FFT – 高速アルゴリズムの発見 –

石岡圭一

1　はじめに

FFT(Fast Fourier Transform = 高速フーリエ変換) は, DFT(Discrete Fourier Transform = 離散フーリエ変換) を高速に実行する手法です. ここでは, フーリエ変換の意味や性質については既知として, DFT を簡単に導入したうえで, FFT そのものの構成や歴史に絞って解説していきます.

2　離散フーリエ変換

周期 2π もつ関数 $f(x)$ を以下のようにフーリエ級数展開することを考えます:

$$f(x) = \sum_{k=-\infty}^{\infty} a_k e^{ikx}. \tag{1}$$

$f(x)$ の値が一周期にわたって連続的に分かっていれば, 展開係数 a_k の値は以下のフーリエ積分で求めることができます:

$$a_k = \frac{1}{2\pi}\int_0^{2\pi} f(x)e^{-ikx}dx. \tag{2}$$

しかし, 関数 $f(x)$ が観測データ等である場合, 関数値は連続的には得られず, 離散的な x の値に対してしか関数値が与えられないことが普通です. 例えば, N 個の等間隔の点 x_j:

$$x_j = \frac{2\pi}{N}j, \quad (j = 0, 1, \cdots, N-1), \tag{3}$$

での関数値 f_j:

$$f_j = f(x_j), \quad (j = 0, 1, \cdots, N-1), \tag{4}$$

のみが得られている場合を考えましょう. このとき, フーリエ展開 (1) における展開関数 e^{ikx} は N 個の点 x_j でしか評価されませんので, N 次元のベクトルとみなされます. N 次元のベクトルですので, 当然一次独立な成分は N 個しかありえません. 特に今の場合のように点 x_j を等間隔にとっている場合には,

$$e^{i(k+N)x_j} = e^{i(k+N)\frac{2\pi}{N}j} = e^{2\pi ij}e^{ik\frac{2\pi}{N}j} = e^{ik\frac{2\pi}{N}j} = e^{ikx_j} \tag{5}$$

となり, 波数 k を N だけずらしたベクトルが波数 k のベクトルと全く同じものになります. これは, 有限の点で関数を評価するために, ある波数の成分が別の波数の成分と区別がつかなくなって同一視されてしまうことによります. この現象は, ある波数の成分に「別名」を与えてしまうということで,「エリアジング」と呼ばれています. 従って, 今の場合, フーリエ級数展開 (1) は無限個の和を考える代わりに, 独立な N 個の成分の和:

$$f_j = \sum_{k=0}^{N-1} a_k e^{ikx_j}, \quad (j = 0, 1, \cdots, N-1), \tag{6}$$

のみを考えるべきであることが分かります.

では, 展開 (6) において, 与えられた関数値 f_j から展開係数 a_k を求めるにはどうすればよいでしょうか? すぐ思いつく方法は,「(6) の左辺が既知なのだから, (6) を N 元連立一次方程式として解けば a_k が求ま

る」というものですが, 実はもっとうまい方法があります. 等比級数の和の公式により, 整数 l が N の倍数でない場合には,

$$\sum_{j=0}^{N-1} e^{ilx_j} = \sum_{j=0}^{N-1} e^{il\frac{2\pi}{N}j} = \sum_{j=0}^{N-1} \left(e^{il\frac{2\pi}{N}}\right)^j = \frac{\left(e^{il\frac{2\pi}{N}}\right)^N - 1}{e^{il\frac{2\pi}{N}} - 1} = 0, \tag{7}$$

となることが分かりますし, 整数 l が N の倍数の場合には

$$\sum_{j=0}^{N-1} e^{ilx_j} = \sum_{j=0}^{N-1} e^{il\frac{2\pi}{N}j} = \sum_{j=0}^{N-1} 1^j = N, \tag{8}$$

となることが分かります. 従って, (6) の両辺に $e^{-ikx_j} \quad (k = 0, 1, \cdots, N-1)$ を掛けて j について総和をとると,

$$(\text{左辺}) = \sum_{j=0}^{N-1} f_j e^{-ikx_j}, \tag{9}$$

$$(\text{右辺}) = \sum_{j=0}^{N-1} \left(\sum_{k'=0}^{N-1} a_{k'} e^{ik'x_j}\right) e^{-ikx_j} = \sum_{k'=0}^{N-1} a_{k'} \left(\sum_{j=0}^{N-1} e^{i(k'-k)x_j}\right) = Na_k, \tag{10}$$

が得られます. ここで, (6) における k を k' に書換え, $k' - k$ が N の倍数になるのは, $k' = k$ となるときだけであることを用いました. 以上から, a_k を求めるための公式:

$$a_k = \frac{1}{N} \sum_{j=0}^{N-1} f_j e^{-ikx_j}, \quad (k = 0, 1, \cdots, N-1), \tag{11}$$

が得られたことになります. $f(x)$ が連続量として与えられていた場合のフーリエ積分の式 (2) と見くらべると, この式は, 積分を区分求積法的な和に置き換えたものになっていることが分かります.

以上により, 関数の評価点が離散的に与えられた場合の展開係数 a_k と関数値 f_j との間の関係式 (6),(11) が導かれましたが, これらの式は離散フーリエ変換 (Discrete Fourier Transform = DFT) と呼ばれています.

3 高速フーリエ変換

DFT の 2 つの式 (11) と (6) とは, 互いに変換と逆変換との関係にありますが, 殆ど同じ形をしていることに気がつきます. 実際, (11) の両辺を N 倍して複素共役をとると,

$$(Na_k)^* = \sum_{j=0}^{N-1} f_j^* e^{ikx_j} = \sum_{j=0}^{N-1} f_j^* e^{ik\frac{2\pi}{N}j} = \sum_{j=0}^{N-1} f_j^* e^{ij\frac{2\pi}{N}k} = \sum_{j=0}^{N-1} f_j^* e^{ijx_k}, \tag{12}$$

となり, (6) と全く同じ形に帰着されます. 従って, DFT の計算について考える場合, (6) 式の形のみを考えるだけで十分であることが分かります. というわけで, 以下 (6) 式の計算について考えていくわけですが, 便宜上, $x_j = 2\pi j/N$ であることを使って, 以下のような形に書換えておきます:

$$F(j) = \sum_{k=0}^{N-1} W_N^{jk} A(k), \quad (j = 0, 1, \cdots, N-1). \tag{13}$$

ここに, $F(j) = f_j$, $A(k) = a_k$ と書き直し, $W_N = e^{\frac{2\pi i}{N}}$ と定めています.

さて, (13) 式の計算について考えていきましょう. この計算は, $N \times N$ 行列 (W_N^{jk}) に N 次元ベクトル a_k を掛けることに他なりませんから, 必要な計算量は, 積が N^2 回と和が $N(N-1)$ 回で, ともに $\mathcal{O}(N^2)$

回ということになります．従って，例えば $N = 10000$ の場合を考えると，DFT の計算のために $N^2 = 1$ 億回オーダーの演算が必要になります．今では，パソコンの性能が向上して，最速のものになると 1GFlops(1 秒間に 10 億回の浮動小数点演算を実行できる) に迫っていますから，この程度の計算ならパソコンでやれないこともありません．しかし，電子計算機の草創期ではそんな余裕なことを言っていられませんでした．1960 年代になってやっとスーパーコンピューターと呼ばれる高速計算機が現れてきましたが，それでもたかだか 1MFlops(1 秒間に 100 万回の浮動小数点演算を実行できる) 程度の能力しかありませんでした．というわけで，当時は最速の計算機を使っても，上記の DFT の計算が 100 秒以上かかってしまうという状況であり，N が大きい場合の DFT は気軽には実行できませんでした．

そこで，「DFT を計算するための何かうまい手法はないか?」というのが時代の要請であり，「必要は発明の母」の諺どおりに高速フーリエ変換 (Fast Fourier Transform = FFT) が生まれることになりました．ここでは，やや冗長にはなりますが，FFT の「発見者」である Cooley and Tukey (1965) の論文に沿ってその概要を見ていきましょう．

まず，変換の長さ N が $N = N_1 P$ と 2 つの整数 N_1, P の積になっている場合を考えます．このとき，(13) の添字 j, k それぞれを 2 つの添字の組 (j_p, j_1), (k_1, k_p) によって，

$$j = N_1 j_p + j_1, \quad (j_1 = 0, 1, \cdots, N_1 - 1;\ j_p = 0, 1, \cdots, P - 1), \tag{14}$$

$$k = P k_1 + k_p, \quad (k_1 = 0, 1, \cdots, N_1 - 1;\ k_p = 0, 1, \cdots, P - 1), \tag{15}$$

のように表示することにします．これは，C などのプログラミング言語において，$N_1 \times P$ および $P \times N_1$ の 2 次元配列を使っていると考えると分かりやすいかもしれません．この表示を使うと，(13) は以下のように書換えられます:

$$F(j_p, j_1) = \sum_{k_1=0}^{N_1-1} \sum_{k_p=0}^{P-1} W_N^{(N_1 j_p + j_1)(P k_1 + k_p)} A(k_1, k_p). \tag{16}$$

ここで，

$$\begin{aligned} W_N^{(N_1 j_p + j_1)(P k_1 + k_p)} &= W_N^{N_1 P j_p k_1 + P j_1 k_1 + (N_1 j_p + j_1) k_p} \\ &= W_N^{N j_p k_1} W_N^{P j_1 k_1} W_N^{(N_1 j_p + j_1) k_p} \\ &= (W_N^N)^{j_p k_1} W_N^{P j_1 k_1} W_N^{(N_1 j_p + j_1) k_p} \\ &= W_N^{P j_1 k_1} W_N^{(N_1 j_p + j_1) k_p} \end{aligned}$$

であることに注意すると，(16) は，

$$F(j_p, j_1) = \sum_{k_p=0}^{P-1} W_N^{(N_1 j_p + j_1) k_p} \sum_{k_1=0}^{N_1-1} W_N^{P j_1 k_1} A(k_1, k_p), \tag{17}$$

すなわち，

$$F(j_p, j_1) = \sum_{k_p=0}^{P-1} W_N^{(N_1 j_p + j_1) k_p} A_1(j_1, k_p), \quad (j_1 = 0, 1, \cdots, N_1 - 1;\ j_p = 0, 1, \cdots, P - 1), \tag{18}$$

$$A_1(j_1, k_p) = \sum_{k_1=0}^{N_1-1} W_N^{P j_1 k_1} A(k_1, k_p), \quad (j_1 = 0, 1, \cdots, N_1 - 1;\ k_p = 0, 1, \cdots, P - 1), \tag{19}$$

と 2 段階に分解されます．さて，このように分解すると，演算回数がどう変化するか数えてみましょう．まず，(19) には積が $N_1^2 \times P$ 回，和が $N_1(N_1 - 1) \times P$ 回必要で，(19) については，積が $P^2 \times N_1$ 回，和が $P(P-1) \times N_1$ 回となりますから，合計で，積が $N_1 P(N_1 + P) = N(N_1 + P)$ 回，和が $N_1 P(N_1 + P - 2) =$

$N(N_1+P-2)$ 回必要ということになります. 従って, $\mathcal{O}(N^2)$ の計算量が $\mathcal{O}(N(N_1+P))$ になったことになります. 最初に $N=10000$ の場合の計算量を例にとりましたが, この場合, $N_1=P=100$ ととってあげると, $N^2=1$ 億回の計算量が $N(N_1+P)=200$ 万回と, $1/50$ に減じられることになります.

これでも十分にすばらしい改良なのですが, さらに, $P=N_2Q$ と P が 2 つの整数の積に分解できる場合には, j_p, k_p それぞれを 2 つの添字の組 (j_q, j_2), (k_2, k_q) によって,

$$j_p = N_2 j_q + j_2, \quad (j_2 = 0, 1, \cdots, N_2-1;\ j_q = 0, 1, \cdots, Q-1), \tag{20}$$

$$k_p = Q k_2 + k_q, \quad (k_2 = 0, 1, \cdots, N_2-1;\ k_q = 0, 1, \cdots, Q-1), \tag{21}$$

と表すことにより, (18) の計算は以下のように変形できます (以下, 添字の範囲については省略しています):

$$\begin{aligned}
F(j_q, j_2, j_1) &= \sum_{k_q=0}^{Q-1}\sum_{k_2=0}^{N_2-1} W_N^{(N_1(N_2j_q+j_2)+j_1)(Qk_2+k_q)} A_1(j_1, k_2, k_q) && (22)\\
&= \sum_{k_q=0}^{Q-1} W_N^{(N_1(N_2j_q+j_2)+j_1)k_q} \sum_{k_2=0}^{N_2-1} W_N^{Q(N_1(N_2j_q+j_2)+j_1)k_2} A_1(j_1, k_2, k_q) && (23)\\
&= \sum_{k_q=0}^{Q-1} W_N^{(N_1(N_2j_q+j_2)+j_1)k_q} A_2(j_1, j_2, k_q). && (24)
\end{aligned}$$

ここに, $W_N^{QN_1N_2} = W_N^N = 1$ であることに注意して,

$$A_2(j_1, j_2, k_q) = \sum_{k_2=0}^{N_2-1} W_N^{Q(N_1j_2+j_1)k_2} A_1(j_1, k_2, k_q) \tag{25}$$

と定めました. このように分解すると, 計算量は $\mathcal{O}(N(N_1+N_2+Q))$ に下げられることになります. さらに $Q=N_3R$ と表されるときに (24)を分解すると $\cdots$ と, この分解を繰り返していくことを考えると, 結局, $N=N_1N_2\cdots N_m$ のとき, $F(j)$ は以下のような漸化式的手順で計算できることになります.

1. まず, 以下のように初期値 A_1 を A から定めます:

$$\begin{gathered}
Q_1 = N_1; \quad P_1 = N/Q_1,\\
A_1(j_1, k_p) = \sum_{k_1=0}^{N_1-1} W_N^{P_1j_1k_1} A(k_1, k_p),\\
(j_1 = 0, 1, \cdots, N_1-1;\ k_p = 0, 1, \cdots, P_1-1).
\end{gathered} \tag{26}$$

2. あとは $l=2, 3, \cdots, m$ について, 漸化式的に順に A_l を求めていきます:

$$\begin{gathered}
Q_l = N_1N_2\cdots N_l; \quad P_l = N/Q_l,\\
A_l(j_1, j_2, \cdots, j_{l-1}, j_l, k_p) = \sum_{k_l=0}^{N_l-1} W_N^{P_l(Q_{l-1}j_l+Q_{l-2}j_{l-1}+\cdots+Q_1j_2+j_1)k_l} A_{l-1}(j_1, j_2, \cdots, j_{l-1}, k_l, k_p)\\
(j_r = 0, 1, \cdots, N_r-1\ (r=1, 2, \cdots, l);\ k_p = 0, 1, \cdots, P_l-1).
\end{gathered} \tag{27}$$

3. 以上を繰り返していくと最終的に A_m が求まるので, それから F を添字の並べ替えによって以下のように求めます:

$$\begin{gathered}
F(j_m, j_{m-1}, \cdots, j_2, j_1) = A_m(j_1, j_2, \cdots, j_{m-1}, j_m),\\
(j_r = 0, 1, \cdots, N_r-1\ (r=1, 2, \cdots, m)).
\end{gathered} \tag{28}$$

これで, 入力 A から出力 F を得ることができました. 要される計算量は, 数えてみればすぐ分かりますように, $\mathcal{O}(N(N_1+N_2+\cdots+N_m))$ にまで減らされることになります. 特に, N がある整数 r の冪によって, $N=r^m$ と書ける場合には, $N_1=N_2=\cdots=N_m=r$ ととると, $\mathcal{O}(Nrm)=\mathcal{O}(Nr\log_r N)$ と表せます. 最初の $N=10000$ の場合について, $r=10$ ととると, 計算量は, $\mathcal{O}(Nr\log_r N)=\mathcal{O}(10^4\times 10\times 4)=\mathcal{O}(400000)$, となり, 結局 DFT を直接計算するのに必要な 1 億回の計算量が, 40 万回となり, 1/250 にまで減じられることになります[1] .

以上のアルゴリズムによって, $\mathcal{O}(N^2)$ の計算量が必要だった DFT が $\mathcal{O}(N\log N)$ の計算量でできることが分かります. このアルゴリズムは Cooley and Tukey (1965) によって発表され[2] , Fast Fourier Transform = FFT と呼ばれています. このアルゴリズムを使うことによる計算量の減少は劇的ですが, アルゴリズム自体は非常に簡明なので 1965 年まで発見されなかったということを不思議に思われる方もいらっしゃるかもしれません. この疑問はもっともで, 最初に紹介した部分で「発見者」とかぎ括弧を付けたのもそのためです. 実際, Cooley and Tukey (1965) 以前にも同様のアルゴリズムを独立に発見し, 実際, プログラミングまで行っていた人もいたのですが, FFT を実際に世に広めたのはこの 5 ページの簡潔な論文でしたので, 今では FFT の発見者の栄誉は Cooley and Tukey (1965) に与えられています. このあたりの FFT の歴史については, Brigham “The Fast Fourier Transform” (1974) (邦訳: 宮川・今井訳「高速フーリエ変換」(1978)) の第 1 章の末尾に詳しくまとめられています.

4 行列表示による FFT の理解

前節で述べた内容で FFT の基本的な構成法は尽されていて, これだでも FFT を計算するプログラムが作れるはずですが, FFT の手法に慣れないうちは「確かに数えてみると計算量が劇的に減るのは分かるけど, 添字が入り乱れていて手品みたい.」という印象を持つ方も多いと思います. そのような場合は, 全体の計算の流れを行列計算の形に表示しておくと理解が容易になります. N が 2 の冪の場合についてはいろいろな教科書に載っていますので, ここでは $N=6=3\times 2$ の場合について前節の FFT アルゴリズムを行列の形で書き下してみましょう.

まず, DFT の定義式 (13) を $N=6$ の場合について書いておきます:

$$\begin{bmatrix} F(0)\\ F(1)\\ F(2)\\ F(3)\\ F(4)\\ F(5) \end{bmatrix} = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0\\ \omega^0 & \omega^1 & \omega^2 & \omega^3 & \omega^4 & \omega^5\\ \omega^0 & \omega^2 & \omega^4 & \omega^6 & \omega^8 & \omega^{10}\\ \omega^0 & \omega^3 & \omega^6 & \omega^9 & \omega^{12} & \omega^{15}\\ \omega^0 & \omega^4 & \omega^8 & \omega^{12} & \omega^{16} & \omega^{20}\\ \omega^0 & \omega^5 & \omega^{10} & \omega^{15} & \omega^{20} & \omega^{25} \end{bmatrix} \begin{bmatrix} A(0)\\ A(1)\\ A(2)\\ A(3)\\ A(4)\\ A(5) \end{bmatrix}. \tag{29}$$

ここに, $\omega=W_6$ としています. さて, 前節で述べた FFT アルゴリズムを $N_1=3, N_2=2$ として書き下し

[1] もっとも, この場合 $10000=(2\times 5)^4=2^4\times 5^4$ とより細かく分解した方が, 計算量が $\mathcal{O}(10^4\times(2\times 4+5\times 4))=\mathcal{O}(280000)$ と, さらに少なくなります.

[2] Cooley and Tukey (1965) では, 具体的な表式としては $N=2^m$ の場合のみが示されますので, N が上記のように一般の N_r で分解される場合についての表式は, その若干の拡張である Bergland (1967) によっています.

みます:

$$\begin{bmatrix} F(0) \\ F(1) \\ F(2) \\ F(3) \\ F(4) \\ F(5) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \omega^0 & \omega^0 & 0 & 0 & 0 & 0 \\ \omega^0 & \omega^3 & 0 & 0 & 0 & 0 \\ 0 & 0 & \omega^0 & \omega^1 & 0 & 0 \\ 0 & 0 & \omega^0 & \omega^4 & 0 & 0 \\ 0 & 0 & 0 & 0 & \omega^0 & \omega^2 \\ 0 & 0 & 0 & 0 & \omega^0 & \omega^5 \end{bmatrix} \begin{bmatrix} \omega^0 & 0 & \omega^0 & 0 & \omega^0 & 0 \\ 0 & \omega^0 & 0 & \omega^0 & 0 & \omega^0 \\ \omega^0 & 0 & \omega^2 & 0 & \omega^4 & 0 \\ 0 & \omega^0 & 0 & \omega^2 & 0 & \omega^4 \\ \omega^0 & 0 & \omega^4 & 0 & \omega^8 & 0 \\ 0 & \omega^0 & 0 & \omega^4 & 0 & \omega^8 \end{bmatrix} \begin{bmatrix} A(0) \\ A(1) \\ A(2) \\ A(3) \\ A(4) \\ A(5) \end{bmatrix}. \tag{30}$$

両者がちゃんと同じ演算になっていることは, (30) の右辺の 3 つの正方行列の積を計算し, $\omega^6 = \omega^0 = 1$ であることに注意すれば簡単に確認できます. FFT のアルゴリズムによって計算量が減少するのも (30) を見れば一目瞭然で, (29) の右辺の行列が (30) では非零成分の少ない疎行列の積に分解されているため, 演算を右から行うと計算回数が少なくて済むということに他ならないことが分かります.

また, このような行列表示を使うと, FFT アルゴリズムの別の形式についての見通しも良くなります. 例えば, (29) の右辺の行列は対称行列であるため, 転置をとっても同じ形になります. そこで, (30) の右辺に現れている 3 つの正方行列の積全体の転置をとっても良いことになり, そうすると A のベクトルに演算される行列の順番が変えられることになります. こうして Cooley and Tukey のアルゴリズムの "転置" に相当するアルゴリズムが導かれますし, また, (30) では最後に並べ替えの行列が掛けられていますが, この並べ替えの操作を途中の行列に埋め込んだ "self-sorting" 型のアルゴリズムも提案されています. これらの FFT のいろいろな形態については, Temperton (1983) に詳細に整理されています.

5 Faster Fourier Transform?

FFT の基本的アルゴリズムが Cooley and Tukey (1965) によって発表されて以来, さらに計算を効率化するための研究が数多く行われました. FFT が計算量 $\mathcal{O}(N^2)$ を $\mathcal{O}(N \log N)$ に減少させたようには劇的ではありませんが, 場合によっては 2 倍程度の効率化につながりますので, 以下, その効率化の概略を見ていきましょう.

5.1 N が素数の場合の DFT

FFT は変換の項数 N が $N = N_1 N_2 \cdots N_m$ と分解できる場合の DFT の高速化を図るものであり, N が素数の場合にはこの手は使えません. 例えば, $N = 5$ の場合の DFT:

$$\begin{bmatrix} F(0) \\ F(1) \\ F(2) \\ F(3) \\ F(4) \end{bmatrix} = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 & \omega^4 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 & \omega^8 \\ \omega^0 & \omega^3 & \omega^6 & \omega^9 & \omega^{12} \\ \omega^0 & \omega^4 & \omega^8 & \omega^{12} & \omega^{16} \end{bmatrix} \begin{bmatrix} A(0) \\ A(1) \\ A(2) \\ A(3) \\ A(4) \end{bmatrix}, \tag{31}$$

を考えてみましょう. ここで, $\omega = W_5$ と略記しています. この場合, $\omega^0 = 1$ なので, この成分との乗算が必要ないことを考慮しても, この計算には乗算 16 回, 加算 20 回はどうしても必要なように見えます. しかし, 以下のように計算過程を工夫すると, 少ない演算回数でこの計算が可能になります. まず, $\theta = 2\pi/5$ として 5 つの定数:

$$c_1 = \frac{\cos\theta + \cos 2\theta}{2} - 1,\ c_2 = \frac{\cos\theta - \cos 2\theta}{2},\ c_3 = i(\sin\theta + \sin 2\theta),\ c_4 = i\sin 2\theta,\ c_5 = i(\sin\theta - \sin 2\theta), \tag{32}$$

を前もって計算しておくと, $F(0), \cdots, F(4)$ は,

$$\begin{array}{lllll} s_1 = A(1) + A(4), & s_2 = A(1) - A(4), & s_3 = A(3) + A(2), & s_4 = A(3) - A(2), & \\ t_1 = s_1 + s_3, & t_2 = s_1 - s_3, & t_3 = s_2 + s_4, & F(0) = t_1 + A(0), & \\ u_1 = c_1 t_1, & u_2 = c_2 t_2, & u_3 = c_3 s_2, & u_4 = c_4 t_3, & u_5 = c_5 s_4, \\ v_1 = F(0) + u_1, & v_2 = v_1 + u_2, & v_3 = v_1 - u_2, & v_4 = u_3 - u_4, & v_5 = u_4 + u_5, \\ F(1) = v_2 + v_4, & F(2) = v_3 + v_5, & F(3) = v_3 - v_5, & F(4) = v_2 - v_4 & \end{array} \tag{33}$$

の手順で計算できます. この手順に要する演算回数は, 乗算 5 回, 加減算 17 回と, DFT の定義式どおりの計算よりもかなり少なくなっています. しかも, (33) に現れる乗算はすべて, (実数または純虚数)$\times$ 複素数の形になっているため, 計算機の内部のように複素数を 2 つの実数の組として表した場合の乗算回数は, $5 \times 2 = 10$ 回になりますが, (31) の方はすべて複素数 $\times$ 複素数の形ですから, 複素数を 2 つの実数の組として表した場合の乗算回数は $16 \times 4 = 64$ 回であり, 改善が大きいことが分かります. このような効率化は, 最初 Rader (1968) によって提案されましたが, Winograd (1978) によって様々な N に対して系統的に導く方法が整理されました.

また, この手法は FFT そのものの高速化にも応用できます. FFT のアルゴリズムの中の漸化式 (27) は以下のように 2 段階に分離できます:

$$A'_{l-1}(j_1, j_2, \cdots, j_{l-1}, k_l, k_p) = W_N^{P_l(Q_{l-2}j_{l-1} + \cdots + Q_1 j_2 + j_1)k_l} A_{l-1}(j_1, j_2, \cdots, j_{l-1}, k_l, k_p), \tag{34}$$

$$A_l(j_1, j_2, \cdots, j_{l-1}, j_l, k_p) = \sum_{k_l=0}^{N_l-1} W_{N_l}^{j_l k_l} A'_{l-1}(j_1, j_2, \cdots, j_{l-1}, k_l, k_p). \tag{35}$$

ここで, $W_N^{P_l Q_{l-1}} = W_N^{N/N_l} = W_{N_l}$ であることを用いました. このようにすると, (35) の方は長さ N_l の DFT ですから, この DFT の計算を 5.1 のように最適化しておけば全体の FFT の高速化につながります. これは (27) の計算には $N \cdot N_l$ 回の乗算が必要なのに対して, (34) は N 回, (35) は $(N/N_l) \cdot N_l^2$ 回の乗算が必要で, 定義式どおりでは 2 段階に分けると (34) の分だけ乗算回数が増えてしまいますが, 5.1 の最適化を行っておくと, 後者の演算における乗算回数がかなり減らせるため効率化につながるわけです. また, このように変形した際の (34) の右辺の係数は回転因子 (= twiddle factor) と呼ばれています.

5.2 PFA = Prime Factor Algorithm

通常の FFT のアルゴリズムでは回転因子の存在は不可避なのですが, 回転因子を伴わないアルゴリズムも存在します.

変換の長さ N が $N = N_1 N_2$ と 2 つの整数の積となっている場合を考えます. ただし, FFT の場合と異なり, N_1 と N_2 とは互いに素であるという条件を付けておきます.

まず, N_1 と N_2 とは互いに素であるという条件より,

$$p_1 N_2 = p_2 N_1 + 1, \quad q_2 N_1 = q_1 N_2 + 1, \quad (0 < p_1, q_1 < N_1,\ 0 < p_2, q_2 < N_2) \tag{36}$$

を満す整数 p_1, p_2, q_1, q_2 が選べます[3]. このとき, DFT の式 (13) の添字 j, k それぞれを 2 つの添字の組 (j_1, j_2), (k_1, k_2) によって以下のように表すことにします:

$$j = (p_1 N_2 j_1 + q_2 N_1 j_2) \bmod N, \quad (j_1 = 0, 1, \cdots, N_1 - 1;\ j_2 = 0, 1, \cdots, N_2 - 1), \tag{37}$$

$$k = (N_2 k_1 + N_1 k_2) \bmod N, \quad (k_1 = 0, 1, \cdots, N_1 - 1;\ k_2 = 0, 1, \cdots, N_2 - 1). \tag{38}$$

[3] ユークリッドの互除法を使っていけば, p_1, p_2, q_1, q_2 が具体的に定められます. 例えば, $N_1 = 25, N_2 = 16$ の場合には, $p_1 = 11, p_2 = 7, q_1 = 14, q_2 = 9$ となります.

ここで j, k のすべての範囲をこれでカバーできるのか気になるところですが, これは, (36) 式を利用すると, それぞれの逆写像が,

$$j_1 = (j)\mathrm{mod}N_1, \quad j_2 = (j)\mathrm{mod}N_2, \tag{39}$$

$$k_1 = (p_1 k)\mathrm{mod}N_1, \quad k_2 = (q_2 k)\mathrm{mod}N_2, \tag{40}$$

と作れることで保証されます. さて, $W_N^N = 1$ であることに注意すると,

$$\begin{aligned}
W_N^{jk} &= W_N^{(p_1 N_2 j_1 + q_2 N_1 j_2)(N_2 k_1 + N_1 k_2)} \\
&= W_N^{p_1 N_2 N_2 j_1 k_1 + q_2 N_1 N_1 j_2 k_2 + N_1 N_2 (p_1 j_1 k_2 + q_2 j_2 k_1)} \\
&= W_N^{p_1 N_2 N_2 j_1 k_1 + q_2 N_1 N_1 j_2 k_2} \\
&= W_N^{(p_2 N_1 + 1) N_2 j_1 k_1 + (q_1 N_2 + 1) N_1 j_2 k_2} \\
&= W_N^{N_2 j_1 k_1 + N_1 j_2 k_2} \\
&= W_{N_1}^{j_1 k_1} W_{N_2}^{j_2 k_2},
\end{aligned}$$

となることが分かりますから, 元の DFT の式 (13) は,

$$F(j_1, j_2) = \sum_{k_2=0}^{N_2-1} W_{N_2}^{j_2 k_2} \sum_{k_1=0}^{N_1-1} W_{N_1}^{j_1 k_1} A(k_1, k_2), \tag{41}$$

と変形できることになります. FFT の場合とは異なり, この分解の場合は長さ N_1 の DFT と長さ N_2 の DFT が繰り返されるだけで, 余分な回転因子の積などがなく, 乗算が FFT に較べて少なくて済むことになります. また, この分解は FFT と同様に, N_2 がさらに 2 つの互いに素な整数の積で表される場合 $\cdots$, のように続けていくことができます. このようにして, N を互いに素な整数同士の積に分解して, その分解をもとに DFT を上記のように分解していくアルゴリズムは, PFA(= Prime Factor Algorithm) と呼ばれています.

PFA の発見は FFT より古く, Good (1958) にさかのぼるのですが, N を互いに素な整数に分解しなければならないという制約があることと, (37), (38) による並べ替えが複雑なことから, 当初はあまり実用的と考えられていなかったようです. しかし, 計算量が FFT より少なくて済むなどの魅力的な性質を持っているため, その後改良も加えられ, FFT と並ぶ DFT の有力な計算法と見なされています. また, Winograd (1978) は PFA の手法と 5.1 の手法を組み合わせることによって, さらに乗算回数を減らした計算法 (Winograd のアルゴリズム) を提案しています. この辺の詳細は Temperton (1988) にレビューされていますし, FFT と PFA を組合せた高度な実装については, Temperton (1992) によって論じられています.

5.3 計算機に合わせた最適化

ここまでの効率化の議論は, 計算に要求される乗算や加算の回数を数えるといいった, いわば「机上」の議論であり, 実際に FFT を計算機用のプログラムとして効果的に実装するためには, その計算機の CPU やメモリの構造を考慮に入れなければなりません. 例えば, たとえ演算回数が少なくても, メモリ参照が多いために実際には遅くなってしまうようなアルゴリズムはざらにあります. また, スーパーコンピューターなどでは乗算器と加算器がセットになっているため, 乗算の回数のみを減らすだけでは意味がないなど, 計算機毎に注意しなければならない点は異なります.

このため, 最近では, アルゴリズムの選択や, 5.1 で述べたような最適化されたプログラムの作成を計算機に合わせて自動的に行うような技術が開発されてきています. その中でも, MIT の Frigo and Johnson による FFTW(= The Fastest Fourier Transform in the West) は最新の研究成果もとりこまれた優れたフリーソフトウェアであり, インターネットで入手可能です (`http://theory.lcs.mit.edu/~fftw/`). この

FFTW は, 一般的によく使われているパソコンやワークステーションの CPU に合せた最適化が十分になされており, 他のフリーな FFT ソフトウェアの倍速またはそれ以上のパフォーマンスを示しています. もしお手元の計算機で FFT を行う必要がある場合にはぜひ入手して使ってみられることをお薦めします.

6 FFT の応用

FFT アルゴリズムの発見によって, N の長い DFT が気軽に計算できるようになった恩恵は非常に大きくて, FFT は実に多様な分野で応用されています. 最も直接的な応用は, 様々な信号の時系列データのスペクトル解析で, 与えられたデータにどの周波数の成分が多く含まれるかといった計算には, N を長くとれるという FFT の性質が特に威力を発揮します. データのスペクトル解析と, それに対する FFT の応用については, 日野 (1977) を参照されると良いでしょう.

FFT の高速性が可能にしたものの一つに, スペクトル法と呼ばれる偏微分方程式の解法があります. 例えば,

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0, \quad \text{境界条件: } u(x+2\pi, t) = u(x, t), \tag{42}$$

のような偏微分方程式を数値的に解く場合を考えましょう. 古典的な差分法では x 軸上に等間隔の分点 $x_j = (2\pi/N)j, \quad (j = 0, 1, \cdots, N-1)$ をとって, 偏微分 $\partial u/\partial x$ を分点間の差分で近似するのですが, スペクトル法は, FFT によって $u(x,t)$ の Fourier 級数展開を求め, 偏微分をその Fourier 級数展開式に適用することによって差分誤差をなくして, より精密な解を求めようとする解法です. この解法は, 乱流のシミュレーションや, 全球的な天気予報の計算に用いられています. この解説でしばしば名前を出した Clive Temperton は, そもそも ECMWF(= European Centre for Medium-Range Weather Forecasts: ヨーロッパ中期天気予報センター) のスタッフであり, 天気予報の計算に使うために, FFT の実装についての研究を行っていたわけです. スペクトル法については, Canuto, *et al.* (1988) が非常に詳しい教科書ですので, 興味のある方はそちらを参照してみて下さい.

また, 最近では, JPEG や MPEG のように, 静止画や動画のデータの圧縮にも FFT が使われています[4]. 画像圧縮に FFT がどのように使われているかについては, 田原 (1997) に紹介されています.

7 おわりに

ここまで, ざっと FFT の基本について解説してきましたが, FFT のようなアルゴリズムの真の理解のためには, 紙の上での計算だけではなく, 自分自身で計算機プログラムを書いてみることが早道ですので, プログラミングできる環境を持っている方はぜひ「自分なりの手作り FFT プログラム」を作ってみられるとよいと思います. 当然 FFTW のような「プロ」の書いた既製のソフトウェアにはなかなか太刀打ちできませんから,「既にあるものを再発明しようとするのはムダ」と思われるかもしれません. しかし, アシモフの「銀河帝国興亡史」ではありませんが, FFT のような「技術」を「魔術」にしてしまわないためにも, ぜひ, 先人の知恵を追体験したうえで,「よっしゃ, 自分がもっとええソフトを書いたる.」というくらいの気概を持つ方が多く現れてほしいと思っています.

8 参考文献

Bergland, G. D., 1967: The Fast Fourier Transform Recursive Equations for Arbitrary Length Records, *Math. Comp.*, **21**, 236–238.

[4] 厳密には, DCT(= Discrete Cosine Transform: 離散コサイン変換) ですが, DFT の実数部分だけをとりだしたようなもので, FFT と全く同じようなアルゴリズムで高速化されます.

Brigham, E. Oran, 1974: *The Fast Fourier Transform*, Prentice-Hall. (邦訳: 宮川洋・今井秀樹 訳, 1978, 「高速フーリエ変換」科学技術出版社, 262pp.)

Good, I. J., 1958: The Interaction of Algorithm and Practical Fourier Series, *J. Roy. Statist. Soc. Ser. B*, **20**, 361–372.

Canuto, C., M. Y. Hussaini, A. Quarteroni, and T. A. Zang, 1988: *Spectral Methods in Fluid Dynamics.* Springer, 567pp.

Cooley, J. W. and J. W. Tukey, 1965: An Algorithm for the Machine Calculation of Complex Fourier Series, *Math. Comp.*, **19**, 297–301.

日野幹雄, 1977: スペクトル解析, 朝倉書店, 300pp.

Rader, C. M., 1968: Discrete Fourier Transforms When the Number of Data Samples is Prime, *Proc. IEEE*, **5**, 1107–1108.

田原勝己, 1997: MPEG –インターネット時代の映像圧縮, bit 別冊「インターネット時代の数学」, 共立出版.

Temperton, C., 1983: Self-Sorting Mixed-Radix Fast Fourier Transforms, *J. Comp. Phys.*, **52**, 1–23.

Temperton, C., 1988: Nesting Strategies for Prime Factor FFT Algorithms, *J. Comp. Phys.*, **82**, 247–268.

Temperton, C., 1992: A Generalized Prime Factor FFT Algorithm for Any N=$2^p3^q5^r$, *SIAM J. Sci. Stat. Comput.*, **13**, 676–686.

Winograd, S., 1978: On Computing the Discrete Fourier Transform, *Math. Comp.*, **32**, 175–199.