Lehrstuhl für Multimediakommunikation und Signalverarbeitung

Prof. Dr.-Ing. André Kaup

Forschungspraktikum

im Studiengang
"Elektrotechnik, Elektronik und Informationstechnik (EEI)"

von

Simon Appel

zum Thema

Entwicklung einer optimierten 2D-FFT-Bibliothek

Betreuer: Nils Genser, M.Sc.
Beginn: 01.05.2017

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und, dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde.
Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 2. September 2017

Simon Appel

Kurzfassung

Dieses Forschungspraktikum behandelt eine effiziente Implementierung der Fourier Transformation von zweidimensionalen Signalen. Die benötigten Funktionen werden in C++ realisiert. Zu Beginn werden die Grundlagen der 2D-Fouriertransformation für digitale Signale vorgestellt. Anschließende werden zwei Implementierungen der 1D-FFT vorgestellt. Eine davon ist rekursiv und die andere nicht-rekursiv. Die beiden Varianten werden unter Zuhilfenahme einer bereits existierenden Implementierung erstellt. Diese wurde in C geschrieben und dient für die folgende Arbeit als Vorlage. Zum Vergleich wird die FFT auch in MATLAB berechnet. Die gewünschten Erweiterungen für Spezialfälle und die Implementierung für 2D-Signale wird im letzten Kapitel, vor der Zusammenfassung, erklärt. Hier wird zuerst die nicht-rekursive Variante der 1D-FFT auf eine 2D-FFT erweitert. Als letzter Schritt wird nur ein Teil des FFT-Spektrums berechnet, um es dann anschließend durch Kopieren entsprechender Werte zu vervollständigen. Hierbei werden die Symmetrieeigenschaften eines zweidimensionalen Spektrums ausgenutzt. Die berechneten FFT-Werte sind außerdem skalierbar.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Ein wichtiges Arbeitsgebiet der digitalen Signalverarbeitung beschäftigt sich mit der Rekonstruktion von Bild- und Videosignalen. Dabei ist es oftmals notwendig ein digitales Signal in seine Frequenzanteile zu zerlegen und diese anschließend weiter zu analysieren und zu verarbeiten. Üblicherweise wird zur effizienten Berechnung der Diskreten Fourier-Transformation (DFT) das Verfahren der schnellen Fourier-Transformation (FFT) verwendet.

In der Bildverarbeitung ist besonders die reelwertige 2D-FFT von Interesse, bei der sich besondere DFT-Symmetrieeigenschaften ergeben. Diese lassen sich ausnutzen, um nur einen Teil des Spektrums berechnen und die fehlenden Werte im Anschluss kopieren zu müssen, womit der erforderliche Rechenaufwand stark absinkt. Ein weiterer Punkt betrifft die Skalierung der FFT. Hier gibt es mehrere Möglichkeiten, die FFT bzw. die inverse FFT zu formulieren. Da je nach Verarbeitung der Werte eine andere Skalierung von Vorteil sein kann, soll diese voreinstellbar bzw. vollständig deaktivierbar sein. Viele FFT-Implementierungen arbeiten als rekursiver Algorithmus. Das heißt, die Eingangswerte werden rekursiv in Listen zerlegt, die FFT berechnet und die Listen wieder zusammengefügt ("Butterfly"-Prinzip). Eine rekursive Zerlegung ist rechenaufwendig, da für jeden Funktionsaufruf der Kontext gesichert und Methodeneintrittscode gespeichert werde muss. Durch eine nicht-rekursive bzw. iterative FFT-Umsetzung wird eine bessere Performanz erreicht.

1.2 Aufgabenstellung

In diesem Forschungspraktikum wird eine effiziente C++ Implementierung der FFT umgesetzt. Diese nutzt sowohl die zweidimensionalen Symmetrieeigenschaften aus und ist auch variabel skalierbar. Für eine schnelle Berechnung wird die FFT als nicht-rekursiver Algorithmus implementiert. Hierfür wird als Vorlage eine nicht-rekursive 1D-FFT verwendet [1]. Diese wird den entsprechenden Anforderungen angepasst.

2 Diskrete zweidimensionale Fourier-Transformation

In der Einleitung wurde bereits gesagt, dass in der digitalen Signalverarbeitung die Diskrete Fourier-Transformation (DFT), bei der Bearbeitung von Bild- und Videosignalen zum Einsatz kommt. Da diese Signale zweidimensional sind, kommt hier die 2D-FFT zum Einsatz [2]. Die Berechnungsvorschrift des Fourier-Spektrums ergibt sich zu:

$$\begin{aligned} X[k,l] = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} \left(\frac{1}{\sqrt{M}} \sum_{m=0}^{M-1} x[m,n] e^{-j2\pi \frac{mk}{M}} \right) e^{-j2\pi \frac{nl}{N}} = \\ = \sum_{n=0}^{N-1} \omega_N[n,l] \left(\sum_{M=0}^{M-1} \omega_M[m,k] x[m,n] \right) \end{aligned} \tag{2.1}$$

.

Hierbei bezeichnet x das zweidimensionale Eingangssignal, dessen Fourier-Spektrum berechnet werden soll. M und N sind die Anzahl der Abtastpunkte in x und y-Richtung. k und l definieren die Anzahl der Zeilen und Spalten des Spektrums X. Die Drehfaktoren ω_M und ω_N sind dabei definiert als

$$\omega_M[m,k] = \frac{1}{\sqrt{M}} e^{-j2\pi \frac{mk}{M}} \quad \text{bzw.} \quad \omega_N[n,l] = \frac{1}{\sqrt{N}} e^{-j2\pi \frac{ml}{N}} \tag{2.2}$$

.

Um die FFT einer zweidimensionalen Matrix zu berechnen, werden zuerst die einzelnen Spalten dieser Matrix in den Frequenzbereich transformiert. Daraus resultiert eine Matrix $X'[k,n]$. Anschließend werden die Zeilen der Matrix X' in den Frequenzbereich transformiert und man erhält die 2D-FFT Matrix X des ursprünglichen Eingangssignals x. Die Berechnungsvorschrift für die FFT der Spalten ist in (2.3) zu sehen.

$$X'[k,n] = \sum_{m=0}^{M-1} \omega_M[m,k] x[m,n]; \qquad (k = 0, 1, ..., M-1) \tag{2.3}$$

Hierbei bezeichnet m den Zeilenindex der Matrix $x[m,n]$. Der Ausdruck ist also die 1D-FFT der n-ten Spalte von x. Die FFT der Zeilen wird als

$$X[k,l] = \sum_{n=0}^{N-1} \omega_N[n,l] X'[k,n]; \qquad (l = 0, 1, ..., N-1) \tag{2.4}$$

bestimmt. Dieser Ausdruck ist die 1D-FFT der k-ten Zeile von X'. Alternativ kann auch zuerst die FFT der Zeilen und dann die der Spalten berechnet werden [2].

3 Implementierung

3.1 1D-FFT

Als erster Schritt wird die 1D-FFT, unter Zuhilfenahme des C-Codes aus [1], sowohl rekursiv als auch nicht rekursiv in C++ implementiert. Im Folgenden wird nur der Code der nicht-rekursiven Implementierung betrachtet, da dieser anschließend weiterverwendet wird, um die 2D-FFT nicht-rekursiv zu implementieren. Die Implementierung der 1D-FFT als rekursiver Algorithmus in C++ dient lediglich zu Vergleichszwecken und zur Überprüfung und Verifizierung der berechneten FFT-Werte einer Sinusfolge.

Quelltextbeispiel 3.1 zeigt die Funktion zur Realisierung des Bit-Reversings.

Quelltextbeispiel 3.1: Durchführung des Bit-Reversings

```
// C++ function for Bit reversing
void BitInvert(complex<double> *a, int n)
// "a" is the 1D-array as inputsignal, "n" is the number of samples
{
  int i, mv = n/2;
  int k, rev = 0;
  complex<double> b;
  for (i = 1; i < n; i++) // run through all the indexes from 1 to n
  {
    k = i;
    mv = n / 2;
    rev = 0;
    while (k > 0) // invert the actual index
    {
      if ((k % 2) > 0)

        rev = rev + mv;
        k = k / 2;
        mv = mv / 2;

    }
    { // switch the actual sample and the bitinverted one
      if (i < rev)
      {
        b = a[rev];
        a[rev] = a[i];
        a[i] = b;
      }
    }
  }
}
```

Für jeden Index geht man mit der Laufvariable i durch alle Bits mit Hilfe von k durch und invertiert diese. Durch die Modulodivision wird überprüft, ob das LSB 1 oder 0 ist und ob es folglich umsortiert werden muss, oder nicht. Mit dem MSB wird gestartet. Durch die Division von k durch 2 wird das zweite LSB eine Position nach rechts verschoben. So werden sukzessiv alle Bits vom LSB bis zum MSB durchgegangen und die umsortierte Bitfolge wird in rev gespeichert. Jetzt wird noch der Index i und rev vertauscht, aber nur wenn i kleiner als rev ist. Ansonsten werden alle Indizes vertauscht, sobald i größer als rev wird.

Die Funktion zur Berechnung der $\frac{N}{2}$-FFT bzw. sub-FFT ist in Quelltextbeispiel 3.2 dargestellt.

Quelltextbeispiel 3.2: Berechnung der sub-FFT

```
// C++ function for the calculation of the sub-FFT
void CalcSubFFT(complex<double> *a, int n) // "a" is the 1D-array as
    inputsignal, "n" is the number of samples
{
  int i, k, m; // variables for the loops
  complex<double> w; // variable for the twiddle factor "w"
  complex<double> v; // variable for buffering the input "a"
  complex<double> h; // variable for buffering the complex product of the
      elements of "a" with "w"

  k = 1;
  while (k <= n/2) // we start at stage 0 with 2 samples per transformation
  {
    m = 0;
    while (m <= (n-2*k))
    {
      for (i = m; i < m + k; i++)
      {
        w = complex<double>(cos(M_PI * (double)(i - m) / (double)(k)), sin(M_PI
            * (double)(i - m) / (double)(k))); // computation of the twiddle
            factor

        // butterfly operation
        h = a[i + k] * w; // complex product of the elements of "a" with "w"
        v = a[i];
        a[i] = a[i] + h; // complex sum
        a[i + k] = v - h; // complex difference
      }
      m = m + 2 * k;
    }
    k = k * 2;
  }
}
```

Bis auf wenige Änderungen ist diese C++ Funktion nahezu identisch mit der C-Funktion aus [1].

Zur Berechnung der vollständigen FFT werden die beiden vorgestellten Funktionen in einer gemeinsamen Funktion aufgerufen. Diese ist in Quelltextbeispiel 3.3 zu sehen.

Quelltextbeispiel 3.3: Berechnung der 1D-FFT

```
// C++ function for calculation of the 1D-FFT
void CalcFFT(complex<double> *signalY, int signalLength)
{
    complex<double> *y = signalY;
    int N = signalLength;
    int i;
    BitInvert(y, N); // Bit reversing
    CalcSubFFT(y, N); // calculation of the sub-FFT
    for (i = 0; i < N; i++) {
    //y[i] = y[i] / (double) N * 2.0;
    //y[i].real(-y[i].real());
    y[i].imag(-y[i].imag()); // adaption of the imaginary part
    }
    //y[0] = y[0] / 2.0;
}
```

Das Eingangssignal wird in dem eindimensionalen array y gespeichert und die Länge des Signals gibt die Anzahl der Abtastpunkte für das Bit-Reversing und die sub-FFT vor.

In ähnlicher Art und Weise wird der C-Code aus [1] für die rekursive 1D-FFT in C++ implementiert. Beiden Implementierungen (rekursiv als auch nicht-rekursiv) wird als Eingangssignal eine Sinusfolge der Werte “0“ bis “7“ übergeben. Die Anzahl der Abtastpunkte N wird in diesem Fall also auf “8“ gesetzt. Gleichzeitig wird die FFT dieser Sinuswerte in MATLAB berechnet, um die Funktionalität beider Algorithmen zu überprüfen. In Tabelle x sind die Ergebnisse aller drei FFT-Berechnungen (C++ rekursiv und nicht-rekursiv und MATLAB) dargestellt.

Tabelle 3.1: Berechnung der FFT der Sinuswerte von 0 bis 7

x	0	1	2	3
sin(x)	0	0,84	0,90	0,14
FFT[sin(x)]	0,55 + 0,00 i	2,39 - 2,09 i	-1,38 + 0,91 i	-0,88 + 0,28 i

x	4	5	6	7
sin(x)	-0,75	-0,95	-0,27	0,65
FFT[sin(x)]	-0,80 + 0,00 i	-0,88 - 0,28 i	-1,38 - 0,91 i	2,39 + 2,09 i

Alle drei Berechnungen der FFT dieser Sinuswerte liefern als Resultat das gleiche Ergebnis.

3.2 2D-FFT

3.2.1 Standardverfahren

Um die FFT eines zweidimensionalen Signals zu berechnen, muss erst die FFT der einzelenen Zeilen und dann die der einzelnen Spalten bzw. anders herum berechnet werden. Es ist grundsätzlich egal, welche Variante verwendet wird. Das bedeutet, dass die drei vorgestellten Funktionen des Kapitels “1D-FFT“ lediglich durch passende Schleifenaufrufe iterativ verwendet werden müssen, um die FFT einer zweidimensionalen Matrix berechnen zu können. Im Folgenden werden zuerst die Zeilen zur FFT-Berechnung herangezogen und dann die Spalten.

Quelltextbeispiel 3.4 zeigt die Funktionen die zur Extraktion einzelner Zeilen bzw. Spalten einer 2D-Matrix verwendet werden.

Quelltextbeispiel 3.4: Extraktion einzelner Zeilen und Spalten einer 2D-Matrix

```
1  // C++ function for extraction of a column
2  complex<double>* getcolumn(complex<double> **a, int rowCount, int colNumber)
3  // "a" is the input signal, "rowCount" is the number of rows
4  {
5    complex<double> *col = new complex<double> [rowCount]; // variable for the
         received column
6     // iteration through the rows of the input signal
7     for (int i = 0; i < rowCount; i++){
8       col[i] = a[i][colNumber]; //column is saved to an 1D-array
9     }
10    return col;
11 }
12
13 // C++ function for extraction of a row
14 complex<double>* getrow(complex<double> **a, int colCount, int rowNumber)
15 // "a" is the input signal, "colCount" is the number of columns
16 {
17   // variable for the received row
18   complex<double> *row = new complex<double> [colCount];
19
20    // iteration through the columns of the input signal
21    for (int i = 0; i < colCount; i++){
22      row[i] = a[rowNumber][i]; //row is saved to an 1D-array
23    }
24    return row;
25 }
```

Wie man sieht sind diese beiden Funktionen sehr intuitiv geschrieben und einfach zu verstehen. Diese werden nun zusammen mit der Funktion (CalcFFT) aus Kapitel (1D-FFT) verwendet, um die Berechnung der Fourier-Transformation der einzelnen Zeilen bzw. Spalten des zweidi-

mensionalen Eingangssignals zu implementieren. Der Programmcode ist in Quelltextbeispiel 3.5 dargestellt.

Quelltextbeispiel 3.5: Berechnung der FFT-Matrix der Zeilen bzw. Spalten

```
1  // C++ function for the calculation of the FFT of the columns
2  complex<double>** FFTcolumnscalc(complex<double> **a, int rowCount, int
       colCount) // "a" is the input signal, "rowCount" is the number of rows, "
       colCount" is the number of columns
3  {
4    // FFT-matrix for the columns
5    complex<double> **FFTcols = new complex<double>*[rowCount];
6    for(int i = 0; i < rowCount; ++i){
7      FFTcols[i] = new complex<double>[colCount];
8    }
9    complex<double> *cols; // array for the columns
10
11   for(int i = 0; i < rowCount; i++){
12     // iterative extraction of the columns and their FFT-computations
13     cols = getcolumn(a,rowCount,i);
14     CalcFFT(cols,rowCount);
15     for(int j = 0; j < rowCount; j++){
16       FFTcols[j][i] = complex<double>(cols[j].real(), -cols[j].imag()); //
             creation of the columns transform matrix and adaption of the imaginary
              part
17     }
18   }
19   return FFTcols;
20 }
21
22 // C++ function for the calculation of the FFT of the rows
23 complex<double>** FFTrowscalc(complex<double> **a, int rowCount, int colCount)
       // "a" is the input signal, "rowCount" is the number of rows, "colCount" is
        the number of columns
24 {
25   // FFT-matrix for the rows
26   complex<double> **FFTrows = new complex<double>*[rowCount];
27   for(int i = 0; i < rowCount; ++i){
28     FFTrows[i] = new complex<double>[colCount];
29   }
30   complex<double> *rows; // array for the rows
31
32   for(int i = 0; i < colCount; i++){
33     // iterative extraction of the rows and their FFT-computations
34     rows = getrow(a,rowCount,i);
35     CalcFFT(rows,colCount);
36     for(int j = 0; j < colCount; j++){
37       FFTrows[i][j] = rows[j]; // creation of the rows transform matrix
38     }
39   }
40   return FFTrows;
41 }
```

Beide Funktionen laufen nach dem gleichen Muster ab. Im ersten Schritt werden die einzelnen Vektoren (Zeilen bzw. Spalten) der Matrix extrahiert. Da diese Vektoren eindimensionale Signale darstellen, erfolgt die Berechnung ihrer FFT-Werte sukzessive mit der Funktion “CalcFFT“ für 1D-Arrays. Im nächsten Schritt werden, die berechneten Vektoren, der Reihe nach, in eine 2D-Matrix gespeichert.
Um nun die FFT eines zweidimensionalen Signals zu berechnen, wird erste die Funktion `FFTrowscalc` aufgerufen und die daraus resultierende Matrix anschließend der Funktion `FFTcolumnscalc` als Eingangssignal übergeben. Daraus erhält man das Endergebnis.

Zur Überprüfung der vorgestellten Implementierungen, wird als Eingangssignal die Matrix aus [2] verwendet. Gleichzeitig wird die 2D-FFT auch in MATLAB berechnet. Wird die FFT der folgenden Matrix berechnet:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 70 & 80 & 90 & 0 & 0 & 0 \\ 0 & 0 & 90 & 100 & 110 & 0 & 0 & 0 \\ 0 & 0 & 110 & 120 & 130 & 0 & 0 & 0 \\ 0 & 0 & 130 & 140 & 150 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Sowohl MATLAB als auch die vorgestellte C++ Implementierung liefern für den Realteil,

$$\begin{bmatrix} 1320,0 & -791,1 & 80,0 & -168,9 & 440,0 & -168,9 & 80,0 & -791,1 \\ -504,9 & -90,7 & 221,4 & 105,9 & -168,3 & 12,7 & -261,4 & 685,3 \\ 120,0 & -0,0 & -40,0 & -23.4 & 40,0 & 0,0 & 40,0 & -136,6 \\ -335,1 & 134,1 & 21,4 & 50,7 & -111,7 & 34,7 & -61,4 & 267,3 \\ 120,0 & -68,3 & 0,0 & -11,7 & 40.0 & -11,7 & -0,0 & -68,3 \\ -335,1 & 267,3 & -61,4 & 34,7 & -111,7 & 50,7 & 21,4 & 134,1 \\ 120,0 & -136,6 & 40,0 & -0,0 & 40,0 & -23,4 & -40,0 & 0,0 \\ -504,9 & 685,3 & -261,4 & 12,7 & -168,3 & 105,9 & 221,4 & -90,7 \end{bmatrix}$$

und für den Imaginärteil

$$\begin{bmatrix} 0,0i & -711,1i & 440,0i & 88,9i & 0,0i & -88,9i & -440,0i & 711,1i \\ -724,3i & 713,6i & -216,6i & 55,6i & -241,4i & 134,1i & 120,0i & 159,0i \\ 120,0i & -136,6i & 40,0i & 0,0i & 40,0i & -23,4i & -40,0i & 0,0i \\ -124,3i & 255,6i & -120,0i & -6,4i & -41,4i & 39,0i & 103,4i & -105,9i \\ 0,0i & -68,3i & 40,0i & 11,7i & 0,0i & -11,7i & -40,0i & 68,3i \\ 124,3i & 105,9i & -103,4i & -39,0i & 41,4i & 6,4i & 120,0i & -255,6i \\ -120,0i & 0,0i & 40,0i & 23,4i & -40,0i & 0,0i & -40,0i & 136,6i \\ 724,3i & -159,0i & -120,0i & -134,1i & 241,4i & -55,6i & 216,6i & -713,6i \end{bmatrix}$$

.

Um auf die gleichen Werte wie aus [2] zu kommen, müssen die berechneten FFT-Werte durch den Faktor 8 geteilt werden. Hierzu wird, wie auch in der Aufgabenstellung gefordert, eine einfache Multiplikationsfunktion in C++ geschrieben, die jedes Element der zu skalierenden Matrix mit einem beliebigen Faktor multipliziert. Diese Rechenoption ist optional implementiert und kann nach Bedarf verwendet werden.

3.2.2 Ausnutzung der Symmetrieeigenschaften

Betrachtet man die FFT-Resultate aus „Standardverfahren“ genauer, so sieht man dass das Fourier-Spektrum sowohl im Real-, als auch im Imaginärteil punkt- sowie achsensymmetrische Komponenten aufweist.

Daher werden die vorgestellten C++ Funktionen neu implementiert, um nur einen Teil des FFT-Spektrums zu berechnen. Die vollständigen FFT-Matrizen erhält man anschließend durch einfaches Kopieren der berechneten Werte.

Zuerst werden die Funktionen `getcolumn` und `getrow` in `getcolumnpart` und `getrowpart` umgeschrieben. Diese sind in Quelltextbeispiel 3.6 zu sehen.

Quelltextbeispiel 3.6: Extraktion einzelner Teil-Zeilen und Teil-Spalten einer 2D-Matrix

```
1  // C++ function for extraction of a columnpart
2  complex<double>* getcolumnpart(complex<double> **a, int rowCount, int colNumber
       ) // "a" is the input signal, "rowCount" is the number of rows
3  {
4    // variable for the received column
5    complex<double> *col = new complex<double> [rowCount];
6
7      // iteration through the rows of the input signal
8      for (int i = 0; i <= (rowCount/2); i++){
9        col[i] = a[i][colNumber]; //columnpart is saved to an 1D-array
10     }
11     return col;
12 }
13
14 // C++ function for extraction of a rowpart
15 complex<double>* getrowpart(complex<double> **a, int colCount, int rowNumber)
       // "a" is the input signal, "colCount" is the number of columns
16 {
17   complex<double> *row = new complex<double> [colCount]; // variable for the
         received row
18     // iteration through the columns of the input signal
19     for (int i = 0; i <= (colCount/2); i++){
20       row[i] = a[rowNumber][i]; //rowpart is saved to an 1D-array
21     }
22     return row;
23 }
```

Man sieht, dass sich für den Quelltextausschnitt kaum Änderungen ergeben. Der einzige Unterschied zu den ursprünglichen Funktionen besteht darin, dass die Laufvariable i der beiden for-Schleifen nur bis zur Hälfte der entsprechenden Matrixdimension inkrementiert wird. Dadurch werden nur die notwendigen Elemente der Matrix zur effizienten Berechnung der FFT extrahiert.

Als nächstes werden die Funktionen `FFTcolumnscalc` und `FFTrowscalc` in `FFTcolumnscalcpart` und `FFTrowscalcpart` umgeschrieben. Die beiden Funktionen verwenden die Funktionen `getcolumnpart` und `getrowpart`. Sie sind in Quelltextbeispiel 3.7 dargestellt.

Quelltextbeispiel 3.7: Berechnung des notwendigen Teils der FFT-Matrix der Zeilen bzw. Spalten

```
// C++ function for the calculation the necessary part
// of the FFT of the columns
complex<double>** FFTcolumnscalcpart(complex<double> **a, int rowCount, int
    colCount)
// "a" is the input signal, "rowCount" is the number of rows,
// "colCount" is the number of columns
{
  // FFT-matrix for the columns
  complex<double> **FFTcols = new complex<double>*[rowCount];
  for(int i = 0; i < rowCount; ++i){
    FFTcols[i] = new complex<double>[colCount];
  }
  complex<double> *cols; // array for the columns

  for(int i = 0; i < rowCount; i++){
    // iterative extraction of the columns and their FFT-computations
    cols = getcolumnpart(a,rowCount,i);
    CalcFFT(cols,rowCount);
    for(int j = 0; j <= (rowCount/2); j++){
    // creation of the columns transform matrix and
    // adaption of the imaginary part
      FFTcols[j][i] = complex<double>(cols[j].real(), -cols[j].imag());
    }
  }
  return FFTcols;
}
// C++ function for the calculation the necessary part
// of the FFT of the rows
complex<double>** FFTrowscalcpart(complex<double> **a, int rowCount, int
    colCount)
// "a" is the input signal, "rowCount" is the number of rows,
// "colCount" is the number of columns
{
  // FFT-matrix for the rows
  complex<double> **FFTrows = new complex<double>*[rowCount];
  for(int i = 0; i < rowCount; ++i){
    FFTrows[i] = new complex<double>[colCount];
  }
  complex<double> *rows; // array for the rows

  for(int i = 0; i <= (colCount/2); i++){
    // iterative extraction of the rows and their FFT-computations
    rows = getrowpart(a,rowCount,i);
    CalcFFT(rows,colCount);
    for(int j = 0; j < colCount; j++){
      FFTrows[i][j] = rows[j]; // creation of the rows transform matrix
    }
  }
  return FFTrows;
}
```

Wendet man diese beiden Funktionen auf das bereits vorgestellte zweidimensionale Signal aus [2] an, resultiert daraus folgendes Spektrum für den Realteil,

$$\begin{bmatrix} 1320,0 & -791,1 & 80,0 & -168,9 & 440,0 & -168,9 & 80,0 & -791,1 \\ -504,9 & -90,7 & 221,4 & 105,9 & -168,3 & 12,7 & -261,4 & 685,3 \\ 120,0 & -0,0 & -40,0 & -23.4 & 40,0 & 0,0 & 40,0 & -136,6 \\ -335,1 & 134,1 & 21,4 & 50,7 & -111,7 & 34,7 & -61,4 & 267,3 \\ 120,0 & -68,3 & 0,0 & -11,7 & 40.0 & -11,7 & -0,0 & -68,3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

und für den Imaginärteil,

$$\begin{bmatrix} 0,0i & -711,1i & 440,0i & 88,9i & 0,0i & -88,9i & -440,0i & 711,1i \\ -724,3i & 713,6i & -216,6i & 55,6i & -241,4i & 134,1i & 120,0i & 159,0i \\ 120,0i & -136,6i & 40,0i & 0,0i & 40,0i & -23,4i & -40,0i & 0,0i \\ -124,3i & 255,6i & -120,0i & -6,4i & -41,4i & 39,0i & 103,4i & -105,9i \\ 0,0i & -68,3i & 40,0i & 11,7i & 0,0i & -11,7i & -40,0i & 68,3i \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

.

Durch entsprechendes Kopieren und Einfügen, werden der Real- und der Imaginärteil des Spektrums vervollständigt. Da es sich bei den achsensymmetrischen Komponenten nur um Vektoren handelt, können hierfür einfach die Funktionen `getcolumnpart` und `getrowpart` verwendet werden. Die Implementierung hierfür ist in Quelltextbeispiel 3.8 zu sehen.

Quelltextbeispiel 3.8: Kopieren der achsensymmetrischen Komponenten des FFT-Spektrums

```
// C++ function to complete the spectrum with the axially symmetric components
complex<double>** copyaxially(complex<double> **a, int rowCount, int colCount)
  // "a" is the input signal, "rowCount" is the number of rows,
  // "colCount" is the number of columns
{
  // arrays for the required columns
  complex<double> *colLeft;
  complex<double> *colRight;
  // arrays for the required rows
  complex<double> *rowUp;
  complex<double> *rowDown;
  // saving the columns
  colLeft = getcolumnpart(a,rowCount,0);
  colRight = getcolumnpart(a,rowCount,(colCount/2));
  // saving the rows
  rowUp = getrowpart(a,colCount,0);
  rowDown = getrowpart(a,colCount,(rowCount/2));

      // iterative pasting of the columns with adaption of the imaginary part
      for(int j = rowCount-1; j >= (rowCount/2); j--){
        a[j][0] = complex<double>(colLeft[rowCount-j].real(), -colLeft[rowCount-
            j].imag());
        a[j][(colCount/2)] = complex<double>(colRight[rowCount-j].real(), -
            colRight[rowCount-j].imag());
      }

      // iterative pasting of the rows with adaption of the imaginary part
      for(int j = colCount-1; j >= (colCount/2); j--){
        a[0][j] = complex<double>(rowUp[colCount-j].real(), -rowUp[colCount-j].
            imag());
        a[(rowCount/2)][j] = complex<double>(rowDown[colCount-j].real(), -
            rowDown[colCount-j].imag());
      }
    return a;
}
```

Zuerst werden, die entsprechenden Vektoren zwischengespeichert und die Werte anschließend über for-Schleifen iterativ in die 2D-Matrix eingefügt.

Zum Kopieren der punktsymmetrischen Komponenten, wird eine Funktion `getmatrixpart` implementiert, da hier Untermatrizen des Spektrums kopiert werden müssen. Die Funktion ist in Quelltextbeispiel 3.9 zu sehen.

Quelltextbeispiel 3.9: Extrahieren von Untermatrizen aus einer 2D-Matrix

```
// C++ function for extraction of a sub-matrix
complex<double>** getmatrixpart(complex<double> **a, int rowCount, int colCount
    , int type) {
  // matrix for the copied elements of the inputsignal a
  complex<double> **MatrixPart = new complex<double>*[rowCount];
  for(int i = 0; i < rowCount; ++i){
    MatrixPart[i] = new complex<double>[colCount];
  }

  if(type == 0){ // left part
    for(int i = 1; i <= ((rowCount/2)-1); i++){
      for(int j = 1; j <= ((colCount/2)-1); j++){
        MatrixPart[i][j] = a[i][j];
      }
    }
  }
  else if (type == 1){ // right part
    for(int i = 1; i <= ((rowCount/2)-1); i++){
      for(int j = ((colCount/2)+1); j < colCount; j++){
        MatrixPart[i][j] = a[i][j];
      }
    }
  }
    return MatrixPart;
}
```

Abhängig von der Variable `type`, wird entweder der rechte bzw. der linke Teil des Spektrums aus dem Eingangssignal extrahiert. Das Aufsammeln der Werte wird jeweils über zwei for-Schleifen realisiert, die so gewählt sind dass sie an den entsprechenden Stellen der Eingangsmatrix mit der Iteration starten. Im 2D-Array `MatrixPart` werden die Werte zur Weiterverarbeitung gespeichert.

Als letzter Schritt wird die Funktion `copypointsymmetry` implementiert, um mit Hilfe der Funktion `getmatrixpart` das Spektrum zu vervollständigen und das Endergebnis zu erhalten. Dieses ist natürlich auch wieder nach Bedarf skalierbar. Die Funktion ist in Quelltextbeispiel 3.10 dargestellt.

Quelltextbeispiel 3.10: Kopieren der punktsymmetrischen Komponenten des FFT-Spektrums

```
// C++ function to complete the spectrum with the point symmetric components
complex<double>** copypointsymmetry(complex<double> **a, int rowCount, int
    colCount) {

  // arrays for the requiered matrix parts
  complex<double>** leftMatrixPart;
  complex<double>** rightMatrixPart;
  // saving the matrix parts
  leftMatrixPart = getmatrixpart(a,rowCount,colCount,0);
  rightMatrixPart = getmatrixpart(a,rowCount,colCount,1);

  // iterative pasting of the leftMatrixPart with adaption of the imaginary
      part
  for(int i = rowCount-1; i >= ((rowCount/2)+1); i--){
    for(int j = colCount-1; j >= ((colCount/2)+1); j--){
      FFTpointsymmetry[i][j] = complex<double>(leftMatrixPart[rowCount-i][
          colCount-j].real(), -leftMatrixPart[rowCount-i][colCount-j].imag());
      a[i][j] = complex<double>(leftMatrixPart[rowCount-i][colCount-j].real(),
          -leftMatrixPart[rowCount-i][colCount-j].imag());
    }
  }

  // iterative pasting of the rightMatrixPart with adaption of the imaginary
      part
  for(int i = rowCount-1; i >= ((rowCount/2)+1); i--){
    for(int j = ((colCount)/2-1); j > 0; j--){
      FFTpointsymmetry[i][j] = complex<double>(rightMatrixPart[rowCount-i][
          colCount-j].real(), -rightMatrixPart[rowCount-i][colCount-j].imag());
      a[i][j] = complex<double>(rightMatrixPart[rowCount-i][colCount-j].real(),
           -rightMatrixPart[rowCount-i][colCount-j].imag());
    }
  }
  return a;
}
```

Das Prinzip ist hierbei das gleiche, wie bei der Funktion `copyaxially`. Zuerst werden, die entsprechenden Untermatrizen zwischengespeichert und die Werte anschließend über zwei for-Schleifen iterativ in die 2D-Matrix eingefügt.

Werden alle vorgestellten Funktionen dieses Unterkapitels auf das Signal aus [2] angewendet, erhält man das gleiche FFT-Spektrum, wie mit der normalen C++ Implementierung ohne Ausnutzung der Symmetrieeigenschaften, oder der Standard 2D-FFT aus MATLAB.

4 Zusammenfassung

Im Rahmen dieses Forschungspraktikums, wurde eine effiziente C++ Implementierung der Fourier Transformation von zweidimensionalen Signalen realisiert. Nach der Vorstellung des Prinzips der diskreten zweidimensionalen Fourier Transformation, wurde anhand einer vorgegeben C Implementierung zur 1D-FFT eine rekursive als auch nicht-rekursive C++ Implementierung entwickelt. Für die weiteren Arbeitsschritte wurde nur die nicht-rekursive Variante verwendet, da die Rekursive zwar intuitiv, aber ineffizient ist. Die 1D-FFT wurde auf eine 2D-FFT erweitert. Hierfür wird prinzipiell einfach die Funktion, zur Berechnung eines eindimensionalen Spektrums, iterativ zuerst auf die Zeilen und dann auf die Spalten des zweidimensionalen Eingangssignals angewendet. Um das Ganze noch effizienter zu machen, wurde als letzter Schritt nur ein Teil des Spektrums berechnet, um dessen Symmetrieeigenschaften auszunutzen. Über anschließende Schleifenoperationen wurden die fehlenden Werte des FFT-Spektrums zuerst kopiert und dann iterativ an die entsprechenden Stellen der FFT-Matrix eingefügt, um das vollständige Spektrum zu erhalten.

Tabellenverzeichnis

Quelltextverzeichnis

Literaturverzeichnis

[1] H.-P. Moser, “Quick FFT - CodeProject.” [Online]. Available: https://www.codeproject.com/Articles/619688/Quick-FFT

[2] “Two-Dimensional Fourier Transform,” 15.11.2007. [Online]. Available: http://fourier.eng.hmc.edu/e101/lectures/Image_Processing/node6.html