Die schnelle Fouriertransformation

Proseminar "Rund um JPEG"
SS 2003
Universität Tübingen

Holger Szillat
szillat@informatik.uni-tuebingen.de
betreut durch Jürgen Schweizer

24. Juni 2003

Zusammenfassung

Die (naive) Berechnung der zur Fourier Transformation benötigten "Fourier Koeffizienten" hat einen Aufwand der Größe $O(N^2)$, welches es für große Eingabelängen praktisch unbrauchbar macht. Die "schnelle Fourier Transformation" (FFT) ermöglicht die Berechnung der Koeffizienten mit einem Aufwand von $O(N \log N)$, ist allerdings auf den ersten Blick etwas undurchschaubar und damit auch schwieriger zu implementieren. Dieser Vortrag versucht die schnelle Fourier Transformation mit Hilfe von Matrizen zu erläutern, in der Hoffnung das Verfahren dadurch verständlicher und die Implementierung einfacher zu machen.

Inhaltsverzeichnis

1 *Geschichte*

Joseph Fourier (1768-1830) verfaßte im Jahre 1807 seine erste Arbeit über die mathematische Behandlung von Wärmeleitungsgleichungen. Zur Lösung der Gleichungen benutzte er trigonometrische Reihen und schuf damit ein leistungsstarkes Werkzeug in der allgemeinen Theorie der partiellen Differentialgleichungen mit vorgegebenen Randbedingungen.

Andere Wissenschaftler seiner Zeit verwendeten ähnliche Verfahren um z.B. die Umlaufbahnen von Planeten zu berechnen. Euler erfand ein Verfahren, dessen Prinzip auch der schnellen Fourier Transformation zu Grunde liegt.

Die Fourier Transformation wird in fast allen wissenschaftlichen Disziplinen und in der Praxis eingesetzt. Weite Verbreitung findet die Fourier Transformation in der Signalverarbeitung, aber auch bei der Auswertung von Polynomen kann die schnelle Fourier Transformation eingesetzt werden.

Zum Einsatz mit dem Computer ist allerdings nur die diskrete Fourier Transformation sinnvoll, die nur eine endliche Anzahl an Stützstellen braucht. Das Problem bei diesem Verfahren ist allerdings die Berechnung der Fourier Koeffizienten: Prinzipiell sind immer Multiplikationen der Größe N^2 notwendig um alle Koeffizienten zu berechnen. Für große N ist dies natürlich unpraktikabel. Eine bessere Alternative bietet die "schnelle Fourier Transformation" (FFT) welche die Koeffizienten mit $N \log_2(N)$ Multiplikationen berechnet.

Der FFT-Algorithmus geht auf einen Artikel von James W. Cooley und John W. Tukey aus dem Jahre 1965 zurück, in dem ein Algorithmus zur schnellen Berechnung der diskreten Fouriertransformation vorgestellt wurde. Der Algorithmus funktioniert aber nur für Eingabegrößen, die eine Zweierpotenz sind.

Später stellte sich heraus, daß auch andere Wissenschaftler an ähnlichen Verfahren und Ideen arbeiteten.

Der FFT-Algorithmus von Cooley und Tukey arbeitet zwar nur für eine spezielle Klasse von Problemen optimal, dennoch existiert für fast jede Klasse ein "spezieller" FFT-Algorithmus, der für die jeweilige Problem-Klasse optimal zugeschnitten ist.

2 *Mathematisches*

Bevor man sich an die Implementierung machen kann, ist es notwendig, sich die mathematischen Grundlagen der Fourier Transformation nochmal ins Gedächtnis zu rufen.

2.1 Trigonometrische Approximation

In der Numerik wird die "trigonometrische Approximation" verwendet, um eine Funktion f durch die trigonometrischen Funktionen cos und sin möglichst gut anzunähern[1]. Das praktische Ziel dabei ist es, die "gesuchte", unbekannte Funktion f durch die "bekannten" Funktionen darzustellen und damit besser oder schneller zu berechnen oder Aussagen über die unbekannte Funktion anhand der "bekannten" Funktionen zu treffen.

Dazu betrachtet man zunächst den Vektorraum V der integrierbaren Funktionen auf dem Interval $[0; 2\pi]$:

$$V := \{f : [0; 2\pi] \to \mathbb{R} \mid f \text{ ist integrierbar und beschränkt}\}$$

Die Beschränkung der Funktionen auf das Intervall $[0; 2\pi]$ ist willkürlich. Durch Umskalieren der Definitionsbereiche ist es aber leicht jedes andere beschränkte Intervall zu betrachten.

Auf diesem Vektorraum V definiert man nun ein Skalarprodukt durch:

$$< f \mid g >:= \int_0^{2\pi} f(x)g(x)dx$$

Die dazugehörige Norm ist:

$$||f|| := \sqrt{< f \mid f >}$$

Diese Definitionen lassen sich auch auf den Vektorraum der komplexwertigen Funktionen anwenden. Dazu betrachtet man nun den Vektorraum der komplexwertigen Funktionen W, definiert durch:

$$W := \{f : [0; 2\pi] \to \mathbb{C} \mid f \text{ ist integrierbar und beschränkt}\}$$

Das Skalarprodukt ändert sich dann in:

$$< f \mid g >:= \int_0^{2\pi} \overline{f(x)}g(x)dx$$

Man erkennt leicht, daß $V \subseteq W$ und daß das Skalarprodukt auf V eigentlich dasselbe Skalarprodukt auf W ist, da für eine reellwertige Funktion f gilt: $\bar{f} = f$.

[1] Der Ausdruck "möglichst gut" bezieht sich dabei implizit auf eine geeignete Norm.

2.2 Basisfunktionen in $\mathbb{C}$

Damit nun eine beliebige Funktion $f \in V$ durch die Funktionen cos und sin approximiert werden kann, müssen cos und sin eine geeignete Basis bilden.

Allgemein ist eine Menge $b_i, 1 \leq i \leq n$ genau dann eine Orthonormalbasis für einen n-dimensionalen Vektorraum, wenn gilt:

$$\begin{aligned} < b_i \mid b_j > &:= \begin{cases} 1 & \text{für } i = j, \\ 0 & \text{sonst} \end{cases} \\ &= \delta_{ij} \end{aligned}$$

Nun läßt sich zeigen, daß die Funktionen cos und sin auf V folgende Eigenschaften haben:

$$\begin{aligned} \int_0^{2\pi} \cos(jx)\cos(kx)dx &= \begin{cases} 0 & \text{für alle } j \neq k \\ 2\pi & \text{für } j = k = 0 \\ \pi & \text{für } j = k > 0 \end{cases} \\ \int_0^{2\pi} \sin(jx)\sin(kx)dx &= \begin{cases} 0 & \text{für alle } j \neq k; j, k > 0 \\ \pi & \text{für } j = k > 0 \end{cases} \\ \int_0^{2\pi} \cos(jx)\sin(kx)dx &= 0 \quad \text{für alle } j \geq 0, k > 0. \end{aligned}$$

Das Problem ist aber, daß diese Funktionen leider keine Orthonormalbasis bilden, da für $i \neq j$ die Integrale ungleich 1 sind.

Um das Problem zu lösen braucht man eigentlich nur einen geeigneten Skalierungsfaktor. Dieser ist aber zunächst nicht offensichtlich.

Betrachtet man aber den "allgemeineren" Vektorraum W, kann man mit Hilfe der *Eulerschen Formel*[2] die etwas umständlichen Funktionen cos und sin handlicher darstellen:

$$e^{iz} = \cos(z) + i\sin(z), z \in \mathbb{Z}$$

Nun lassen sich die obigen Gleichungen einfacher darstellen:

$$\begin{aligned} < e^{ikt} \mid e^{ijt} > &= \frac{1}{2\pi} \int_0^{2\pi} e^{ikt}\overline{e^{ijt}}dt \\ &= \frac{1}{2\pi} \int_0^{2\pi} e^{i(k-j)t}dt \\ &= \begin{cases} \frac{1}{2\pi}\int_0^{2\pi} 1dt = 1 & \text{für } k = j \\ \frac{1}{2\pi}\frac{1}{i(k-j)} e^{i(k-j)t}|_0^{2\pi} = 0 & \text{für } k \neq j \end{cases} \end{aligned}$$

Definiert man nun:

[2]Die *Euler'sche Formel* ist eine moderne Fassung der *de Moivre'schen* Formel, siehe auch [RR95]

$$\Phi_k(z) := \frac{1}{\sqrt{2\pi}} e^{ikz}, k \in \mathbb{Z} \tag{1}$$

(Zu beachten ist hier, daß es sich bei den Φ_k um eine zweiseitige Folge von Funktionen handelt.), dann läßt sich leicht zeigen, daß gilt:

$$< \Phi_i \mid \Phi_j >= \delta_{ij} \tag{2}$$

Die Φ_i bilden jetzt also eine Orthonormalbasis. Damit ist es nun möglich, jede Funktion $f \in W$ als "Unendliche Linearkombination" dieser Basisfunktionen darzustellen.

$$f(x) = \sum_{k=-\infty}^{\infty} c_k \Phi_k(x) = \frac{1}{\sqrt{2\pi}} \sum_{k=-\infty}^{\infty} c_k e^{ikx} \tag{3}$$

Diese Reihen werden "Fourierreihen" genannt und machen natürlich nur Sinn, wenn sie konvergieren. Konvergieren sie, dann konvergieren sie gegen die "ursprüngliche" Funktion f.

3 *Die Berechnung der Fourier Transformation*

3.1 Die "*Diskrete Fourier Transformation*"

Mit Hilfe der *Fourierreihen* aus (3) läßt sich also eine beliebige Funktion (aus W) darstellen. Das einzige Problem sind nun nur noch die unbekannten Koeffizienten c_k, die sogenannten "*Fourierkoeffizienten*".

Sie sind einfach herzuleiten, indem man Gleichung (3) nach c_k auflöst:

$$c_k(f) := \int_{-\infty}^{\infty} f(x)e^{-ikx}dx,\ k \in \mathbb{Z} \tag{4}$$

Interessant ist vielleicht, daß die Umkehrung, die sog. "*Inverse Fourier Transformation*" es möglich macht, aus den Fourier Koeffizienten wieder die ursprünglichen Funktionswerte zu berechen:

$$f(t) = \int_{-\infty}^{\infty} c_k(f)e^{ikx}dx,\ k \in \mathbb{Z} \tag{5}$$

Das Problem, das sich nun aber ergibt ist, daß die Berechnung der Fourier Koeffizienten mit dem Computer in dieser Form natürlich nicht möglich ist (weil es unendlich viele sind).

Um also die Fourier Transformation mit dem Computer berechnen zu können, muß man sich zuerst auf den endlichen Fall zurückziehen.

Angenommen also, man hat eine Menge von N Datenpunkten der Funktion f auf einem beliebigen Intervall $[k_0; k_0 + (N-1)]$. Zur Vereinfachung sei angenommen, daß N gerade ist und daß die Datenpunkte äquidistant auf dem Intervall liegen:

$$f_k := f(t_k),\ t_k := k_0 + k\Delta,\ k = 0, 1, \ldots, N-1$$

(Die Funktion sei an den Stellen außerhalb des Intervalls $= 0$).

Dann läßt sich die Fourier Transformation F bei geeigneter Wahl von N, Δ und k_0 mit Hilfe von numerischen Integrationsformeln approximieren durch:

$$c_k(f) = \int_{-\infty}^{\infty} f(x)e^{-ixt}dx \approx \sum_{k=0}^{N-1} f_k e^{-ikt_k}\Delta$$

Diese letzte Summe wird die "Diskrete Fourier Transformation" (*DFT*) genannt; der Faktor Δ wird einfach weggelassen. Ferner kann das beliebige Intervall, auf dem die Funktion f approximiert wird, durch Verschiebung auf das Intervall $[0; 2\pi]$ abgebildet werden.

Nun lassen sich die Fourier Koeffizienten, die vorher nicht mit dem Computer zu berechnen waren, durch die folgende Gleichung berechnen:

$$F(k) = \sum_{n=0}^{N-1} f(n)e^{-2\pi ikn/N} \ \text{ mit } k = 0, \ldots, N-1 \tag{6}$$

(Um die "endlichen" Koeffizienten von den "unendlichen" zu unterscheiden, bezeichne ich die "endlichen" im Folgenden als $F(k)$, die "unendlichen" werden weiterhin mit c_k bezeichnet.)

Wieder lassen sich die ursprünglichen Funktionswerten aus den Koeffizienten zurückberechnen analog zu (5). Diese Umkehroperation heißt nun *inverse Diskrete Fourier Transformation* (*IDFT*):

$$f(n) = \sum_{k=0}^{N-1} F(k) e^{2\pi i n k/N} \quad \text{mit } k = 0, \ldots, N-1 \tag{7}$$

Da eine Basis existiert (siehe Gleichung (1)) kann man diese Transformation auch als Matrixmultiplikation schreiben:

$$F = DFT_N \; f^T$$

Der Vektor F enthält dann die Fourier Koeffizienten. Die Matrix DFT_N ist definiert als:

$$DFT_N := (e^{-2\pi i k l/N})_{0 \le k,l \le N-1} \tag{8}$$

Beispiele für die Fälle $N = 1, 2, 4$:

$$N = 1 : DFT_1 = \big[F(0)\big] = \big[1\big] \, \big[f(0)\big]$$

$$N = 2 : DFT_2 = \begin{bmatrix} F(0) \\ F(1) \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} +1 & +1 \\ +1 & -1 \end{bmatrix} \begin{bmatrix} f(0) \\ f(1) \end{bmatrix}$$

$$N = 4 : DFT_4 = \begin{bmatrix} F(0) \\ F(1) \\ F(2) \\ F(3) \end{bmatrix} = \frac{1}{\sqrt{4}} \begin{bmatrix} +1 & +1 & +1 & +1 \\ +1 & -i & -1 & +i \\ +1 & -1 & +1 & -1 \\ +1 & +i & -1 & -i \end{bmatrix} \begin{bmatrix} f(0) \\ f(1) \\ f(2) \\ f(3) \end{bmatrix}$$

Man beachte, daß hier mit Absicht die Fälle so gewählt wurden, daß $N = 2^p, p \in \mathbb{N}$ gilt. Wird N anders gewählt, gibt es natürlich auch eine Matrixdarstellung, doch ist diese etwas "unschöner" und für die "schnelle Fourier Transformation" unbrauchbar, wie sich noch zeigen wird.

Es läßt sich zeigen, daß die Matrix DFT_N nicht-singulär ist, d.h. $DFT_N^{-1} = DFT_N^*$, wodurch die Umkehrung der Abbildung leicht zu gewinnen ist, indem alle Einträge der Matrix komplex konjugiert werden. (Eine Spiegelung an der Diagonalen kann entfallen, da die Matrix symmetrisch ist.)

Die Darstellung als Matrix bedeutet auch, daß die *DFT* eine lineare Abbildung ist, d.h. sind $\tilde{f}$ und $\tilde{g}$ *DFT*en der Funktionen f und g, dann ist die *DFT* von $af(x) + bg(x)$ (mit a und b Konstanten): $a\tilde{f}(\omega) + b\tilde{g}(\omega)$.

Das Problem bei der Matrixdarstellung (und bei der damit verbundenen Berechnung) ist, daß die Berechnung mindestens N^2 Multiplikation notwendig sind. In der Informatik kann man einen solchen Aufwand getrost als "schlecht" ansehen.

Definiert man nun:

$$\omega_N := e^{-2\pi i/N}$$

so läßt sich die Matrix DFT_N etwas einfacher schreiben:

$$DFT_N := (\omega_N^{kl})_{0 \leq k,l \leq N-1} \tag{9}$$

Von Bedeutung für die spätere "schnelle Fourier Transformation" ist, daß diese ω_N die N-te Einheitswurzel in $\mathbb{C}$ bilden. D.h. die Potenzen ω_N^k liegen auf dem Einheitskreis und bilden die Eckpunkte eines regelmäßigen N-Ecks. Das hat zur Folge, daß sich die Exponenten modulo N reduzieren lassen.

Für $N = 4$ hat die Matrix DFT_4 die folgende Gestalt:

$$DFT_4 = \begin{pmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 \\ \omega^0 & \omega^3 & \omega^6 & \omega^8 \end{pmatrix} \tag{10}$$

$$= \frac{1}{\sqrt{4}} \begin{pmatrix} +1 & +1 & +1 & +1 \\ +1 & -i & -1 & +i \\ +1 & -1 & +1 & -1 \\ +1 & +i & -1 & -i \end{pmatrix} \in \mathbb{C}^{4\times 4} \tag{11}$$

Es gilt dabei: $\omega^1 = -i$, $\omega^3 = -\omega^1 = i$ und $\omega^2 = -1 = \omega^0$. Die Matrix DFT_4 hat nun eine relativ übersichtliche Gestalt angenommen.

Im Beispiel (11) ist leider auch zu sehen, daß die Matrix DFT_4 *vollbesetzt* ist, d.h. bei der Implementierung der Matrixmultiplikation ließe sich "nichts einsparen", da alle Multiplikationen durchgeführt werden müssen.

4 *Die schnelle Fourier Transformation*

Durch die Darstellung der Diskrete Fourier Transformation als Matrix in (9) wird klar, daß die Berechnung der DFT_N $O(N^2)$ Multiplikationen erfordert. Wäre die Matrix "dünn" besetzt, dann ließen sich zumindest einige Multiplikationen einsparen, da die entsprechenden Einträge $= 0$ nicht wirklich durchgeführt werden müßten. Es ist auf den ersten Blick nicht offensichtlich, daß es überhaupt eine Möglichkeit gibt auf eine solche Darstellung zu kommen. Erst fast 200 Jahre nach der "Erfindung" der Fourier Transformation kamen zwei Wissenschaftler von IBM auf die Lösung.

4.1 Die Idee der *FFT*

Betrachtet man nochmal die Matrix DFT_4 aus dem obigen Beispiel (11) (im folgenden bezeichnet ω immer ω_4, was der besseren Übersicht dienen soll):

$$F = DFT_4 \; f^T \tag{12}$$

$$\Leftrightarrow \begin{pmatrix} F_0 \\ F_1 \\ F_2 \\ F_3 \end{pmatrix} = \left(\begin{array}{cc|cc} 1 & 1 & 1 & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 \\ \hline 1 & \omega^2 & 1 & \omega^2 \\ 1 & \omega^3 & \omega^2 & \omega^1 \end{array}\right) \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \end{pmatrix} \tag{13}$$

Sortiert man nun den F-Vektor nach geraden und ungeraden Indizes, behält dabei aber die Reihenfolge des f-Vektors bei, dann ergibt sich:

$$\hat{F} = \widehat{DFT_4} f^T \tag{14}$$

$$\Leftrightarrow \begin{pmatrix} F_0 \\ F_2 \\ F_1 \\ F_3 \end{pmatrix} = \left(\begin{array}{cc|cc} 1 & 1 & 1 & 1 \\ 1 & \omega^2 & 1 & \omega^2 \\ \hline 1 & \omega^1 & \omega^2 & \omega^3 \\ 1 & \omega^3 & \omega^2 & \omega^1 \end{array}\right) \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \end{pmatrix} \tag{15}$$

Für die Untermatrizen $\widehat{\mathrm{W}}_{ik}$ von $\widehat{DFT_4}$ im obigen Beispiel (15) ergibt sich der folgende Zusammenhang:

$$\widehat{\mathrm{W}}_{11} = \widehat{\mathrm{W}}_{12}, \widehat{\mathrm{W}}_{22} = \omega^2 \widehat{\mathrm{W}}_{21}$$

Für die nachfolgende Gleichung (18) ist es vielleicht sinnvoll, DFT_2 einmal explizit hinzuschreiben:

$$DFT_2 = \begin{pmatrix} 1 & 1 \\ 1 & \omega^2 \end{pmatrix}$$

Nun ist, bis auf eine Permutationsmatrix P_4 für die Vertauschung in (13), die folgende Faktorisierung von DFT_4 möglich:

$$DFT_4 = P_4 \left(\begin{array}{cc|cc} 1 & 1 & 1 & 1 \\ 1 & \omega^2 & 1 & \omega^2 \\ \hline 1 & \omega^1 & \omega^2 & \omega^3 \\ 1 & \omega^3 & \omega^2 & \omega^1 \end{array}\right) \tag{16}$$

$$= P_4 \left(\begin{array}{cc|cc} 1 & 1 & 0 & 0 \\ 1 & \omega^2 & 0 & 0 \\ \hline 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & \omega^2 \end{array}\right) \left(\begin{array}{cc|cc} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ \hline 1 & 0 & \omega^2 & 0 \\ 0 & \omega^1 & 0 & \omega^3 \end{array}\right) \tag{17}$$

$$= P_4 \begin{pmatrix} DFT_2 & 0 \\ 0 & DFT_2 \end{pmatrix} \begin{pmatrix} I_2 & I_2 \\ D_2 & -D_2 \end{pmatrix} \tag{18}$$

Wobei I_2 die 2×2-Einheitsmatrix und D_2 die 2×2-Diagonalmatrix mit $D_2 = \text{diag}(\omega^0, \omega^1)$ ist, die hier ausgeschrieben so aussieht:

$$\begin{aligned} D_2 &= \begin{pmatrix} \omega^0 & 0 \\ 0 & \omega^1 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ 0 & \omega \end{pmatrix} \end{aligned}$$

Nun kann man erkennen, daß ein Problem der Größe N auf ein Problem der Größe $\frac{N}{2}$ zurückgeführt werden konnte. (Dies legt bei der Implementierung einen "Divide and Conquer"-Algorithmus nahe.) Läßt sich diese Verfahren fortsetzen, dann würde das bedeuten, daß bei einem Problem der Größe $N = 2^p, p \in \mathbb{N}$, die Fourierkoeffizienten nach spätestens p Schritten bestimmt wären. (Dies entspräche gerade der Tiefe des Baumes, der durch das rekursive Faktorisieren der Matrizen entstehen würde.)

Im obigen Beispiel muß man nun noch die Faktorisierung der DFT_2-Matrizen durchführen, nun mit einer Permutationsmatrix P_2:

$$\begin{aligned} DFT_2 &= P_2 \begin{pmatrix} +1 & +1 \\ +1 & -1 \end{pmatrix} \\ &= P_2 \begin{pmatrix} DFT_1 & 0 \\ 0 & DFT_1 \end{pmatrix} \begin{pmatrix} I_1 & I_1 \\ D_1 & -D_1 \end{pmatrix} \end{aligned}$$

Da aber $DFT_1 = \big(1\big)$ und $I_1 = \big(1\big)$, sowie $D_1 = \big(1\big)$ ist, ist dieser Schritt natürlich trivial.

Doch wie sieht nun konkret die Permutationsmatrix P_4 im obigen Beispiel (18) aus? Das einzige was diese Matrix macht ist, daß sie die Sortierung nach geraden und ungeraden Indizes wieder zurücknimmt. Ihre Darstellung ist sehr einfach:

$$\begin{aligned} P_4 &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\ &= (e_1, e_3, e_2, e_4)^T \end{aligned}$$

Wobei die e_i gerade dem i-tem Einheitszeilenvektor entspricht, d.h. an der i-ten Stelle im Zeilenvektor steht eine 1, sonst 0.

Die Idee, die *FFT* als Matrixmultiplikation aufzufassen läßt sich auch für $N = 2^p, p \in \mathbb{N}$ verallgemeinern, wie das folgende Lemma aus [Wer92] zeigt.

Lemma: Für $N \in \mathbb{N}$ sei $DFT_N \in \mathbb{C}^{N\times N}$ definiert durch:

$$DFT_N := (\omega_N^{jk})_{0\leq j,k\leq N-1} \text{ mit } \omega_N := e^{-2\pi i/N}$$

Bei gegebenem geraden $N \in \mathbb{N}$ sei $m := \frac{1}{2}N$ und $P_N \in \mathbb{C}^{N\times N}$ die Permutationsmatrix

$$P_N := (e_1\ e_3 \dots e_{N-1}|e_2\ e_4 \dots e_N)^T,$$

wobei e_j der j-te Einheitsvektor im $\mathbb{C}^N$ ist. Dann ist

$$DFT_N = P_N \begin{pmatrix} DFT_m & 0_m \\ 0_m & DFT_m \end{pmatrix} \begin{pmatrix} I_m & I_m \\ D_m & -D_m \end{pmatrix}$$

mit $D_m := \text{diag}(\omega_N^0, \omega_N^1, \dots, \omega_N^{m-1})$.

Hierbei ist I_m die $m \times m$-Einheitsmatrix und O_m die $m \times m$-Nullmatrix.

Der Beweis ist in [Wer92] nachzulesen. □

Die Idee der *FFT*, das Problem auf ein Teilproblem zurückzuführen funktioniert mit der Summendarstellung der *DFT* (siehe (6)) natürlich genauso gut.

Zunächst definiert man die Transformation FFT_N, wobei man den Skalierungsfaktor einfach wegläßt:

$$FFT_N(k, f) = \sum_{n=0}^{N-1} f(n)e^{-i2\pi kn/N}$$

Das bekannte Prinzip des Sortierens in gerade und ungerade Indizes wendet man hier nun analog an, unter der Voraussetzung, daß N gerade ist:

$$\begin{aligned} FFT_N(k, f) &= \sum_{n=0}^{N/2-1} f(n)e^{-i2\pi kn/N} + \sum_{n=N/2}^{N-1} f(n)e^{-i2\pi kn/N} \\ &= \sum_{n=0}^{N/2-1} (f(n) + f(n + N/2)e^{-i\pi k})e^{-i2\pi kn/N} \\ &= \sum_{n=0}^{N/2-1} (f(n) + f(n + N/2)(-1)^k)e^{-i2\pi kn/N} \end{aligned}$$

Betrachtet man dieses Ergebnis für die geraden und ungeraden k einzeln, dann ergibt sich:

- Für gerades $k = 2k'$ mit $k' = 0, \dots, N/2 - 1$:

$$FFT_{N/2}(k', f_E) = \sum_{n=0}^{N/2-1} f_E(n)e^{-i2\pi k'n/(N/2)}$$

mit $f_E(n) = f(n) + f(n + N/2)$.

- Für ungerades $k = 2k' + 1$ mit $k' = 0, \ldots, N/2 - 1$:

$$FFT_{N/2}(k', f_O) = \sum_{n=0}^{N/2-1} f_O(n) e^{-i2\pi k' n/(N/2)}$$

mit $f_O(n) = (f(n) - f(n + N/2))e^{-i2\pi n/N}$.

Auch hier wird ein Problem der Größe N auf ein Problem der halben Größe zurückgeführt.

4.2 Mehrdimensionale Fourier Transformation

Im Bereich der Bildverarbeitung müssen häufig mehrdimensionale Eingabedaten verarbeitet werden. Wenn also eine Funktion $f(k_x, k_y)$ vorliegt, die auf einem Gitter mit den Koordinaten $0 \leq k_x \leq N_x - 1$ und $0 \leq k_y \leq N_y - 1$ erklärt ist, so ist die Fourier Transformation $F(k_x, k_y)$ auf einem Gitter derselben Größe definiert als:

$$F(k_x, k_y) := \frac{1}{\sqrt{N_x N_y}} \sum_{n_x=0}^{N_x-1} \sum_{n_y=0}^{N_y-1} f(n_x, n_y) e^{-2\pi i k_x n_x/N_x} e^{-2\pi i k_y n_y/N_y}$$

Da man hier aber die Reihenfolge der Summationen vertauschen kann, läßt sich auch die Reihenfolge der Fourier Transformationen vertauschen. Die FT für zwei Dimensionen läßt sich also auf eine eindimensionale FT reduzieren, für die man bereits einen schnellen Algorithmus kennt.

$$F(k_x, k_y) = f''(k_x, k_y) \tag{19}$$

$$= \frac{1}{\sqrt{N_x}} \sum_{n_x=0}^{N_x-1} \Big(\frac{1}{\sqrt{N_y}} \sum_{n_y=0}^{N_y-1} f(n_x, n_y) e^{-2\pi i k_y n_y/N_y} \Big) e^{-2\pi i k_x n_x/N_x} \tag{20}$$

Nun kann man (20) auffassen als:

$$f''(k_x, k_y) = \frac{1}{\sqrt{N_x}} \sum_{n_x=0}^{N_x-1} f'(n_x, k_y) e^{-2\pi i k_x n_x/N_x} \tag{21}$$

mit

$$f'(n_x, k_y) = \frac{1}{\sqrt{N_y}} \sum_{n_y=0}^{N_y-1} f(n_x, n_y) e^{-2\pi i k_y n_y/N_y} \tag{22}$$

Nun sind (21) und (22) beides "einfache" eindimensionale FT, auf die man den FFT-Algorithmus anwenden kann. Das einzige Problem, mit dem man sich bei der Implementierung auseinandersetzen muß ist, daß nun ein "in-situ"-Algorithmus nicht mehr davon ausgehen kann, daß sich die Array-Elemente nebeneinander im Speicher befinden. Dem Algorithmus muß nun der "Abstand" zwischen den Elementen mitgegeben werden, was aber kein großes Problem darstellen sollte.

5 *Die Komplexität der FFT*

Der Aufwand, der beim Umsortieren der Matrix entsteht, drängt natürlich die Frage auf, ob das *FFT*-Verfahren tatsächlich schneller ist, als eine naive *DFT*-Implementierung?

In diesem Abschnitt möchte ich zeigen, wieviele Rechenoperationen insgesamt bei der *FFT* notwendig sind und dies anhand zweier Beispiele deutlich machen.

5.1 Rechenoperationen bei der *FFT*

Da auf modernen Architekturen die Kosten für Multiplikationen gegenüber den Kosten für Additionen dominieren, betrachte ich hier zunächst nur die Kosten für die komplexen Multiplikationen bei einer Eingabe der Größe $N = 2^p, p \in \mathbb{N}$:

$$C_{mult}(N) = \sum_{i=0}^{p-1} \sum_{j=0}^{2^i-1} 2^{p-i-1} = \sum_{i=0}^{p-1} 2^{p-1} = \frac{1}{2} p 2^p$$

Da in der innersten Schleife des Algorithmus für $k = 0$ mit 1 multipliziert wird, sich hier also die komplexe Multiplikation einsparen ließe, so reduziert sich die Anzahl der komplexen Multiplikationen auf:

$$C_{mult}(N) = \sum_{i=0}^{p-1} \sum_{j=0}^{2^i-1} (2^{p-i-1}) = \frac{1}{2} p 2^p - \sum_{i=0}^{p-1} 2^i = \frac{1}{2} p 2^p - (2^p - 1) = (p-2)2^{p-1} + 1. \quad (23)$$

Vorausgesetzt wird dabei natürlich, daß die Potenzen ω^i im Voraus berechnet wurden und so nicht in diese Analyse einfließen.

Substituiert man in (23) p durch $N = 2^p$, so ergibt sich:

$$C'_{mult}(N) = \frac{1}{2} N \log_2(N)$$

Die Anzahl der Additionen ist in jedem Schritt $= N$. Die Anzahl der Schritte ist die Tiefe der Rekursion, also $\log_2(N)$. Damit ergibt sich für die Anzahl der Additionen:

$$C_{add}(N) = N \log_2(N) \quad (24)$$

Insgesamt sind die Kosten für den *FFT*-Algorithmus also:

$$\begin{aligned} C(N) &= C'_{mult}(N) + C_{add}(N) \\ &= \frac{1}{2} N \log_2(N) + N \log_2(N) \\ &= \frac{3}{2} N \log_2(N) \\ &= O(N \log_2(N)) \end{aligned}$$

Für den Algorithmus im 2-dimensionalen Fall wie in (20) gilt:

$$\begin{aligned} C''(N_x, N_y) &= N_x C'(N_y) + N_y C'(N_x) \\ &= \frac{N_x N_y}{2} \log_2(N_y) + \frac{N_y N_x}{2} \log_2(N_x) \\ &= \frac{N_x N_y}{2} \log_2(N_x N_y) \\ &= C'(N_x N_y) \end{aligned}$$

wobei für die Anzahl der komplexen Multiplikationen $C'(N)$ gilt:

$$C'(N) = \frac{N}{2} \log_2(N)$$

Die Berechnung einer 2-dimensionalen FFT der Größe $N_x \times N_y$ braucht also dieselbe Anzahl an Multiplikationen wie die Berechnung einer ein-dimensionalen FFT der Größe $N_x N_y$.

5.2 Die Rechenschritte für $\boldsymbol{FFT}_4$

Betrachtet wir einmal konkret die Rechenschritte für den Fall $N = 4$:

Input		1.Schritt		2.Schritt	Output
f_0	0	$f_0^1 := f_0 + f_2$	0	$f_0^2 := f_0^1 + f_1^1$	F_0
f_1	1	$f_1^1 := f_1 + f_3$	2	$f_1^2 := (f_0^1 - f_1^1)\omega^0$	F_2
f_2	2	$f_2^1 := (f_0 - f_2)\omega^0$	1	$f_2^2 := f_2^1 + f_3^1$	F_1
f_3	3	$f_3^1 := (f_1 - f_3)\omega^1$	3	$f_3^2 := (f_2^1 - f_3^1)\omega^0$	F_3

Die Anzahl der Multiplikationen (auch die trivialen mit $\omega_0 = 1$) beträgt 4, die Anzahl der Additionen 8. Eine gewaltige Einsparung gegenüber den zu erwartenden 16 Multiplikationen und 12 Additionen der DFT_4 aus (11).

Als einziges Problem bleibt nun noch die Permutation P_4 vom Anfang, die ja erst auf das Muster in DFT_4 führte. Auf den ersten Blick mag man keine Regelmäßigkeit erkennen:

Input		...		Output
f_0	0	...	0	F_0
f_1	1	...	2	F_2
f_2	2	...	1	F_1
f_3	3	...	3	F_3

Betrachtet man aber die Eingabe-Indizes f_k und die Ausgabe-Indizes F_k sowohl dezimal als auch binär so ergibt sich Verbüffendes:

$$\begin{aligned} 0_{10} = 00_2 &\rightarrow 00_2 = 0_{10} \\ 1_{10} = 01_2 &\rightarrow 10_2 = 2_{10} \\ 2_{10} = 10_2 &\rightarrow 01_2 = 1_{10} \\ 3_{10} = 11_2 &\rightarrow 11_2 = 3_{10} \end{aligned}$$

Die Indizes des Resultats sind gerade die Indizes der Eingabe in Binärdarstellung, rückwärts gelesen. Dies gilt nicht nur "zufälligerweise" für $N = 4$, sondern tatsächlich für jedes $N = 2^p, p \in \mathbb{N}$.

5.3 Ein weiteres Beispiel für $N = 8$

Sei nun also $N = 8 = 2^3$ und zur Abkürzung $\omega := \omega_8 = e^{-2\pi i/8}$. Im ersten Schritt ergibt sich:

$$DFT_8 f = P_8 \begin{pmatrix} DFT_4 & 0 \\ 0 & DFT_4 \end{pmatrix} \begin{pmatrix} I_4 & I_4 \\ D_4 & -D_4 \end{pmatrix} f$$

mit $D_4 = \mathrm{diag}(\omega^0, \omega^1, \omega^2, \omega^3)$.
Dann läßt sich f^1 (wie in der Darstellung in Abschnitt (5.2)) berechnen:

$$\begin{aligned} f^1 &:= \begin{pmatrix} I_4 & I_4 \\ D_4 & -D_4 \end{pmatrix} f \\ &= \left(\begin{array}{c} f_0 + f_4 \\ f_1 + f_5 \\ f_2 + f_6 \\ f_3 + f_7 \\ \hline (f_0 - f_4)\omega^0 \\ (f_1 - f_5)\omega^1 \\ (f_2 - f_6)\omega^2 \\ (f_3 - f_7)\omega^3 \end{array}\right) = \left(\begin{array}{c} f_0^1 \\ \hline f_1^1 \end{array}\right) \end{aligned}$$

Zusammengefaßt also:

$$DFT_8 f = P_8 \begin{pmatrix} DFT_4 & 0 \\ 0 & DFT_4 \end{pmatrix} \begin{pmatrix} f_0^1 \\ f_1^1 \end{pmatrix} = P_8 \begin{pmatrix} DFT_4\ f_0^1 \\ DFT_4\ f_1^1 \end{pmatrix}$$

Nun folgt der zweite Schritt, die rekursive Anwendung mit DFT_4 auf f_0^1 und f_1^1:

$$DFT_4 f_0^1 = P_4 \begin{pmatrix} DFT_2 & 0 \\ 0 & DFT_2 \end{pmatrix} \begin{pmatrix} I_2 & I_2 \\ D_2 & -D_2 \end{pmatrix} f_0^1$$

sowie:

$$DFT_4 f_1^1 = P_4 \begin{pmatrix} DFT_2 & 0 \\ 0 & DFT_2 \end{pmatrix} \begin{pmatrix} I_2 & I_2 \\ D_2 & -D_2 \end{pmatrix} f_1^1$$

mit $D_2 := \mathrm{diag}(\omega_4^0, \omega_4^1) = \mathrm{diag}(\omega^0, \omega^2)$. Es ergibt sich dann analog zum ersten Schritt:

$$P_4 \begin{pmatrix} DFT_4\ f_0^1 \\ DFT_4\ f_1^1 \end{pmatrix} = P_4 \begin{pmatrix} DFT_2\ f_0^2 \\ DFT_2\ f_1^2 \\ DFT_2\ f_2^2 \\ DFT_2\ f_3^2 \end{pmatrix}$$

Für f^2 gilt dabei:

$$f^2 := \begin{pmatrix} f_0^2 \\ f_1^2 \\ f_2^2 \\ f_3^2 \end{pmatrix} = \left(\begin{array}{c} \begin{pmatrix} I_2 & I_2 \\ D_2 & -D_2 \end{pmatrix} f_0^1 \\ \hline \begin{pmatrix} I_2 & I_2 \\ D_2 & -D_2 \end{pmatrix} f_1^1 \end{array}\right)$$

Schlußendlich der letzte Schritt:

$$DFT_2 f_0^2 = P_2 \begin{pmatrix} DFT_1 & 0 \\ 0 & DFT_1 \end{pmatrix} \begin{pmatrix} I_1 & I_1 \\ D_1 & -D_1 \end{pmatrix} \begin{pmatrix} f_0^2 \\ f_1^2 \end{pmatrix}$$

Da $DFT_1 = (1)$ und $D_1 = (\omega^0) = (1)$ ergibt sich nun trivialerweise:

$$DFT_2 f_0^2 = P_2 \begin{pmatrix} f_0^2 + f_1^2 \\ (f_0^2 - f_1^2)\omega^0 \end{pmatrix} = \begin{pmatrix} f_0^3 \\ f_1^3 \end{pmatrix}$$

Die übrigen Koeffizienten ergeben sich analog. Bis auf eine Permutation der Komponenten steht nun in f^3 die gesuchte Fourier Transformierte von f. (Die "Berechnung" von DFT_1 kann man sich sparen, da $DFT_1 = (1)$ der Identität entspricht.)

Die durchzuführenden Berechnungen sind in der folgenden Tabelle aufgeschlüsselt:

Input		1.Schritt	2.Schritt	3.Schritt	Output
f_0	0	$f_0^1 := f_0 + f_4$	$f_0^2 := f_0^1 + f_2^1$	$f_0^3 := f_0^2 + f_1^2$	F_0
f_1	1	$f_1^1 := f_1 + f_5$	$f_1^2 := f_1^1 + f_3^1$	$f_1^3 := (f_0^2 - f_1^2)\omega^0$	F_4
f_2	2	$f_2^1 := f_2 + f_6$	$f_2^2 := (f_0^1 - f_2^1)\omega^0$	$f_2^3 := f_2^2 + f_3^2$	F_2
f_3	3	$f_3^1 := f_3 + f_7$	$f_3^2 := (f_1^1 - f_3^1)\omega^2$	$f_3^3 := (f_2^2 - f_3^2)\omega^0$	F_6
f_4	4	$f_4^1 := (f_0 - f_4)\omega^0$	$f_4^2 := f_4^1 + f_6^1$	$f_4^3 := f_4^2 + f_5^2$	F_1
f_5	5	$f_5^1 := (f_1 - f_5)\omega^1$	$f_5^2 := f_5^1 + f_7^1$	$f_5^3 := (f_4^2 - f_5^2)\omega^0$	F_5
f_6	6	$f_6^1 := (f_2 - f_6)\omega^2$	$f_6^2 := (f_4^1 - f_6^1)\omega^0$	$f_6^3 := f_6^2 + f_7^2$	F_3
f_7	7	$f_7^1 := (f_3 - f_7)\omega^3$	$f_7^2 := (f_5^1 - f_7^1)\omega^2$	$f_7^3 := (f_6^2 - f_7^2)\omega^0$	F_7

Bei diesem *FFT*-Verfahren werden sukzessive Vektoren f^1 (bestehend aus 2 Blöcken f_0^1 und f_1^1 der Länge 4), f^2 (bestehend aus 4 Blöcken $f_0^2, \ldots, f_3^2$ der Länge 2) und f^3 (bestehend aus 8 "Blöcken" $f_0^3, \ldots, f_7^3$ der Länge 1) berechnet.

In f^3 stehen dann die gesuchten $F_0, \ldots, F_7$ in einer permutierten Reihenfolge.

Die Anzahl der benötigten komplexen Multiplikationen (einschließlich derjenigen mit $\omega^0 = 1$ ist offenbar $12 = \frac{1}{2} 8 \log_2 8$.

In diesem Spezialfall ist die Gesetzmäßigkeit der Permutation leicht zu erkennen, wenn man sich die Binärdarstellung der Zahlen in der zweiten und drittletzten Spalte genauer ansieht:

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

Offenbar bestätigt sich, daß die Permutation dadurch gegeben ist, daß man die Binärdarstellung von hinten nach vorne liest.

6 *Implementierung*

Wie wird dieses Verfahren aber nun effizient implementiert? Ein leicht verständlicher Algorithmus findet sich in [Eng], den ich hier wiedergebe. Er ist in einer Pascal-ähnlichen Sprache geschrieben, die über komplexe Zahlen verfügt und es erlaubt Felder (Arrays) als Parameter an Prozeduren zu übergeben und als Ergebnis zurückzugeben.

Algorithm 1 *DIF:* Decimation in Frequency

Require: $N' \in \mathbb{N}, f \in \mathbb{C}[N]$
Ensure: $F \in \mathbb{C}[N]$
1: **if** N = 1 **then**
2: **DIF** $\leftarrow f$
3: **else**
4: $N' \leftarrow N/2$;
5: **for all** $n := 0 \ldots N' - 1$ **do**
6: $f_e[n] \leftarrow f[n] + f[n + N']$;
7: $f_o[n] \leftarrow (f[n] - f[n + N']) * \mathbf{T}(N, n)$
8: **end for**
9: $F_e \leftarrow \mathbf{DIF}(N', f_e)$;
10: $F_o \leftarrow \mathbf{DIF}(N', f_o)$;
11: **for all** $k' := 0 \ldots N' - 1$ **do**
12: $F[2 * k'] \leftarrow F_e[k']$;
13: $F[2 * k' + 1] \leftarrow F_o[k']$
14: **end for**
15: **DIF** $\leftarrow F$
16: **end if**

Im Algorithmus (1) wird eine Funktion T benutzt, die definiert ist als:

$$T_N(n) = e^{-2\pi i n/N}$$

Die Implementierung ist natürlich trivial:

Algorithm 2 *T:* Hilfsfunktion

Require: $N, n \in \mathbb{N}$
Ensure: $T \in \mathbb{C}$
1: $\mathbf{T} \leftarrow \exp(-i2\pi n/N)$

Der Grund für diese "Auslagerung" ist einfach: Die e Funktion muß dann lediglich an dieser Stelle implementiert werden. Dies kann dann sehr effizient geschehen, z.B. mit Tabellen.

A *Literatur und Quellen*

Literatur

[Dew95] A.K. Dewdney. *Der Turing Omnibus*, chapter 32: Schnelle Fourier Transformation. Springer Verlag, 1995.

[Eng] Engineering Productivity Tools Ltd. The FFT Demystified. `http://www.eptools.com/tn/T0001/PT00.HTM`.

[Hof] Forrest M. Hoffman. An introduction to fourier theory.

[HP94] Hüttenhofer/Lesch and Peyerimhoff. *Mathematik in Anwendung mit C++*, chapter 9.2. Quelle und Meyer, UTB für Wissenschaft, 1994.

[Lip81] John D. Lipson. *Elements of Algebra and Algebraic Computing*, chapter IX. The Fast Fourier Transform. Benjamin/Cummings Publishing Company, Inc., 1981.

[RR95] Reinhold Remmert. *Funktionentheorie 1*, chapter 3, page 117. Springer Verlag, 1995.

[Sch77] H.R. Schwarz. Elementare Darstellung der schnellen Fouriertransformation. *Computing*, 18:107–116, 1977.

[Sch93] Hans Rudolf Schwarz. *Numerische Mathematik*, chapter 4: Funktionsapproximation. B.G. Teubner, Stuttgart, 1993.

[Sch03] Jürgen Schweizer. Fourierreihen – Quick and Clean. Paper zur Vorbereitung auf das Proseminar, 2003.

[Sto94] Josef Stoer. *Numerische Mathematik 1, 7.Auflage*, chapter 2.3. Springer Lehrbuch, 1994.

[Übe95] C. Überhuber. *Computer-Numerik 2*, chapter 11: Fourier Transformation. Springer Verlag, 1995.

[Wer92] Jochen Werner. *Numerische Mathematik 1*, chapter 3.2: Trigonometrische Interpolation. Vieweg Verlag, 1992.