Wave-Datei-Analyse via FFT

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...

Überblick

- Vorüberlegungen
- Fourier-Transformation
- Fast Fourier Transformation und Algorithmen
- Wave-Datei-Analyse

Vorüberlegungen

- Werte zwischen -128 und 127 (8-bit) bzw. -32768 und 32767 (16-bit)
- Konstante Abstände zwischen den einzelnen Werten

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

- Endliche Anzahl von Daten, jedoch in großen Mengen, vorhanden

Fourier-Transformation

Fourier-Transformation

- Jede mögliche Schwingung auf einer Periode T läßt sich als unendliche Summe von Sinus- und Kosinus-Funktionen darstellen:

$$f(t) = \sum_{k=0}^{\infty} \left(A_k \cos \omega_k t + B_k \sin \omega_k t \right)$$

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

$$\text{und } B_0 = 0$$

- Das Wichtigste ist es nun, die Koeffizienten A_k und B_k zu berechnen.

Fourier-Koeffizienten

- Dies geht, wie man durch arithmetische Operationen zeigen kann, so:

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

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

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

Umwandlung in komplexe Schreibweise

- Die Eulersche Identität läßt uns Sinus und Kosinus auch im komplexen Fall fassen:

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

- Wir können nun unser $f(t)$ komplex auffassen:

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

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

- Natürlich lassen sich dann auch die Koeffizienten C_k komplex formulieren:

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

Diskrete Fourier-Transformation

- Die bisher betrachteten Formeln beziehen sich auf kontinuierliche Funktionen. Hier haben wir jedoch nur „Stichproben“, unsere Samples $\{f_k\}$. Umformuliert lautet die Formel für das Spektrum $\{F_j\}$ also so:

$$F_j = \frac{1}{N} \sum_{k=0}^{N-1} f_k W_N^{-kj}$$

$$\text{mit } W_N = \mathrm{e}^{\frac{2\pi \mathrm{i}}{N}}$$

Fast Fourier Transformation und Algorithmen

Laufzeit Fourier-Transformation

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

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

Laufzeit Fast Fourier Transformation

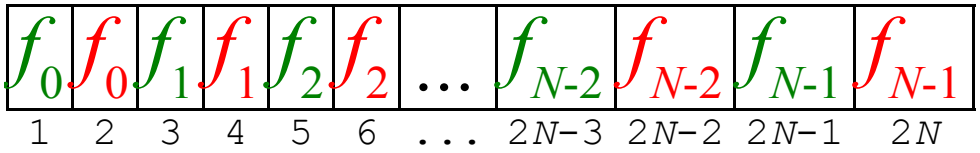
- Durch geschickte Überlegungen (dass man aus zwei Fourier-Reihen eine machen kann, und dass bei einer Fourier-Reihe der Länge 1 $F_0 = f_0$ ist) kann man einen Algorithmus der Laufzeit $\mathrm{O}(N \log_2 N)$ schreiben.

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

Algorithmus

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

- Der Algorithmus selbst besteht aus zwei Teilen, der erste sortiert die Daten neu für den Algorithmus und der zweite berechnet die Fourier-Transformationen.
- Die Daten werden wie folgt an den Algorithmus übergeben:



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

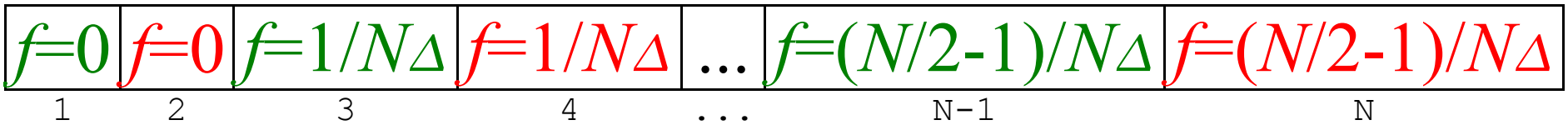
- `nn` ist N, `isign=1` bedeutet FFT, `isign=-1` inverse FFT.

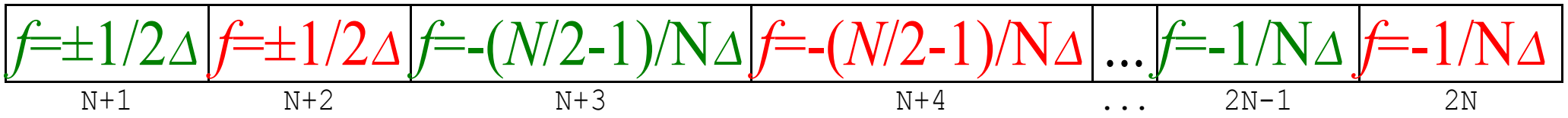
```
public static void four1(double data[], int nn, int isign) {
    int n, mmax, m, j, istep, i;
    double wtemp, wr, wpr, wpi, wi, theta;
                                          // Double precision for the
                                          // trigonometric recurrences.
    double tempr,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

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

   // 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)
```

`four1()`: *Output-Daten*

- Die Daten werden in folgendem Format wieder zurückgegeben:





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

Komfortable Methoden

- `complexFFT()` und `complexFFTinv()` übernehmen die Bestimmung von N und reorganisieren das [0..2N-1]-Datenfeld in ein [1..2N]-Datenfeld und zurück:

```
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[])
```

Reelle Fast Fourier Transformation

- Eine reelle Fast Fourier Transformation ist nichts anderes als eine komplexe Fast Fourier Transformation, wo alle imaginären Teile der $f_n = 0$.
- Es gilt: $F_n = F_{N-n}^*$ (komplex konjugiert). Deshalb braucht nur das positive Frequenzspektrum gespeichert zu werden.
- Es ist möglich, zwei reelle FFT in dem Algorithmus für komplexe FFT gleichzeitig durchzuführen (`twofft()`).

Komfortable Methode

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

- Input:

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

- Output (f=0 und f=±1/(2Δ) haben keinen imaginären Teil):

f=0	f=±1/(2Δ)	f=1/(NΔ)	f=1/(NΔ)	...
0	1	2	3	...

...	f=(N/2-1)/(NΔ)	f=(N/2-1)/(NΔ)
...	N-2	N-1

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

Wave-Datei-Analyse

Format der Daten

- Die Daten eines Audio-Kanals liegen in einem `int[] data` sortiert nach ihrem zeitlichen Auftreten vor.
- Zu jedem `int[] data` gibt es eine Sample-Rate `float samplingRate`. Sie bezeichnet die Anzahl der Samples pro Sekunde.
- Die Werte der einzelnen `int`s liegen zwischen -128 und 127 (8-bit) bzw. -32768 und 32767 (16-bit).

Analyse der FFT-Daten

- Wichtig sind die absoluten Größen der Intensitäten (also der komplexen Zahlen). Diese errechnen sich durch folgende Formel:

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

- Die Frequenzen zu den dazugehörigen Intensitäten können durch die schon öfters erwähnte Formel errechnet werden:

$$F_i = \frac{i}{N\,\Delta}$$

$$\text{mit } \Delta = \frac{1}{SamplingRate}$$

`analyseWaveData()`: *Ein Stück Wave-Daten analysieren*

- Der folgende Algorithmus transportiert einen Teil der Wave-Daten in den `realFFT()`-Algorithmus und gibt die Analyse zurück. Zuerst Kopf und Parameter-Check:

```
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.");
```

```
public static double[][] analyseWaveData(int[] data, int start, int end,
                                         float samplingRate) {

  // 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)
```

`analyseWaveData()`: *Output-Daten*

- Zurückgegeben wird ein zweidimensionales Array von Doubles `double[][]`, das für jede errechnete Intensität bei einer bestimmten Frequenz ein Pärchen bereithält, das die Frequenz und die Intensität beinhaltet:
  `double[i][0]`=Frequenz
  `double[i][1]`=Intensität bei Frequenz
- Die Anzahl der Pärchen ist genau `end` – `start` +1 (also immer eine Potenz von 2+1).

`analyseWaveFile()`: *einen Teil einer Wave-Datei analysieren*

- Nun haben wir fast alles, um eine echte Wave-Datei zu analysieren. Was nun noch fehlt sind ein paar Routinen, die die Daten aus einer Wave-Datei holen und an `analyseWaveData()` schicken. Dabei kommt die schon bekannte Klasse `AudioData` mit ins Spiel:

```
AudioData AudioData(java.io.File audiofile)
int[] getLeftChannel()
int[] getRightChannel()
float getSampleRate()
```

`analyseWaveFile()`: *einen Teil einer Wave-Datei analysieren*

- Der Rest ist nun trivial:

```
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)
```

- Die Output-Daten sind die, die von `analyseWaveData()` zurückgegeben werden.

`showAnalysisWindow()`: *die Daten auf den Bildschirm bringen*

- Die Daten sind nun soweit, dass sie visuell dargestellt werden können. Da es hier um die Analyse der Daten geht und nicht um `java.awt.*`. Deshalb nur das Wichtigste.
- Für die Darstellung ist eine logarithmische Skala der Y-Achse praktisch. Somit wird die Lautstärke linear in dB dargestellt. Hinzu kommt noch die Normalisierung, die hier jedoch ohne weitere Bedeutung ist.

```
// 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
```

$$\forall x : \log\left(\frac{x}{c}\right) = \log(x) - \log(c)$$

- Das Objekt `FrequencyWindow` ist ein Fenster, das die Daten plottet. Der wichtigste Teil dabei ist die Methode `paint()`, die die eigentliche Arbeit übernimmt.

Literatur

- Tilman Butz: *„Fouriertransformation für Fußgänger“*, Stuttgart, Leipzig, 2000.
- Numerical Recipes Software: *„NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING“*, Cambridge, 1992, http://www.nr.com/.
- Sun Microsystems: *„Java™ 2 SDK, Standard Edition – Documentation – Version 1.3.1“*, http://java.sun.com/j2se/1.3/docs.html.
- Materialien zur Sitzung: http://www.t-online.de/~datzko/musik/wissenschaft/fft.zip.