高速フーリエ変換の実装

はじめに

フーリエ変換は周波数解析だけではなく, 畳み込み演算, 相関演算など高速演算手法として信号解析や画像処理などの分野で使用される数学的変換である。 フーリエ変換がそれほど多くの分野で応用されるのは, 畳み込みや相関演算が周波数領域での処理に置き換えられることも理由ではあるが, 高速フーリエ変換という高速演算が存在することが最大の理由であろう。

N 項の複素系列 a_n $(n = 0, 1, \ldots, N-1)$ が与えられたとする。 この系列の離散フーリエ変換 A_k は

$$A_k = \sum_{n=0}^{N-1} a_n W_N^{nk} \tag{1}$$

で与えられる。 この式に見られる W_N は回転因子と呼ばれ, 具体的には複素関数における単位元 1 の N 乗根の一つ, 具体的には, $W_N \equiv \exp(-2\pi i/N)$ である。 この式を見ると, 特定の k に関してフーリエ変換を得るには N 回の複素乗算が必要であることがわかる。 さらに, N 項の系列から得られる周波数成分が N 個あるので, すべての周波数成分に関するフーリエ変換を得るには N^2 回程度の複素乗算が必要になる。 これに対し, 複素加算の回数も N^2 回が必要である。よって, 離散フーリエ変換の演算量は $O(N^2)$ と書かれる。 これは, 演算量がほぼ N^2 に比例することを意味している。以降で説明する高速フーリエ変換 (FFT) では演算量が $O(N \log N)$ にすることができる。

本書の第 1 章では, 連続関数のフーリエ変換から離散フーリエ変換への導入を説明する。その後, フーリエ変換を高速化するための手法として, FFT をサンプルコードを交えて解説する。

目 次

第1章　フーリエ級数とフーリエ変換

フーリエ変換は, 時間または空間の座標で定義された関数をその座標に対応する周波数の座標で定義される関数に変換する数学変換である。そのため, 周波数解析を利用する波動解析や電気工学などの分野でフーリエ変換は応用されている。フーリエ変換は、フランスの数学者ジョゼフ・フーリエが熱伝導に関する研究のために導入したフーリエ級数に端を発する。本章はフーリエ級数から離散フーリエ変換までの一連の流れを説明する。フーリエ変換に関する知識を十分に持っている読者, または, 複素関数を用いた説明を特に必要としない読者は本章を飛ばしてもよい。

1.1　フーリエ級数

フーリエ級数は, 周期性をもつ任意の関数を基本的な周期関数の無限和によって表現する方法である。その表現に用いる基本的な周期関数としては三角関数を用いる。級数展開の表現法として, 実数による正弦関数と余弦関数を用いた実級数展開と, 複素数の指数関数による三角関数の表現を利用した複素級数展開の手法がある。本節では, 実級数展開から始め, 次に, それを複素級数展開に拡張する。

1.1.1　実級数展開

周期性をもつ関数 $f(t)$ を考えよう。この関数は時間軸 t 上で定義された関数であり, その周期を T であるとする。つまり, $f(t+T) = F(t)$ が必ず成立する。そのような関数は, $-T/2 \leq t < T/2$ について具体的な関数値を定義すれば, すべての実数 t における振る舞いが定義されたのと同じことである。フーリエ級数は, そのような周期関数 $f(t)$ を,

$$f(t) = a_0 + \sum_{n=1}^{\infty} \left(a_n \cos \frac{2\pi nt}{T} + b_n \sin \frac{2\pi nt}{T} \right) \tag{1.1}$$

なる級数展開で表現する。この級数展開のために導入した $\cos(2\pi nt/T)$ と $\sin(2\pi nt/T)$ は時間 T の間に n 周期する関数であるので, $f(t+T) = f(t)$ を満たす関数である。また, 定数 a_0 も $f(t+T) = f(t)$ を満たす関数である。言い換えると, フーリエ級数は周期 T の関数を, 基本的な周期 T の関数を用いて展開する手法である。

フーリエ級数の展開係数を a_0, a_n, b_n $(n = 1, 2, \ldots)$ を求めてみよう。それらの係数を計算するには, 次の積分公式を用いればよい。

$$\int_{-\pi}^{\pi} \cos mx \cos nx \,\mathrm{d}x = \begin{cases} \pi, & (m = n) \\ 0, & (m \neq n) \end{cases}$$

$$\int_{-\pi}^{\pi} \sin mx \sin nx \,\mathrm{d}x = \begin{cases} \pi, & (m = n) \\ 0, & (m \neq n) \end{cases}$$

$$\int_{-\pi}^{\pi} \cos mx \sin nx \,\mathrm{d}x = 0.$$

これらの積分公式は, 三角関数の加法定理を用いれば容易に導出できる。これらの積分公式を用いてフーリエ級数の展開係数を計算すると,

$$a_0 = \frac{1}{T} \int_{-T/2}^{T/2} f(t) \,\mathrm{d}t, \tag{1.2a}$$

$$a_n = \frac{2}{T} \int_{-T/2}^{T/2} f(t) \, \cos nt \,\mathrm{d}t, \quad b_n = \frac{2}{T} \int_{-T/2}^{T/2} f(t) \, \sin nt \,\mathrm{d}t, \tag{1.2b}$$

が得られる。なお, この展開係数の添え字 n をフーリエ級数展開の次数と呼ぶ。この次数は, 関数 $f(t)$ の周期 T において, 展開に用いる関数成分が周期する回数を表す。たとえば, 1 次の関数成分は周期 T で 1 周期しかしない。その関数を基本波成分と呼ぶ。一方, n 次の関数成分は周期 T で n 周期する。その関数を n 倍波成分と呼ぶ。

矩形波の級数展開　フーリエ級数展開の例として矩形波を展開してみよう。取り扱う矩形波は, 区間 $[-\pi, \pi)$ で,

$$f(t) = \begin{cases} 0, & (-\pi \leq t < 0) \\ 1, & (0 \leq t < \pi) \end{cases}$$

のように定義され, 周期 $T = 2\pi$ であるとする。この定義式を, (1.2a) と (1.2b) に代入すると, 展開係数 a_0, a_n, b_n が得られる。得られた展開係数を用いて展開式を書くと,

$$f(t) = \frac{1}{2} + \sum_{n=0}^{\infty} \frac{\sin(2n+1)t}{\pi\,(2n+1)}$$

となる。つまり, 余弦関数 (cosine) の展開係数はすべてゼロ, 正弦関数 (sine) については偶数次の展開係数がゼロになっている。さらに, この展開式を計算した結果は, 図 1.1 のように, 展開次数の増加とともに矩形波に近づいていることが確認できる。

関数の偶奇性　関数の偶奇性を利用すると, 無用な展開係数の計算を省略することができる。実際に計算すれば容易に確認できるとおり, $f_{\mathrm{even}}(t) = f_{\mathrm{even}}(-t)$ なる偶関数の展開係数について, $b_n = 0$ が必ず成立する。偶関数 $f_{\mathrm{even}}(t)$ のフーリエ級数展開の係数は,

$$a_0 = \frac{2}{T} \int_{0}^{T/2} f_{\mathrm{even}}(t) \,\mathrm{d}t, \quad a_n = \frac{4}{T} \int_{0}^{T/2} f_{\mathrm{even}}(t) \, \cos nt \,\mathrm{d}t, \quad b_n = 0,$$

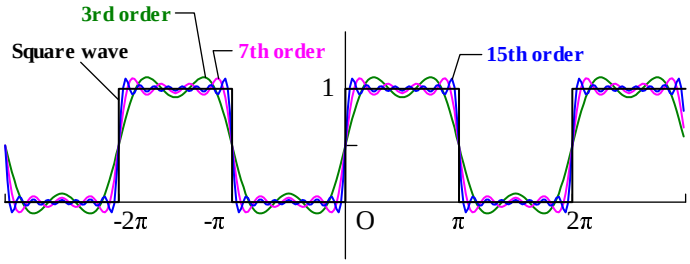


図 1.1: 矩形波のフーリエ級数展開の結果

のようになる。一方, 奇関数 $f_{\mathrm{odd}}(t) = -f_{\mathrm{odd}}(-t)$ なる奇関数については, $a_0 = a_n = 0$ が必ず成立する。よって, 偶関数 $f_{\mathrm{odd}}(t)$ のフーリエ級数展開の係数は,

$$a_0 = 0, \quad a_n = 0, \quad b_n = \frac{4}{T}\int_0^{T/2} f_{\mathrm{odd}}(t)\,\sin nt\,\mathrm{d}t,$$

となる。

のこぎり波の級数展開 次のフーリエ級数展開の例としてのこぎり波を展開してみよう。取り扱うのこぎり波は, 区間 $[-\pi, \pi)$ で,

$$f(t) = x, \quad (-\pi \leq t < \pi)$$

のように定義され, 周期 $T = 2\pi$ であるとする。こののこぎり波は奇関数であるので, 明らかに $a_0 = a_n = 0$ となる。したがって, b_n だけを計算すればよい。計算された展開係数を用いて展開式を書くと,

$$f(t) = \sum_{n=1}^{\infty}(-1)^{n+1}\frac{\sin nt}{n}$$

となる。この展開式を計算した結果は, 図 1.2 のように, 展開次数の増加とともにのこぎり波に近づいていることが確認できる。

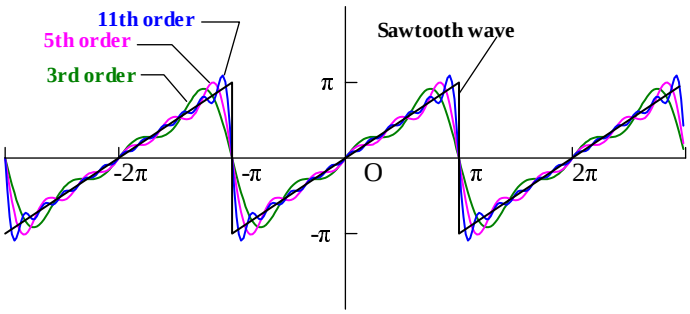


図 1.2: のこぎり波のフーリエ級数展開の結果

1.1.2　複素級数展開

オイラーの関係式 $e^{iz} = \cos z + i \sin z$ により, 三角関数が複素数の指数関数によって簡単に取り扱うことができる。フーリエ級数展開も指数関数を用いて, 数学表記を簡略化することが可能である。これ以降, 本章ではフーリエ級数展開 (さらには, 後に説明するフーリエ変換) を指数関数によって表記することとする。本節では, それに先立ち, 実級数展開から複素級数展開を導出する。

周期関数 $f(t)$ は, 既に示したように, その周期 T のなかで整数回繰り返す周波数成分の重ね合わせ, すなわち, (1.1) なる級数展開で表現することができる。この級数展開の展開係数 a_0, a_n, b_n は (1.2a) と (1.2b) で計算できる。

オイラーの関係式から, $\cos z = (e^{iz} + e^{-iz})/2$, $\sin z = (e^{iz} - e^{-iz})/2i$ であることが導かれる。これらの式を (1.1) に代入すると,

$$f(t) = a_0 + \sum_{n=1}^{\infty} \left(\frac{a_n - ib_n}{2} e^{2\pi i\, nt/T} + \frac{a_n + ib_n}{2} e^{-2\pi i\, nt/T} \right)$$

が得られる。ここで, 展開係数を

$$c_0 \equiv a_0, \quad c_n \equiv \frac{a_n - ib_n}{2}, \quad c_{-n} \equiv \frac{a_n + ib_n}{2}$$

のように置き換えると, フーリエ級数展開は

$$f(t) = \sum_{n=-\infty}^{\infty} c_n e^{2\pi i\, nt/T} \tag{1.3}$$

のように簡略化することができる。新たに導出された展開式は, 総和記号における添え字 n の稼動範囲が $-\infty$ から $+\infty$ に変化していることが注意すべき点である。また, 展開係数の計算公式も容易に書き換えることができ,

$$c_n = \frac{1}{T} \int_{-T/2}^{T/2} f(t) e^{2\pi i\, nt/T} \mathrm{d}t, \quad (n = 0, \pm 1, \pm 2, \ldots) \tag{1.4}$$

が得られる。この式によって算出される係数 c_n は複素数である。このように複素数に拡張した指数関数によって, フーリエ級数展開を簡潔な形で記述することができた。この形態ゆえ, 本書では原則的に複素数表記を用いる。

1.2　フーリエ変換

前節で, 周期 T の周期性をもつ関数がフーリエ級数展開できることを示した。その級数展開において, 周波数成分は $e^{2\pi i\, nt/T}$ で表されされている。この周波数成分は, 角周波数

$\omega = 2\pi n/T$ に対応する。つまり, 周期 T の周期関数は, $\Delta\omega = 2\pi/T$ の間隔で離散的に配置した角周波数に対応する成分で構成されていることを意味する。周期 T が長くなると, 構成する周波数成分の間隔 $\Delta\omega$ は小さくなる。さらに, $T \to \infty$ の極限では, $\Delta\omega \to 0$ となるので, 周波数成分に対応する展開係数 c_n も, 離散的ではなく, 連続的な角周波数 ω の関数となるはずである。しかも, $T \to \infty$ とすることで, 周期関数でない一般の関数についてもフーリエ級数のようが議論が可能になるはずである。本節では, $T \to \infty$ の極限を議論してみよう。

1.2.1 フーリエ変換とフーリエ逆変換

上で述べたように, $\omega \equiv 2\pi n/T$ なる角周波数 ω を用いて議論しよう。その角周波数を用いて, フーリエ級数展開の数式を書くと,

$$f(t) = \sum_{n=-\infty}^{\infty} c_n e^{i\omega t} = \frac{1}{2\pi} \sum_{n=-\infty}^{\infty} c_n T\, e^{i\omega t} \Delta\omega$$

のようになる。ここで, $\Delta\omega$ も上で紹介した離散的な角周波数の間隔である。極限 $T \to \infty$ とすると, $\Delta\omega$ はゼロに近づき, 展開係数 c_n は連続関数になる。そこで, そのような極限のもとで,

$$\sum_{n=-\infty}^{\infty} \Delta\omega \mapsto \int_{-\infty}^{\infty} \mathrm{d}\omega, \quad c_n T \mapsto F(\omega)$$

のように[1]置き換えると, フーリエ級数展開は,

$$f(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} F(\omega)\, e^{i\omega t} \mathrm{d}t \tag{1.5}$$

となる。同様に, (1.4) にも置き換えを適用すると,

$$F(\omega) = \int_{-\infty}^{\infty} f(t)\, e^{-i\omega t} \mathrm{d}t \tag{1.6}$$

が導出される。導かれた数式のうち, (1.6) をフーリエ変換と呼ぶ。時間 t で定義された関数を, 角周波数 ω の関数に変換するという意味である。一方, (1.5) はその逆の操作をする変換式であるので, フーリエ逆変換と呼ばれる。

1.2.2 デルタ関数

フーリエ変換の結果をフーリエ逆変換するともとの関数に戻ることを示そう。また, その過程でフーリエ変換を議論する上で欠かせないデルタ関数という特殊な関数を紹介する。

[1]なぜ $c_n \mapsto F(\omega)$ と置き換えないのかと思う人もいるだろう。しかし, そのよう置き換えると数式に T が残ってしまうのである。前提としている極限 $T \to \infty$ のため, 数式に T が残っていると都合が悪いのである。だから, $c_n T \mapsto F(\omega)$ なる置き換えがベストなのだ。

まず, 関数 $f(t)$ のフーリエ変換をフーリエ逆変換して元に戻ることを示そう。ただし, 現時点で $f(t)$ は複素数全体まで拡張した t において正則であるとしておく。フーリエ変換 $F(\omega)$ のフーリエ逆変換を実際に計算してみると,

$$\begin{aligned}
\frac{1}{2\pi}\int_{-\infty}^{\infty} F(\omega)\, e^{i\omega t}\mathrm{d}\omega &= \frac{1}{2\pi}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(\tau)\, e^{-i\omega\tau}\mathrm{d}\tau\cdot e^{i\omega t}\mathrm{d}\omega \\
&= \frac{1}{2\pi}\int_{-\infty}^{\infty} f(\tau)\int_{-\infty}^{\infty} e^{-i\omega(\tau-t)}\mathrm{d}\omega\,\mathrm{d}\tau \\
&= \frac{1}{2\pi}\lim_{R\to\infty}\int_{-\infty}^{\infty} f(\tau)\int_{-R}^{R} e^{-i\omega(\tau-t)}\mathrm{d}\omega\,\mathrm{d}\tau \\
&= \frac{1}{2\pi}\lim_{R\to\infty}\int_{-\infty}^{\infty} \frac{e^{iR(\tau-t)}-e^{-iR(\tau-t)}}{i(\tau-t)} f(\tau)\,\mathrm{d}\tau \\
&= \frac{1}{\pi}\lim_{R\to\infty}\int_{-\infty}^{\infty} \frac{\sin R(\tau-t)}{\tau-t} f(\tau)\,\mathrm{d}\tau \\
&= \frac{1}{\pi}\lim_{R\to\infty}\int_{-\infty}^{\infty} \frac{\sin R\tau}{\tau} f(\tau+t)\,\mathrm{d}\tau
\end{aligned} \tag{1.7}$$

のように数式変形できる。これ以上の計算を実行するには, ゼロでない任意の実数 A と任意の関数 $g(\tau)$ を含む積分公式[2]:

$$\int_{-\infty}^{\infty} \frac{\sin A\tau}{\tau} g(\tau)\,\mathrm{d}\tau = i\pi\, g(0)$$

を利用すればよい。ただし, この積分公式が成立するのは関数 $g(\tau)$ が複素平面全体で正則であることが条件である。この積分公式を適用すると, (1.7) の右辺は,

$$\text{RHS of (1.7)} = \lim_{R\to\infty} f(t) = f(t)$$

となり, フーリエ変換の逆変換がもとの関数に戻ることが示された。

デルタ関数　上の数式変形の中に現れる $e^{-i\omega(\tau-t)}$ の積分を, ある関数として定義してみよう。すなわち,

$$\delta(\tau) \equiv \frac{1}{2\pi}\int_{-\infty}^{\infty} e^{-i\omega\tau}\mathrm{d}\omega \tag{1.8}$$

なる関数 $\delta(\tau)$ を定義するのである。この定義式を (1.7) に代入すると,

$$f(t) = \int_{-\infty}^{\infty} f(\tau)\,\delta(\tau-t)\,\mathrm{d}\tau$$

となるはずである。ここで定義した $\delta(\tau)$ は, 無限積分と組み合わせると任意関数 $f(\tau)$ から局所的な値を取り出す効果を示す。そのような効果を示す関数であるためには,

$$\delta(\tau) = 0 \ \ (\text{if } \tau \neq 0), \qquad \int_{\infty}^{-\infty} \delta(\tau)\,\mathrm{d}\tau = 1 \tag{1.9}$$

[2]一見, 不思議な等式であるが, この積分公式は留数定理とコーシーの主値積分を用いて証明できる。興味のある読者は複素関数論のテキストを参照すればよい。

でなければならない。この関数 $\delta(\tau)$ は, $\tau = 0$ のときのみゼロでない値をとる特殊な関数である。その意味で, デルタ関数は一般の関数ではなく, 超関数として分類される。

いくつもの関数が, 性質 (1.9) をもつ関数の例として可能である。たとえば,

$$\delta_1(\tau) \equiv \lim_{\varepsilon \to 0} \frac{\varepsilon}{\pi\left(\tau^2 + \varepsilon^2\right)} \tag{1.10}$$

$$\delta_2(\tau) \equiv \lim_{\varepsilon \to 0} \frac{1}{\sqrt{2\pi}\,\varepsilon} e^{-\tau^2/2\varepsilon^2} \tag{1.11}$$

などは (1.9) の性質を満足するのでデルタ関数の例となる。

それでは, 複素指数関数の無限積分 (1.8) によって定義された関数ではどうであろうか? この関数は, 既に示したように無限積分が 1 となるのであるが, $\tau \neq 0$ に対して $\delta(\tau) = 0$ と主張するには無理があるのである。無限積分 (1.8) は,

$$\delta(\tau) = \frac{1}{2\pi} \lim_{R \to \infty} e^{i\omega\tau} \mathrm{d}\omega = \frac{1}{\pi} \lim_{R \to \infty} \frac{\sin R\tau}{\tau}$$

のように書くことができる。パラメータ R が小さな値のとき, この関数は図 1.3 (a) の曲線を描き, $\tau = 0$ での最大値 R/π 付近の形状は R を大きくとればさらに先鋭になる。ところが, $R \to \infty$ としても $\tau \neq 0$ での値が確定しないのである。この関数がゼロとなるのは $R\tau$ が π の整数倍である場合に限られる。たとえ $R \to \infty$ としても, 具体的な R の選び方によって, この関数は $-1/\tau \leq \delta(\tau) \leq 1/\tau$ のように無秩序に値が分散するのである。その様子を示したのが図 1.3 (b) である。このグラフは, R を変化させたときの $\delta(\tau)$ の値を調べたモンテカルロ・シミュレーションの結果である。値が確定しないとはいえ, $\tau \neq 0$ のとき, R の変化に対して $\delta(\tau)$ は正と負で対象に分散するので, $\delta(\tau)$ がとる値の期待値はゼロになる。よって, 期待値の意味で無限積分 (1.8) はデルタ関数となるのである。しかも, フーリエ変換に関する記述が便利であることから, デルタ関数の定義として (1.8) を使用することが多い。

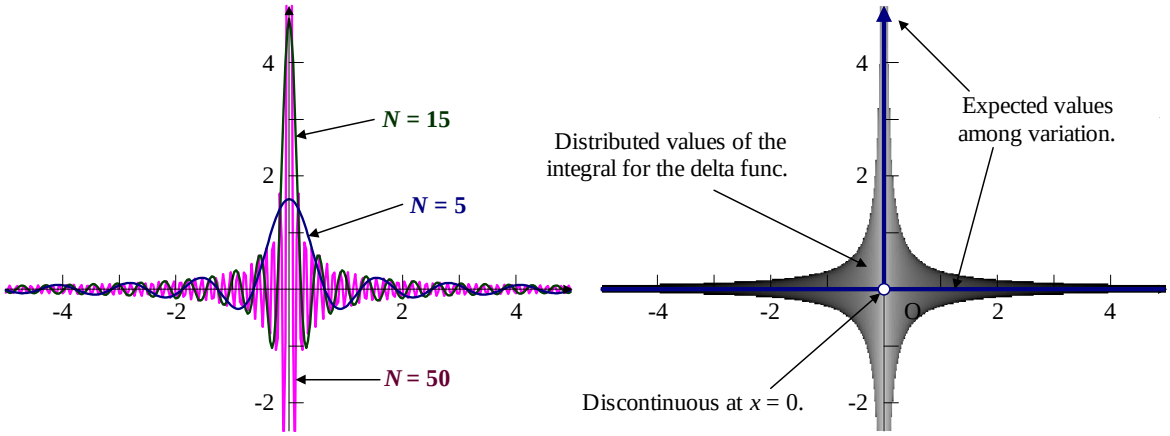
1.2.3 デルタ関数の性質

本節ではデルタ関数の性質をいくつか紹介する。デルタ関数 $\delta(x)$ は $x = 0$ 付近では具体的な値をもたないため, デルタ関数を含む計算はデルタ関数の性質を利用することになる。デルタ関数を含む数式の計算には, 次の 4 つの性質が有用である。

$$\delta(x) = \delta(-x), \tag{1.12}$$

$$\delta(ax) = \frac{\delta(x)}{|a|}, \tag{1.13}$$

$$\delta(f(x)) = \sum_{\{x_k \mid f(x_k)=0\}} \frac{\delta(x - x_k)}{|f(x_k)|}, \tag{1.14}$$



(a) Basic functions for the delta func with small *R*s.　(b) Distribution of values for the delta func with infinite R.

図 1.3: 複素指数関数の積分によって定義されたデルタ関数

$$x\,\delta'(x) = -\delta(x). \tag{1.15}$$

第 1 の性質 (1.12) はデルタが偶関数であることを意味している。その性質は,

$$\int_{-\infty}^{\infty} f(x)\,\delta(-x)\,\mathrm{d}x = \int_{-\infty}^{\infty} f(-x)\,\delta(x)\,\mathrm{d}x = f(0) = \int_{-\infty}^{\infty} f(x)\,\delta(x)\,\mathrm{d}x$$

が任意の関数 $f(x)$ について成立することから証明される。この性質は, (1.8) によって証明することもできる。¶

第 2 の性質 (1.13) は, 物理学における数式変形で頻繁に利用される。まず, 第 1 の性質 $\delta(x) = \delta(-x)$ を利用すると,

$$\int_{-\infty}^{\infty} f(x)\,\delta(ax)\,\mathrm{d}x = \int_{-\infty}^{\infty} f(x)\,\delta(|a|\,x)\,\mathrm{d}x = \frac{1}{|a|}\int_{-\infty}^{\infty} f(x/|a|)\,\delta(x)\,\mathrm{d}x = \frac{f(0)}{|a|}$$

が成立することがわかる。この式の右辺は,

$$\frac{f(0)}{|a|} = \int_{-\infty}^{\infty} f(x/|a|)\frac{f(x)}{|a|}\,\delta(x)\,\mathrm{d}x$$

のように書くこともできる。この等式が任意の関数 $f(x)$ に対して成立するので第 2 の性質が導かれる。この性質も, (1.8) によって証明することもできる。¶

第 3 の性質 (1.14) は第 2 の性質の応用である。数式において, x_k は $f(x) = 0$ の解のうち, (何らかの順序付けをしたと仮定して) k 番目の解を表す。総和の記号は, $f(x) = 0$ のすべての解に対する総和であることを意味する。解 x_k の近傍 $(x_k - \varepsilon, x_k + \varepsilon)$ では, 1 次のテーラー級数展開 $f(x) = (x - x_k)\,f'(x_k)$ が成立する。第 2 の性質を利用すると,

$$\delta(f(x)) = \delta((x - x_k)\,f'(x_k)) = \frac{\delta(x - x_k)}{|f'(x_k)|}, \quad (x \in (x_k - \varepsilon, x_k + \varepsilon))$$

が成立する。上に記載したように, この等式は x_k の近傍でしか成立しない。任意の x で成立させるには, $f(x)=0$ のすべての解の近傍での影響を加算する必要がある。したがって, 第 3 の性質が証明される。¶

第 3 の性質の応用として $\delta(x^2-a^2)$ を考えてみよう。デルタ関数の変数 $f(x)=x^2-a^2$ がゼロになる条件: x^2-a^2 の解は $x=\pm a$ である。さらに, その解における $f(x)$ の導関数が $f'(\pm a)=2|a|$ となるので,

$$\delta(x^2-a^2)=\frac{\delta(x+a)+\delta(x-a)}{2|a|}$$

が導かれる。

次に第 4 の性質 (1.15) を証明しよう。部分積分を利用すると,

$$\int_{-\infty}^{\infty} f(x)\,\delta(x)\,\mathrm{d}x=\Big[F(x)\,\delta(x)\Big]_{-\infty}^{\infty}-\int_{-\infty}^{\infty}F(x)\,\delta'(x)\,\mathrm{d}x=-\int_{-\infty}^{\infty}F(x)\,\delta'(x)\,\mathrm{d}x$$

なる等式が成立することがわかる。ここで, $F(x)$ は $f(x)$ の原始関数とする。この原始関数は, $x=0$ の近傍で $F(x)\simeq x\,f(0)$ が成立する。これを利用すると,

$$\int_{-\infty}^{\infty} f(x)\,\delta(x)\,\mathrm{d}x=-f(0)\int_{-\infty}^{\infty}x\,\delta'(x)\,\mathrm{d}x=-\int_{-\infty}^{\infty}x\,f(x)\,\delta'(x)\,\mathrm{d}x$$

が得られる。この等式が任意の関数 $f(x)$ について成立するための条件として, 第 4 の成立が導かれる。

1.2.4 デルタ関数の級数公式

デルタ関数を周期 T で繰り返すように周期的に並べた関数 $\delta_T(t)$ を考えよう。形式的には,

$$\delta_T(t)\equiv\sum_{n=-\infty}^{\infty}\delta(t-nT)$$

のように定義できる。この関数は周期関数であるので, フーリエ級数で表現できるはずである。そこで, フーリエ級数の展開係数を計算すると,

$$c_n=\frac{1}{T}\int_{-T/2}^{T/2}\delta(t)\,e^{-2\pi i\,nt/T}\mathrm{d}t=\frac{1}{T}$$

が得られる。つまり,

$$\delta_T(t)=\frac{1}{T}\sum_{n=-\infty}^{\infty}e^{2\pi i\,nt/T}$$

のように展開できる。次に, $\delta_T(t)$ のフーリエ変換 $\Delta_T(\omega)$ を計算してみよう。まず, $\delta_T(t)$ の定義式から計算すると,

$$\Delta_T(\omega)=\int_{-\infty}^{\infty}\sum_{n=-\infty}^{\infty}\delta(t-nT)\,e^{i\omega t}\mathrm{d}t=\sum_{n=-\infty}^{\infty}e^{i\omega nT}$$

となる。一方, $\delta_T(t)$ のフーリエ級数展開から計算すると,

$$\Delta_T(\omega) = \int_{-\infty}^{\infty} \frac{1}{T} \sum_{n=-\infty}^{\infty} e^{2\pi i\, nt/T} \cdot e^{i\omega t} \mathrm{d}t = \frac{2\pi}{T} \sum_{n=-\infty}^{\infty} \delta\left(\omega - \frac{2\pi n}{T}\right)$$

が得られる。これら2つの結果を等号で結ぶと,

$$\sum_{n=-\infty}^{\infty} e^{i\omega nT} = \frac{2\pi}{T} \sum_{n=-\infty}^{\infty} \delta\left(\omega - \frac{2\pi n}{T}\right) \tag{1.16}$$

なる関係式が導出される。また, フーリエ変換において時間 t と角周波数 ω が対称であることを考えると,

$$\sum_{n=-\infty}^{\infty} e^{2\pi i\, nt/T} = T \sum_{n=-\infty}^{\infty} \delta(t - nT) \tag{1.17}$$

が成立することがわかる。これらの関係式は, 本章で何度か使用する有用な公式である。

1.3 周期関数と離散関数のフーリエ変換

本章の冒頭で, 周期関数がフーリエ級数展開できることを述べた。また, その事実がフーリエ変換の発端になっている。それでは, 周期関数をフーリエ変換すると何が得られるであろうか? それを計算してみよう。

フーリエ変換する関数 $f(t)$ が周期 T をもっているとする。その関数 $f(t)$ のフーリエ変換は次のように計算できる。

$$\begin{aligned} F(\omega) &= \int_{-\infty}^{\infty} f(t)\, e^{-i\omega t} \mathrm{d}t = \sum_{n=-\infty}^{\infty} \int_{-T/2}^{T/2} f(t+nT)\, e^{-i\omega\,(t+nT)} \mathrm{d}t \\ &= \sum_{n=-\infty}^{\infty} \int_{-T/2}^{T/2} f(t)\, e^{-i\omega\,(t+nT)} \mathrm{d}t = \sum_{n=-\infty}^{\infty} e^{-i\omega nT} \int_{-T/2}^{T/2} f(t)\, e^{-i\omega t} \mathrm{d}t. \end{aligned}$$

この計算をさらに進めるには, デルタ関数の級数に関する公式を利用する。その公式を適用すると,

$$F(\omega) = \frac{2\pi}{T} \sum_{n=-\infty}^{\infty} \delta\left(\omega - \frac{2\pi n}{T}\right) \int_{-T/2}^{T/2} f(t)\, e^{-i\omega t} \mathrm{d}t \tag{1.18}$$

が得られる。デルタ関数を含んでいることから明らかなように, 周期 T をもつ関数は, $\omega = 2\pi n/T$ の離散的な角周波数でしか周波数成分をもたない。その離散的な角周波数とは, フーリエ級数展開で現れた角周波数と同一である。しかも, デルタ関数の振幅はフーリエ級数展開の展開係数の 2π 倍の値になっている。

1.3.1 離散関数のフーリエ変換

次にτ間隔の離散的な時刻で定義された関数をフーリエ変換しよう。その離散的な関数は形式的には,

$$f(t) = \sum_{n=-\infty}^{\infty} a_n \delta(t - n\tau) \tag{1.19}$$

のように表現できる。この離散的な関数とは, アナログ-ディジタル変換器 (AD 変換器) によって一定の標本化周期τで標本化されたデータ列a_nに対応すると考えるとよい。フーリエ変換と逆フーリエ変換が表裏一体であることから, 周期関数のフーリエ変換が離散関数であるから, 離散関数のフーリエ変換は周期関数になると予想される。検証のため, 実際に計算してみると,

$$F(\omega) = \int_{-\infty}^{\infty} \sum_{n=-\infty}^{\infty} a_n \delta(t - n\tau)\, e^{-i\omega t} \mathrm{d}t = \sum_{n=-\infty}^{\infty} a_n e^{-i\omega n\tau}$$

が得られる。このフーリエ変換の結果が, $F(\omega) = F(\omega + 2\pi/\tau)$ を満足することが容易にわかる。つまり, そのフーリエ変換結果は, 角周波数について$\omega_\mathrm{s} = 2\pi/\tau$の周期をもつ周期関数である。

つぎに, 角周波数における周期性を利用して, $F(\omega)$ をフーリエ逆変換してみよう。

$$\begin{aligned}
f(t) &= \frac{1}{2\pi} \int_{-\infty}^{\infty} F(\omega)\, e^{i\omega t} \mathrm{d}\omega = \frac{1}{2\pi} \sum_{n=-\infty}^{\infty} \int_{\omega_\mathrm{s}/2}^{\omega_\mathrm{s}/2} F(\omega)\, e^{i(\omega + n\omega_\mathrm{s})t} \mathrm{d}\omega \\
&= \frac{1}{2\pi} \sum_{n=-\infty}^{\infty} e^{i\, n\omega_\mathrm{s} t} \int_{\omega_\mathrm{s}/2}^{\omega_\mathrm{s}/2} F(\omega)\, e^{i\omega t} \mathrm{d}\omega \\
&= \frac{1}{2\pi} \sum_{n=-\infty}^{\infty} \frac{2\pi}{\omega_\mathrm{s}} \delta\left(t - \frac{2\pi n}{\omega_\mathrm{s}}\right) \int_{\omega_\mathrm{s}/2}^{\omega_\mathrm{s}/2} F(\omega)\, e^{i\omega t} \mathrm{d}\omega \\
&= \frac{\tau}{2\pi} \sum_{n=-\infty}^{\infty} \delta(t - n\tau) \int_{\omega_\mathrm{s}/2}^{\omega_\mathrm{s}/2} F(\omega)\, e^{i\omega t} \mathrm{d}\omega.
\end{aligned}$$

この計算結果は, $\delta(t - n\tau)$ を含んでいるため, 確かに標本化周期τの離散的なデータ列になっている。この計算結果と, 関数 $f(t)$ の定義 (1.19) を比較すると,

$$a_n = \frac{\tau}{2\pi} \int_{-\omega_\mathrm{s}/2}^{\omega_\mathrm{s}/2} F(\omega)\, e^{in\omega t} \mathrm{d}\omega \tag{1.20}$$

が得られる。

1.3.2 周期性をもつ離散関数のフーリエ変換

周期関数のフーリエ変換が離散関数となり, 離散関数のフーリエ変換が周期関数となることは前節までに示したとおりである。すると, 離散関数が周期関数でもあった場合, その関数のフーリエ変換は同様に, 離散関数でもあり周期関数でもあるはずである。

周期 τ で標本化した離散関数 $f(t) = \sum a_n \delta(t - n\tau)$ が N 点の標本ごとに同一のデータ列を繰り返しているとしよう。形式的には, $a_{n+N} = a_n$ を仮定するのである。その条件でフーリエ変換を計算してみよう。周期性に注意しながら, 前節で導出した離散関数のフーリエ変換の公式を計算すると,

$$\begin{aligned} F(\omega) &= \sum_{n=-\infty}^{\infty} a_n e^{-i\omega n\tau} = \sum_{k=-\infty}^{\infty} \sum_{n=0}^{N-1} a_{n+kN} e^{-i\omega(n+kN)\tau} \\ &= \sum_{k=-\infty}^{\infty} e^{-i\omega kN\tau} \sum_{n=0}^{N-1} a_n e^{-i\omega n\tau} \end{aligned}$$

が得られるが, $e^{-i\omega kN\tau}$ の総和についてデルタ関数の級数の公式を利用すると,

$$F(\omega) = \frac{2\pi}{N\tau} \sum_{k=-\infty}^{\infty} \delta\left(\omega - \frac{2\pi k}{N\tau}\right) \sum_{n=0}^{N-1} a_n e^{i\omega n\tau}$$

が得られる。確かに, $F(\omega)$ は離散関数になっている。そこで,

$$F(\omega) = \frac{2\pi}{N\tau} F_k \, \delta(\omega - k\omega_{\mathrm{s}}/N)$$

とおいてみよう。ただし, $\omega_{\mathrm{s}} = 2\pi/\tau$ である。すると,

$$F_k = \sum_{n=0}^{N-1} a_n \exp\left(-\frac{2\pi i}{N} nk\right) \tag{1.21}$$

が得られる。この結果を離散関数のフーリエ逆変換の公式に代入すると,

$$\begin{aligned} a_n &= \frac{\tau}{2\pi} \int_0^{\omega_{\mathrm{s}}} f(\omega) \, e^{i\omega n\tau} \mathrm{d}\omega \\ &= \frac{\tau}{2\pi} \frac{2\pi}{N\tau} \int_0^{\omega_{\mathrm{s}}} F_k \, \delta(\omega - k\omega_{\mathrm{s}}/N) \, e^{i\omega n\tau} \mathrm{d}\omega \\ &= \frac{1}{N} \sum_{0 \le \omega < \omega_s} F_k \, \exp\left(\frac{2\pi i}{N} nk\right) = \frac{1}{N} \sum_{n=0}^{N-1} F_k \, \exp\left(\frac{2\pi i}{N} nk\right) \end{aligned}$$

が得られる。結果を再度書くと, 周期的な離散関数に関するフーリエ変換とフーリエ逆変換は,

$$F_k = \sum_{n=0}^{N-1} a_n \exp\left(-\frac{2\pi i}{N} nk\right) \tag{1.22a}$$

$$a_n = \frac{1}{N} \sum_{n=0}^{N-1} F_k \, \exp\left(\frac{2\pi i}{N} nk\right) \tag{1.22b}$$

のように書くことができる。

第2章　高速フーリエ変換

高速フーリエ変換 (Fast Fourier Transform; FFT) は, 変換の対称性を利用して分割統治による劇的な演算量削減をしたアルゴリズムである。 FFT が広く知られるようになったのは, 1965 年の Cooley と Tukey の論文以降であるが, 対象性を利用したその種の高速演算手法は 1805 年に Gauss が見つけていたとも言われる。他にも数人の数学者が同様のアルゴリズムを発見していたにも関わらず, アルゴリズムは知れ渡ることなく, コンピュータの発明によって巨大なデータの演算が可能になって, ようやく FFT が脚光を浴びることとなった。

Cooley と Tukey が発表したアルゴリズムは, 長 N の系列を R 等分し, 分割によって長さ N/R になった系列のフーリエ変換を統合することによって, 長さ N のフーリエ変換を得る手法である。このような手法は基数 R の FFT と呼ばれる。離散フーリエ変換を定義どおりに計算した場合の演算量が $O(N^2)$ であるのに対して FFT の演算量は $O(N \log N)$ となる。 Coolery-Tukey 型の FFT には, 基数 2, 基数 4 などの様々な基数のアルゴリズムがあるが, 演算量はすべて $O(N \log N)$ のオーダであり, 基数が変わると演算量の比例的数が変わる程度である。Cooley-Tukey 型のうち演算量の比例係数が最も小さいものは基数 2 と基数 4 を混合した Split Radix アルゴリズムと呼ばれる手法である。本章は Cooley-Tukey 型アルゴリズムとして有名な基数 2, 基数 4, および, Split Radix アルゴリズムについて説明する。

2.1　基数2のアルゴリズム

基数 2 のアルゴリズムは Cooley-Tukey 型 FFT の中で最も基本的なアルゴリズムである。変換する系列 a_n の長さ N が偶数であれば, 回転因子には $W_N^p = -W_N^{p+N/2}$ なる対称性がある。この回転因子の対象性を利用してフーリエ変換の高速化を実現できる。

回転因子の対象性を利用して (1) を書き直すと,

$$\begin{aligned} A_{2k} &= \sum_{n=0}^{N/2-1} a_n W_N^{2nk} + \sum_{n=0}^{N/2-1} a_{n+N/2} W_N^{2k(n+N/2)} \\ &= \sum_{n=0}^{N/2-1} (a_n + a_{n+N/2}) W_{N/2}^{n} k \end{aligned} \tag{2.1}$$

$$A_{2k+1} = \sum_{n=0}^{N/2-1} a_n W_N^{n(2k+1)} + \sum_{n=0}^{N/2-1} a_{n+N/2} W_N^{(n+N/2)(2k+1)}$$
$$= \sum_{n=0}^{N/2-1} (a_n - a_{n+N/2}) W_N^2 \cdot W_{N/2}^n k \tag{2.2}$$

が得られる。この数式が意味するのは, N が偶数であるとき, 系列 a_n のフーリエ変換のうち, 偶数次の周波数成分 A_{2k} は長さ $N/2$ 項の系列 $a_n + a_{n+N/2}$ のフーリエ変換に等しい。また, 奇数次の周波数成分 A_{2k+1} は長さ $N/2$ 項の系列 $(a_n - a_{n+N/2})W_N^n$ のフーリエ変換に等しい。すなわち, N が偶数ならば $N/2$ 項のフーリエ変換を 2 回実行することによって N 項のフーリエ変換を計算できる。定義式どおりの積和演算による計算では乗算回数が N^2 に比例するので, N 項のフーリエ変換には $N/2$ 項のときの 4 倍の演算量が必要である。しかし, 上式の結果を使えば N 項のフーリエ変換に必要な演算量は, $N/2$ 項のときの 2 倍程度の量に抑えられる。このアルゴリズムは偶数次と奇数次の周波数成分に分離して導かれていることから**周波数間引きアルゴリズム**[1]と呼ばれる。

分割した系列の長さ $N/2$ が偶数ならば, 当然, さらに同様に系列を分割して演算量を抑えることができる。もし, N が 2 のべき乗ならば, 長さ 2 の FFT に落ち着くまで系列の分割を繰り返すことができる。

例えば $N = 4$ の場合を考えてみよう。まず, (2.1) にしたがい, $a_0 + a_2$ と $a_1 + a_3$ を長さ 2 の FFT に入力する。 ところで, 長さ 2 のフーリエ変換は離散フーリエ変換の定義式によって直接計算すると $A_0 = a_0 + a_1$, $A_1 = a_0 - a_1$ となるので, シグナルフローグラフを描くと図 2.1(a) のようになる。この演算は蝶が羽根を広げているように見えることからバタフライ演算と呼ばれる。さて, (2.1) によると長さ 2 の FFT の $\{A_0, A_1\}$ 出力が, 長さ 4 の FFT の $\{A_0, A_2\}$ 出力に対応している。一方, (2.2) にしたがい, $a_0 - a_2$ と $(a_1 - a_3)W_4$ を長さ 2 の FFT に入力する。この長さ 2 の FFT に関しては, $\{A_0, A_1\}$ 出力が長さ 4 の FFT の $\{A_1, A_3\}$ 出力に対応していることが (2.2) からわかる。 よって, 長さ 4 の FFT は図 2.1(b) のシグナルフローグラフによって実現できる。

$N = 8$ についても同様に, 長さ 4 の FFT を後段に配置して図 2.1(c) のようなシグナルフローをつくれば長さ 8 の FFT が実現できるはずである。

これで規則性が見えてきたのではないだろうか。任意の N (ただし 2 のべき乗) については, 図 2.2 に示すように第 1 段目のバタフライ演算を通過したデータを入力して長さ $N/2$ の FFT を実行すれば長さ N の FFT が計算できる。その長さ $N/2$ の FFT も同様に 1 段のバタフライ演算と長さ $N/4$ の FFT によって構成されているはずであるから, フーリエ変換する系列はバタフライ演算を通過するたびに長さが 2 分の 1 になっていく。このような分割は系列の長さが 2 になるまで繰り返される。

[1] これに対し, $A_k = \sum a_{2m} W_N^{2mk} + \sum a_{2m+1} W_N^{(2m+1)k}$ のように数式変形しても類似のアルゴリズムが得られる。この方法は時間軸上で定義された入力データを間引いて導出されているため, 次官間引きアルゴリズムと呼ばれる。

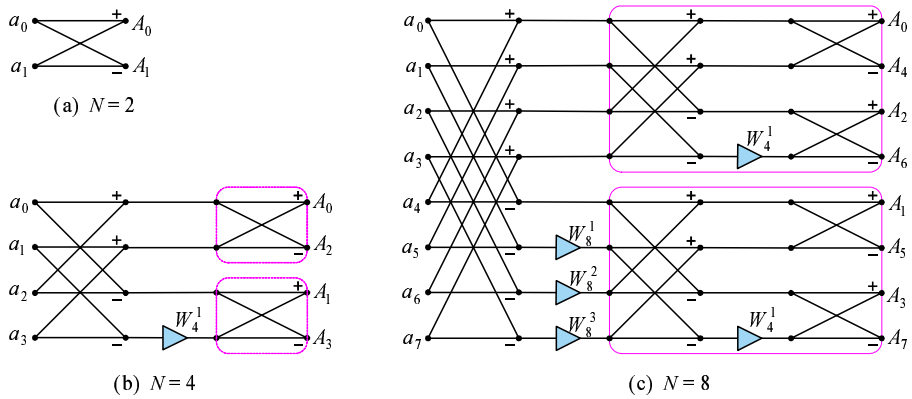


図 2.1: 基数 2 のアルゴリズムにおけるシグナルフローグラフ

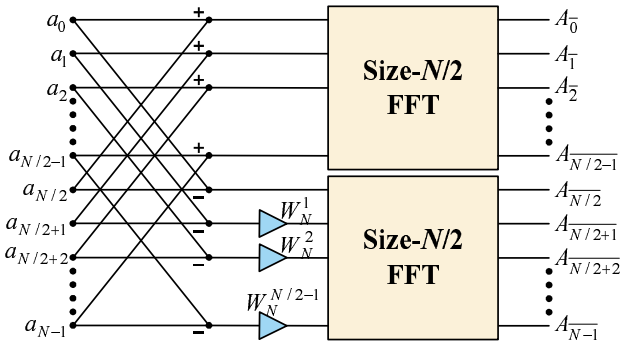


図 2.2: 一般的な基数 2 の FFT のシグナルフローグラフ

ここで注意することは, 出力データの順序である。 先ほどの長さ 4 の FFT や長さ 8 の FFT の場合においても, 出力データが $\{A_0, A_1, A_2, \ldots\}$ のような連番になっていない。実は, 出力されるデータはビット逆順と呼ばれる順序になっている。 例えば $N = 8$ の場合, 出力データ A_m の添え字 m は 0 から 7 の整数であり, その値を 2 進数で表現すると $(b_2b_1b0)_2$ のように 3 桁表記できる。まず, これらを連番に並べた数列 $\{0, 1, 2, 3, 4, 5, 6, 7\}$ を考え, その各要素に対して, 2 進数表現 $(b_2b_1b0)_2$ の桁を逆転させ $(b_0b_1b2)_2$ と読み替えた数列をつくると,$\{0, 4, 2, 6, 1, 5, 3, 7\}$ が得られる。このような数値の順序を**ビット逆順**と呼ぶ。$N = 8$ の場合のビット逆順は確かに図 2.1 に示す長さ 8 の FFT の出力の順序と一致する。長さ N の FFT の出力 A_m の添え字は $\log_2 N$ 桁の 2 進数におけるビット逆順になっている。これは次のようにして証明される。

Proof フーリエ変換する系列の長さが $N = 2^1$ の場合, 出力結果を順に並べると A_0, A_1 である。この添え字の数値 $\{0, 1\}$ は 1 桁の 2 進数で表現でき, その桁を逆転させても同じ数値である。よって, $N = 2^1$ のときの FFT の出力はビット逆順である。

次に, $N = 2^k$ のときの FFT の出力がビット逆順であることを仮定する。すなわち, 出力される成分 A_m の添え字 m が k 桁の 2 進数で表現され, それらの順に並べると,

0 番目:	$(00000\cdots00000)_2$
1 番目:	$(10000\cdots00000)_2$
2 番目:	$(01000\cdots00000)_2$
3 番目:	$(11000\cdots00000)_2$
$\vdots$	$\vdots$
2^k-3 番目:	$(01111\cdots11111)_2$
2^k-2 番目:	$(01111\cdots11111)_2$
2^k-1 番目:	$(11111\cdots11111)_2$

となっているはずである。その仮定の上で長さ $N = 2^{k+1}$ の FFT の結果を考えてみる。図 2.1 のようなシグナルフローグラフを考えたとき, バタフライ演算を 1 段通過した後, 系列は前半と後半に分離され, それぞれが長さ 2^k の FFT に入力される。基数 2 のアルゴリズム (2.1), (2.2) によると, 前半の系列を入力した長さ 2^k の FFT の A_m 出力が最終的な長さ 2^{k+1} の FFT の A_{2m} 出力となり, 後半の系列を入力した長さ 2^k の FFT の A_m 出力が最終的な長さ 2^{k+1} の FFT の A_{2m+1} 出力なる。このことから, 長さ 2^{k+1} の FFT の出力結果の添え字を 2 進数表現して順に並べると,

$$
\begin{array}{ll}
0\text{ 番目:} & (00000\cdots000000)_2 \\
1\text{ 番目:} & (10000\cdots000000)_2 \\
2\text{ 番目:} & (01000\cdots000000)_2 \\
3\text{ 番目:} & (11000\cdots000000)_2 \\
\vdots & \vdots \\
2^k-2\text{ 番目:} & (01111\cdots111110)_2 \\
2^k-1\text{ 番目:} & (11111\cdots111110)_2 \\
2^k\text{ 番目:} & (00000\cdots000001)_2 \\
2^k+1\text{ 番目:} & (10000\cdots000001)_2 \\
2^k+2\text{ 番目:} & (01000\cdots000001)_2 \\
\vdots & \vdots \\
2^{k+1}-2\text{ 番目:} & (01111\cdots111111)_2 \\
2^{k+1}-1\text{ 番目:} & (11111\cdots111111)_2
\end{array}
$$

となる。これらの 2 進数の桁を逆転して読んでみると 0 から $2^{k+1}-1$ の連番になっていることが確認できる。FFT 出力の添え字がビット逆転になっていることは $N=2^1$ のときに成り立ち, さらに, $N=2^k$ のときに成り立つと仮定すれば $N=2^{k+1}$ のときにも矛盾なく成り立つ。よって, 任意の 2 のべき乗 N を長さとする FFT の出力の添え字はビット逆順になっている。¶

図 2.2 に示す FFT の演算量を評価してみよう。この図を見ると第 1 段目のバタフライ演算には $N/2-1$ この複素乗算が含まれていることがわかる。ただし, そのうちの 1 つは $W_4\equiv -i$ による乗算であり, それは単に複素数の実部と虚部を入れ替えるだけの操作で実現できる。よって, 第 1 段目のバタフライ演算において実際に乗算命令を要する複素乗算は $N/2-2$ 個である。引き続き実行される長さ $N/2$ の FFT も同じ構造になっているはずであるから, その第 1 段目のバタフライ演算には $N/4-2$ 個の複素乗算が含まれる。このようにして複素乗算の個数を積み上げていくと, 長さ N の FFT に含まれる複素乗算の総数は

$$\nu_{\mathrm{rad2}}=\sum_{m=0}^{\log_2 N-2}2^m\left(\frac{N}{2^{m+1}}-1\right)=\frac{1}{2}N\log_2 N-\frac{3}{2}N+2$$

となることがわかる。もう一方, 複素加算の総数は $N\log_2 N$ であることが容易にわかる。さらに, この結果を発展させて現実的な実装に必要とする加算命令と乗算命令の回数を計算することができる。 導出は付録に記載するが, 加算回数を $\nu_{\mathrm{rad2}}^{(\mathrm{add})}$, 乗算回数を $\nu_{\mathrm{rad2}}^{(\mathrm{mul})}$ とすると,

$$\begin{aligned}\nu_{\mathrm{rad2}}^{(\mathrm{add})}&=3N\log_2 N-3N+4,\\ \nu_{\mathrm{rad2}}^{(\mathrm{mul})}&=2N\ pg_2N-7N+12\end{aligned}$$

となる。よって, 長さ N の FFT の演算量は $O(N\log N)$ となり, 離散フーリエ変換の定義式 (1) を直接計算する場合に比べ演算量を大きく改善できる。

さて, 基数 2 の FFT が再帰構造をもっていることを利用すると, 下に示すように C 言語による再帰呼び出しを用いて FFT を実装することができる。ただし, このプログラムは

原理をわかりやすく説明するためのプログラムであるので実行速度は速くない。例えば, バタフライ演算を連続的に実行するためのループの中で回転因子を逐一計算しているのは高速演算には向かない。高速演算を実現するにはさらに最適化が必要である。

引数について説明すると, *lpdfX* と *lpdfY* は, それぞれ, フーリエ変換する系列の実部と虚部を格納した配列へのポインタである。第 3 引数 *Len* はフーリエ変換する系列の長さ (2 のべき乗) である。このプログラムは inplace フーリエ変換のという方法の処理であり, 変換結果は *lpdfX* と *lpdfY* で指定される配列に上書きされる。

プログラムの処理に関して説明すると, FFT_Base2 関数はサブルーチンコールをしているだけであり, 実際の計算をしているのは fft_Base2 関数である。その関数は第 1 段目のバタフライ演算を実行した後, 自分自身 (fft_Base2 関数) を再帰呼び出ししている。ただし, 長さ 2 の FFT である場合には再帰呼び出しではない経路が実行される。また, fft_Base2 関数から戻ってきた時点では, *lpdfX* と *lpdfY* で指定される配列には図 2.2 のようにビット逆順で結果が格納されているため, 正しい順序に並べ替える必要がある。その順序並べ替えは fft_BitRev 関数で実行されるが, その並べ替えアルゴリズムについては後に説明するので、fft_BitRev 関数はここではブラックボックスであることにしておく。

```
#define _PI     3.14159265358979323846
#define _2PI    (2.0 * _PI)

static void fft_Base2(double *lpdfX, double *lpdfY, int Len);
static void fft_BitRv(double *lpdfX, double *lpdfY, int Len);

void Fft_Base2(double *lpdfX, double *lpdfY, int Len)
{
    fft_Base2(lpdfX, lpdfY, Len);
    fft_BitRv(lpdfX, lpdfY, Len);
}

static void fft_Base2(double *lpdfX, double *lpdfY, int Len)
{
    double dfX0, dfY0;
    double dfX1, dfY1;

    if (Len == 2) {
        dfX0 = lpdfX[0];    dfY0 = lpdfY[0];
        dfX1 = lpdfX[1];    dfY1 = lpdfY[1];

        lpdfX[0] = dfX0 + dfX1;    lpdfY[0] = dfY0 + dfY1;
        lpdfX[1] = dfX0 - dfX1;    lpdfY[1] = dfY0 - dfY1;
    } else {
        int i;

        dfX0 = lpdfX[0];               dfY0 = lpdfY[0];
        dfX1 = lpdfX[Len/2];           dfY1 = lpdfY[Len/2];
        lpdfX[0]     = dfX0 + dfX1;    lpdfY[0]     = dfY0 + dfY1;
        lpdfX[Len/2] = dfX0 - dfX1;    lpdfY[Len/2] = dfY0 - dfY1;
```

```
        for (i=0; i < Len/2; i++) {
            double dfWx =  cos(_2PI * (double)i / (double)Len);
            double dfWy = -sin(_2PI * (double)i / (double)Len);

            dfX0 = lpdfX[i];        dfY0 = lpdfY[i];
            dfX1 = lpdfX[i+Len/2];  dfY1 = lpdfY[i+Len/2];

            lpdfX[i] = dfX0 + dfX1;
            lpdfY[i] = dfY0 + dfY1;
            dfX0 -= dfX1;
            dfY0 -= dfY1;
            lpdfX[i] = dfX0 * dfWx - dfY0 * dfWy;
            lpdfY[i] = dfX0 * dfWy + dfY0 * dfWx;
        } /* <----- end for */

        fft_Base2(lpdfX,       lpdfY,       Len/2);
        fft_Base2(lpdfX+Len/2, lpdfY+Len/2, Len/2);
    }
}
```

2.2 基数 4 のアルゴリズム

フーリエ変換する系列の長さ N を 4 分割して, 長さ $N/4$ の FFT を統合することによって長さ N の FFT を得る方法を基数 4 のアルゴリズムという。このアルゴリズムは基数 2 よりも演算量を小さくできることが知られている。

フーリエ変換 A_m を, 4 分の 1 に周波数間引きした 3 つの系列 A_{4k}, A_{4k+1}, A_{4k+2}, A_{4k+3} 分け, それぞれを離散フーリエ変換の定義式に代入してみると,

$$A_{4k} = \sum_{n=0}^{N/4-1} \Big[(a_n + a_{n+N/2}) + (a_{n+N/4} + a_{n+3N/4})\Big] \cdot W_{N/4}^{nk} \tag{2.3}$$

$$A_{4k+1} = \sum_{n=0}^{N/4-1} \Big[(a_n - a_{n+N/2}) - i(a_{n+N/4} - a_{n+3N/4})\Big] W_N^n \cdot W_{N/4}^{nk} \tag{2.4}$$

$$A_{4k+2} = \sum_{n=0}^{N/4-1} \Big[(a_n + a_{n+N/2}) - (a_{n+N/4} + a_{n+3N/4})\Big] W_N^{2n} \cdot W_{N/4}^{nk} \tag{2.5}$$

$$A_{4k+3} = \sum_{n=0}^{N/4-1} \Big[(a_n - a_{n+N/2}) + i(a_{n+N/4} - a_{n+3N/4})\Big] W_N^n \cdot W_{N/4}^{nk} \tag{2.6}$$

$$\tag{2.7}$$

を得る。これらの数式からわかるように, 4 分の 1 に周波数間引きしたフーリエ変換は長さ $N/4$ の FFT で表現されている。この結果をシグナルフローとして描くと図 2.3 のようになる。図のようにバタフライ演算を 2 段通過した結果を, 長さ $N/4$ の FFT に入力すれば長さ N の FFT が計算される。

基数 4 のアルゴリズムでは変換する系列の長さ N を 4 分の 1 に分割して再帰的に FFT

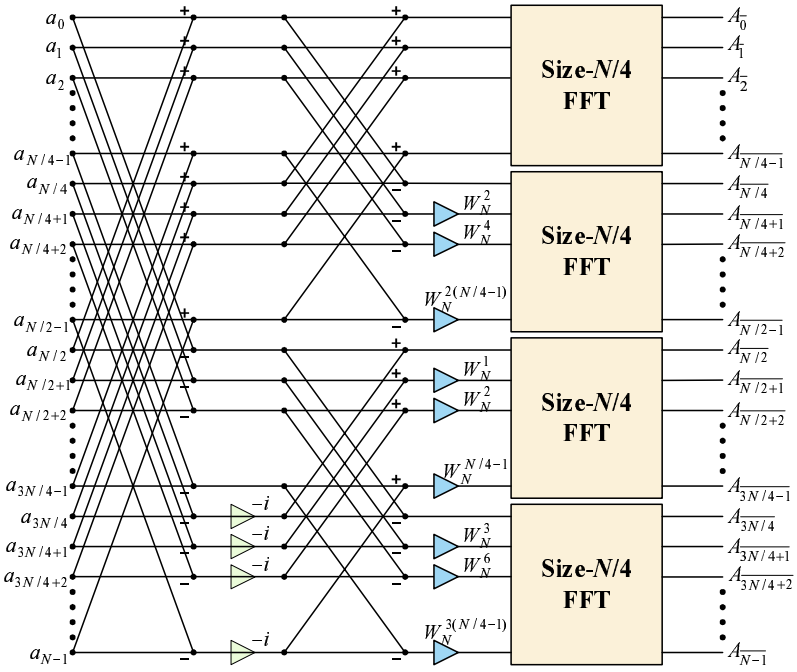


図 2.3: 一般的な基数 4 の FFT のシグナルフローグラフ

を実行する。この分割を繰り返すたびに系列の長さは 4 分の 1 になっていくのだが, 最初の系列の長さ N に関して $\log_2 N$ が偶数である場合, 最終的には長さ 4 の FFT まで細分化される。それに対して, $\log_2 N$ が奇数である場合, 最終的には長さ 2 の FFT まで細分化される。その例として, $N = 8$, $N = 16$ のときのシグナルフローグラフを描くと図 2.4 のようになる。図 2.4(a) のように $\log_2 N$ が奇数であれば基数 4 アルゴリズムの末端は長さ 2 の FFT に, 図 2.4(b) のように $\log_2 N$ が偶数であれば末端は長さ 4 の FFT になる。つまり, この末端部における条件分岐さえできていれば奇数 4 の FFT も基数 2 の場合とほぼ同様に実現できる。

すでに示したシグナルフローからわかるように, 第 1 段目のバタフライ演算に後置する乗算器の乗数がすべて $-i$ になっている。これは実部と虚部を交換するだけの演算であり, 乗算器を必要としない。(その意味で, シグナルフローも他の乗算器とは区別して描いている。) このため, 基数 4 の FFT は基数 2 よりも乗算回数を少なくでき, 高速化が可能である。基数 4 の周波数間引きアルゴリズムによって長さ N の FFT を実現する場合の加算回数 $\nu_{\mathrm{rad4}}^{(\mathrm{add})}$ と乗算回数 $\nu_{\mathrm{rad4}}^{(\mathrm{mul})}$ は

$$\nu_{\mathrm{rad4}}^{(\mathrm{add})} = \frac{11}{4} N \log_2 N - \frac{15 + (-1)^{\log_2 N}}{8} N + 3 - (-1)^{\log_2 N} \tag{2.8}$$

$$\nu_{\mathrm{rad4}}^{(\mathrm{mul})} = \frac{3}{4} N \log_2 N - \frac{15 + (-1)^{\log_2 N}}{4} N + 10 \tag{2.9}$$

となるので, 基数 2 アルゴリズムに比べ加算回数は 0.92 倍に, 乗算回数は 0.75 倍に減少し

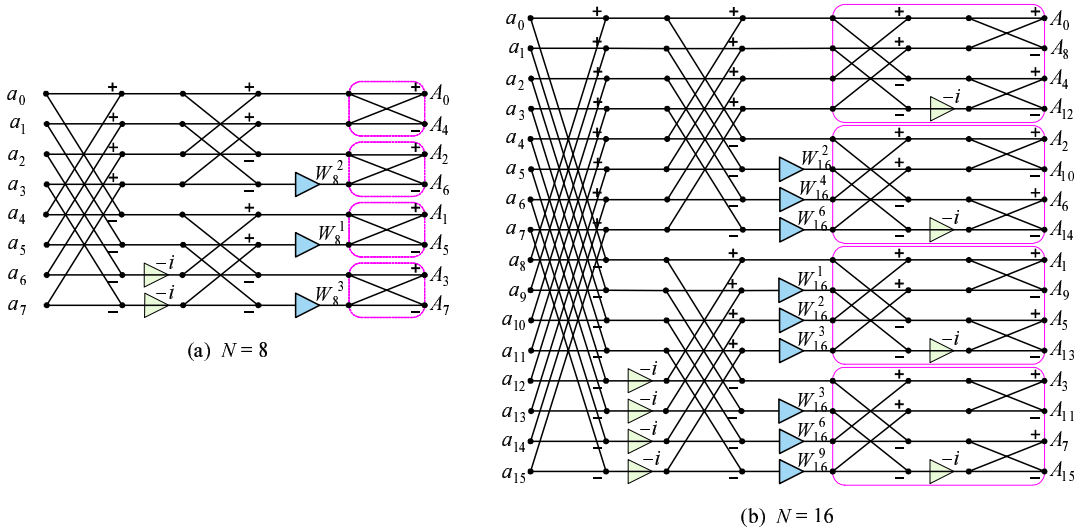


図 2.4: 基数 4 の FFT のシグナルフローの例

ている。基数 4 のアルゴリズムによる乗算回数の減少は図 2.5 を見るとわかりやすい。この図は FFT における最初の 2 段のバタフライ演算を表している。特徴をつかみやすくするため, a_k, $a_{k+N/4}$, $a_{k+N/2}$, $a_{k+3N/4}$ のみを抽出して描いている。 図 2.5(a) において, 第 1 段目のバタフライ演算に後置する乗算器 W_N^k と $W_N^{k+N/4}$ は共通因子 W_N^k をもっている。この共通因子をはずし, 次段のバタフライ演算に後置させると図 2.5(b) のように, 第 2 段目のバタフライ演算に後置する乗算器が 1 つ増えるが第 1 段目のバタフライ演算に後置する乗算器を削除できる。この図において, 基数 2 のアルゴリズムでは乗算器が 4 個必要であったが, 基数 4 アルゴリズムでは乗算器が 3 個になっている。FFT 全体ではこのような乗算器の削減がシグナルフロー全体に拡がっているため, 乗算器の数は基数 2 アルゴリズムの 0.75 倍になっているのである。

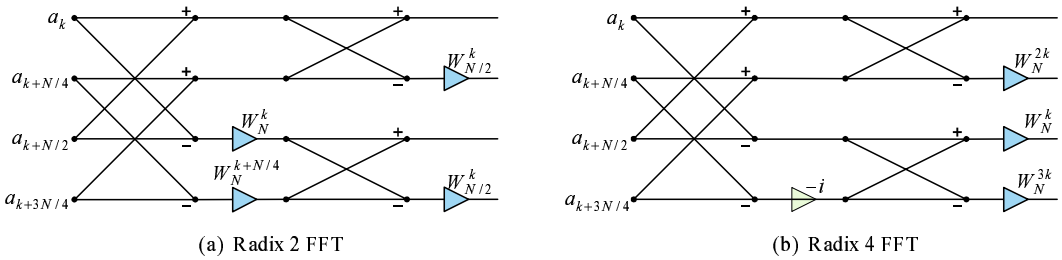


図 2.5: 基数 2 と基数 4 のシグナルフローの比較

基数 4 の FFT を C 言語によって実装してみよう。ここでも FFT の再帰構造を C 言語による再帰呼び出しによって実現してみた。このプログラムも原理説明のためのプログラムであるので, 実行速度は速くない。引数について説明すると, *lpdfX* と *lpdfY* は, それぞれ, フーリエ変換する系列の実部と虚部を格納した配列へのポインタである。第 3 引数

Len はフーリエ変換する系列の長さ (2 のべき乗) である。このプログラムは inplace フーリエ変換という処理であり, 変換結果は *lpdfX* と *lpdfY* がさす配列に上書きされるようになっている。

プログラムの処理に関して説明すると, FFT を計算するのは fft_Base4 関数であるが, その関数は第 1 段目と第 2 段目のバタフライ演算を実行した後, 自分自身 (fft_Base4 関数) を再帰呼び出ししている。ただし、フーリエ変換する系列の長さが 2, または, 4 である場合には再帰呼び出しをしない経路が実行される。また, fft_Base4 関数から戻ってきた時点では, *lpdfX* と *lpdfY* で指定される配列には図 2.3 のようにビット逆順で結果が格納されているため, fft_BitRev 関数によって正しい順序に並べ替えている。

```
#define _PI     3.14159265358979323846
#define _2PI    (2.0 * _PI)

static void fft_Base4(double *lpdfX, double *lpdfY, int Len);
static void fft_BitRv(double *lpdfX, double *lpdfY, int Len);

void Fft_Base4(double *lpdfX, double *lpdfY, int Len)
{
    fft_Base4(lpdfX, lpdfY, Len);
    fft_BitRv(lpdfX, lpdfY, Len);
}

static void fft_Base4(double *lpdfX, double *lpdfY, int Len)
{
    double dfX0, dfX1, dfX2, dfX3;
    double dfY0, dfY1, dfY2, dfY3;

    if (Len == 2) {
        dfX0 = lpdfX[0];  dfY0 = lpdfY[0];
        dfX1 = lpdfX[1];  dfY1 = lpdfY[1];

        lpdfX[0] = dfX0 + dfX1;    lpdfY[0] = dfY0 + dfY1;
        lpdfX[1] = dfX0 - dfX1;    lpdfY[1] = dfY0 - dfY1;
    } else if (Len == 4) {
        dfX0 = lpdfX[0] + lpdfX[2];  dfY0 = lpdfY[0] + lpdfY[2];
        dfX1 = lpdfX[1] + lpdfX[3];  dfY1 = lpdfY[1] + lpdfY[3];
        dfX2 = lpdfX[0] - lpdfX[2];  dfY2 = lpdfY[0] - lpdfY[2];
        dfX3 = lpdfY[1] - lpdfY[3];  dfY3 = lpdfX[3] - lpdfX[1];

        lpdfX[0] = dfX0 + dfX1;      lpdfY[0] = dfY0 + dfY1;
        lpdfX[1] = dfX0 - dfX1;      lpdfY[1] = dfY0 - dfY1;
        lpdfX[2] = dfX2 + dfX3;      lpdfY[2] = dfY2 + dfY3;
        lpdfX[3] = dfX2 - dfX3;      lpdfY[3] = dfY2 - dfY3;
    } else {
        int i0 = 0;
        int i1 = i0 + Len/4;
        int i2 = i1 + Len/4;
        int i3 = i2 + Len/4

        dfX0 = lpdfX[i0] + lpdfX[i2];    dfY0 = lpdfY[i0] + lpdfY[i2];
        dfX1 = lpdfX[i1] + lpdfX[i3];    dfY1 = lpdfY[i1] + lpdfY[i3];
```

```
        dfX2 = lpdfX[i0] - lpdfX[i2];    dfY2 = lpdfY[i0] - lpdfY[i2];
        dfX3 = lpdfY[i1] + lpdfY[i3];    dfY3 = lpdfX[i3] - lpdfX[i1];

        lpdfX[i0] = dfX0 + dfX1;    lpdfY[i0]   = dfY0 + dfY1;
        lpdfX[i0] = dfX0 + dfX1;    lpdfY[i0]   = dfY0 + dfY1;
        lpdfX[i0] = dfX0 + dfX1;    lpdfY[i0]   = dfY0 + dfY1;
        lpdfX[i0] = dfX0 + dfX1;    lpdfY[i0]   = dfY0 + dfY1;

        i0++;  i1++;  i2++;  i3++;
        for ( ; i0 < Len/4; i0++, i1++, i2++, i3++;) {
            dfWx1 =  cos(_2PI     * (double)i0 / (double)Len);
            dfWy1 = -sin(_2PI     * (double)i0 / (double)Len);
            dfWx2 =  cos(_2PI * 2 * (double)i0 / (double)Len);
            dfWy2 = -sin(_2PI * 2 * (double)i0 / (double)Len);
            dfWx3 =  cos(_2PI * 3 * (double)i0 / (double)Len);
            dfWy3 = -sin(_2PI * 3 * (double)i0 / (double)Len);

            dfX0 = lpdfX[i0] + lpdfX[i2];    dfY0 = lpdfY[i0] + lpdfY[i2];
            dfX1 = lpdfX[i1] + lpdfX[i3];    dfY1 = lpdfY[i1] + lpdfY[i3];
            dfX2 = lpdfX[i0] - lpdfX[i2];    dfY2 = lpdfY[i0] - lpdfY[i2];
            dfX3 = lpdfY[i1] + lpdfY[i3];    dfY3 = lpdfX[i3] - lpdfX[i1];

            lpdfX[i0] = dfX0 + dfX1;
            lpdfY[i0] = dfY0 + dfY1;
            dfX0 = dfX0 - dfX1;
            dfY0 = dfY0 - dfY1;
            lpdfX[i1] = dfX0 * dfWx2 - dfY0 * dfXy2;
            lpdfY[i1] = dfX0 * dfWy2 + dfY0 * dfXx2;

            dfX0 = dfX2 + dfX3;
            dfY0 = dfY2 + dfY3;
            lpdfX[i2] = dfX0 * dfWx1 - dfY0 * dfXy1;
            lpdfY[i2] = dfX0 * dfWy1 + dfY0 * dfXx1;
            dfX0 = dfX2 - dfX3;
            dfY0 = dfY2 - dfY3;
            lpdfX[i1] = dfX0 * dfWx3 - dfY0 * dfXy3;
            lpdfY[i1] = dfX0 * dfWy3 + dfY0 * dfXx3;
        } /* <----- end for */

        fft_Base4(lpdfX,           lpdfY,           Len/4);
        fft_Base4(lpdfX+(  Len/4), lpdfY+(  Len/4), Len/4);
        fft_Base4(lpdfX+(  Len/2), lpdfY+(  Len/2), Len/4);
        fft_Base4(lpdfX+(3*Len/4), lpdfY+(3*Len/4), Len/4);
    }
}
```

2.3 Split Radix アルゴリズム

Split Radix アルゴリズムは基数 2 アルゴリズムと基数 4 アルゴリズムを混合したアルゴリズムであり, どの基数のアルゴリズムよりも演算量を小さくできることが知られている。このアルゴリズムは 1968 年に R. Yavne によって発見されたのだが, そのときに用い

た記法が一般には受け入れられず, 一度忘れ去られ, 1984 年に P. Duhamel と H. Hollmann がこのアルゴリズムを再発見し, 論文中で Split Radix という名称を用いて以来, 広く知られるようになった。この節では Split Radix アルゴリズムについて解説する。

前節で紹介した基数 4 アルゴリズムにはもう少し乗算器を削減する余地がある。前節の図 2.5 を見ると, 確かに $a_{k+N/2}$ と $a_{k+3N/4}$ から水平方向に伸びるシグナルフローでは乗算器が減っているが, 系列の前半にあたる a_k と $a_{k+N/4}$ から水平方向に伸びるシグナルフローについては全く変化がない。そのため, 効果がある後半のみを基数 4 で展開し, 前半は基数 2 の展開にとどめるというのが Split Radix アルゴリズムである。

Split Radix アルゴリズムでは図 2.6 に示すように L 字型に処理のステージが進行する。具体的には, 基数 4 が適用される系列の後半ではバタフライ演算を 2 段通過した後に次の処理ステージ, すなわち, 長さ $N/4$ の FFT に移るのだが, 基数 2 が適用される系列の前半ではバタフライ演算を 1 段通過すると早速, 長さ $N/2$ の FFT に移るのである。図 2.6 は基数 4 アルゴリズムと Spilit Raidx アルゴリズムの比較をしている。Split Radix では系列の前半は基数 2 が適用されるため, バタフライ演算を 1 段通過すると第 2 ステージに入り, 第 2 ステージにおける基数 4 の展開によって, 基数 4 アルゴリズムで減らすことができなかった乗算器を減らすことができている。 それによって, Split Radix アルゴリズムは基数 4 アルゴリズムよりも小さい演算量を得ることができる。

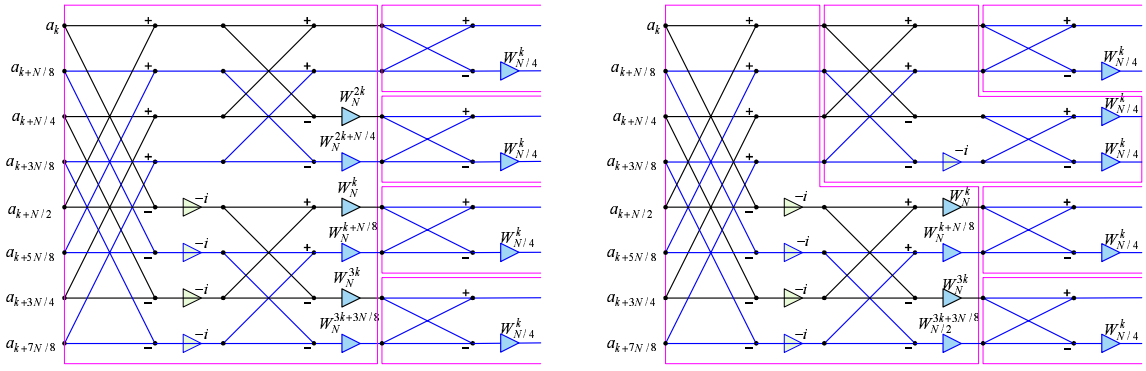


図 2.6: 基数 4 と Split Radix のシグナルフローの比較

Split Radix による長さ N の FFT の一般的なシグナルフローグラフは図 2.7 のように L 字型に組まれたバタフライ演算ブロックと, それに続く細分化された FFT の処理ブロックで構成される。L 字型のバタフライ演算ブロックの出力の半数はバタフライ演算を 1 段しか通過せず, 残りの半数は基数 4 の FFT のようにバタフライ演算を 2 段通過する。バタフライ演算を 1 段しか通過しなかった信号は長さ $N/2$ の FFT に, 2 段通過した信号は長さ $N/4$ の FFT に入力される。それらの長さ $N/2$ の FFT や長さ $N/4$ の FFT の出力を並べると, 基数 2 や基数 4 のときと同様に, 長さ N の系列をフーリエ変換した結果がビット逆順になっている。 内部で実行される長さ $N/2$ の FFT や長さ $N/4$ の FFT も同様に細分

化され, より小さなサイズの FFT が内部で実行される。その際分かは, 長さ 2, または, 長さ 4 の FFT に分割されるまで再帰的に繰り返されている。

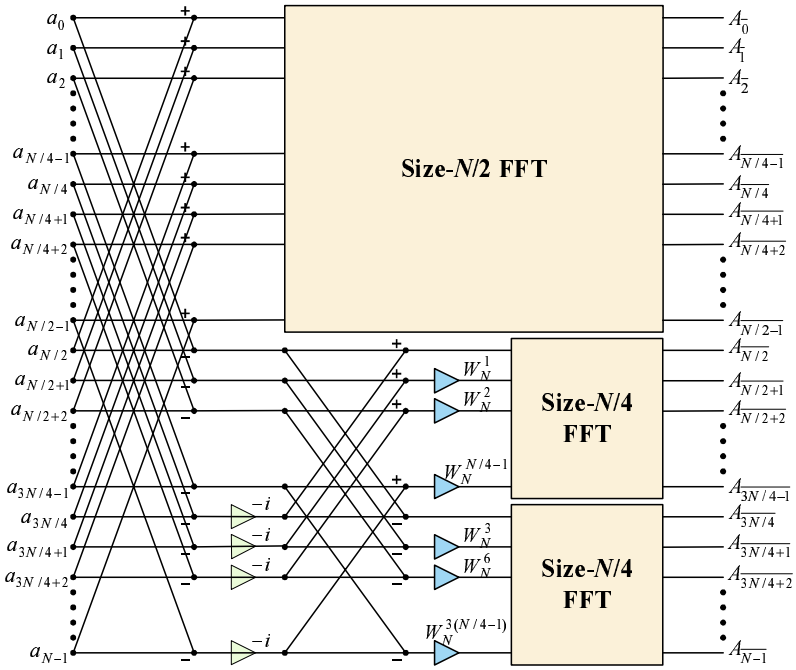


図 2.7: 一般的な Split Radix FFT のシグナルフローグラフ

Split Radix FFT は基数 4 の FFT では改善されなかった場所の乗算器を削減するアルゴリズムである。そのために, 基数 4 では効果がない部分を基数 2 の構成にとどめ, 次のステージに処理を移行させることによって乗算器の削減を狙っている。 導出は付録にまわすとして, 長さ N の FFT を Split Radix アルゴリズムで実行したときの加算回数 $\nu_{\mathrm{spr}}^{(\mathrm{add})}$ と乗算回数 $\nu_{\mathrm{spr}}^{(\mathrm{mul})}$ は

$$\nu_{\mathrm{spr}}^{(\mathrm{add})} = \frac{8}{3} N \log_2 N - \frac{16}{9} N + 2 - \frac{2}{9}(-1)^{\log_2 N}$$
$$\nu_{\mathrm{spr}}^{(\mathrm{mul})} = \frac{4}{3} N \log_2 N - \frac{38}{9} N + 6 - \frac{2}{9}(-1)^{\log_2 N}$$

となる。つまり, Split Radix FFT アルゴリズムは基数 4 に比べて加算回数が 0.97 倍, 乗算回数が 0.89 倍になっている。 基数 2 と比べると加算回数は 0.89 倍, 乗算回数は 0.67 倍である。

Split Radix アルゴリズムもその再帰構造に着目すれば C 言語によって簡単に実現できる。実行速度に関する最適化をしていないが, 原理を説明するためのプログラムを書くと下のサンプルプログラムのようになる。引数について説明すると, *lpdfX* と *lpdfY* は, それぞれ, フーリエ変換する系列の実部と虚部を格納した配列へのポインタである。第 3 引数 *Len* はフーリエ変換する系列の長さ (2 のべき乗) である。このプログラムは inplace フーリエ変換という処理であり, 変換結果は *lpdfX* と *lpdfY* が指す配列に上書きされる。

最上位の関数 FFT_Transform はサブルーチンコールをしているだけである。実際の計算を実行するのは, その下位にあたる fft_SpRad 関数であり, その関数は L 字型のバタフライ演算ブロックを実行し, 自分自身 (fft_SpRad 関数) を再帰呼び出ししている。ただし, 長さ 2, または, 長さ 4 の FFT の場合には再帰呼び出しをしない経路が実行される。また, fft_SpRad 関数から戻ってきた時点では, *lpdfX* と *lpdfY* で指定される配列にはビット逆順で結果が格納されているため, fft_BitRev 関数によって結果を正しい順序に並び替えている。

```
#define _PI     3.14159265358979323846
#define _2PI    (2.0 * _PI)

static void fft_SpRad(double *lpdfX, double *lpdfY, int Len);
static void fft_BitRv(double *lpdfX, double *lpdfY, int Len);

void Fft_SpRad(double *lpdfX, double *lpdfY, int Len)
{
    fft_Base4(lpdfX, lpdfY, Len);
    fft_BitRv(lpdfX, lpdfY, Len);
}

static void fft_SpRad(double *lpdfX, double *lpdfY, int Len)
{
    double dfX0, dfX1, dfX2, dfX3;
    double dfY0, dfY1, dfY2, dfY3;

    if (Len == 2) {
        dfX0 = lpdfX[0];  dfY0 = lpdfY[0];
        dfX1 = lpdfX[1];  dfY1 = lpdfY[1];

        lpdfX[0] = dfX0 + dfX1;    lpdfY[0] = dfY0 + dfY1;
        lpdfX[1] = dfX0 - dfX1;    lpdfY[1] = dfY0 - dfY1;
    } else if (Len == 4) {
        dfX0 = lpdfX[0] + lpdfX[2];  dfY0 = lpdfY[0] + lpdfY[2];
        dfX1 = lpdfX[1] + lpdfX[3];  dfY1 = lpdfY[1] + lpdfY[3];
        dfX2 = lpdfX[0] - lpdfX[2];  dfY2 = lpdfY[0] - lpdfY[2];
        dfX3 = lpdfY[1] - lpdfY[3];  dfY3 = lpdfX[3] - lpdfX[1];

        lpdfX[0] = dfX0 + dfX1;      lpdfY[0] = dfY0 + dfY1;
        lpdfX[1] = dfX0 - dfX1;      lpdfY[1] = dfY0 - dfY1;
        lpdfX[2] = dfX2 + dfX3;      lpdfY[2] = dfY2 + dfY3;
        lpdfX[3] = dfX2 - dfX3;      lpdfY[3] = dfY2 - dfY3;
    } else {
        int i0 = 0;
        int i1 = i0 + Len/4;
        int i2 = i1 + Len/4;
        int i3 = i2 + Len/4

        dfX0 = lpdfX[i0] + lpdfX[i2];    dfY0 = lpdfY[i0] + lpdfY[i2];
        dfX1 = lpdfX[i1] + lpdfX[i3];    dfY1 = lpdfY[i1] + lpdfY[i3];
        dfX2 = lpdfX[i0] - lpdfX[i2];    dfY2 = lpdfY[i0] - lpdfY[i2];
        dfX3 = lpdfY[i1] + lpdfY[i3];    dfY3 = lpdfX[i3] - lpdfX[i1];
```

```
        lpdfX[i0] = dfX0;           lpdfY[i0]   = dfY0;
        lpdfX[i1] = dfX1;           lpdfY[i1]   = dfY1;
        lpdfX[i2] = dfX2 + dfX3;    lpdfY[i2]   = dfY2 + dfY2;
        lpdfX[i3] = dfX2 - dfX3;    lpdfY[i3]   = dfY3 - dfY3;

        i0++;  i1++;  i2++;  i3++;
        for ( ; i0 < Len/4; i0++, i1++, i2++, i3++;) {
            dfWx1 =  cos(_2PI     * (double)i0 / (double)Len);
            dfWy1 = -sin(_2PI     * (double)i0 / (double)Len);
            dfWx3 =  cos(_2PI * 3 * (double)i0 / (double)Len);
            dfWy3 = -sin(_2PI * 3 * (double)i0 / (double)Len);

            dfX0 = lpdfX[i0] + lpdfX[i2];    dfY0 = lpdfY[i0] + lpdfY[i2];
            dfX1 = lpdfX[i1] + lpdfX[i3];    dfY1 = lpdfY[i1] + lpdfY[i3];
            dfX2 = lpdfX[i0] - lpdfX[i2];    dfY2 = lpdfY[i0] - lpdfY[i2];
            dfX3 = lpdfY[i1] + lpdfY[i3];    dfY3 = lpdfX[i3] - lpdfX[i1];

            lpdfX[i0] = dfX0;    lpdfY[i0] = dfY0;
            lpdfX[i1] = dfX1;    lpdfY[i1] = dfX1;

            dfX0 = dfX2 + dfX3;
            dfY0 = dfY2 + dfY3;
            lpdfX[i2] = dfX0 * dfWx1 - dfY0 * dfXy1;
            lpdfY[i2] = dfX0 * dfWy1 + dfY0 * dfXx1;
            dfX0 = dfX2 - dfX3;
            dfY0 = dfY2 - dfY3;
            lpdfX[i1] = dfX0 * dfWx3 - dfY0 * dfXy3;
            lpdfY[i1] = dfX0 * dfWy3 + dfY0 * dfXx3;
        } /* <----- end for */

        fft_SpRad(lpdfX,           lpdfY,           Len/2);
        fft_Base4(lpdfX+(  Len/2), lpdfY+(  Len/2), Len/4);
        fft_Base4(lpdfX+(3*Len/4), lpdfY+(3*Len/4), Len/4);
    }
}
```

第3章　ビット逆順

前章のように, FFT のアルゴリズムにしたがってシグナルフローを流すと, 何段にもわたるバタフライ演算を通過した後, ビット逆順で変換結果が出力される。そのため, この出力を正しいデータ順に並べ替えることが必要となる。この章ではビット逆順に並べ替えるアルゴリズムを説明する。

3.1　直接的なビット逆順

ビット逆順を得るには, 系列 A_m の添え字 m を 2 進数表記したとき $m = (b_{M-1}b_{M-2}\cdots b_2b_1b_0)_2$ とすると, 直接的に桁を反転させて得られる $k = (b_0b_1b_2 \cdot b_{M-2}b_{M-1})_2$ を求め, A_m と A_k を交換すればよい。直接的に桁を逆転させるとは, 図 3.1 のような処理を考えればよい。この図では, フーリエ変換の結果 A_m の添え字 m が M ビットの 2 進数であることを想定している。 まず, 入力された添え字を右側に 1 ビットシフトする。左側の空いたビットには 0 が入ってきて, 右側からあふれたビットは出力変数の最も右の (最下位) ビットに入る。出力変数は左方向にビットシフトが進行する。ビットシフトを繰り返し, 入力変数が 0 になったとき, 出力変数にはビットの順序を逆転した結果が格納されている。

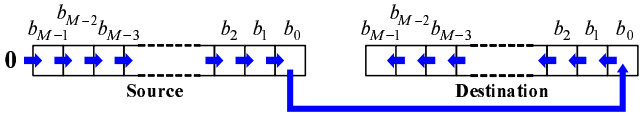


図 3.1: 直接的にビット順を逆転させる方法

上で説明したアルゴリズムを C 言語によって実装すると, 下のサンプルプログラムのようになる。サンプルプログラムに実装される fft_BitRev 関数はこれまでに名前は出てきていたがブラックボックスとして扱われていた。この関数が, ポインタ *lpdfX* と *lpdfY* で指定される配列の要素をビット逆順に並べ替える。

その fft_BitRev 関数はさらに bitrv_Reverse 関数をコールしているが, この関数がビットの順序を逆転させる関数である。このサブルーチンは引数で渡された値のビットを下から調べ, その結果を上のビットから配置していくことによってビットを逆順に並べ替えて

いる。

また, fft_BitRev 関数に戻ると, その関数は配列の添え字 i のビットを逆転させた値 k に代入し, 配列の i 番目と k 番目を交換することによって配列の要素をビット逆順に並べ替えている。ただし, $i = k$ のときはビットを逆転させても値が変化しないので交換の必要がない。また, $i > k$ のときは, すでに交換が完了しているので交換の必要はない。

```
static unsigned long bitrv_Reverse(unsigned long dwVal, int nBits):

static void fft_BitRv(double *lpdfX, doubel *lpdfY, int Len)
{
    int i, nBits;

    i = Len;
    for (nBits=0; i != 0; nBits++)    i >>= 1;

    for (i=0; i < Len; i++) {
        double dfTmpX, dfTmpY;
        int    k = bitrv_Reverse(i, nBits);

        if (i >= k)    continue;
        dfTmpX   = lpdfX[i];  dfTmpY   = lpdfY[i];
        lpdfX[i] = lpdfX[k];  lpdfY[i] = lpdfY[k];
        lpdfX[k] = dfTmpX;    lpdfY[k] = dfTmpY;
    } /* <----- end for */
}

static unsigned long bitrv_Reverse(unsigned long dwVal, int Len)
{
    unsigend long dwRev = 0;
    int           i;

    for (i=0; i < nBits; i++) {
        dwRev = (dwRev << 1) | (dwVal & 0x01);
        dwVal >>= 1;
    } /* <----- end for */
    return dwRev;
}
```

3.2 ビット逆順カウンタ

フーリエ変換におけるビット逆順は, 最終結果における配列の要素の並べ替えだけであるので, 任意の数値のビットを逆転させる必要はなく, $0, 1, 2, \ldots$ のような連番に対応するビット逆順を生成する規則だけでも十分である。例えば, ある数値が与えられたとき, その数値を 1 だけ増加させるような方法で次のビット逆順の数値を得るような仕組みをビット逆順カウンタと呼ぶ。

通常の (ビット逆順していない) 数値について単純なカウンタを得るには, 半加算器と呼ばれる論理回路を用いるとよい。半加算器は図 3.2 のように, 入力された 2 つの 2 値信号

A, B の排他的論理和 S と論理積 C を出力する回路である。この回路は, 図 3.2 に示すような 2 進数の加算結果を出力する。この回路の排他的論理和 S は A と B の加算結果, C は桁上がりである。最下位ビットに 1 を加算し, そのときに生じた桁上がりを上のビットに加え, その桁上がりをさらに上のビットに加算, という具合に半加算器をカスケード接続すれば通常のカウンタを実現できる。

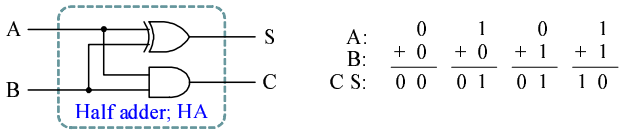


図 3.2: 半加算器の論理構成と実行結果

ビット逆順カウンタの場合は, 通常のカウンタとは逆方向に, すなわち, 上位ビットから下位ビットへ半加算器をカスケード接続することによって実現できる。図 3.3 は, 一例として, M ビットのビット逆順カウンタを示している。ここで, カウント値が $(b_{M-1}b_{M-2}\cdots b_2b_1b_0)_2$ なる 2 進数で表現されているとする。この回路は, 最上位ビット b_{M-1} に 1 を加算し, そのとき生じる桁上がりを下のビットに加えていく。それ以降は, 上のビットからの桁上がりを加算しながら下のビットへ処理を進行させる。この操作が通常の加算とは桁上がりの方向が逆であるので, この回路はビット逆順カウンタとして動作する。

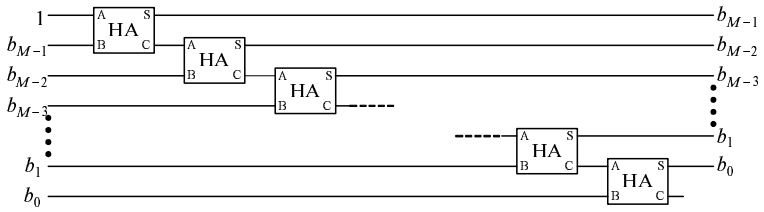


図 3.3: 半加算器による M ビットのビット逆順カウンタ

半加算器によるビット逆順カウンタを C 言語で実現した例が下のプログラムである。この例において, bitrv_Inc 関数がビット逆順カウンタであり, 配列をビット逆順に並べ替える fft_BitRv 関数からコールされている。そのビット逆順カウンタにおいて, 変数 $dwBit$ は半加算器が加算をするビットを表し, $dwCy$ がそのときに生じる桁上がりである。

```
static unsigned long bitrv_Inc(unsigned long dwVal, int nBits):

static void fft_BitRv(double *lpdfX, doubel *lpdfY, int Len)
{
    int i, k;

    for (i=0, k=0; i < Len; i++) {
        double dfTmpX, dfTmpY;
```

```
        k = bitrv_Inc(k, Len);
        if (i >= k)    continue;

        dfTmpX   = lpdfX[i];  dfTmpY   = lpdfY[i];
        lpdfX[i] = lpdfX[k];  lpdfY[i] = lpdfY[k];
        lpdfX[k] = dfTmpX;    lpdfY[k] = dfTmpY;
    } /* <----- end for */
}

static unsigned long bitrv_Inc(unsigned long dwVal, int Len)
{
    unsigend long dwBit, dwCy;

    dwBit = Len >> 1;
    while (wBit >= dwVal) {
        dwCy   = dwVal & dwBit;
        dwVal ^= dwBit;
        dwBit>>= 1;
    } /* <----- end while */
    return dwVal;
}
```

このビット逆順カウンタ bitrv_Inc はもう少し工夫することができる。まず, 桁上がり *dwCy* は桁上がりが生じるとき, 必ず, *dwBit* と同じ値であるので, これらを 1 つにまとめることができる。 さらに, カウンタを 1 つ増加させる処理において, 途中の桁で桁上がりがなくなると, それ以降の桁では 0 を加算する演算となるため桁上がりが生じない。つまり, 桁上がりが生じなくなった時点でループを脱出すればわずかであるが, 計算量を小さくできる。その考えによって bitrv_Inc 関数を改良すると次のようになる。

```
static unsigned long bitrv_Inc(unsigned long dwVal, int Len)
{
    unsigend long dwCy;

    dwCy = Len >> 1;
    while (dwCy > (dwVal ^= dwCy))    dwCy >>= 1;
    return dwVal;
}
```

この改良した bitrv_Inc 関数の while ループは, 桁上がりがある限り半加算器を実行する条件になっている。 ビット逆順カウンタの桁上がりに関する条件は下の例がわかりやすい。 ビット逆順カウンタは上から下の桁へ桁上がりを加算させながら進行する。しかも, 桁上がりが生じる場合, 桁上がりを加算した桁が 0 となっているはずであるから, 第 m ビット目の桁上がりを加算をしているとき, 被加算数の第 m ビットより上の桁はすべて 0 になっている。この被加算数の第 m ビット目に桁上がりを加算した結果さらに桁上がりが生じるには, 加算結果の第 m ビット目が 0 になっているはずであるである。つまり, 加算結果が桁上がりより小さな数値になっていれば, さらに桁上がりが生じたことを意味す

る。逆に, 桁上がりが生じない場合, 加算結果の第 m ビット目が 1 であるので, 加算結果は桁上がりと等しい, もしくは, 桁上がりより大きな数値となっている。よって, 上に示した改良版 bitrv_Inc 関数のようなプログラムが書ける。

(a) 桁上がりがある場合

被加算数	$000\cdots001{*}{*}{*}\cdots$
桁上がり	$000\cdots001000\cdots$
加算結果	$000\cdots000{*}{*}{*}\cdots$

(b) 桁上がりがない場合

被加算数	$000\cdots000{*}{*}{*}\cdots$
桁上がり	$000\cdots001000\cdots$
加算結果	$000\cdots001{*}{*}{*}\cdots$

第4章　高速フーリエ変換の応用例

4.1　畳み込み

実数空間全体にわたって定義された関数 $f(x)$, $g(x)$ に関して, 次のように定義される $(f \circ g)(x)$ を関数 $f(x)$ と $g(x)$ の畳み込みという。

$$(f \circ g)(x) = \int_{-\infty}^{\infty} f(\xi)\, g(x-\xi) d\xi. \tag{4.1}$$

畳み込みはフーリエ変換と密接な関係があることがよく知られている。関数 $f(x)$ と $g(x)$ のフーリエ変換を, それぞれ, $F(k)$, $G(k)$ とすると,

$$(f \circ g)(x) = \frac{1}{2\pi} \int_{-\infty}^{\infty} F(k)\, G(k)\, e^{ikx} dk$$

が成り立つ。つまり, フーリエ変換の積 $F(k)\, G(k)$ を逆フーリエ変換すると原関数 $f(x)$ と $g(x)$ の畳み込みが得られることになる。このようなフーリエ変換と畳み込みの密接な関係は, 離散系列においても成り立つ。しかし, 計算機による信号処理で扱う系列は, 長さが有限であるため, 上のような関係とは少し異なる。ここでは, 長さが有限な複素系列の畳み込みとフーリエ変換の関係について考察する。

長さ N の離散系列 a_n と b_n のフーリエ変換を, それぞれ, A_k, B_k としよう。 その 2 つのフーリエ変換の積 $A_k B_k$ を逆フーリエ変換すると,

$$\begin{aligned}
\frac{1}{N} \sum_{k=0}^{N-1} A_k B_k W_N^{-kn} &= \frac{1}{N} \sum_{k=0}^{N-1} \left(\sum_{r=0}^{N-1} a_r W_N^{kr} \cdot \sum_{s=0}^{N-1} b_s W_N^{ks} \right) W_N^{-kn} \\
&= \frac{1}{N} \sum_{k=0}^{N-1} \sum_{r=0}^{N-1} \sum_{s=0}^{N-1} a_r b_s W_N^{k(r+s-n)}
\end{aligned}$$

を得る。ここで, 回転因子 W_N^{kn} の総和が

$$\sum_{k=0}^{N-1} W_N^{kn} = \begin{cases} N, & (n \text{ が } N \text{ の倍数の場合}) \\ 0, & (\text{それ以外}) \end{cases}$$

となることを考慮して上式の総和を評価すると, 離散系列におけるフーリエ変換と畳み込みの関係:

$$\frac{1}{N} \sum_{k=0}^{N-1} A_k B_k W_N^{-kn} = \sum_{r=0}^{n} a_r b_{n-r} + \sum_{r=n+1}^{N-1} a_r b_{n-r+N} \tag{4.2}$$

が得られる。 この結果を見ると, 妙な第 2 項が現れている。 有限な離散系列の畳み込みとフーリエ変換の関係が, 無限に続く連続関数と少し違うと言ったのはこの第 2 項があるからである。 ここで, (4.2) の解釈のため, $n < 0$, および, $n \geq N$ のとき, $a_n \equiv b_n \equiv 0$ とおいて a_n, b_n を無限数列として拡張してみる。また, 連続関数における畳み込みの定義 (4.1) からの類推によって, 離散系列 a_n と b_n の畳み込み $(a \circ b)_n$ を

$$(a \circ b)_n = \sum_{r=-\infty}^{\infty} a_r b_{n-r} \tag{4.3}$$

と定義する。 すると, (4.2) の右辺は

$$\text{RHS of (4.2)} = \sum_{r=-\infty}^{\infty} a_r b_{n-r} + \sum_{r=-\infty}^{\infty} a_r b_{n-r+N} = (a \circ b)_n + (a \circ b)_{n+N}$$

と書くことができる。添え字 $n = 0, 1, 2, \ldots, N-1$ に対して a_n と b_n が必ずしもゼロではない値であれば, (4.3) のように定義された畳み込み $(a \circ b)_n$ は $n = 0, 1, 2, \ldots, 2N-2$ に対して必ずしもゼロではない。その畳み込みの結果を第 0 項から第 $N-1$ 項に制限し, その範囲を逸脱した第 N 項から第 $2N-2$ 項を巡回させ, 第 0 項から順に重ね合わせると (4.2) の右辺に等しくなる。その意味で, (4.2) を**循環畳み込み**という。

4.2　相互相関関数

畳み込みと類似した定義であるが, 実数空間全体にわたって定義された関数 $f(x)$, $g(x)$ について

$$(f * g)(x) = \int_{-\infty}^{\infty} f^*(\xi)\, g(\xi + x) d\xi \tag{4.4}$$

によって定義される量 $(f * g)(x)$ は相互相関関数とよばれる。ここで, 右肩のアスタリスク $(*)$ は複素共役を意味する。 相互相関関数 $(f * g)(x)$ は 2 つの関数の類似度を示す尺度として用いあられる量で, 具体的には, $f(\xi)$ と $g(\xi)$ の変数 ξ を x だけずらしたときこれら 2 つの関数がどれだけ相似であるかを表す。そのため, 相互相関関数はパターンマッチングなどの信号処理手法としてしばしば使用される。畳み込みと同様, 相互相関関数はフーリエ変換と密接な関係があることがよく知られている。関数 $f(x)$ と $g(x)$ のフーリエ変換を, それぞれ, $F(k)$, $G(k)$ とすると,

$$(f * g)(x) = \frac{1}{2\pi} \int_{-\infty}^{\infty} F^*(k)\, G(k)\, e^{ikx} dk$$

が成り立つ。つまり, フーリエ変換の積 $F^*(k)\, G(k)$ を逆フーリエ変換すると原関数 $f(x)$ と $g(x)$ の相互相関関数が得られることになる。このようなフーリエ変換と相関係数の密接な関係は, 離散系列においても成り立つ。しかし, 計算機による信号処理で扱う系列は, 有限時間内でしか定義されない系列であるため, 上のような関係とは少し異なる。前節で

得られた知識から, 有限時間における系列の場合, 循環相関係数なるものに落ち着くと推測されるが, その様子を確認してみよう。

長さ N の離散系列 a_n と b_n のフーリエ変換を, それぞれ, A_k, B_k としよう。 その 2 つのフーリエ変換の積 $A_k^* B_k$ を逆フーリエ変換すると,

$$\begin{aligned}\frac{1}{N}\sum_{k=0}^{N-1} A_k^* B_k W_N^{-kn} &= \frac{1}{N}\sum_{k=0}^{N-1}\left(\sum_{r=0}^{N-1} a_r^* W_N^{-kr}\cdot\sum_{s=0}^{N-1} b_s W_N^{ks}\right) W_N^{-kn} \\ &= \frac{1}{N}\sum_{k=0}^{N-1}\sum_{r=0}^{N-1}\sum_{s=0}^{N-1} a_r^* b_s W_N^{k(-r+s-n)}\end{aligned}$$

を得る。ここでも, 回転因子 W_N^{kn} の総和に注意をして上の式を評価すると, 離散系列におけるフーリエ変換と相関係数の関係:

$$\frac{1}{N}\sum_{k=0}^{N-1} A_k^* B_k W_N^{-kn} = \sum_{r=0}^{N-r-1} a_r^* b_{r+n} + \sum_{r=N-r}^{N-1} a_r^* b_{r+n-N} \tag{4.5}$$

が得られる。 この結果を見ると, (予想通り) 無限の連続関数における相関係数は異なる第 2 項が付加されていることが確認できる。ここで, (4.5) の解釈のため, $n < 0$, および, $n \geq N$ のとき, $a_n \equiv b_n \equiv 0$ とおいて a_n, b_n を無限数列として拡張してみる。また, 連続関数における相互相関関数の定義 (4.4) からの類推によって, 離散系列 a_n と b_n の相互相関関す $(a*b)_n$ を

$$(a*b)_n = \sum_{r=-\infty}^{\infty} a_r^*\, b_{r+n} \tag{4.6}$$

と定義する。 すると, (4.5) の右辺は

$$\text{RHS of (4.5)} = \sum_{r=-\infty}^{\infty} a_r^*\, b_{r+n} + \sum_{r=-\infty}^{\infty} a_r^*\, b_{r+n-N} = (a*b)_n + (a*b)_{n-N}$$

と書くことができる。添え字 $n = 0, 1, 2, \ldots, N-1$ に対して a_n と b_n が必ずしもゼロではない値であれば, (4.6) のように定義された相互相関関数 $(a*b)_n$ は $n = -N+1, -N+2, \ldots, 0, 1, \ldots, N-1$ に対して必ずしもゼロではない。その相互相関関数を第 0 項から第 $N-1$ 項に制限し, その範囲を逸脱した第 $-N+1$ 項から第 -1 項を巡回させ, 第 0 項から順に重ね合わせると (4.5) の右辺に等しくなる。その意味で, (4.5) を**循環相互相関関数**という。

4.3 実数フーリエ変換

離散系列 a_n $(N = 0, 1, \ldots, N-1)$ が実数であるなら, そのフーリエ変換を実部と虚部に分けて $A_k \equiv \mathrm{Re}A_k + i\,\mathrm{Im}A_k$ と書くと,

$$\mathrm{Re}A_k = \mathrm{Re}A_{N-k}, \qquad \mathrm{Im}A_k = -\,\mathrm{Im}A_{N-k} \tag{4.7}$$

なる関係が成立する。 この関係式は, 実数列をフーリエ変換すると, その結果の実部が偶関数, 虚部が奇関数であることを意味している。

Proof　フーリエ変換の定義式にしたがって数式変形すると,

$$\begin{aligned}
A_k &= \sum_{n=0}^{N-1} a_n W_N^{nk} \\
&= \sum_{n=0}^{N-1} a_n \cos\frac{2\pi nk}{N} - i\sum_{n=0}^{N-1} a_n \sin\frac{2\pi nk}{N} \\
&= \sum_{n=0}^{N-1} a_n \cos\left(2\pi - \frac{2\pi nk}{N}\right) + i\sum_{n=0}^{N-1} a_n \sin\left(2\pi - \frac{2\pi nk}{N}\right) \\
&= \sum_{n=0}^{N-1} a_n \cos\frac{2\pi n(N-k)}{N} + i\sum_{n=0}^{N-1} a_n \sin\frac{2\pi n(N-k)}{N} \\
&= \mathrm{Re}\, A_{N-k} - i\,\mathrm{Im}\, A_{N-k}
\end{aligned}$$

となるので, $\mathrm{Re}A_k = \mathrm{Re}A_{N-k}$, $\mathrm{Im}A_k = -\mathrm{Im}A_{N-k}$ が成立することが示された。¶

このような性質があるため, 実数のフーリエ変換に限り, いくつか便利なアルゴリズムが知られている。

4.3.1　実数系列 2 つの変換

実数系列 a_n と b_n $(n = 0, 1, \ldots, N-1)$ が与えられたとき, $c_n \equiv a_n + ib_n$ とすることによって, 1 回のフーリエ変換で a_n と b_n のフーリエ変換 A_k と B_k $(k = 0, 1, \ldots, N-1)$ を得ることができる。具体的には, c_n のフーリエ変換, C_k と書くと,

$$A_k = \frac{C_k^{(\mathrm{r})} + C_{N-k}^{(\mathrm{r})}}{2} + i\frac{C_k^{(\mathrm{i})} - C_{N-k}^{(\mathrm{i})}}{2}, \tag{4.8}$$

$$B_k = \frac{C_k^{(\mathrm{i})} + C_{N-k}^{(\mathrm{i})}}{2} - i\frac{C_k^{(\mathrm{r})} - C_{N-k}^{(\mathrm{r})}}{2} \tag{4.9}$$

となる。ただし, 簡単のため $C_k^{(\mathrm{r})} \equiv \mathrm{Re}C_k$, $C_k^{(\mathrm{i})} \equiv \mathrm{Im}C_k$ という記号を用いた。

Proof　実数系列 a_n と b_n のフーリエ変換をそれぞれ, A_k, B_k とすると, その実数フーリエ変換の実部と虚部は (4.7) を満たしているはずである。ここでも, $A_k^{(\mathrm{r})} \equiv \mathrm{Re}A_k$, $A_k^{(\mathrm{i})} \equiv \mathrm{Im}A_k$, $B_k^{(\mathrm{r})} \equiv \mathrm{Re}B_k$, $B_k^{(\mathrm{i})} \equiv \mathrm{Im}B_k$ のように実部と虚部を分離して書くことにする。さて, $c_n \equiv a_n + ib_n$ のフーリエ変換を C_k とおき, この量を計算してみると,

$$C_k = (A_k^{(\mathrm{r})} - B_k^{(\mathrm{i})}) + i(A_k^{(\mathrm{i})} + B_k^{(\mathrm{r})})$$

を得る。フーリエ変換 C_k についても, $C_k^{(\mathrm{r})} \equiv \mathrm{Re}C_k$, $C_k^{(\mathrm{i})} \equiv \mathrm{Im}C_k$ とおき, 実部と虚部を具体的に書くと, $C_k^{(\mathrm{r})} = A_k^{(\mathrm{r})} - B_k^{(\mathrm{i})}$, $C_k^{(\mathrm{i})} = A_k^{(\mathrm{i})} + B_k^{(\mathrm{r})}$ となるわけである。 ここで, A_k と B_k が (4.7) を満たすことより,

$$A_k^{(\mathrm{r})} = \frac{C_k^{(\mathrm{r})} + C_{N-k}^{(\mathrm{r})}}{2}, \qquad A_k^{(\mathrm{i})} = \frac{C_k^{(\mathrm{i})} - C_{N-k}^{(\mathrm{i})}}{2},$$
$$B_k^{(\mathrm{r})} = \frac{C_k^{(\mathrm{i})} + C_{N-k}^{(\mathrm{i})}}{2}, \qquad B_k^{(\mathrm{i})} = -\frac{C_k^{(\mathrm{r})} - C_{N-k}^{(\mathrm{r})}}{2}$$

が得られるので, (4.8) と (4.9) が成立する。¶

実数系列に限られるが, 本来 2 回必要だったフーリエ変換がこのアルゴリズムを使うと 1 回ですむため演算量が 2 分の 1 に抑えられる。

4.3.2 αN/2 系列の変換

長さ N の実数系列 a_n の偶数番目 (先頭を 0 番目とする) の要素を実部に, 奇数番目の要素を虚部においた長さ $N/2$ の複素系列 $c_m \equiv a_{2m} + i\, a_{2m+1}$ $(m = 0, 1, \ldots, N/2-1)$ をフーリエ変換することによって a_n のフーリエ変換 A_k を計算することができる。複素系列 c_m のフーリエ変換 C_k $(k = 0, 1, \ldots, N/2-1)$ について, $\mathrm{Re}C_k \equiv C_k^{(\mathrm{r})}$, $\mathrm{Im}C_k \equiv C_k^{(\mathrm{i})}$ のように実部と虚部を別々に記述すると, 求めたいフーリエ変換 A_k は,

$$A_0 = C_0^{(\mathrm{r})} + C_0^{(\mathrm{i})}, \tag{4.10}$$
$$A_k = (C_k^{(\mathrm{r}+)} + iC_k^{(\mathrm{i}-)}) + (C_k^{(\mathrm{i}+)} - iC_k^{(\mathrm{r}-)})W_N^k, \tag{4.11}$$
$$A_{N/2} = C_0^{(\mathrm{r})} - C_0^{(\mathrm{i})} \tag{4.12}$$

のように書くことができる。ただし,

$$C_k^{(\mathrm{r}+)} = \frac{C_k^{(\mathrm{r})} + C_{N/2-k}^{(\mathrm{r})}}{2}, \qquad C_k^{(\mathrm{r}-)} = \frac{C_k^{(\mathrm{r})} - C_{N/2-k}^{(\mathrm{r})}}{2},$$
$$C_k^{(\mathrm{i}+)} = \frac{C_k^{(\mathrm{i})} + C_{N/2-k}^{(\mathrm{i})}}{2}, \qquad C_k^{(\mathrm{i}-)} = \frac{C_k^{(\mathrm{r})} - C_{N/2-k}^{(\mathrm{r})}}{2}$$

とする。系列の長さを 2 分の 1 にすることによってフーリエ変換の計算時間を 2 分の 1 より短くすることができる。求めたいフーリエ変換 A_k を得るには (4.10) から (4.12) のような後処理が必要であるが, それでも直接フーリエ変換を実行する場合に比べても十分に計算時間は短くなる。

Proof 実数系列 a_n $(n = 0, 1, \ldots, N-1)$ を $c_m = a_{2m} + i\, a_{2m+1}$ $(m = 0, 1, \ldots, N/2-1)$ のように変換して複素系列 c_m をつくる。このとき, c_m のフーリエ変換を C_k とし, これを

定義式に代入すると,

$$C_k = \sum_{m=0}^{N/2-1} c_m W_{N/2}^{mk} = \sum_{m=0}^{N/2-1} (a_{2m} + i\, a_{2m+1}) W_{N/2}^{mk}$$
$$= \sum_{m=0}^{N/2-1} a_{2m} W_{N/2}^{mk} + i \sum_{m=0}^{N/2-1} a_{2m+1} W_{N/2}^{mk} \tag{4.13}$$

となる。 ここで, (4.13) の右辺 第 1 項と第 2 項は, それぞれ, 実数列 a_{2m} と a_{2m+1} のフーリエ変換である。 これらのフーリエ変換に関して,

$$A_k^{(\mathrm{er})} + iA_k^{(\mathrm{ei})} = \sum_{m=0}^{N/2-1} a_{2m} W_{N/2}^{mk}, \qquad A_k^{(\mathrm{or})} + iA_k^{(\mathrm{oi})} = \sum_{m=0}^{N/2-1} a_{2m+1} W_{N/2}^{mk}$$

とおくと, (4.13) は

$$C_k = (A_k^{(\mathrm{er})} - A_k^{(\mathrm{oi})}) + i(A_k^{(\mathrm{ei})} + A_k^{(\mathrm{or})}) \tag{4.14}$$

と書くことができる。これらの量 $A_k^{(\mathrm{er})}$, $A_k^{(\mathrm{ei})}$, $A_k^{(\mathrm{or})}$, $A_k^{(\mathrm{oi})}$ は実数フーリエ変換の実部と虚部であるので, (4.7) を満足している。 つまり, C_k を実部と虚部に分離して $C_k^{(\mathrm{r})} \equiv \mathrm{Re}C_k$, $C_k^{(\mathrm{i})} \equiv \mathrm{Im}C_k$ という記号を用いると,

$$C_k^{(\mathrm{r})} + C_{N/2-k}^{(\mathrm{r})} = 2A_k^{(\mathrm{er})}, \qquad C_k^{(\mathrm{i})} - C_{N/2-k}^{(\mathrm{i})} = 2A_k^{(\mathrm{ei})}, \tag{4.15}$$
$$C_k^{(\mathrm{i})} + C_{N/2-k}^{(\mathrm{i})} = 2A_k^{(\mathrm{or})}, \qquad C_k^{(\mathrm{r})} - C_{N/2-k}^{(\mathrm{r})} = -2A_k^{(\mathrm{oi})} \tag{4.16}$$

なる関係が成り立つ。 これを踏まえて a_n $(n = 0, 1, \ldots, N-1)$ のフーリエ変換を計算してみると,

$$A_k = \sum_{n=0}^{N-1} a_n W_N^{nk} = \sum_{m=0}^{N/2-1} a_{2m} W_N^{2mk} + \sum_{m=0}^{N/2-1} a_{2m+1} W_N^{(2m+1)k}$$
$$= (A_k^{(\mathrm{er})} + iA_k^{(\mathrm{ei})}) + (A_k^{(\mathrm{or})} + iA_k^{(\mathrm{oi})}) W_N^k \tag{4.17}$$

となる。 ここで, $k = 1, 2, \ldots, N/2-1$ のとき, (4.17) に (4.15) と (4.16) を代入すると (4.11) を得る。さらに, $k = 0, N/2$ については, フーリエ変換の周期性 $C_{N/2} = C_0$ を用いると (4.10) と (4.12) を得る。¶

実際の計算を考えるなら, フーリエ変換 A_k を実部と虚部に分離して書いたほうが便利だろう。 変換式 (4.10) から (4.12) が $k = N/4$ を中心に対称性をもっていることから, それらの変換式をさらに展開すると,

$$A_0^{(\mathrm{r})} = C_0^{(\mathrm{r})} + C_0^{(\mathrm{i})}, \qquad A_0^{(\mathrm{i})} = 0, \tag{4.18}$$

$$\begin{aligned} A_k^{(\mathrm{r})} &= C_k^{(\mathrm{r})} + C_k^{(\mathrm{i}+)} \cos\frac{2\pi k}{N} - C_k^{(\mathrm{r}-)} \left(1 + \sin\frac{2\pi k}{N}\right), \\ A_k^{(\mathrm{i})} &= C_k^{(\mathrm{i})} - C_k^{(\mathrm{r}-)} \cos\frac{2\pi k}{N} - C_k^{(\mathrm{i}+)} \left(1 + \sin\frac{2\pi k}{N}\right), \end{aligned} \tag{4.19}$$

$$A^{(\mathrm{r})}_{N/4} = C^{(\mathrm{r})}_{N/4}, \qquad A^{(\mathrm{i})}_{N/4} = -C^{(\mathrm{i})}_{N/4}, \tag{4.20}$$

$$\begin{aligned} A^{(\mathrm{r})}_{N/2-k} &= C^{(\mathrm{r})}_k - C^{(\mathrm{i}+)}_k \cos\frac{2\pi k}{N} + C^{(\mathrm{r}-)}_k \left(1 + \sin\frac{2\pi k}{N}\right), \\ A^{(\mathrm{i})}_{N/2-k} &= C^{(\mathrm{i})}_k - C^{(\mathrm{r}-)}_k \cos\frac{2\pi k}{N} - C^{(\mathrm{i}+)}_k \left(1 + \sin\frac{2\pi k}{N}\right), \end{aligned} \tag{4.21}$$

$$A^{(\mathrm{r})}_{N/2} = C^{(\mathrm{r})}_0 - C^{(\mathrm{i})}_0, \qquad A^{(\mathrm{i})}_{N/2} = 0 \tag{4.22}$$

が得られる。

ここで導出した公式の使い方についてもう一度説明する。長さ N の実数系列 a_n が与えられたとき, まず, $c_m \equiv a_{2m} + a_{2m+1}$ の

$$\begin{aligned} \hat{R}_k &= \mathrm{Re}(\hat{A}_k), \\ \check{R}_k &= -\mathrm{Re}(\check{A}_k)\sin\frac{2\pi k}{N} - \mathrm{Im}(\hat{A}_k)\cos\frac{2\pi k}{N} \\ \hat{I}_k &= \mathrm{Re}(\check{A}_k)\cos\frac{2\pi k}{N} - \mathrm{Im}(\hat{A}_k)\sin\frac{2\pi k}{N} \\ \check{I}_k &= \mathrm{Im}(\check{A}_k) \end{aligned}$$

となるので, 変換式 (4.10) から (4.12) に対する逆変換

$$C^{(\mathrm{r})}_0 = \frac{A^{(\mathrm{r})}_0 + A^{(\mathrm{r})}_{N/2}}{2}, \qquad C^{(\mathrm{r})}_0 = \frac{A^{(\mathrm{r})}_0 - A^{(\mathrm{r})}_{N/2}}{2}, \tag{4.23}$$

$$\begin{aligned} C^{(\mathrm{r})}_k &= A^{(\mathrm{r})}_k - A^{(\mathrm{i}+)}_k \cos\frac{2\pi k}{N} - A^{(\mathrm{r}-)}_k \left(1 + \sin\frac{2\pi k}{N}\right), \\ C^{(\mathrm{i})}_k &= A^{(\mathrm{i})}_k + A^{(\mathrm{r}-)}_k \cos\frac{2\pi k}{N} - A^{(\mathrm{i}+)}_k \left(1 + \sin\frac{2\pi k}{N}\right), \end{aligned} \tag{4.24}$$

$$C^{(\mathrm{r})}_{N/4} = A^{(\mathrm{r})}_{N/4}, \qquad C^{(\mathrm{i})}_{N/4} = -A^{(\mathrm{i})}_{N/4}, \tag{4.25}$$

$$\begin{aligned} C^{(\mathrm{r})}_{N/2-k} &= A^{(\mathrm{r})}_{N/2-k} + A^{(\mathrm{r}-)}_k \left(1 + \sin\frac{2\pi k}{N}\right) + A^{(\mathrm{i}+)}_k \cos\frac{2\pi k}{N}, \\ C^{(\mathrm{i})}_{N/2-k} &= A^{(\mathrm{i})}_{N/2-k} - A^{(\mathrm{i}+)}_k \left(1 + \sin\frac{2\pi k}{N}\right) + A^{(\mathrm{r}-)}_k \cos\frac{2\pi k}{N}, \end{aligned} \tag{4.26}$$

を得ることができる。

4.4 Bluestein のアルゴリズム

Bluestein のアルゴリズムは 2 のべき乗ではない一般的な長さ N の系列を $O(N\log N)$ の演算量でフーリエ変換する手法である。系列の長さ N が 2 のべき乗であるとき, すでに見たように FFT によって $O(N\log N)$ の計算量でフーリエ変換が計算できる。一方, 2 のべき乗でない一般的な長さ N に関して, フーリエ変換には $O(N^2)$ の計算量が必要だと考えられていた。しかし, 1968 年に Bluestein が発表したアルゴリズムによって一般のフーリエ変換が $O(N\log N)$ の計算量で求められることがわかった。

長さ N の系列 a_n の離散フーリエ変換の定義式

$$A_k = \sum_{n=0}^{N-1} a_n e^{-\frac{2\pi i}{N}nk}$$

を, $(k-n)^2 = k^2 - 2nk + n^2$ なる関係に着目して変形すると,

$$A_k = e^{-\frac{\pi i}{N}k^2} \sum_{n=0}^{N-1} a_n e^{-\frac{\pi i}{N}n^2} \cdot e^{\frac{\pi i}{N}(k-n)^2}$$

を得る。ここで,

$$\tilde{a}_n \equiv a_n e^{-\frac{\pi i}{N}n^2}, \qquad b_n \equiv e^{\frac{\pi i}{N}k^2}$$

なる系列 a_n, b_n を定義すると離散フーリエ変換の定義式は

$$A_k = b_k^* \sum_{n=0}^{N-1} \tilde{a}_n b_{k-n} \tag{4.27}$$

となる。ここで, 右肩のアスタリスク $(*)$ は複素共役を表す。系列 $\tilde{a}$ と b の畳み込みについて $\tilde{a}\circ b$ なる記法を用いれば, 上の結果は $A_k = b_k^* \cdot (\tilde{a}\circ b)_k$ と書くことができる。

畳み込みとフーリエ変換に密接な関係があることを思い出すと (**??**) を $O(N\log N)$ の計算量で求めるアルゴリズムが得られる。具体的に言うと, $M \geq 2N-1$ を満足する 2 のべき乗 M を用いて, 系列 $\tilde{a}_n$, b_n を

$$\tilde{a}_m \equiv \begin{cases} a_m e^{-\frac{\pi i}{N}m^2} & (m = 0, 1, 2, \ldots, N-1) \\ 0 & (m = N, N+1, \ldots, M-1) \end{cases}$$

$$b_m = b_{M-m} \equiv e^{\frac{\pi i}{N}m^2} \qquad (m = 0, 1, 2, \ldots, M/2)$$

長さ M の系列として拡張する。

付 録A　演算量の定式化

A.1　基数2アルゴリズム

第n層バタフライ演算ブロックの数をν_n, 第n層の各バタフライ演算ブロックに含まれる複素乗算の数をξ_nとすると,

$$\nu_n = 2^n, \qquad \xi_n = \frac{N}{2^{n+1}} - 2$$

である。 ただし, 複素乗算の数ξ_nについては, 実部と虚部の交換のみで実現できる$-i$倍の演算は除いている。複素乗算が含まれる最も深いバタフライ演算ブロックの番号は$n_{\max} = \log_2 N - 3$であるので, 基数2のFFTに含まれる複素乗算の数は

$$n_{\text{base2}} = \sum_{n=0}^{n_{\max}} \nu_n \xi_n = \sum_{n=0}^{n_{\max}} \left(\frac{N}{2} - 2^{n+1} \right) = \frac{1}{2} N \log_2 N - \frac{3}{2} N + 2$$

となる。 現実的なプロセッサでは複素乗算は一命令で動作するわけではなく, 実数乗算と実数加算を組み合わせて動作する。 例えば, 複素数$x_0 + iy_0$と$x_1 + iy_1$の積を計算するには, $(x_0x_1 - y_0y_1) + i(x_0y_1 + x_1y_0)$を数式どおりに計算する。すなわち, 1回の複素乗算を実行するには4回の実数乗算が必要となる。 しかしながら, 回転因子の中には$W_4 \equiv \sqrt{1/2}\,(1-i)$のような特別な複素数がある。この回転因子による乗算は, $W_4(x+iy) = \sqrt{1/2}\,[(x+y) + i(y-x)]$であることから2回の実数乗算で実現できる。このような特別な複素数による乗算は第$n_{\max}$層までのバタフライブロックに必ず2つずつ含まれているので, その回数分を調整すると, 基数2のFFTに必要な実数乗算回数は

$$\nu_{\text{rad2}}^{(\text{mul})} = 4\nu_{\text{rad2}} - 4 \sum_{n=0}^{n_{\max}} \nu_n = 2N \log_2 N - 7N + 12$$

となる。 一方, 加算回数に関しては, 1回の複素加算には2回の実数加算が必要である。また, 上記の検討から1回の複素乗算が2回の実数加算を必要とすることから, 基数2のFFTに必要な実数加算回数は

$$\nu_{\text{rad2}}^{(\text{add})} = 2N \log_2 N + 2\nu_{\text{rad2}} = 3N \log_2 N - 3N + 4$$

となる。

A.2 基数 4 アルゴリズム

第 n 層バタフライ演算ブロックの数を ν_n, 第 n 層の各バタフライ演算ブロックに含まれる複素乗算の数を ξ_n とすると,

$$\nu_n = 4^n, \qquad \xi_n = 3\left(\frac{N}{4^{n+1}} - 1\right)$$

よって, 基数 4 の FFT に含まれる複素乗算の総数は

$$\nu_{\mathrm{mul}} = \sum_{n=0}^{n_{\max}} \nu_n \xi_n = 3 \sum_{n=0}^{n_{\max}} 4^n \left(\frac{N}{4^{n+1}} - 1\right)$$

となる。 ここで, $n_{\max}$ は 1 つ以上の複素乗算が存在する最も深いバタフライ演算ブロック層の番号, すなわち, $\log_2 N$ が偶数ならば $n_{\max} = (1/2)\log_2 N - 2$, 奇数ならば $n_{\max} = (1/2)\log_2 N - 3/2$ である。 これを上式に代入すると,

$$\nu_{\mathrm{rad4}} = \begin{cases} \dfrac{3}{8} N \log_2 N - N + 1, & \text{if } N \text{ is even,} \\ \dfrac{3}{8} N \log_2 N - \dfrac{7}{8} N + 1, & \text{if } N \text{ is odd} \end{cases}$$

を得る。

ここで先ほどのように実数乗算と実数加算の演算回数を調べてみよう。 複素乗算は 4 回の実数乗算が必要であるが, W_4 のような特別な回転因子では実数乗算が 2 回でよい。 このような特別な回転因子はバタフライ演算ブロックに必ず 3 つずつ含まれている。 よって, 基数 4 の FFT が必要とする実数乗算回数は,

$$\nu_{\mathrm{rad4}}^{(\mathrm{mul})} = 4\nu_{\mathrm{rad4}} - 6 \sum_{n=0}^{n_{\max}} \nu_n = \begin{cases} \dfrac{3}{2} N \log_2 N - \dfrac{11}{2} N + 10, & \text{if } N \text{ is even,} \\ \dfrac{3}{2} N \log_2 N - \dfrac{13}{2} N + 10, & \text{if } N \text{ is odd} \end{cases}$$

となる。 また, 加算回数に関しては, 1 回の複素加算につき 2 回, 1 回の複素乗算につき 2 回の実数加算が必要であるので, 基数 4 の FFT が必要とする実数加算回数は,

$$\nu_{\mathrm{rad4}}^{(\mathrm{add})} = 2N \log_2 N + 2\nu_{\mathrm{rad2}} = \begin{cases} \dfrac{11}{4} N \log_2 N - 2N + 4, & \text{if } N \text{ is even,} \\ \dfrac{11}{4} N \log_2 N - \dfrac{7}{4} N + 4, & \text{if } N \text{ is odd} \end{cases}$$

となる。 ここで, $\log_2 N$ が整数しか取らないことを利用して, 上の結果から場合分けを省略すると

$$\nu_{\mathrm{rad4}}^{(\mathrm{mul})} = \frac{3}{2} N \log_2 N - \frac{15 + (-1)^{\log_2 N}}{4} N + 10$$

$$\nu_{\mathrm{rad4}}^{(\mathrm{add})} = \frac{11}{4} N \log_2 N - \frac{15 + (-1)^{\log_2 N}}{8} N + 3 - (-1)^{\log_2 N}$$

と書くことができる。

A.3 Split Radix

第 n 層バタフライ演算ブロックの数を ν_n とする。このブロック数は $\nu_{n+2} = \nu_{n+} + 2\nu_n$ なる漸化式を満たす。さらに, 第 0 層と第 1 層のバタフライ演算ブロックはともに 1 つしか存在しないので, $\nu_0 = \nu_1 = 1$ という初期条件を用いて, この漸化式から一般項を求めると

$$\nu_n = \frac{1}{3}\left[2^{n+1} - (-1)^{n+1}\right]$$

が得られる。 第 n 層のバタフライ演算ブロックに含まれる複素乗算の数を ξ_n とすると,

$$\xi_n = N\left(\frac{1}{2}\right)^{n+1} - 2$$

が成り立つ。 また, 複素乗算が 1 つ以上含まれる最も深いバタフライ演算ブロックの番号は $n_{\max} = \log_2 N - 3$ であるので, Split Radix FFT に含まれる複素乗算回数は

$$\begin{aligned}
\nu_{\text{sprad}} &= \sum_{n=0}^{\log_2 N-2} \nu_n \xi_n \\
&= \frac{1}{3}\sum_{n=0}^{\log_2 N-2}\left[N - 2^{n+2} - N\left(-\frac{1}{2}\right)^{n+1} + 2(-1)^{n+1}\right] \\
&= \frac{1}{3}N\log_2 N - \frac{8}{9}N + 1 - \frac{1}{9}(-1)^{\log_2 N}
\end{aligned}$$

となる。

ここでも実数乗算と実数加算の演算回数を調べてみよう。 複素乗算は 4 回の実数乗算が必要であるが, $W_N^{N/4}$ のような特別な回転因子では実数乗算が 2 回でよい。 このような特別な回転因子はバタフライ演算ブロックに必ず 2 つずつ含まれている。 よって, Split RTadix FFT が必要とする実数乗算回数は,

$$\nu_{\text{sprad}}^{(\text{mul})} = 4\nu_{\text{rad2}} - 4\sum_{n=0}^{n_{\max}} \nu_n = \frac{4}{3}N\log_2 N - \frac{38}{9}N + 6 + \frac{2}{9}(-1)^{\log_2 N}$$

となる。 一方, 加算回数に関しては, 1 回の複素加算には 2 回の実数加算が必要である。また, 上記の検討から 1 回の複素乗算が 2 回の実数加算を必要とすることから, Split Radix FFT に必要な実数加算回数は

$$\nu_{\text{sprad}}^{(\text{add})} = 2N\log_2 N + 2\nu_{\text{rad2}} = \frac{8}{3}N\log_2 N - \frac{16}{9}N + 2 - \frac{2}{9}(-1)^{\log_2 N}$$

となる。