
} // int getMinHeight()

/**
 * Sets color for background of frequency graph
```

```
 * @author Christian Datzko
 * @version 1.0
 * @param c some Color
 */
public void setBackgroundColor(Color c) {
  if (c != null) {
    backColor = c;
  } // if
} // void setBackgroundColor(Color c)

/**
 * Gets color for background of frequency graph
 * @author Christian Datzko
 * @version 1.0
 * @return background color of frequency graph
 */
public Color getBackgroundColor() {
  return backColor;
} // void setBackgroundColor(Color c)

/**
 * Sets color for coordinate system
 * @author Christian Datzko
 * @version 1.0
 * @param c some Color
 */
public void setCoordinateSystemColor(Color c) {
  if (c != null) {
    coordColor = c;
  } // if
} // void setCoordinateSystemColor(Color c)

/**
 * Gets color for coordinate system
 * @author Christian Datzko
 * @version 1.0
 * @return coordinate system color
 */
public Color getCoordinateSystemColor() {
  return coordColor;
} // void setCoordinateSystemColor(Color c)

/**
 * Sets color for frequency graph
 * @author Christian Datzko
 * @version 1.0
 * @param c some Color
 */
public void setGraphColor(Color c) {
  if (c != null) {
    graphColor = c;
  } // if
} // void setGraphColor(Color c)

/**
 * Gets color for frequency graph
 * @author Christian Datzko
 * @version 1.0
 * @return frequency graph color
 */
public Color getGraphColor() {
  return graphColor;
} // void setGraphColor(Color c)

/**
 * Sets space to be left between border of window and border of graph
 * @author Christian Datzko
 * @version 1.0
 * @param left space from left
 * @param right space from right
 * @param top space from top
 * @param bottom space from bottom
```

```
 */
public void setSpace(int left, int right, int top, int bottom) {
  if ((left > 0 && right > 0 && top > 0 && bottom > 0)
      && (left + right < minWidth)
      && (top + bottom < minHeight)) {
    spaceLeft = left;
    spaceRight = right;
    spaceTop = top;
    spaceBottom = bottom;
  } // if
} // void setSpace(int left, int right, int top, int bottom)

/**
 * Gets space from left between between border of window and border of graph
 * @author Christian Datzko
 * @version 1.0
 * @return space from left between between border of window and border of graph
 */
public int getSpaceLeft() {
  return spaceLeft;
} // int getSpaceLeft()

/**
 * Gets space from right between between border of window and border of graph
 * @author Christian Datzko
 * @version 1.0
 * @return space from right between between border of window and border of graph
 */
public int getSpaceRight() {
  return spaceRight;
} // int getSpaceRight()

/**
 * Gets space from top between between border of window and border of graph
 * @author Christian Datzko
 * @version 1.0
 * @return space from top between between border of window and border of graph
 */
public int getSpaceTop() {
  return spaceTop;
} // int getSpaceTop()

/**
 * Gets space from bottom between between border of window and border of graph
 * @author Christian Datzko
 * @version 1.0
 * @return space from bottom between between border of window and border of graph
 */
public int getSpaceBottom() {
  return spaceBottom;
} // int getSpaceBottom()

/**
 * Does the plotting section of the <CODE>FrequencyWindow</CODE>. This method
 * is called each time the window is redrawn. You don't need to call this
 * method.
 * @author Christian Datzko
 * @version 1.0
 * @param g the graphics object to draw to
 */
public void paint(Graphics g) {
  Dimension d = getSize();
  if (d.width < minWidth) {
    setSize(minWidth, d.height);
  } // if
  d = getSize();
  if (d.height < minHeight) {
    setSize(d.width, minHeight);
  } // if
  d = getSize();

  // delete old contents
```

```
    g.clearRect(1, 1, d.width, d.height);
    // draw coordinate system
    if (dataInitialized) {
      int w = d.width - spaceLeft - spaceRight - 1;
      int h = d.height - spaceTop - spaceBottom - 1;

      // graph background
      g.setColor(backColor);
      g.fillRect(spaceLeft, spaceTop, d.width - spaceLeft - spaceRight,
                 d.height - spaceTop - spaceBottom);

      // coordinate system
      g.setColor(coordColor);
      int offsetY = (int)Math.round((double)h * (double)(max) /
                                    (double)(max - min)) + spaceTop;
      g.drawLine(spaceLeft, d.height - spaceBottom,
                 d.width - spaceRight - 1, d.height - spaceBottom);
      g.drawLine(d.width - spaceRight - 6, d.height - spaceBottom - 5,
                 d.width - spaceRight - 1, d.height - spaceBottom);
      g.drawLine(spaceLeft, spaceTop, spaceLeft, d.height - spaceBottom);
      g.drawLine(spaceLeft, spaceTop, spaceLeft + 5, spaceTop + 5);

      // graph
      g.setColor(graphColor);
      int x1, x2, y1, y2;
      x2 = spaceLeft;
      y2 = (int)Math.round((double)h * (double)(max - frequencyTable[0][1]) /
                           (double)(max - min)) + spaceTop;
      for (int i = 1; i < frequencyTable.length; i++) {
        x1 = x2;
        x2 = (int)Math.round((double)(i * w) /
                             (double)(frequencyTable.length - 1)) + spaceLeft;
        y1 = y2;
        y2 = (int)Math.round((double)h * (double)(max - frequencyTable[i][1]) /
                             (double)(max - min)) + spaceTop;
        g.drawLine(x1, y1, x2, y2);
      } // for
    } // if
  } // void paint (Graphics g)
} // FrequencyWindow
```

5.1.4. `AudioData.java`

```
import javax.sound.sampled.*;
import java.io.File;

public class AudioData {
  private int[] leftChannel  = null;    // Sample-Daten des linken Kanals
  private int[] rightChannel = null;    // Sample-Daten des rechten Kanals
  private int numChannels  = 0;         // Anzahl der Kanäle
  private long millis = 0;              // Dauer in Millisekunden
  private int sampleDepth = 0;          // Bits pro Sample
  private float sampleRate = 0;         // Samples pro Sekunde
  private boolean valid = false;        // true, wenn Audiodaten gültig

  public AudioData(File audiofile) {
      if (audiofile != null && audiofile.isFile()) try {
        AudioInputStream inputstream = AudioSystem.getAudioInputStream(audiofile);
        AudioFormat format = inputstream.getFormat();
        numChannels = format.getChannels();
        millis = (long)(1000 * inputstream.getFrameLength()/format.getFrameRate());
        sampleDepth = format.getSampleSizeInBits();
        sampleRate  = format.getSampleRate();

        int frameLength = (int)inputstream.getFrameLength();

        leftChannel = new int[frameLength];
        if (numChannels == 2)
          rightChannel = new int[frameLength];

        byte frame[] = new byte[format.getFrameSize()];
```

```
        if (sampleDepth == 16)
          for (int i=0; i < frameLength; i++) {
            inputstream.read(frame, 0, frame.length);
            leftChannel[i] = toInt(frame[0], frame[1], format);
            if (numChannels == 2)
              rightChannel[i] = toInt(frame[2], frame[3], format);
          } // for
      } // if
      else if (sampleDepth == 8) {
        for (int i=0; i < frameLength; i++) {
          inputstream.read(frame, 0, frame.length);
          leftChannel[i] = toInt(frame[0], format);
          if (numChannels == 2)
            rightChannel[i] = toInt(frame[1], format);
        } // for
      } // if
      inputstream.close();
      valid = true;                    // wenn wir hier angekommen sind, wurden
                                       // gültige Audiodaten gelesen
    } // try
    catch (Exception e) {
      System.out.println("Fehler beim Öffnen von '" + audiofile + "'");
      e.printStackTrace();
    } // catch
  } // AudioData(File audiofile)

/*  public void print() {
    for (int i=0; i < leftChannel.length; i++) {
      System.out.print(i + ": " + leftChannel[i]);
      if (rightChannel != null)
        System.out.println(", " + rightChannel[i]);
    } // for
  } // void print()
*/

  public int getNumChannels() {
    return numChannels;
  } // int getNumChannels()

  public long getMillis() {
    return millis;
  } // long getMillis()

  public int[] getLeftChannel() {
    return leftChannel;
  } // int[] getLeftChannel()

  public int[] getRightChannel() {
    return rightChannel;
  } // int[] getRightChannel()

  public void setLeftChannel(int[] lc) {
    leftChannel = lc;
  } // void setLeftChannel(int[] lc)

  public void setRightChannel(int[] rc) {
    rightChannel = rc;
  } // void setRightChannel(int[] rc)

  public boolean isValid() {
    return valid;
  } // boolean isValid()

  public float getSampleRate() {
    return sampleRate;
  } // float getSampleRate()

  public int getSampleDepth() {
    return sampleDepth;
  } // int getSampleDepth()

  // wandelt ein einzelnes Byte gemäß angegebenem AudioFormat in eine Integer-
```

```
  // Zahl um
  protected static int toInt(byte b, AudioFormat format) {
    int res;
    if (format.getEncoding() == AudioFormat.Encoding.PCM_UNSIGNED)
      res = b & 0xff;
    else
      res = b;
    if (format.getEncoding() == AudioFormat.Encoding.PCM_UNSIGNED)
      res -= 0x7f;
    return res;
  } // int toInt(byte b, AudioFormat format)

  // wandelt ein Byte-Paar gemäß angegebenem AudioFormat in eine Integer-Zahl
  // um
  protected static int toInt(byte b1, byte b2, AudioFormat format) {
    int upper_byte, lower_byte;
    if (format.isBigEndian()) {
      upper_byte = b1;
      lower_byte = b2;
    } // if
    else {
      upper_byte = b2;
      lower_byte = b1;
    } // else
    int res = (upper_byte << 8) | (lower_byte & 0xff);
    if (format.getEncoding() == AudioFormat.Encoding.PCM_UNSIGNED)
      res -= 0x7fff;
    return res;
  } // int toInt(byte b1, byte b2, AudioFormat format)

/*  public static void main(String[] argv) {
    AudioData ad = new AudioData(new File("c:\\windows\\media\\chord.wav"));
    int left[]  = ad.getLeftChannel();
    int right[] = ad.getLeftChannel();
    System.out.println("Kanäle: " + ad.getNumChannels());
    for (int i=0; i < left.length; i++)
      System.out.println(left[i] + "   " + right[i]);
    } // void main(String[] argv)*/
} // AudioData
```

5.2. Komplexe Zahlen[12]

Für die Gleichung $x^2=-1$ gibt es in den reellen Zahlen $\mathbb{R}$ keine Lösung. Da man aber mit dem Ergebnis einer solchen Gleichung in besonderen Fällen weiter rechnen möchte, definiert man einfach ein besonderes Zeichen als Ergebnis: $\mathrm{i}=\sqrt{-1}$. Somit kann man alle Wurzeln negativer Zahlen lösen.

Mit diesem Zeichen kann man weiterrechnen, wie man weiter rechnen, wie man es gewohnt ist, Addition und Subtraktion, Multiplikation und Division, die üblichen Berechnungen kann man weiterhin machen. Dabei gelten folgende Rechenregeln:

$$\begin{gathered}(a+b\,\mathrm{i})+(c+d\,\mathrm{i})=(a+c)+(b+d)\,\mathrm{i}\\(a+b\,\mathrm{i})\cdot(c+d\,\mathrm{i})=a\cdot c-b\cdot d+(a\cdot d+b\cdot c)\,\mathrm{i}\\|a+b\,\mathrm{i}|=\sqrt{a^2+b^2}\\(a+b\,\mathrm{i})^*=(a-b\,\mathrm{i})\end{gathered}\tag{10}$$

Man kann sich eine komplexe Zahl $z=a+b\,\mathrm{i}$ gut in einem zweidimensionalen Koordinatensystem vorstellen. Dann sind die oben genannten Rechenregeln auch sofort einsichtig.

12 Siehe auch: Peter Meyer-Nieberg: *„Analysis – Vorlesung SS 2000“*, Uni Osnabrück, 2000.

5.3. Literatur

Sun Microsystems: *„Java™ 2 SDK, Standard Edition – Documentation – Version 1.3.1“*, http://java.sun.com/j2se/1.3/docs.html.

Tilman Butz: *„Fouriertransformation für Fußgänger“*, Stuttgart, Leipzig, 2000.

Sydney VisLab: *„1. The Fourier Series: Background“*, http://www.vislab.usyd.edu.au/CP3/Four1/node2.html, 30. Juni 2002.

Mathematica: http://www.wolfram.com/.

Oliver Vornberger, Olaf Müller und Ralf Kunze: *„Algorithmen – Vorlesung im WS 2001/2002“*, Osnabrück, 2001

Numerical Recipes Software: *„NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING“*, Cambridge, 1992, http://www.nr.com/.

Peter Meyer-Nieberg: *„Analysis – Vorlesung SS 2000“*, Uni Osnabrück, 2000.

Diese Ausarbeitung, die Präsentation und die `.java`-Dateien: http://www.t-online.de/~datzko/.