Xpert.press

Die Reihe **Xpert.press** vermittelt Professionals in den Bereichen Softwareentwicklung, Internettechnologie und IT-Management aktuell und kompetent relevantes Fachwissen über Technologien und Produkte zur Entwicklung und Anwendung moderner Informationstechnologien.

Oliver Kluge

Praktische Informationstechnik mit C#

Anwendungen und Grundlagen

Mit 115 Abbildungen und 18 Tabellen

Springer

Dr. Oliver Kluge
90489 Nürnberg
oliver.kluge@siemens.com

Bibliografische Information der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de abrufbar.

ISSN 1439-5428
ISBN-10 3-540-20812-7 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-20812-9 Springer Berlin Heidelberg New York

Springer ist ein Unternehmen von Springer Science+Business Media
springer.de

Satz und Herstellung: LE-TEX, Jelonek, Schmidt & Vöckler GbR, Leipzig
Umschlaggestaltung: KünkelLopka Werbeagentur, Heidelberg
Gedruckt auf säurefreiem Papier 33/3100 YL – 5 4 3 2 1 0

Vorwort

„Scientific computing“, „computational intelligence“ und „computational engineering“ sind Schlagworte der modernen Informationstechnik. Diese Begriffe stehen für verschiedene Konzepte der digitalen Informationsverarbeitung, wie sie in Wissenschaft und Technik zum Einsatz kommen. Ob bei der Analyse von Meßwertreihen, der automatischen Schrifterkennung oder bei der Simulation von elektrischen Ausgleichsvorgängen, Kollege Computer ist immer dabei. Das vorliegende Buch gibt einen Einblick in die Funktionsweise der entsprechenden Programme. Zahlreiche Implementierungsbeispiele erleichtern das Verständnis. Aufgrund des breiten Anwendungsspektrums der Informationstechnik werden sehr unterschiedliche Schwerpunkte behandelt.

Signale sind Träger von Information. Die digitale Signalverarbeitung ist daher eine Schlüsseltechnologie unserer heutigen Informationsgesellschaft. Im ersten Kapitel steht die Verarbeitung von Abtastsignalen im Zeit- und Frequenzbereich im Vordergrund. Neben der Frequenzanalyse und dem Kurzzeit-Spektrum wird ausführlich auf die Berechnung digitaler Filter eingegangen. Hieran schließen sich statistische Verfahren an. Lage-, Streu- und Formparameter sind geeignete Kenngrößen, um Rauschphänomene zu beschreiben. Stochastische Abhängigkeiten in Meßreihen lassen sich mit Hilfe der Korrelationsanalyse erkennen. Zusätzlich werden Regressionsverfahren und statistische Rangordnungsfilter vorgestellt. Als Ergänzung zu den konventionellen Verfahren behandelt das dritte Kapitel Methoden der künstlichen Intelligenz. Mit neuronalen Netzen und Fuzzy-Systemen hat die Signalverarbeitung nach biologischem Vorbild bereits in vielen Anwendungen Einzug gefunden, bei denen unscharfe oder gestörte Informationen vorliegen. Das vierte Kapitel ist eine Einführung in die Simulationstechnik. Mit der numerischen Lösung von Differentialgleichungen oder Monte-Carlo-Methoden kommt man der Realität zum Greifen nahe. Doch was nützt Information, wenn sie nicht verfügbar ist? Die Kommunikation über Netzwerke ist eine weitere Säule der Informationstechnik, weshalb sich das letzte Kapitel mit der Netzwerkprogrammierung beschäftigt.

Nürnberg, Januar 2006 *Oliver Kluge*

Inhaltsverzeichnis

1 Digitale Signalverarbeitung 1
1.1 Einführung 1
1.2 Fourier-Reihen 2
1.3 Die Diskrete Fourier-Transformation (DFT) 6
1.4 Arithmetiktuning – Die FFT 11
1.5 Pulse und Pulsfolgen 22
1.6 Der Abtastvorgang 26
1.7 Das Abtasttheorem 29
1.8 Leakage 30
1.9 Nichtstationäre Signale – Die Grenzen der DFT 33
1.10 Die Zeit-Frequenz-Analyse 35
1.11 Digitale Filter 40
1.11.1 Frequenzselektive Eigenschaften 41
1.11.2 Die *z*-Transformation 43
1.11.3 Die Übertragungsfunktion und der Frequenzgang 47
1.11.4 Mittelwertfilter 49
1.11.5 FIR-Filter 51
1.11.6 IIR-Filter 68
1.11.7 Der Phasengang 81
1.11.8 Vergleich zwischen FIR- und IIR-Filtern 82
1.11.9 FFT-Filter 83
1.12 Experimentelle Systemanalyse 84
1.12.1 Identifikation im Frequenzbereich 84
1.12.2 Identifikation im Zeitbereich 94

2 Statistische Signalverarbeitung 97
2.1 Einführung 97
2.2 Zufallszahlen – Dem Rauschen auf der Spur 98
2.2.1 Gleichverteilte Zufallszahlen 99
2.2.2 Normalverteilte Zufallszahlen 100
2.2.3 Beliebig verteilte Zufallszahlen 101
2.2.4 Summen von Zufallsvariablen – Der Grenzwertsatz 103

2.3 Die Normalverteilung ... 103
2.4 Grafische Methoden der statistischen Analyse ... 105
2.4.1 Das Histogramm ... 105
2.4.2 Das Streudiagramm ... 109
2.5 Lage-, Streu- und Formparameter in der Statistik ... 112
2.5.1 Der arithmetische Mittelwert ... 112
2.5.2 Der Median ... 113
2.5.3 Die Spannweite ... 114
2.5.4 Die mittlere absolute Abweichung ... 115
2.5.5 Die Standardabweichung und die Varianz ... 115
2.5.6 Die Schiefe ... 117
2.5.7 Die Kurtosis ... 117
2.6 Stichprobe und Grundgesamtheit ... 122
2.7 Standardisierte Maßzahlen – Die *z*-Transformation ... 123
2.8 Die Korrelationsanalyse ... 124
2.8.1 Empirische Korrelation ... 124
2.8.2 Korrelation im Streudiagramm ... 130
2.8.3 Korrelation und Kausalität ... 132
2.9 Die Regressionsanalyse ... 133
2.9.1 Lineare Regression ... 133
2.9.2 Regression einer allgemeinen Polynomfunktion ... 137
2.9.3 Regression einer Exponentialfunktion ... 143
2.9.4 Regression einer Potenzfunktion ... 145
2.10 Rangordnungsfilter ... 148

3 Computational Intelligence ... 157
3.1 Einführung ... 157
3.2 Neuronale Netze ... 159
3.2.1 Biologische Grundlagen ... 159
3.2.2 Künstliche Neuronen ... 161
3.2.3 Netzstrukturen ... 163
3.2.4 Der Backpropagation-Lernalgorithmus ... 166
3.2.5 Lerndatenaufbereitung ... 176
3.2.6 Die Lerndatei ... 177
3.2.7 Anwendungen neuronaler Netze ... 181
3.3 Fuzzy-Logik ... 187
3.3.1 Die unscharfe Menge ... 188
3.3.2 Unscharfes Schließen ... 191
3.3.3 Die Struktur von Fuzzy-Systemen ... 195
3.3.4 Implementierung von Fuzzy-Systemen ... 197
3.4 Neuronales Netz oder Fuzzy-System? ... 213

4 Simulationstechnik 215
4.1 Einführung 215
4.2 Modellbildung 215
4.3 Die analytische Lösung 217
4.4 Numerische Lösungsmethoden von Differentialgleichungen 220
4.4.1 Das Polygonzug-Verfahren 222
4.4.2 Das verbesserte Polygonzug-Verfahren 223
4.4.3 Das Euler-Cauchy-Verfahren 224
4.4.4 Das Runge-Kutta-Verfahren 225
4.4.5 Das Adams-Bashforth-Verfahren 227
4.4.6 Die Wahl der Methode 229
4.4.7 Mehrdimensionale Betrachtungen 230
4.4.8 Schrittweitensteuerung 243
4.4.9 Differentialgleichungen höherer Ordnung 243
4.5 Monte-Carlo-Methoden 245

5 Netzwerke 249
5.1 Einführung 249
5.2 Historische Entwicklung 249
5.3 Physikalische Grundlagen 252
5.3.1 Leitungsgebundene Signalübertragung 253
5.3.2 Drahtlose Signalübertragung 254
5.3.3 Optische Signalübertragung 256
5.4 Netzwerkstrukturen 259
5.4.1 Sternstruktur 259
5.4.2 Ringstruktur 260
5.4.3 Busstruktur 260
5.4.4 Vermaschte Struktur 261
5.5 Netzwerkkomponenten 261
5.5.1 Repeater 262
5.5.2 Gateways 262
5.5.3 Router 262
5.6 TCP/IP und UDP/IP 262
5.7 Sockets 263
5.8 Das Client-Server-Modell 264

6 Anhang 271
6.1 Lineare Gleichungssysteme 271
6.1.1 Die Gauß-Elimination 272

6.2 Sortieren 288
6.2.1 Sortieren durch Vertauschen – Selectionsort 288
6.2.2 Quicksort 289
6.2.3 Vergleich beider Verfahren 291

Literaturverzeichnis 297

Sachverzeichnis 301

1 Digitale Signalverarbeitung

1.1 Einführung

Es besteht gar kein Zweifel, wir leben in einer Informationsgesellschaft. Ganz egal, ob wir Musik hören, mit dem Auto unterwegs sind oder mit guten Freunden telefonieren, ständig kommen wir mit Systemen der digitalen Signalverarbeitung in Berührung. Ein großer Teil der Information die uns erreicht ist bereits in mehreren vorangegangenen Schritten digital verarbeitet worden, auch wenn wir uns dessen nicht immer bewußt sind. Träger der Information sind Signale, die im mathematischen Sinne Funktionen einer oder mehrerer unabhängiger Variablen (Zeit, Raumkoordinaten) sind. Die Aufgabe der Signalverarbeitung besteht vor allem in der Manipulation, Analyse und Interpretation von Signalen. Sie kann zu Recht als eine entscheidende Schlüsseltechnologie des modernen Lebens gesehen werden. Interessanterweise wurden die mathematischen Grundlagen hierzu bereits in einer Zeit erarbeitet, als an Computer, Unterhaltungselektronik oder auch nur die Elektrifizierung der privaten Haushalte noch nicht einmal zu denken war.

Einer der Wegbereiter war Jean-Baptiste Joseph Fourier (1768–1830), der im französischen Auxerre als Sohn eines Schneiders geboren wurde. Er besuchte u. A. die dortige Militärschule, wo sein mathematisches Interesse geweckt wurde. Hier machte er durch seine Studien zur Mathematik und Mechanik auf sich aufmerksam. Trotzdem entschied Fourier sich zunächst, nicht unüblich für die damalige Zeit, für das Priesteramt und begann eine entsprechende Ausbildung in der Abtei St. Benoit-sur-Loire. Er mußte sich aber bald eingestehen, daß sein Herz doch mehr für die Mathematik schlug. Zurück in Auxerre arbeitete er als Mathematiklehrer. Die Wirren der Französischen Revolution gingen auch an Fourier nicht spurlos vorüber und hätten ihm beinahe, im wahrsten Sinne des Wortes, Kopf und Kragen gekostet. 1795 zog er nach Paris und vollendete seine Studien an der École Polytechnique, an der er später auch selber lehrte. Fourier war Mitglied der Académie des Sciences und der Académie Française. Sein Ruhm gründet vor allem auf Arbeiten zur Mathematik und zur mathematischen Physik. 1822 entstand sein Hauptwerk, die „Théorie analytique de la chaleur“, in dem er den Wärmetransport und die Temperaturverteilung im

Abb. 1.1. Jean-Baptiste Fourier

Inneren homogener Körper mit Hilfe von partiellen Differentialgleichungen beschreibt.

1.2 Fourier-Reihen

Fourier verwendete in seiner Arbeit trigonometrische Reihen, die man heute als *Fourier-Reihen* kennt. Er erkannte, daß sich periodische Funktionen in einfache Basisfunktionen zerlegen lassen. Die von ihm verwendeten trigonometrischen Funktionen *Sinus* und *Cosinus* bilden gerade einen hierfür geeigneten Funktionsbaukasten. Die notwendige Periodizität liegt genau dann vor, wenn für alle ganzzahligen n die folgende Bedingung erfüllt ist.

$$x(t) = x(t + nT) \tag{1.1}$$

Das bedeutet, der Funktionsverlauf wiederholt sich mit der Periodendauer T, deren Kehrwert bekanntlich die Frequenz ist.

$$f = \frac{1}{T} \tag{1.2}$$

Anstelle der Frequenz f wird jedoch für periodische Vorgänge üblicherweise die sogenannte Kreisfrequenz ω verwendet.

$$\omega = 2\pi f = \frac{2\pi}{T} \tag{1.3}$$

Mit diesem Rüstzeug gelingt die Darstellung einer periodischen Funktion als unendliche Summe der trigonometrischen Basisfunktionen.

$$x(t) = a_0 + \sum_{n=1}^{\infty} (a_n \cos(n\omega t) + b_n \sin(n\omega t)) \tag{1.4}$$

mit

$$a_0 = \frac{1}{T} \int_T x(t)dt \tag{1.5}$$

$$a_n = \frac{2}{T} \int_T x(t) \cos(n\omega t)\, dt \tag{1.6}$$

$$b_n = \frac{2}{T} \int_T x(t) \sin(n\omega t)\, dt \tag{1.7}$$

Der Koeffizient a_0 ist der Mittelwert der Zeitfunktion $x(t)$, stellt also den darin enthaltenen Gleichanteil dar. Die Amplituden a_n und b_n repräsentieren die Gewichtung der verschiedenen Frequenzen $n\omega$. Wie später noch gezeigt wird, kommt ihnen bei der Signalanalyse eine ganz besondere Bedeutung zu. In der praktischen Anwendung bricht man eine solche Reihe nach endlich vielen Gliedern ab und begnügt sich mit einer genügend genauen Approximation. Als Beispiel sei an dieser Stelle die Entwicklung einer Rechteckfunktion vorgestellt.

$$x(t) = \sum_{n=1}^{\infty} \frac{\sin((2k-1)\omega t)}{(2k-1)}$$

$$= \sin(\omega t) + \frac{\sin(3\omega t)}{3} + \frac{\sin(5\omega t)}{5} + \frac{\sin(7\omega t)}{7} + \ldots$$

In diesem einfachen Beispiel sind keine Cosinus-Anteile enthalten ($a_n = 0$ für alle n) und die Amplituden der Sinus-Anteile ergeben sich zu $b_1 = 1$, $b_2 = 0$, $b_3 = 1/3$, $b_4 = 0$, $b_5 = 1/5$, $b_6 = 0$, $b_7 = 1/7$, ... usw. Die gewünschte Annäherung an den idealen Funktionsverlauf gelingt aber erst mit einer sehr großen Anzahl von Reihengliedern, wie die nachfolgende Abbildung deutlich macht.

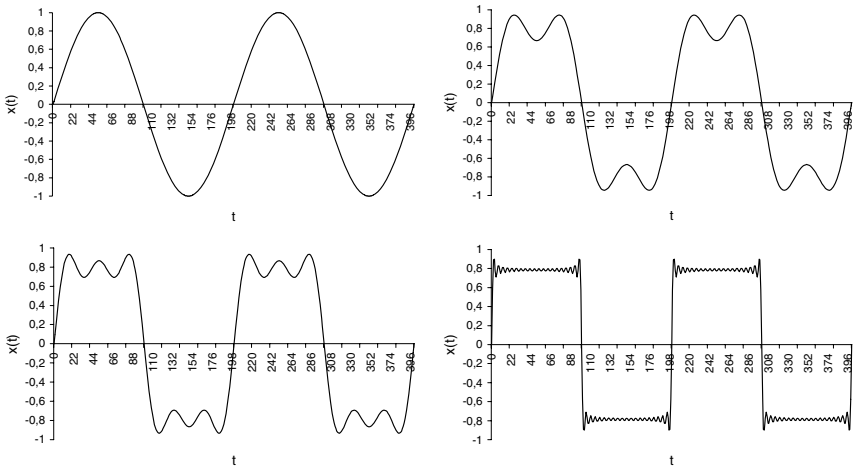


Abb. 1.2. Entwicklung einer Rechteckfunktion mit 1, 2, 3 und 20 Reihengliedern

Offensichtlich bereiten steile Signalflanken gewisse Probleme, denn zu beiden Seiten der Sprungstelle ist ein interessanter Effekt zu beobachten. Es zeigt sich ein deutliches Überschwingen, dessen Amplitude auch mit steigender Anzahl von Summanden nicht abnimmt. Dieses Verhalten ist als *Gibbsches Phänomen* bekannt. Die Ursache liegt in der letztlich doch immer nur endlichen Anzahl von Reihengliedern bei einer solchen Approximation. Auch das zweite Beispiel in Tabelle 1.1 zeigt diesen Effekt.

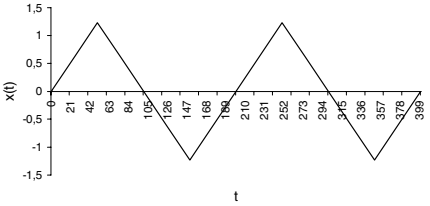
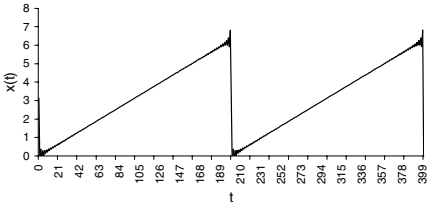
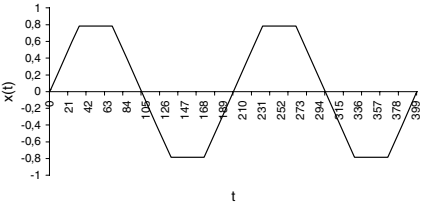
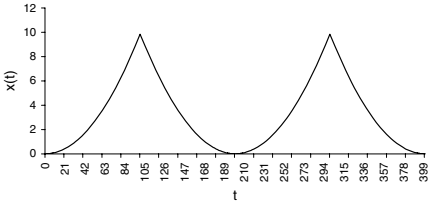
Neben der hier vorgestellten reellen Fourier-Reihe existiert noch eine komplexe Version, die man durch Anwendung der Eulerschen Formel ($e^{j\varphi} = cos\varphi + j \cdot sin\varphi$) erhält. Die resultierende Reihe kann als orthogonale Entwicklung von $x(t)$ nach einem System komplexer Exponentialfunktionen verstanden werden.

$$x(t) = \sum_{n=-\infty}^{+\infty} c_n e^{jn\omega t} \tag{1.8}$$

$$c_n = \frac{1}{T} \int_T x(t)\, e^{-jn\omega t}\, dt \tag{1.9}$$

Die von Fourier beschriebene Reihenentwicklung ist nun keineswegs die einzige Möglichkeit eine Funktion $x(t)$ mit Hilfe elementarer Basisfunktionen darzustellen. Vielmehr gibt es beliebig viele solcher Funktionssysteme, die als *Aufbaufunktionen* dienen können, vergleichbar mit einem Vektor, der ebenfalls in verschiedenen Koordinatensystemen (rechtwinklig, schiefwinklig, Polarkoordinaten, ...) darstellbar ist. Allerdings kommt

Tabelle 1.1. Beispiele für Fourier-Reihen

$x(t) = \sum_{n=1}^{\infty} (-1)^{n+1} \cdot \frac{\sin((2n-1)\omega t)}{(2n-1)^2}$	
$x(t) = \pi - 2\left(\sum_{n=1}^{\infty} \frac{\sin(n\omega t)}{n}\right)$	
$x(t) = \sum_{n=1}^{\infty} \frac{\sin(2n-1) \cdot \sin((2n-1)\omega t)}{(2n-1)^2}$	
$x(t) = \frac{\pi^2}{3} - 4\left(\sum_{n=1}^{\infty} (-1)^{n+1} \cdot \frac{\cos(n\omega t)}{n^2}\right)$	

es in einer Anwendung ganz entschieden darauf an, die jeweils günstigste Darstellung zu wählen. Das gilt auch für die Wahl des Funktionssystems bei der Approximation von $x(t)$. Fourier verwendete für seine periodischen Funktionen die trigonometrischen Basisfunktionen Sinus und Cosinus, und er tat gut daran. Wie sich wahrscheinlich jeder leicht vorstellen kann, ist es immer günstig gerade solche Aufbaufunktionen zu wählen, die eine gewisse Ähnlichkeit mit $x(t)$ aufweisen, da die Approximation dann besonders schnell konvergiert. Eine derartige Reihenentwicklung läßt sich ganz allgemein formulieren:

$$x(t) = \sum_{n=0}^{\infty} a_n \varphi_n(t) \tag{1.10}$$

Wie auch schon bei der Fourier-Reihe, ist $x(t)$ eine Linearkombination der Aufbaufunktionen $\varphi_n(t)$, wobei es gilt, die Entwicklungskoeffizienten a_n zu bestimmen. Hierzu sind beide Seiten der Gleichung mit den konjugiert komplexen Funktionen $\varphi^*_k(t)$ zu multiplizieren und über das Intervall T zu integrieren. Der Rechenaufwand vereinfacht sich erheblich, wenn das gewählte Funktionssystem $\varphi_n(t)$ die *Orthogonalitätsbedingung* erfüllt.

$$\int_T \varphi_n(t) \cdot \varphi_m^*(t) dt = \begin{cases} 0 & (n \neq m) \\ k_n & (n = m) \end{cases} \tag{1.11}$$

Sind zusätzlich alle $k_n = 1$, ist $\varphi_n(t)$ als orthonormal. Die Multiplikation von Gln. (1.10) mit $\varphi^*_k(t)$ und anschließende Integration führen auf die nachfolgende Beziehung.

$$\int_T x(t) \cdot \varphi_m^*(t) dt = \int_T \sum_{n=0}^{\infty} a_n \varphi_n(t) \cdot \varphi_m^*(t) dt$$

Vorausgesetzt die Summe (1.10) ist gleichmäßig konvergent und man beachtet die Bedingung (1.11), folgt hieraus die Formel zur Berechnung der gesuchten Koeffizienten.

$$a_n = \frac{1}{k_n} \int_T x(t) \cdot \varphi_n^*(t) dt \tag{1.12}$$

Neben den trigonometrischen Aufbaufunktionen sind noch die 1923 von J. L. Walsh definierten *Walsh-Funktionen* und die nur ein Jahr zuvor von H. Rademacher entwickelten *Rademacher-Funktionen* von praktischer Bedeutung. Beide sind Erweiterungen der 1910 von A. Haar vorgestellten orthogonalen Funktionsbasis, den sogenannten *Haar-Funktionen*.

1.3 Die Diskrete Fourier-Transformation (DFT)

Schaut man auf das bisher Gezeigte, liegt die Vermutung nahe, daß es auch eine Möglichkeit gibt, den umgekehrten Weg zu beschreiten, also eine Funktionsapproximation wieder in ihre erzeugenden Basisfunktionen zu zerlegen. Ein solches Vorgehen wäre dann geeignet, ein Signal in Bezug auf seine spektrale Zusammensetzung zu analysieren. Während man die Gleichungen (1.4) und (1.8) häufig auch als *Synthesegleichungen* bezeichnet, beschreiben die Gleichungen (1.5), (1.6), (1.7) und (1.9) das Signal im Frequenzbereich und werden daher entsprechend *Analysegleichungen* genannt. Ein Digitalrechner kann aber immer nur zeitdiskrete Werte erfassen und verarbeiten. Die zeitkontinuierlichen Signalverläufe realer

physikalischer Größen müssen infolgedessen diskretisiert werden. Möchte man eine Frequenzanalyse für ein Abtastsignal durchführen, müssen dann auch die Transformationen (1.6) und (1.7) zeitdiskret angepaßt werden. Dabei gehen die zeitkontinuierlichen Integrale der orthogonalen Komponenten, die den Realteil und den Imaginärteil des Signals repräsentieren, in zeitdiskrete Summen über.

$$a_n = \frac{2}{N}\sum_{k=0}^{N-1} x(k)\cdot\cos(2\pi n\frac{k}{N}) \tag{1.13}$$

$$b_n = \frac{2}{N}\sum_{k=0}^{N-1} x(k)\cdot\sin(2\pi n\frac{k}{N}) \tag{1.14}$$

$$\left|X_n\right| = \sqrt{a_n^2 + b_n^2} \tag{1.15}$$

$$\varphi_n = \arctan\left(\frac{b_n}{a_n}\right) \tag{1.16}$$

Aus diesen Summen können dann wiederum nach Gln. (1.15) das *Amplitudenspektrum*, oft auch Betragsspektrum genannt, und das *Phasenspektrum* nach Gln. (1.16) bestimmt werden. Hierbei stellt k die diskretisierten Abtastzeitpunkte dar, N die Anzahl der Abtastwerte je Periode der Grundfrequenz und n die gerade betrachtete Frequenz. Sie sind somit die diskreten Gegenstücke zu den zeitkontinuierlichen Größen t, T und ω. Insgesamt erfordert die Frequenzanalyse eines abgetasteten Signals mit Hilfe der vorgestellten reellen *DFT* nur einen vergleichsweise geringen Implementierungsaufwand und eignet sich damit ganz hervorragend für eigene experimentelle Untersuchungen.

```
class Spektrum
{
  const int Abtastwerte = 128;
  const double PI       = 3.141592653589793;

  // Signalverlauf
  public double[] x  = new double[Abtastwerte];
  // Realteil und Imaginärteil
  public double[] Re = new double[Abtastwerte];
  public double[] Im = new double[Abtastwerte];
  // Amplitudenspektrum und Phasenspektrum
  public double[] AS = new double[Abtastwerte];
  public double[] PS = new double[Abtastwerte];
```

```
  ...
  public void DFT()
  {
    // über alle Frequenzen
    for (int f=0; f<Abtastwerte; f++)
    {
      Re[f] = 0;
      Im[f] = 0;

      // über alle Abtastwerte
      for (int k=0; k<Abtastwerte; k++)
      {
        Re[f] += x[k]*Math.Cos(2*PI*f*k/Abtastwerte);
        Im[f] += x[k]*Math.Sin(2*PI*f*k/Abtastwerte);
      }
      Re[f] *= 2.0/Abtastwerte;
      Im[f] *= 2.0/Abtastwerte;

      AS[f] = Math.Sqrt(Re[f]*Re[f] + Im[f]*Im[f]);
      PS[f] = Math.Atan2(Im[f], Re[f]);
    }
    Re[0] /= 2.0; // Korrektur Gleichanteil
    AS[0] = Re[0];
  }
  ...
}
```

Die Korrektur des Gleichanteils wird vorgenommen, da in der Regel a_0 nicht nach Gln. (1.5) berechnet wird, sondern nach Gln. (1.13). Man kann bei der Implementierung aber auch auf die Korrektur verzichten. In der Praxis kommt das sogar relativ häufig vor, wohlwissend, daß die Spektrallinie AS[0] dann dem doppelten Gleichanteil des Signals entspricht.

In den meisten Fällen ist es völlig ausreichend, sich bei der Darstellung des Signalspektrums auf das Amplitudenspektrum zu beschränken. Es ist für jeden Signalabschnitt identisch und dadurch von eventuellen zeitlichen Verschiebungen des periodischen Signals unabhängig. Ganz anders sieht das aber beim Phasenspektrum aus, das sich aus dem Quotienten von Imaginärteil und Realteil berechnet. Wie man leicht nachvollziehen kann, ist das Phasenspektrum sehr wohl abhängig von dem gewählten zeitlichen Bezugspunkt. Aus diesem Grund ist es sehr viel weniger aussagekräftig und trägt nicht selten sogar eher zur Verwirrung bei. In den folgenden Beispielen betrachten wir daher ausschließlich das Amplitudenspektrum.

Sollte es die Aufgabenstellung erfordern den zeitlichen Verlauf eines Signals aus einem vorgegebenen Spektrum zu bestimmen, ist das ebenfalls

leicht möglich und vollzieht sich in zwei Schritten. Aus dem Amplitudenspektrum ist zwar sofort erkennbar, wie stark die einzelnen Frequenzen vertreten sind, es ist aber zunächst nicht klar, ob diese als Sinus- oder als Cosinusanteil in den Signalverlauf eingehen. Wie schon bei der Signalanalyse, sind daher durch Korrelation mit den trigonometrischen Aufbaufunktionen der Realteil und der Imaginärteil des Spektrums zu ermitteln. Mit dessen orthogonalen Komponenten als Gewichtsfaktoren gelingt dann die Synthese des zeitlichen Verlaufs. Eine erste Implementierungsvariante der inversen DFT (*IDFT*) zeigt das folgende Code-Beispiel.

```
class Spektrum
{
  ...
  public void InverseDFT_Variante1()
  {
    // über alle Frequenzen
    for (int f=0; f<Abtastwerte; f++)
    {
      // Berechnung der orthogonalen Komponenten von
      // XS(f) durch Korrelation mit Cos und Sin
      Re[f] += XS[f]*Math.Cos(2*PI*f/Abtastwerte);
      Im[f] += XS[f]*Math.Sin(2*PI*f/Abtastwerte);

      // Synthese von x[k] durch Gewichtung
      // der orthogonalen Komponenten von XS(f)
      // mit den Aufbaufunktionen
      for (int k=0; k<Abtastwerte; k++)
      {
        x[k] += Re[f]*Math.Cos(2*PI*f*k/Abtastwerte)+
                Im[f]*Math.Sin(2*PI*f*k/Abtastwerte);
      }
    }
  }
  ...
}
```

Wie schon bei der DFT sind auch bei der IDFT zwei geschachtelte Schleifen für die Transformation notwendig. Die Berechnung der orthogonalen Spektralkomponenten erfolgt zunächst in der äußeren Schleife, die Synthese des zeitlichen Verlaufs läuft dann in der inneren Schleife ab. Ähnlich wie bei einer mathematischen Formel kann man aber auch zweckmäßig umformen. In unserer zweiten Implementierung vereinfacht sich dadurch die Synthese zur simplen Addition.

```
class Spektrum
{
  ...
  public void InverseDFT_Variante2()
  {
   for (int k=0; k<Abtastwerte; k++)
   {
    Re[k] = 0;
    Im[k] = 0;

    for (int f=0; f<Abtastwerte; f++)
    {
     Re[k] += XS[f]*Math.Cos(2*PI*k*f/Abtastwerte);
     Im[k] += XS[f]*Math.Sin(2*PI*k*f/Abtastwerte);
    }

    x[k] = Re[k] + Im[k];
   }
  }
  ...
}
```

Diese zweite Variante der IDFT ähnelt der DFT doch sehr auffällig. Das ist natürlich kein Zufall, sondern liegt in der schlichten Tatsache begründet, daß die Hin- bzw. Rücktransformation einander sehr ähnlich sind. Allgemein läßt sich das an den Gleichungen (1.10) und (1.12) erkennen, speziell für die Fourier-Transformation an den Gleichungen (1.8) und (1.9). Diskretisiert man hier die Analysegleichung indem man das Integral durch die Summe ersetzt, unterscheiden sich Hin- bzw. Rücktransformation nur durch eine Normierung und verschiedene Vorzeichen in der komplexen Exponentialfunktion. Eine komplexe Exponentialfunktion ist aber nichts weiter als ein Drehzeiger in der komplexen Zahlenebene, wobei das Vorzeichen die Drehrichtung angibt. Ein Wechsel des Vorzeichen ist hier etwa vergleichbar mit der Summation von links nach rechts, oder umgekehrt. Das Ergebnis ist in jedem Fall genau dasselbe. Somit sind auch die Implementierung der DFT und der IDFT im Kern identisch. In Anwendungen, in denen tatsächlich auch beide Transformationen benötigt werden, sind diese dann üblicherweise in einer einzigen Funktion realisiert. Ein zusätzlicher Parameter beim Funktionsaufruf dient zur Unterscheidung. Aus optischen Gründen ist es dann zweckmäßig anstelle von `x[]` und `XS[]` neutrale Bezeichner wie `data1[]` und `data2[]` zu verwenden. In keinem Fall darf man jedoch vergessen, daß die Ergebnisse im Frequenz- bzw. Zeitbereich entsprechend unterschiedlich skaliert sind.

1.4 Arithmetiktuning – Die FFT

Das Frequenzspektrum einer Folge von Abtastwerten zu berechnen, ist jetzt kein Problem mehr. Mit Hilfe der DFT können wir auf die erweiterten Möglichkeiten einer Meßwertanalyse im Frequenzbereich zurückgreifen. Doch wieviel Rechenzeit darf eine solche Analyse in Anspruch nehmen? Oft ist die DFT nur ein Glied in einer ganzen Kette von Verarbeitungsschritten. Bei zeitkritischen Anwendungen, wie z. B. der Sprachsignalverarbeitung, kommt es dann schon darauf an, den Rechenzeitbedarf insgesamt möglichst gering zu halten. Erfahrungsgemäß ist das Einsparpotential gerade dort am größten, wo auch die meiste Rechenzeit verbraucht wird.

Tatsächlich ist der Rechenzeitbedarf in vielen Anwendungen der digitalen Signalverarbeitung ein latentes Problem, dem man grundsätzlich auf zwei Arten begegnen kann. Einerseits durch Optimierung der verwendeten Algorithmen, als Softwareentwickler können wir hier unmittelbar Einfluß nehmen, andererseits durch die Verwendung einer entsprechend leistungsstarken Hardwarebasis, wobei jedoch der Kostenrahmen des Projektes meist enge Grenzen setzt. Es liegt in der Natur der Sache, daß beide Ansätze das Problem immer nur kurzzeitig entschärfen, aber nicht wirklich dauerhaft lösen können, da sich mit den dann zusätzlichen Möglichkeiten auch immer komplexere Aufgabenstellungen finden, die diesen Freiraum schnell wieder aufzehren. Somit bleibt die Rechenzeitproblematik mit großer Sicherheit auch zukünftig bestehen und jeder Entwickler ist gut beraten, dies in seinem Projekt möglichst frühzeitig zu berücksichtigen.

Viele Wege führen nach Rom. Und wahrscheinlich gibt es genauso viele Berechnungsvarianten für die DFT einer Abtastfolge. Eine Variante, die sogenannte Fast-Fourier-Transformation (*FFT*), hat sich jedoch einen ganz besonderen Stellenwert innerhalb der digitalen Signalverarbeitung erobert. Sie gehört ohne Zweifel zur „Königsklasse“ der Signalverarbeitungsalgorithmen und ist wohl auch einer ihrer kompliziertesten Vertreter. Hiervon sollte man sich aber nicht weiter beeindrucken lassen. Im Grunde ist alles doch ganz einfach.

Wie wir zuvor gesehen haben, sind für die Berechnung einer DFT mit N Abtastwerten $N \cdot N = N^2$ komplexe Multiplikationen notwendig. Man kann sich leicht vorstellen, daß eine solche quadratische Abhängigkeit in Anwendungen mit großen Abtastraten Probleme schafft. In solchen Fällen hat sich das Prinzip *Teile-und-Herrsche* (divide-and-conquer) bewährt. Eine aufwendige Berechnung wird in viele kleine Teilberechnungen zerlegt und deren Teilergebnisse anschließend zu einem Gesamtergebnis zusammengeführt. Das sich dieses Vorgehen auch für eine schnellere Berechnung der DFT eignet, wurde schon frühzeitig erkannt, lange bevor es überhaupt

Computer gab. Aber erst die elektronische Rechentechnik hat dieser Idee zum Durchbruch verholfen. Heute werden insbesondere die Arbeiten von Danielson und Lanczos aus dem Jahre 1942, sowie von Cooley und Tukey aus dem Jahre 1965 als Wiederentdeckung der FFT gesehen.

Um das Teile-und-Herrsche-Prinzip zu verdeutlichen betrachten wir zunächst die diskrete Fouriertransformation. Diese läßt sich durch Zerlegung in gerade und ungerade Abtastwerte wie folgt umschreiben:

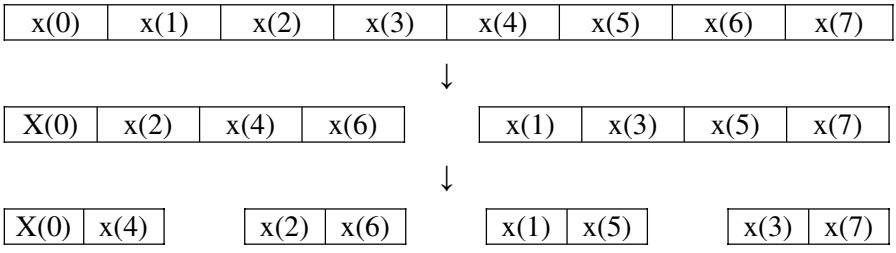
$$X_n = \sum_{k=0}^{N-1} x(k) \cdot e^{-\frac{j2\pi nk}{N}}$$

$$X_n = \sum_{k=0}^{N/2-1} x(2k) \cdot e^{-\frac{j2\pi n(2k)}{N}} + \sum_{k=0}^{N/2-1} x(2k+1) \cdot e^{-\frac{j2\pi n(2k+1)}{N}}$$

$$X_n = \sum_{k=0}^{N/2-1} x(2k) \cdot e^{-\frac{j2\pi nk}{N/2}} + \left(\sum_{k=0}^{N/2-1} x(2k+1) \cdot e^{-\frac{j2\pi nk}{N/2}} \right) \cdot e^{-\frac{j2\pi n}{N}}$$

Eine DFT der Länge N kann also auch aus zwei DFTs der halben Länge berechnet werden, wenn man sie anschließend phasenrichtig addiert. Was das bringt liegt auf der Hand. Anstelle einer N^2 Berechnung hat man es nur noch mit $2 \cdot (N/2) \cdot (N/2) = N^2/2$ Multiplikationen zu tun, was den Aufwand glatt halbiert. Aber es kommt noch besser. Dieses Vorgehen läßt sich rekursiv fortführen, solange die verbleibenden Teilintervalle durch Zwei teilbar sind. Theoretisch könnte man so weitermachen, bis man N DFTs aus N Einzelwerten zu berechnen hat. Das Ergebnis wäre dann der jeweilige Wert selbst. In der Praxis hat es sich aber durchgesetzt zwei Abtastwerte für eine Minimal-DFT zu verwenden. So eine Minimal-DFT wird dann als *Butterfly* bezeichnet.

Natürlich ändert sich durch die fortlaufende Zerlegung in gerade und ungerade Teilfolgen auch die Reihenfolge unserer ursprünglich zeitlich geordneten Abtastwerte. Die Indizes sind nicht mehr aufsteigend sortiert, sondern werden nach einem neuen Schema umgruppiert, wie das nachstehende Beispiel mit $N = 8$ Abtastwerten zeigt.



Der erste Schritt zerlegt die acht Abtastwerte in zwei Teilfolgen mit jeweils vier Werten. In einem zweiten Schritt werden diese nun ihrerseits zerlegt und wir erhalten insgesamt vier Teilfolgen, bestehend aus je zwei Werten. Dieser Vorgang ist an sich nicht weiter kompliziert. Sehr viel schwieriger ist es allerdings die Gesetzmäßigkeit hinter der resultierenden Neuordnung zu erkennen, zumindest solange wir die Indizes ausschließlich als Dezimalzahlen betrachten. Im Dualsystem dagegen offenbart sich das Muster sehr schnell. Der Index 1 (binär 001) wird mit dem Index 4 (binär 100) getauscht, der Index 3 (binär 011) mit der 6 (binär 110), usw., was gerade einer *Bit-Umkehr* (bit reversal) entspricht.

0	=	(000)	→	(000)	=	0
1	=	(001)	→	(100)	=	4
2	=	(010)	→	(010)	=	2
3	=	(011)	→	(110)	=	6
4	=	(100)	→	(001)	=	1
5	=	(101)	→	(101)	=	5
6	=	(110)	→	(011)	=	3
7	=	(111)	→	(111)	=	7

Mit Hilfe der Bit-Umkehr ist es also möglich, die oben beschriebene Zerlegung in gerade und ungerade Teilfolgen nicht explizit durchführen zu müssen, aber dennoch auf identisch angeordnete Abtastwerte zugreifen zu können.

Wir erinnern uns, der ursprüngliche Sinn der Zerlegung war es, mehrere Teil-DFTs zu berechnen und diese dann zum richtigen Gesamtergebnis zusammenzuführen, um Aufwand zu sparen. Die Zusammenführung ist aber gerade die Umkehrung der oben vorgenommenen Zerlegung. Hierbei sind zuerst vier 2-Punkte DFTs aus den Paaren $x(0)$ und $x(4)$, $x(2)$ und $x(6)$, $x(1)$ und $x(5)$ und schließlich $x(3)$ und $x(7)$ zu bilden. Als Ergebnis erhalten wir vier 2-Punkte Spektren, die nun ihrerseits als Eingangswerte für zwei 4-Punkte DFTs dienen. Die beiden resultierenden 4-Punkte Spektren werden abschließend noch einer 8-Punkte DFT unterzogen, und schon haben wir unser Endergebnis. Zugegeben, die Interpretation der Zwischenergebnisse als Spektren ist ein wenig problematisch, schaut man aber auf den Berechnungsaufwand, dann ist das Ergebnis beeindruckend. Für unsere $N = 8$ Abtastwerte sind demnach nur noch $4 \cdot 2 + 2 \cdot 4 + 1 \cdot 8 = 24$ komplexe Multiplikationen nötig, was doch ein ganz ordentliches Stück besser ist als die 64 bei der DFT. Allgemein sind mit dem hier vorgestellten FFT-Algorithmus $N \cdot log_2(N)$ Schritte erforderlich, gegenüber N^2 Schritte bei der normalen DFT.

Neben der Zerlegung bzw. Neuordnung der Abtastwerte ist die phasenrichtige Addition bei den Teil-DFTs die entscheidende Hürde zum Erfolg.

Wir wollen diese daher näher betrachten. In unserer nach geraden und ungeraden Abtastwerten zerlegten Beispielberechnung der DFT taucht ein zusätzlicher Faktor auf, der, abgesehen von der aktuellen Frequenz *f*, nur von der Anzahl der Abtastwerte *N* abhängt. Eine solche komplexe Exponentialfunktion bildet einen *Drehzeiger* (twiddle faktor) in der komplexen Zahlenebene. Genau dieser Zeiger ist das fehlende Glied, das für die Phasenkorrektur unserer Teil-DFT verantwortlich ist.

$$w_N = e^{-\frac{j2\pi f}{N}} \tag{1.17}$$

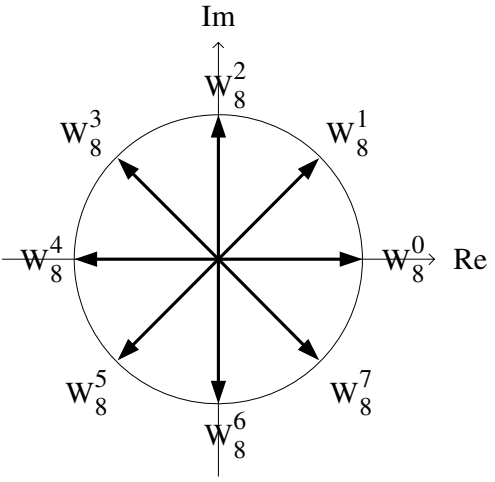


Abb. 1.3. Drehzeiger für $N = 8$

Die Zeiger liegen symmetrisch auf dem Einheitskreis ($r = 1$) und bilden die *N* Einheitswurzeln. Für die Implementierung wird es etwas einfacher, wenn man Gln. (1.17) unter Verwendung der trigonometrischen Funktionen umschreibt, wobei der Index *k* alle Zeiger von 0 bis $N-1$ durchläuft:

$$w_N^k = \cos\left(\frac{2\pi k}{N}\right) + j\sin\left(\frac{2\pi k}{N}\right) \tag{1.18}$$

In dieser Form lassen sich Real- und Imaginärteil des Drehzeigers direkt berechnen. Nutzt man seine Symmetrieeigenschaft, kann man zusätzlich Berechnungsaufwand sparen.

$$w_8^4 = -w_8^0$$

$$w_8^5 = -w_8^1$$

$$w_8^6 = -w_8^2$$

$$w_8^7 = -w_8^3$$

Die Neuordnung der Abtastwerte und der eingeführte Drehzeiger ermöglichen schließlich das effektive Berechnungsschema der FFT.

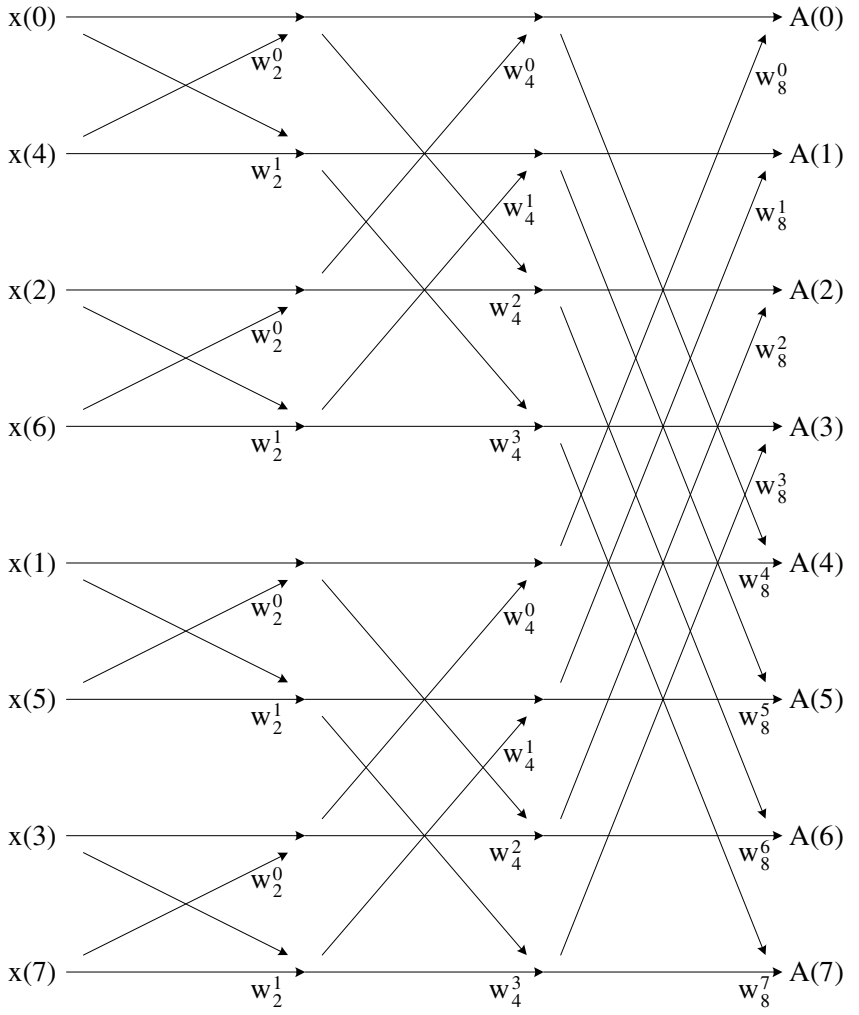


Abb. 1.4. Berechnungsschema für eine FFT mit 8 Abtastwerten

Bei acht Abtastwerten vollzieht sich die Berechnung in drei Stufen. Die Zusammenführung der Pfeile stellt eine Addition dar, ein beistehender Drehzeiger bedeutet eine Multiplikation mit dem entsprechenden Wert. Verständlicher wird es, wenn man einen einzelnen Butterfly betrachtet.

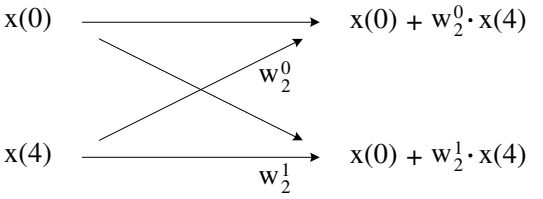


Abb. 1.5. Berechnungsschema eines einzelnen Butterfly

Für das Arbeiten mit komplexen Zahlen benötigt man noch einen entsprechenden Datentyp. Die Berechnungen selbst erfolgen dann jeweils getrennt nach Realteil und Imaginärteil.

```
struct komplex
{
 public double re;
 public double im;
}
```

Echte Meßwerte aus realen Prozessen sind immer reellwertig. In der Regel werden sie auch als solche gespeichert und weiterverarbeitet (üblicherweise noch mit Zeitstempel, physikalischer Einheit, usw.). Für die FFT ist daher die vorherige Umwandlung in komplexe Daten erforderlich.

Auch wenn man bis zu diesem Punkt alles verstanden hat, die wohl größte Schwierigkeit bei der Implementierung liegt ohne Zweifel darin, das Berechnungsschema der FFT über eine geeignete Indizierung tatsächlich umzusetzen. Hierzu führen wir drei Hilfsvariablen ein. Die Variable `stufen` bezeichnet die Anzahl der Berechnungsebenen im Schema und ergibt sich aus dem dualen Logarithmus der Zahl der Abtastwerte ($8 = 2^3$ → drei Stufen). Mit `sprung` wird die notwendige Inkrementierung des Indexes bezeichnet, um vom ersten Zweig des Butterflys zum zweiten Zweig zu gelangen (= 4 in der ersten Ebene: 0 → 4, 2 → 6, 1 → 5 und 3 → 7). Die Variable `schritt` stellt das notwendige Inkrement dar, um vom ersten Zweig eines Butterflys zum ersten Zweig des nächsten Butterflys zu springen (= 2 in der ersten Ebene: 0 → 2 und 1 → 3).

```
class Spektrum
{
  ...
  // komplexe Daten
  public komplex[] data = new komplex[Abtastwerte];

  public void FFT()
  {
    double tempr = 0; // für Tausch bei Bit-Umkehr
    double tempi = 0; // für Tausch bei Bit-Umkehr
    double wreal = 0; // Drehfaktor (Realteil)
    double wimag = 0; // Drehfaktor (Imaginärteil)
    double real1 = 0; // Hilfsvariable
    double imag1 = 0; // Hilfsvariable
    double real2 = 0; // Hilfsvariable
    double imag2 = 0; // Hilfsvariable
    int i = 0;
    int j = 0;
    int k = 0;
    int stufen  = 0;
    int sprung  = 0;
    int schritt = 0;
    int element = 0;

    // Bit-Umkehr
    for (j=0; j<Abtastwerte-1; j++)
    {
      if (j < i)
      {
         tempr = data[j].re;
         tempi = data[j].im;
         data[j].re = data[i].re;
         data[j].im = data[i].im;
         data[i].re = tempr;
         data[i].im = tempi;
      }

      k = Abtastwerte/2;
      while (k <= i)
      {
        i -= k;
        k /= 2;
      }

      i += k;
    }
```

```
stufen = (int)(Math.Log10((double)Abtastwerte)/
               Math.Log10((double)2));
sprung = 2;

for (i=0; i<stufen; i++)
{
  // jede Iterationsstufe startet mit dem 1. Wert
  element = 0;

  for (j=Abtastwerte/sprung; j>=1; j--)
  {
    schritt = sprung/2;

    for (k=0; k<schritt; k++)
    {
      // 1. Zweig des Butterfly
      wreal = Math.Cos(k*2.0*PI/sprung);
      wimag = Math.Sin(k*2.0*PI/sprung);

      real1 = data[element+k].re +
              (wreal*data[element+k+schritt].re -
               wimag*data[element+k+schritt].im);
      imag1 = data[element+k].im +
              (wreal*data[element+k+schritt].im +
               wimag*data[element+k+schritt].re);

      // 2. Zweig des Butterfly
      wreal *= -1.0;
      wimag *= -1.0;

      real2 = data[element+k].re +
              (wreal*data[element+k+schritt].re -
               wimag*data[element+k+schritt].im);
      imag2 = data[element+k].im +
              (wreal*data[element+k+schritt].im +
               wimag*data[element+k+schritt].re);

      // Ergebnisse übernehemen
      data[element+k].re = real1;
      data[element+k].im = imag1;
      data[element+k+schritt].re = real2;
      data[element+k+schritt].im = imag2;
    }
    element += sprung;
  }
  sprung *= 2;
```

```
    }
  }
  ...
}
```

Wie schon bei der DFT unterscheidet sich die inverse FFT (IFFT) von der FFT lediglich durch eine andere Normierung, bzw. Skalierung des Ergebnisses. Unser ursprüngliches Anliegen war es, Rechenzeit zu sparen. Geht man allein nach der Anzahl der benötigten Programmzeilen, dann scheinen wir von diesem Ziel weiter entfernt als zuvor. Aber wer wird sich schon von Äußerlichkeiten blenden lassen, was zählt sind schließlich die inneren Werte. Führt man sich vor Augen, wie viele komplexe Multiplikationen bei DFT bzw. FFT auszuführen sind, wird schnell klar, welches Verfahren die Nase vorn hat.

Ob sich die Einsparung an Rechenoperationen bei der FFT auch im gleichen Maße im Rechenzeitbedarf niederschlägt, läßt sich gut mit Hilfe eines Benchmark-Tests überprüfen. Hierbei wird die Ausführungszeit der Algorithmen miteinander verglichen. Um den relativen Fehler klein zu halten, wird pro Zeitmessung immer eine größere Anzahl von Durchläufen (z. B. 1000) ausgeführt. Wegen der Verwendung eines eigenen Datentyps, wurde bei der FFT die Zuweisung der Meßwerte mit in die Testschleife einbezogen. Genau genommen müßte man noch deren Leerlaufzeit abziehen, diese ist hier aber vernachlässigbar.

Tabelle 1.2. Anzahl der komplexen Multiplikationen bei DFT und FFT

N	DFT	FFT
2	4	2
4	16	8
8	64	24
16	256	64
32	1024	160
64	4096	384
128	16384	896
256	65536	2048
512	262144	4608
1024	1048576	10240

```
class Spektrum
{
  ...
  public void Benchmark()
  {
```

```
    const int testzyklen = 1000;

    DateTime start;
    DateTime stop;

    Console.WriteLine("Benchmark DFT startet jetzt!");
    start = DateTime.Now;

    for (int j=0; j<testzyklen; j++)
    {
      DFT();
    }

    stop = DateTime.Now;

    Console.WriteLine("Dauer = {0}", stop-start);
    Console.WriteLine();

    Console.WriteLine("Benchmark FFT startet jetzt!");
    start = DateTime.Now;

    for (int j=0; j<testzyklen; j++)
    {
      for (int k=0; k<Abtastwerte; k++)
      {
        data[k].re = x[k];
        data[k].im = 0;
      }
      FFT();
    }

    stop = DateTime.Now;

    Console.WriteLine("Dauer = {0}", stop-start);
    Console.ReadLine();
  }
  ...
}
```

Natürlich sind die gemessenen Zeiten in erster Linie von der Leistungsfähigkeit des ausführenden Computers abhängig. Eine normierte Darstellung der Werte ist in solchen Fällen hilfreich.

Tabelle 1.3. Gemessener Zeitbedarf von DFT und FFT (normiert)

N	DFT	FFT
2	1,3	1
4	5,7	2,6
8	24	7
16	94	18
32	355	44
64	1425	103
128	5703	242
256	22972	528
512	92551	1111
1024	371194	2778

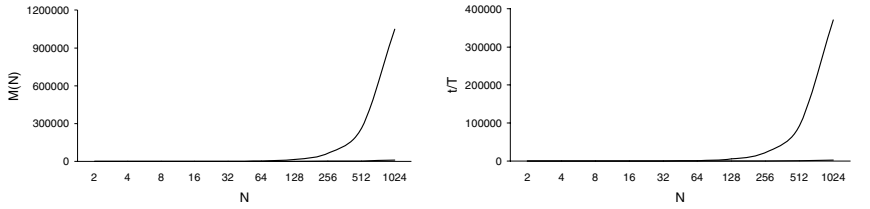


Abb. 1.6. Vergleich theoretischer und gemessener Zeitbedarf von DFT und FFT

Abbildung 1.6 zeigt die gute Übereinstimmung von Theorie und Praxis beim Rechenzeitbedarf beider Verfahren. Die Kurve für die FFT schmiegt sich so dicht an die *x*-Achse an, daß sie kaum sichtbar wird. Für kleine Abtastraten ist der Unterschied weit weniger dramatisch. Tatsächlich gibt es dort für die DFT noch erhebliches Einsparpotential, z. B. indem man die zeitaufwendigen trigonometrischen Funktionsaufrufe für alle k direkt durch ihren jeweiligen Wert ersetzt. Für höhere Abtastraten ist das aber kaum praktikabel, so daß sich die FFT etwa für $N > 16$ lohnt.

Vom Ansatz her produzieren DFT und FFT identische Ergebnisse. Die DFT berechnet die Frequenztransformierte einer diskreten Abtastfolge und genau das gleiche macht auch die FFT. Eigentlich ist sie ja auch nur eine (deutlich effizientere) Berechnungsvariante der DFT. Hierbei könnte man es belassen, wenn da nicht die Zahlendarstellung im Computer mit ihrem begrenzten Auflösungsvermögen wäre. Da es nicht möglich ist alle reellen Zahlenwerte in einem Rechner exakt abzubilden, geht jeder Berechnungsschritt mit einem gewissen Verlust an Präzision einher. Das Ergebnis der FFT ist aus diesem Grund zumindest theoretisch näher am „wahren Wert“, da sich mit der geringeren Anzahl von Berechnungen auch weniger Rundungsfehler aufsummieren. In der Praxis dürfte sich dieser Unterschied allerdings kaum bemerkbar machen. Wer es genauer wissen möchte, kann