FFT アルゴリズム

ブライアン・ガウ, bjg@network-theory.co.uk
(とみなががだいすけ訳 tominaga@cbrc.jp)

May 1997 (訳 May 12, 2010)

1 はじめに

高速フーリエ変換 (Fast Fourier Transforms, FFT) とは、下式で示される離散フーリエ変換 (discrete fourier transform, DFT) を効率良く計算するアルゴリズムである。

$$\begin{aligned} h_a &= \mathrm{DFT}(g_b) &(1)\\ &= \sum_{b=0}^{N-1} g_b \exp(-2\pi i ab/N) \qquad 0 \le a \le N-1 &(2)\\ &= \sum_{b=0}^{N-1} g_b W_N^{ab} \qquad W_N = \exp(-2\pi i/N) &(3)\end{aligned}$$

DFT は、連続関数の値を時間や空間の座標軸上で離散的にサンプリングして計算したものから、その関数を近似する時に使われるのが普通である。DFT は定義から素直に計算すると $\mathbf{W}\vec{g}$ という行列とベクトルの積であり、データの点数が N のとき、その計算量は $O(N^2)$ である。これに対して、行列をその次数 (N) の因数分解 ($N = f_1 f_2 \ldots f_n$) にしたがって小行列に分割し、分割統治法を適用して計算量を $O(N \sum f_i)$ に減らす、というのが FFT の基本的な考え方である。

この章では GSL で実装されている FFT のアルゴリズムと、それを改良するためのヒントについて説明する。FFT 自体についてはデュアメル著 *Fast Fourier Transforms: A Tutorial Review and A State of the Art* と Vetterli[1] を参照するとよい。サンプル・プログラムのついた入門書としては、ブリガム著 *The Fast Fourier Transform*[2] やブラスおよびパークス著 *DFT/FFT and Convolution Algorithms*[3] が勧められる。また IEEE から 1979 年に *Programs for Digital Signal Processing*[4] という FFT の FORTRAN プログラムの大要が出ている。FFT アルゴリズムにもいくつか種類があるが、それぞれについて実装の際の参考となる。デジタル信号処理 (Digital Signal Processing, DSP) プログラムを作成する際に必要となる FFT アルゴリズムやハードウェアについては *Handbook of Real-Time Fast Fourier Transforms*[5] に述べられている。FFT アルゴリズムには数論に基礎を置くものが多い。そういった面からはエリオットおよびラオ著 *Fast transforms: algorithms, analyses, applications*[6]、ブラウト著 *Fast Algorithms for Digital Signal Processing*[7]、マクレランおよびレイダー著 *Number Theory in Digital Signal Processing*[8] が参考になる。ブラスの作った参考書のリストもある [9]。

2 各種の FFT アルゴリズム

FFT アルゴリズムは二種類に大別される。一つはクーリーとテューキーによる方法 (Cooley-Tukey algorithm)、もう一つは因数分解法 (prime factor algorithm) である。これらの相違点は、データ全

体の FFT 計算を複数の小規模な変換に分解する方法である。クーリーとテューキーの方法にはさらに、混合基数法および基数 2 の方法の二つがある。これらはさらに、置換法と追加法、整列法とかき混ぜ法、時間空間法と周波数空間法に分けられる。

混合基数法 (mixed-radix algorithm) は、ベクトルとして与えられる元データを、短いベクトルに分割できる場合に適用できる。この場合、元データに比べてデータ長の短い FFT を行えばよい。FFT の多くの実装は、データ長が 2、3、5 ... などの小さな素数の場合の FFT が高速に行えるように最適化されたルーチンを持っている。それらの短データ長 FFT ルーチンの出力を組み合わせることで、元データの FFT を得ることができる。組み合わせ方により、さまざまな長さのデータに FFT を適用できる。そういった効率の良い短データ長 FFT ルーチンと、計算量 $O(N^2)$ の汎用の DFT ルーチンの両方を実装することで、原理的には任意のデータ長に対して DFT を行うことができる。データ長を因数分解しても大きな素数にしかならないようなデータでは、$O(N^2)$ の計算量が必要となる。

基数 2 の方法 (radix-2 algorithm, 2 の累乗法 power of two algorithm) は、混合基数法を簡略化したものであり、データ長が 2 の累乗に制限される方法である。基数 2 の方法は基本的には加減算だけで行え、アルゴリズムは非常に簡潔である。FFT の効率化のために多くの研究がこのアルゴリズムに対して行われてきたが、その中でも効率のよいものは「基数分割 (split-radix)」と呼ばれる方法である。これは基数 2 の方法と基数 4 の方法 (power of 4 algorithm) から、よい部分だけを取り出して組み合わせたものである [10, 11]。

因数分解法 (prime factor algorithm, PFA) は短データ長の FFT ルーチンを、上とは違う方法で組み合わせて任意のデータ長に適用する方法である [12, 13]。その組み合わせ方は非常に単純でありながらも面白い方法だが、データ長を互いに素な数となる長さに分割せねばならない。そのため、あらかじめ決められたデータ長に特化したルーチンを複数用意しておくのがよい。

2.1 素数長データの FFT

データ長が大きな素数となる場合、上記のいずれのアルゴリズムでも効率のよい計算はできない。しかし数論を応用した方法が適用できる。データ長 p の FFT (p は素数) をデータ長 $(p-1)$ の畳み込み積分に変換できることをレイダーが示している。データ長 N の畳み込み積分と、同じデータ長の FFT は同じ計算であり、したがって $p-1$ が因数分解可能な時は、FFT を使って畳み込み積分が計算できる。このアルゴリズムの原著論文はレイダーによる [14] (リプリントもある [8]) が、上述の参考書にも詳述されている。

2.2 最適化

あらゆる場合に最高の性能を発揮する特定の FFT アルゴリズムと言うものは存在しない。計算科学的に見ると、FFT アルゴリズムは浮動小数点演算、整数演算、メモリアクセスの連続であり、こういった各種の演算の実行速度は計算機プラットフォームによって異なる。したがって FFT アルゴリズムの性能は、それが実装されているハードウェアによって異なるため、「最適化」はこの場合、与えられたハードウェアの性質に応じたもっともよいアルゴリズムを選ぶことである。

たとえばウィノグラッド・フーリエ変換 (Winograd Fourier Transform, WFTA) は浮動小数点演算の回数をできるだけ減らすように工夫されたアルゴリズムである。その代償として加算の回数は他の FFT アルゴリズムよりも多い。したがって WFTA は、浮動小数点演算にくらべるとデータ転送の時間は無視できるような計算機で高速である。しかし最近の計算機ではデータ転送の時間は浮動小数点演算と同等か少し遅い程度のものが多く、演算のバランスがよい (つまり WFTA よりは浮動小数点演算が多くデータ転送が少ない) アルゴリズムの方が高速になるだろう。

効率のよさを詳細に見る時は、種々のプラットフォーム上の種々のアルゴリズムを比較した、メハリック、ルスタン、ルートによる論文 *Effects of Architecture Implementation on DFT Algorithm*

Performance を参照するとよい [15]。この論文は 1980 年代前半に書かれたもので、対象となっている計算機は現在では博物館でもなければまず目にすることのないスパコン、ミニコン類であるが、この論文中の方法論は現在でも通用するものであり、現在の計算機を対象にしても同様の結果が得られるものと考えられる。

3　FFT の考え方

混合基数 FFT における分割統治法では、因数分解がその基本にある。データ長 N が n_f 個の整数 f_i に以下のように因数分解できるとする。

$$N = f_1 f_2 ... f_{n_f} \tag{4}$$

これにより FFT は、データ長が各因数となる短いデータに対する FFT に分割できる。厳密には、データ長 N の元データに対する FFT は、

(N/f_1) (データ長 f_1 の FFT)
(N/f_2) (データ長 f_1 の FFT)
...
(N/f_{n_f}) (データ長 f_1 の FFT)

と分割される。分割された各 FFT の計算量を合計すると $O(N(f_1 + f_2 + ... + f_{n_f}))$ になる。N の因数がすべて小さな整数になる場合、この合計計算量は $O(N^2)$ よりも小さくなる。たとえば N が２の累乗の場合は、データ長が $N = 2^m$ の FFT は $mN/2$ の基数２の FFT に分割され、合計計算量は $O(N \log_2 N)$ になる。これは以下のようにして示される。

素直な定義通りの DFT は、以下である。

$$h_a = \sum_{b=0}^{N-1} g_b W_N^{ab} \qquad W_N = \exp(-2\pi i/N) \tag{5}$$

級数の各項を、基数番目と偶数番目に分けて書くと以下のようになる。

$$= \sum_{b=0}^{N/2-1} g_{2b} W_N^{a(2b)} + \sum_{b=0}^{N/2-1} g_{2b+1} W_N^{a(2b+1)} \tag{6}$$

これにより元のデータ長 N の DFT は、データ長 $N/2$ の二つの DFT に分けられる。

$$h_a = \sum_{b=0}^{N/2-1} g_{2b} W_{(N/2)}^{ab} + W_N^a \sum_{b=0}^{N/2-1} g_{2b+1} W_{(N/2)}^{ab} \tag{7}$$

この最初の項は、データ g の偶数項の DFT である。後の項は g の基数項の DFT であり、これには指数係数 W_N^k が乗じてある (この係数はひねり係数あるいは回転因子、twiddle factor とも呼ばれる)。

$$\mathrm{DFT}(h) = \mathrm{DFT}(g_{even}) + W_N^k \mathrm{DFT}(g_{odd}) \tag{8}$$

DFT を偶数、奇数項に分けることによって、乗算演算の回数を N^2 (長さ N の元データの場合) から $2(N/2)^2$ (長さ $N/2$ の各 DFT) に減らすことができた。分割して得られる結果をまとめるのには回転因子 W_N^k を乗じて、二つの結果をつなげることが必要であり、その計算量は $O(N)$ である。

この分割は再帰的に $\log_2 N$ 回、単項の DFT に分割されるまで繰り返すことができる。単項 DFT は与えられた値を返すだけであり、計算量を要しないが、分割の各ステージで、偶数項と奇

数項に分割して得られる結果を統合するのに $O(N)$ の計算量が必要であり、そのため、最終的には元データの DFT 結果を得るのに $O(N \log_2 N)$ の計算量を要する。データ長を f_1、f_2、$\ldots$ の積に因数分解して、データをその長さに分解しても、同じように計算結果を統合できる。その場合、まずデータを偶数項と奇数項に分ける (因数が 2 の場合に相当する) 代わりに因数 f_1 に分割する。続いて次の因数 f_2 に分割する。この場合の合計計算量は $O(N \sum f_i)$ である。

上記の方法を使えば DFT の計算量を $O(N^2)$ から $O(N \sum f_i)$ に下げることができるが、実際にどのように実装するかについてのヒントにはならない。それについては次章で述べる。

4 基数 2 のアルゴリズム

基数 2 の FFT アルゴリズムの場合、データ長が 2 の累乗に制限されているので、配列要素の添え字を二進数、つまり 1 と 0 で示すのが分かりやすい。これについてはヴィッシャーによる *The FFT: Fourier Transforming One Bit at a Time*[16] がよい参考になる。添え字の二進表示は、基数 2 のアルゴリズムの簡潔さを示すポイントでもある。

添え字 b は $(0 \leq b < 2^{n-1})$、以下のようにして二進数で表記される。

$$b = [b_{n-1} \ldots b_1 b_0] = 2^{n-1} b_{n-1} + \ldots + 2b_1 + b_0 \tag{9}$$

つまり、$b_0, b_1, \ldots, b_{n-1}$ はそれぞれが b の二進表記におけるビット (0 か 1 のどちらかの値を持つ) である。

DFT の元々も定義は、b の各ビットについての和を考えることで以下のように書ける。

$$h(a) = \sum_{b=0}^{N-1} g_b \exp(-2\pi i a b / N) \tag{10}$$

これは以下と同等である。

$$h([a_{n-1} \ldots a_1 a_0]) = \sum_{b_0=0}^{1} \sum_{b_1=0}^{1} \cdots \sum_{b_{n-1}=0}^{1} g([b_{n-1} \ldots b_1 b_0]) W_N^{ab} \tag{11}$$

ここで a は二進表記で $a = [a_{n-1} \ldots a_1 a_0]$ と書かれている。

この和を計算する際の演算回数を減らすために、指数係数が以下のような周期性を持っていることを利用する。

$$W_N^{x+N} = W_N^x. \tag{12}$$

DFT の計算中に出てくる積 ab の値はほとんどの場合 N よりも大きい。したがって指数関数の周期性を考慮すると、この積の値はすべて $0 \ldots N-1$ の範囲内に置き換えられる。したがって N で割った値が同じになる項は、改めて値を計算する必要がない。これを利用して、時間空間法および周波数空間法の二種類のアルゴリズムが導かれる。この二つの違いは、項数をどの空間で減らすかによる。まず最初に時間空間法について述べる。

4.1 基数 2 の時間空間法 (Decimation-in-Time, DIT)

時間空間法を導出するためにはまず、添え字 b の最上位ビット (most significant bit, MSB) を分けなければならない。

$$[b_{n-1} \ldots b_1 b_0] = 2^{n-1} b_{n-1} + [b_{n-2} \ldots b_1 b_0] \tag{13}$$

すると、級数の「もっとも内側」については、b の他のビットへの依存性を消去して計算することができる。

$$h([a_{n-1}\dots a_1a_0]) = \sum_{b_0=0}^{1}\sum_{b_1=0}^{1}\dots\sum_{b_{n-1}=0}^{1} g(b)W_N^{a(2^{n-1}b_{n-1}+[b_{n-2}\dots b_1b_0])} \quad (14)$$

$$= \sum_{b_0=0}^{1}\sum_{b_1=0}^{1}\dots\sum_{b_{n-2}=0}^{1} W_N^{a[b_{n-2}\dots b_1b_0]}\sum_{b_{n-1}=0}^{1} g(b)W_N^{a(2^{n-1}b_{n-1})} \quad (15)$$

$W_N^{a(2^{n-1}b_{n-1})}$ の項からは、同様に a についても、指数関数の周期性から依存性をなくすことができる。

$$W_N^{a(2^{n-1}b_{n-1})} = \exp(-2\pi i[a_{n-1}\dots a_1a_0]2^{n-1}b_{n-1}/2^n) \quad (16)$$

$$= \exp(-2\pi i[a_{n-1}\dots a_1a_0]b_{n-1}/2) \quad (17)$$

$$= \exp(-2\pi i(2^{n-2}a_{n-1}+\dots+a_1+(a_0/2))b_{n-1}) \quad (18)$$

$$= \exp(-2\pi i a_0b_{n-1}/2) \quad (19)$$

$$= W_2^{a_0b_{n-1}} \quad (20)$$

以上のように、この「もっとも内側の項」は b の最上位ビットと a の最下位ビットだけを含む形に簡略化できる。これにより級数は以下の形に書ける。

$$h([a_{n-1}\dots a_1a_0]) = \sum_{b_0=0}^{1}\sum_{b_1=0}^{1}\dots\sum_{b_{n-2}=0}^{1} W_N^{a[b_{n-2}\dots b_1b_0]}\sum_{b_{n-1}=0}^{1} g(b)W_2^{a_0b_{n-1}}. \quad (21)$$

b の最上位ビットの次のビット b_{n-2} についても同様に繰り返すことができる。すると以下のようになる。

$$W_N^{a(2^{n-2}b_{n-2})} = \exp(-2\pi i[a_{n-1}\dots a_1a_0]2^{n-2}b_{n-2}/2^n) \quad (22)$$

$$= W_4^{[a_1a_0]b_{n-2}}. \quad (23)$$

a のビットに対する依存性も、同様に減らすことができる。

$$h([a_{n-1}\dots a_1a_0]) = \sum_{b_0=0}^{1}\sum_{b_1=0}^{1}\dots\sum_{b_{n-3}=0}^{1} W_N^{a[b_{n-3}\dots b_1b_0]}\sum_{b_{n-2}=0}^{1} W_4^{[a_1a_0]b_{n-2}}\sum_{b_{n-1}=0}^{1} g(b)W_2^{a_0b_{n-1}}. \quad (24)$$

これをすべてのビットについて繰り返して級数を簡略化していくことで、基数２の時間空間法の式を得ることができる。

$$h([a_{n-1}\dots a_1a_0]) = \sum_{b_0=0}^{1} W_N^{[a_{n-1}\dots a_1a_0]b_0}\dots\sum_{b_{n-2}=0}^{1} W_4^{[a_1a_0]b_{n-2}}\sum_{b_{n-1}=0}^{1} W_2^{a_0b_{n-1}}g(b) \quad (25)$$

この式をアルゴリズムとして定式化するためには、これを再帰的に展開して、各段階で部分和 g_1、g_2、...、g_n を計算する必要がある。

$$g_1(a_0, b_{n-2}, b_{n-3}, \dots, b_1, b_0) = \sum_{b_{n-1}=0}^{1} W_2^{a_0b_{n-1}} g([b_{n-1}b_{n-2}b_{n-3}\dots b_1b_0]) \quad (26)$$

$$g_2(a_0, \quad a_1, b_{n-3}, \dots, b_1, b_0) = \sum_{b_{n-2}=0}^{1} W_4^{[a_1a_0]b_{n-2}} g_1(a_0, b_{n-2}, b_{n-3}, \dots, b_1, b_0) \quad (27)$$

$$g_3(a_0, \quad a_1, \quad a_2, \ldots, b_1, b_0) = \sum_{b_{n-3}=0}^{1} W_8^{[a_2a_1a_0]b_{n-2}} g_2(a_0, a_1, b_{n-3}, \ldots, b_1, b_0) \tag{28}$$

$$\ldots = \ldots \tag{29}$$

$$g_n(a_0, a_1, a_2, \ldots, a_{n-2}, a_{n-1}) = \sum_{b_0=0}^{1} W_N^{[a_{n-1}\ldots a_1a_0]b_0} g_{n-1}(a_0, a_1, a_2, \ldots, a_{n-2}, b_0) \tag{30}$$

最後の和 g_n が、求める変換結果 h である。

$$h([a_{n-1}\ldots a_1a_0]) = g_n(a_0, a_1, \ldots, a_{n-1}) \tag{31}$$

各ビットを添え字ではなく、プログラム中での関数呼び出し時の引数として使ったことで、部分和を保存しておく計算機メモリ上の領域についてはここまででは触れなかった。この部分和をどのように保存するかは、時間空間法と周波数空間法とで異なる。

メモリを効率良く使う方法について述べる前に、再帰呼び出しの各ステージにおける計算結果 g_1、g_2、...について、「変数値を置き換えて行く」方法について述べる。たとえば g_1 については、いわゆる入力は以下である。

$$g([\underline{b_{n-1}}b_{n-2}b_{n-3}\ldots b_1b_0]) \tag{32}$$

ここで b のとり得る値は $b_{n-1} = (0, 1)$ である。これに対する「出力」は以下である。

$$g_1(\underline{a_0}, b_{n-2}, b_{n-3}, \ldots, b_1, b_0) \tag{33}$$

ここで a の値は $a_0 = (0, 1)$ である。$a_0 = 0, 1$ がどちらの値でも、b_{n-1} についての級数を計算する時には $b_{n-2}, b_{n-3}, \ldots, b_1, b_0$ の値が固定されていて、$a_0 = 0, 1$ の両方の値について、b_{n-1} についての級数を計算する場合、$a_0 = 0$ の時の結果を元々 $b_0 = 0$ だった所に、$a_0 = 1$ の時の結果を元々 $b_0 = 1$ だった所に入れてしまってもよいことは明らかである。この 2 つの入力と 2 つの出力のことを「二重ノード対 (dual node pairs)」と呼ぶ。再帰計算の各ステージで、各二重ノード対は互いに独立に計算できる。それが置換法 (in-place calculation) が可能な理由である。

ここまでの説明では、入力となる二つの項 $g([b_{n-1}, \ldots])$ を出力 $g_1(a_0, \ldots)$ で置き換えるやり方が必要であった。ここで、a の順序が b とは逆であることに注意せねばならない。つまり b の最上位ビット (MSE) が a の最下位ビットで置き換えられるようになっている。a の順序は指数係数 W^{ab} と同じ順序に自然になっているので、b と逆になっているのは扱いにくいとも言える。a と、そのビットを逆に並べたもの $a^{bit-reversed}$ を常に同時に考えることができるならそれでもいいが、ちょっとした工夫でそこまでしなくてもよくなる。そもそもの計算を始める前に、g のビットを逆順にしておけば、a のビットも自動的に逆順になっていく。a を保存しておく領域はもともとが逆順になっているので、大本の入力である g を逆順にすれば a については自然な順序になり、指数係数と同じ順序になる。

この「ちょっとした工夫」は言葉で説明すると複雑なので、以下に $N = 16$ で 4 段階の再帰ステージがある時間空間法 FFT を例示する。最初に与えるデータはビットが逆順になっている。

$$g_1([b_0b_1b_2a_0]) = \sum_{b_3=0}^{1} W_2^{a_0b_3} g([b_0b_1b_2b_3]) \tag{34}$$

$$g_2([b_0b_1a_1a_0]) = \sum_{b_2=0}^{1} W_4^{[a_1a_0]b_2} g_1([b_0b_1b_2a_0]) \tag{35}$$

$$g_3([b_0a_2a_1a_0]) = \sum_{b_1=0}^{1} W_8^{[a_2a_1a_0]b_1} g_2([b_0b_1a_1a_0]) \tag{36}$$

$$h(a) = g_4([a_3a_2a_1a_0]) = \sum_{b_0=0}^{1} W_{16}^{[a_3a_2a_1a_0]b_0} g_3([b_0a_2a_1a_0]) \tag{37}$$

再帰の最初のステージで、ビットが逆順になった b を g の添え字とすることで、逆順でも正しく演算が行われるようにしており、この式では元の式と同じ計算を同じデータで行っていることがわかる。指数係数の中には b のビットは一つしか現れないので、b のビットを

3段階目の再帰ステージ (次式) を詳しく見てみると、

$$g_3([b_0a_2a_1a_0]) = \sum_{b_1=0}^{1} W_8^{[a_2a_1a_0]b_1} g_2([b_0b_1a_1a_0]) \tag{38}$$

和記号の中では、一つのビット b_1 だけが変化し、他は固定である。固定されているビット (b_0、a_1、a_0) は「無関係 (spectator)」である。これらのビットについても全ての組み合わせを計算し、基本的に同じ演算を適切に行って、W_8 の値を得るための指数係数を計算せねばならない。計算で得た値で、それまでの値を置き換えて行く (g_2 を g_3 で置き換えることで $b_1 = 0, 1$ についてと $a_2 = 0, 1$ についてを同時に計算する必要が出てくる)

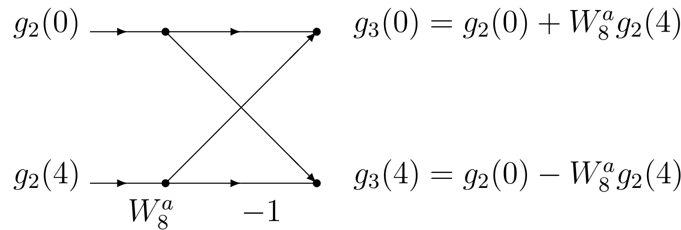
$$\begin{pmatrix} g_3([b_00a_1a_0]) \\ g_3([b_01a_1a_0]) \end{pmatrix} = \begin{pmatrix} g_2([b_00a_1a_0]) + W_8^{[0a_1a_0]} g_2([b_21a_1a_0]) \\ g_2([b_00a_1a_0]) + W_8^{[1a_1a_0]} g_2([b_21a_1a_0]) \end{pmatrix} \tag{39}$$

指数係数 (の指数部分) を簡潔にすることで、より対称的な形に書くことができる。

$$W_8^{[a_2a_1a_0]} = W_8^{4a_2+[a_1a_0]} = (-1)^{a_2} W_8^{[a_1a_0]} \tag{40}$$

$$\begin{pmatrix} g_3([b_00a_1a_0]) \\ g_3([b_01a_1a_0]) \end{pmatrix} = \begin{pmatrix} g_2([b_00a_1a_0]) + W_8^{[a_1a_0]} g_2([b_21a_1a_0]) \\ g_2([b_00a_1a_0]) - W_8^{[a_1a_0]} g_2([b_21a_1a_0]) \end{pmatrix} \tag{41}$$

指数係数 $W_8^{[a_1a_0]}$ は「回転因子 (ひねり係数、twiddle factor)」と呼ばれる。また、この計算式は回転因子により対称的な加減算の形をしており、これは「バタフライ (butterfly) 演算」と呼ばれる。バタフライ演算はおおまかに図示されるだけのことも多いが、$b_0 = a_0 = a_1 = 0$ の場合を示すと以下のようになる。



上図で、入力が左側、出力が右側に示されている。入力された値に、線に沿って示されている係数を乗じて、和をとることで各出力が得られる。

一般的には、二重ノード対の各ビットを Δ で表し a と b の他のビットを $\hat{a}$ と $\hat{b}$ で表す。するとバタフライ演算は以下のように表される。

$$\begin{pmatrix} g(\hat{b}+\hat{a}) \\ g(\hat{b}+\Delta+\hat{a}) \end{pmatrix} \leftarrow \begin{pmatrix} g(\hat{b}+\hat{a}) + W_{2\Delta}^{\hat{a}} g(\hat{b}+\Delta+\hat{a}) \\ g(\hat{b}+\hat{a}) - W_{2\Delta}^{\hat{a}} g(\hat{b}+\Delta+\hat{a}) \end{pmatrix} \tag{42}$$

ここで $\hat{a}$ は $0 \ldots \Delta - 1$、$\hat{b}$ は $0 \times 2\Delta$、$1 \times 2\Delta$、$\ldots$、$(N/\Delta - 1)2\Delta$ である。Δ の値は最初のパス (pass、再帰のステージ) では 1、次のパスでは 2、n 番目のパスで 2^{n-1} である。各パスでは二つの入力を二つの出力で置き換える置き換え計算が $N/2$ 回必要である。

上の例では $\Delta = [100] = 4$、$\hat{a} = [a_1a_0]$、$\hat{b} = [b_0000]$ である。

したがって、$g(0) \ldots g(2^n - 1)$ が長さ 2^n のデータとするとき、これに対する正規の基数２の時間空間法 FFT アルゴリズムは、以下のようになる。

```
bit-reverse ordering of g
Δ ⇐ 1
for pass = 1 ... n do
   W ⇐ exp(−2πi/2Δ)
   for (a = 0; a < Δ; a++) do
      for (b = 0; b < N; b+ =2 ∗ Δ) do
         t0 ⇐ g(b + a) + W^a g(b + Δ + a)
         t1 ⇐ g(b + a) − W^a g(b + Δ + a)
         g(b + a) ⇐ t0
         g(b + Δ + a) ⇐ t1
      end for
   end for
   Δ ⇐ 2Δ
end for
```

4.2 実装の詳細

上に示したアルゴリズムの実装として、単純な基数 2 の時間空間法ルーチンを以下に示す。$a = 0$ の場合を a についてのループの外に出して $W^a = 1$ という意味のない演算を省けば、計算時間を短縮できる。

```
for (b = 0; b < N; b+ =2 ∗ Δ) do
   t0 ⇐ g(b) + g(b + Δ)
   t1 ⇐ g(b) − g(b + Δ)
   g(b) ⇐ t0
   g(b + Δ) ⇐ t1
end for
```

ビットを逆順にするには、速い方法がいくつかある。ここではゴールド・レイダー法 (Gold-Rader algorithm) を示す。この方法は簡潔で、また記憶領域を別途用意する必要がない。

```
for i = 0 ... n − 2 do
   k = n/2
   if i < j then
      swap g(i) and g(j)
   end if
   while k ≤ j do
      j ⇐ j − k
      k ⇐ k/2
   end while
   j ⇐ j + k
end for
```

ゴールド・レイダー法は、ごく単純なビット逆順 (左シフトおよび右シフトでビットを並べ替える方法) に比べるとおおよそ半分の時間で計算できる。GSL には ロドリゲス法 (Rodriguez bit reversal algorithm) も実装されている。これも別途記憶領域を用意する必要がない [17]。ビットを逆順にするには高速な方法が他にもあるが、いずれも別途記憶領域を用意する必要がある [18]。

a のループの内部では、W^a を計算する時に、三角関数が下式のように再帰的に計算できることを利用できる。

$$W^{a+1} = WW^a \tag{43}$$

$$= (\cos(2\pi/2\Delta) + i\sin(2\pi/2\Delta))W^a \quad (44)$$

この計算を使えば、各パスの最初で $\exp(2\pi i/2\Delta)$ を計算するための三角関数の呼び出しは $2\log_2 N$ 回ですむ。

4.3 基数２の周波数空間法 (Decimation-in-Frequency, DIF)

周波数空間法のアルゴリズムの導出は、添え字 a の最下位ビットを分けることから始まる。$N = 16$ の場合を例にして、周波数空間法 FFT の式を書いてみると、

$$\begin{aligned}
W_{16}^{[a_3a_2a_1a_0][b_3b_2b_1b_0]} &= W_{16}^{[a_3a_2a_1a_0][b_2b_1b_0]} W_{16}^{[a_3a_2a_1a_0][b_3000]} && (45)\\
&= W_8^{[a_3a_2a_1][b_2b_1b_0]} W_{16}^{a_0[b_2b_1b_0]} W_2^{a_0b_3} && (46)\\
&= W_8^{[a_3a_2a_1][b_2b_1b_0]} W_{16}^{a_0[b_2b_1b_0]} (-1)^{a_0b_3} && (47)
\end{aligned}$$

のようになる。以下の項を同様のやり方で展開すると、

$$W_8^{[a_3a_2a_1][b_2b_1b_0]} \quad (48)$$

前節とは違った形に簡略化された式を得ることができる。

$$h(a) = \sum_{b_0=0}^{1} (-1)^{a_3b_0} W_4^{a_2b_0} \sum_{b_1=0}^{1} (-1)^{a_2b_1} W_8^{a_1[b_1b_0]} \sum_{b_2=0}^{1} (-1)^{a_1b_2} W_{16}^{a_0[b_2b_1b_0]} \sum_{b_3=0}^{1} (-1)^{a_0b_3} g(b) \quad (49)$$

この式において、前節と同様に再帰的に級数を計算することでアルゴリズムとして実装できるようになる。しかしここでは a のビットの順序が逆になってはいないので、a および b について、そのまま自然な順序で置き換え演算ができる。

$$\begin{aligned}
g_1([a_0b_2b_1b_0]) &= W_{16}^{a_0[b_2b_1b_0]} \sum_{b_3=0}^{1} (-1)^{a_0b_3} g([b_3b_2b_1b_0]) && (50)\\
g_2([a_0a_1b_1b_0]) &= W_8^{a_1[b_1b_0]} \sum_{b_2=0}^{1} (-1)^{a_1b_2} g_1([a_0b_2b_1b_0]) && (51)\\
g_3([a_0a_1a_2b_0]) &= W_4^{a_2b_0} \sum_{b_1=0}^{1} (-1)^{a_2b_1} g_2([a_0a_1b_1b_0]) && (52)\\
h(a) = g_4([a_0a_1a_2a_3]) &= \sum_{b_0=0}^{1} (-1)^{a_3b_0} g_3([a_0a_1a_2b_0]) && (53)
\end{aligned}$$

再帰の最終パスにおいて、$h(a)$ のデータはビット逆順になっているが、これは最後に一度だけ逆順にすればよいだけである。

各パスにおける置き換えの基本的なバタフライ演算は以下のようになり、時間空間法の場合とは若干異なっている。

$$\begin{pmatrix} g(\hat{a}+\hat{b}) \\ g(\hat{a}+\Delta+\hat{b}) \end{pmatrix} \leftarrow \begin{pmatrix} g(\hat{a}+\hat{b}) + g(\hat{a}+\Delta+\hat{b}) \\ W_\Delta^{\hat{b}} \left(g(\hat{a}+\hat{b}) - g(\hat{a}+\Delta+\hat{b}) \right) \end{pmatrix} \quad (54)$$

各パスで $\hat{b}$ は $0 \ldots \Delta-1$、$\hat{a}$ は $0, 2\Delta, \ldots, (N/\Delta-1)\Delta$ である。最初のパスは、ここの例では $\Delta = 16$ で始まり、その後のパスでは Δ の値は $8, 4, \ldots, 1$ となる。

$g(0) \ldots g(2^n - 1)$ が長さ 2^n のデータとするとき、これに対する正規の基数２の周波数空間法 FFT アルゴリズムは、以下のようになる。

```
Δ ⇐ 2^{n−1}
for pass = 1 ... n do
   W ⇐ exp(−2πi/2Δ)
   for (b = 0; b < Δ; b++) do
      for (a = 0; a < N; a+ = 2 ∗ Δ) do
         t_0 ⇐ g(b + a) + g(a + Δ + b)
         t_1 ⇐ W^b (g(a + b) − g(a + Δ + b))
         g(a + b) ⇐ t_0
         g(a + Δ + b) ⇐ t_1
      end for
   end for
   Δ ⇐ Δ/2
end for
bit-reverse ordering of g
```

5 自己整列混合基数複素数 FFT

この章では クライブ・テンパートンの *Self-sorting Mixed-Radix Fast Fourier Transforms*[19] を元に解説する。この文献にはすべてのアルゴリズムが解説されている (アルゴリズムにはさまざまなバリエーションがある) が、ここでは混合基数周波数空間法でもっとも簡潔なものについて導出を行う。

任意のデータ長 N の FFT を行いたい場合、二進表記の基数 2 のアルゴリズムは実用的ではない。データベクトルと行列の積を計算する、混合基数の FFT を構築せねばならない。このために、まず DFT 行列 W_N を計算し、これを、N の各因数に対応する小さくてスパースな行列に分割する。

ここでは行列の要素を表すのに、関数形 (下式右辺) と、一般的な添え字 (下式左辺) と、両方を用いる。

$$M_{ij} = M(i, j) \tag{55}$$

上式における i, j は、(C 言語の配列のように) 0 から $N - 1$ の範囲を取るものとする。また行列の積を、角括弧を使って以下のように表す。

$$[AB]_{ij} = \sum_k A_{ik} B_{kj} \tag{56}$$

混合基数の因数分解を用いた DFT では、3 つの行列が必要になる。それぞれ、単位行列 I、置換行列 P、回転因子行列 D である。D は上述した一般的な DFT の W_n に相当する。

ここでは次数 r の単位行列を $I_r(n, m)$ と表す。

$$I_r(n, m) = \delta_{nm} \tag{57}$$

ここで $0 \leq n, m \leq r - 1$ である。

置換行列 P^a_b は、ベクトル中の要素の順序を入れ替える演算に用いる。ベクトル要素の添え字 $j = 0 \ldots N - 1$ が $j = la + m$ ($0 \leq l \leq b - 1$ かつ $0 \leq m \leq a - 1$) の形に表される時、行列 P による演算で、そのベクトルの $la + m$ 番目の要素と $mb + l$ 番目の要素が入れ替わる (上述のビット逆順演算の一般化である)。

P は大きさが $ab \times ab$ の正方行列で、以下の性質を持つ。

$$P^a_b(j, k) = 1 \text{ if } j = ra + s \text{ and } k = sb + r \tag{58}$$

$$= 0 \text{ otherwise} \tag{59}$$

3 つめの行列、回転因子行列 D_b^a は三角関数の値の和を保持するためのものである。D_b^a は以下で定義される大きさが $ab \times ab$ の正方対角行列である。

$$
\begin{aligned}
D_b^a(j,k) &= \omega_{ab}^{sr} \text{ if } j = k = sb + r & (60)\\
&= 0 \text{ otherwise} & (61)
\end{aligned}
$$

ここで $\omega_{ab} = e^{-2\pi i/ab}$ である。

5.1 クロネッカー積

行列のクロネッカー積は、複数の相異なる部分空間に渡ってなんらかの演算を行うアルゴリズムでは一般的に重要とされる演算である。二つの行列大きさ $a \times a$ の行列 A と $b \times b$ の行列 B の、二つの行列のクロネッカー積 $A \otimes B$ は、以下で表される大きさ $ab \times ab$ として定義される。

$$
[A \otimes B](tb + u, rb + s) = A(t, r)B(u, s) \tag{62}
$$

ここで $0 \leq u, s < b$、$0 \leq t, r < a$ である。例として大きさ 2×2 および 3×3 の二つの行列

$$
A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \quad B = \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} \tag{63}
$$

の場合を考えてみると、これらのクロネッカー積 $A \otimes B$ は以下のようになる。

$$
\begin{aligned}
A \otimes B &= \begin{pmatrix} a_{11}B & a_{12}B \\ a_{12}B & a_{22}B \end{pmatrix} & (64)\\
&= \begin{pmatrix}
a_{11}b_{11} & a_{11}b_{12} & a_{11}b_{13} & a_{12}b_{11} & a_{12}b_{12} & a_{12}b_{13} \\
a_{11}b_{21} & a_{11}b_{22} & a_{11}b_{23} & a_{12}b_{21} & a_{12}b_{22} & a_{12}b_{23} \\
a_{11}b_{31} & a_{11}b_{32} & a_{11}b_{33} & a_{12}b_{31} & a_{12}b_{32} & a_{12}b_{33} \\
a_{21}b_{11} & a_{21}b_{12} & a_{21}b_{13} & a_{22}b_{11} & a_{22}b_{12} & a_{22}b_{13} \\
a_{21}b_{21} & a_{21}b_{22} & a_{21}b_{23} & a_{22}b_{21} & a_{22}b_{22} & a_{22}b_{23} \\
a_{21}b_{31} & a_{21}b_{32} & a_{21}b_{33} & a_{22}b_{31} & a_{22}b_{32} & a_{22}b_{33}
\end{pmatrix} & (65)
\end{aligned}
$$

クロネッカー積 $A \otimes B$ を長さ ab のベクトルにかける演算を考えると、クロネッカー積中の各行列は、そのベクトル中の相異なる部分空間に対する演算になる。ここで i を $i = tb + u$ $(0 \leq u \leq b-1$、$0 \leq t \leq a)$ となるような添え字とすると、その部分空間に対する演算を明示的に表すことができる。

$$
\begin{aligned}
[(A \otimes B)v]_{(tb+u)} &= \sum_{t'=0}^{a-1} \sum_{u'=0}^{b-1} [A \otimes B]_{(tb+u, t'b+u')} v_{t'b+u'} & (66)\\
&= \sum_{t'u'} A_{tt'} B_{uu'} v_{t'b+u'} & (67)
\end{aligned}
$$

行列 B は「添え字」u' に、行列 A は「添え字」t' に対応する。FFT における因数分解でもっとも重要なのは、二つのクロネッカー積の積が、二つの行列の積のクロネッカー積と同じである、という性質である。

$$
(A \otimes B)(C \otimes D) = (AC \otimes BD) \tag{68}
$$

この性質はクロネッカー積の定義から直ちに示される。

5.2 因数が二つ ($N = ab$) の場合

もっとも簡単な例として、データ長 N が二つの数の積 $N = ab$ で表される場合を考える。まず DFT 行列 W_N を、各因数に対応する小行列に分割する。導出を簡潔にするために、因数分解はすでに行われていて、それを確認することから始める (因数分解が非常に単純な場合を一般化することで、任意のデータ長の場合に対応できる)。因数分解が以下で与えられるとする。

$$W_{ab} = (W_b \otimes I_a) P_b^a D_b^a (W_a \otimes I_b). \tag{69}$$

これに、以下のようにベクトルをかける。

$$\begin{aligned} &[(W_b \otimes I_a) P_b^a D_b^a (W_a \otimes I_b)](la + m, rb + s) && (70) \\ &= \sum_{u=0}^{b-1} \sum_{t=0}^{a-1} [(W_b \otimes I_a)](la + m, ua + t)[P_b^a D_b^a (W_a \otimes I_b)](ua + t, rb + s) && (71) \end{aligned}$$

ここではクロネッカー積に対応するように添え字の範囲を $0 \leq m, r \leq a$, $0 \leq l, s \leq b$ と分けている。上の式で、和記号の中の項の前半は以下のような簡単な形になる。

$$\begin{aligned} [(W_b \otimes I_a)](la + m, ua + t) &= W_b(l, u) I_a(m, t) && (72) \\ &= \omega_b^{lu} \delta_{mt} && (73) \end{aligned}$$

後半はそれほど簡単ではない。まず以下のようにクロネッカー積を展開する。

$$\begin{aligned} (W_a \otimes I_b)(tb + u, rb + s) &= W_a(t, r) I_a(u, s) && (74) \\ &= \omega_a^{tr} \delta_{us} && (75) \end{aligned}$$

そしてこれをつかって $P_b^a D_b^a (W_a \otimes I_b)$ を計算する。まず最初に D_b^a をかける。

$$[D_b^a (W_a \otimes I_b)](tb + u, rb + s) = \omega_{ab}^{tu} \omega_a^{tr} \delta_{su} \tag{76}$$

次に置換行列 P_b^a をかけて、最初の添え字を逆順に並べ替える。

$$[P_b^a D_b^a (W_a \otimes I_b)](ua + t, rb + s) = \omega_{ab}^{tu} \omega_a^{tr} \delta_{su} \tag{77}$$

これで和記号の中の前半と後半が得られたので、元の式に代入する。

$$[(W_b \otimes I_a) P_b^a D_b^a (W_a \otimes I_b)](la + m, rb + s) = \sum_{u=0}^{b-1} \sum_{t=0}^{a-1} \omega_b^{lu} \delta_{mt} \omega_{ab}^{tu} \omega_a^{tr} \delta_{su} \tag{78}$$

この和を計算すると、W_a と W_b を含むこの節の最初の式と、DFT 行列 W_{ab} の関係が得られる。

$$\begin{aligned} \sum_{u=0}^{b-1} \sum_{t=0}^{a-1} \omega_b^{lu} \delta_{mt} \omega_{ab}^{tu} \omega_a^{tr} \delta_{su} &= \omega_b^{ls} \omega_{ab}^{ms} \omega_a^{mr} && (79) \\ &= \omega_{ab}^{als+ms+bmr} && (80) \\ &= \omega_{ab}^{als+ms+bmr} \omega_{ab}^{lrab} \quad \text{using } \omega_{ab}^{ab} = 1 && (81) \\ &= \omega_{ab}^{(la+m)(rb+s)} && (82) \\ &= W_{ab}(la + m, rb + s) && (83) \end{aligned}$$

上式の右辺の最後、行列とベクトルの積は、すなわち二数に因数分解できる場合の DFT の行列 W_{ab} そのものである。したがってデータ長が二つの数 a、b の積に分解できる場合の DFT 行列 W_{ab} は、以下の式で表される、W_a、W_b、置換行列 P、回転因子 D を使ったより短いデータ長の二つの変換に分解できることになる。

$$W_{ab} = (W_b \otimes I_a) P_b^a D_b^a (W_a \otimes I_b). \tag{84}$$

この関係が任意データ長の混合基数 FFT の基礎である。

5.3 因数が三つ ($N = abc$) の場合

因数が二つの場合のやり方は、簡単に三つの場合に適用できる。三つの因数の積 abc をまず二数 a と (bc) の積として、次に (bc) を b と c の積として考えればよい。最初は以下のようになる。

$$\begin{aligned} W_{abc} &= W_{a(bc)} && (85)\\ &= (W_{bc} \otimes I_a) P^a_{bc} D^a_{bc} (W_a \otimes I_{bc}). && (86) \end{aligned}$$

因数が二つの場合の結果を使って W_{bc} に関して展開すると、W_{abc} は以下のように分解される。

$$\begin{aligned} W_{abc} &= (((W_c \otimes I_b) P^b_c D^b_c (W_b \otimes I_c)) \otimes I_a) P^a_{bc} D^a_{bc} (W_a \otimes I_{bc}) && (87)\\ &= (W_c \otimes I_{ab})(P^b_c D^b_c \otimes I_a)(W_b \otimes I_{ac}) P^a_{bc} D^a_{bc} (W_a \otimes I_c) && (88) \end{aligned}$$

これは、以下のように三つの行列の積として書くことができる。

$$W_{abc} = T_3 T_2 T_1 \tag{89}$$

ここで T_1, T_2 and T_3 はそれぞれ以下である。

$$\begin{aligned} T_1 &= P^a_{bc} D^a_{bc} (W_a \otimes I_{bc}) && (90)\\ T_2 &= (P^b_c D^b_c \otimes I_a)(W_b \otimes I_{ac}) && (91)\\ T_3 &= (W_c \otimes I_{ab}) && (92) \end{aligned}$$

5.4 任意のデータ長の場合 ($N = f_1 f_2 \ldots f_{n_f}$)

前節までのやり方を拡張することで、任意のデータ長の場合に DFT 行列 $W_{f_1 f_2 \ldots f_{n_f}}$ を分解することができる。前節の最後の式は、以下のようにも書ける。

$$\begin{aligned} T_1 &= (P^a_{bc} D^a_{bc} \otimes I_1)(W_a \otimes I_{bc}) && (93)\\ T_2 &= (P^b_c D^b_c \otimes I_a)(W_b \otimes I_{ac}) && (94)\\ T_3 &= (P^c_1 D^c_1 \otimes I_{ab})(W_c \otimes I_{ab}) && (95) \end{aligned}$$

ここで $P^c_1 = D^c_1 = I_c$ である。一般的に、W_N for $N = f_1 f_2 \ldots f_{n_f}$ は以下の形に書ける。

$$W_N = T_{n_f} \ldots T_2 T_1 \tag{96}$$

この場合 T_i は以下のようになる。

$$T_i = (P^{f_i}_{q_i} D^{f_i}_{q_i} \otimes I_{p_{i-1}})(W_{f_i} \otimes I_{m_i}) \tag{97}$$

上式中の p、q、m は因数の部分積であり、以下のように定義している。

$$\begin{aligned} p_i &= f_1 f_2 \ldots f_i \quad (p_0 = 1) && (98)\\ q_i &= N/p_i && (99)\\ m_i &= N/f_i && (100) \end{aligned}$$

FFT 行列 W を P の前に適用すると、周波数間引き法のアルゴリズムになる。

5.5 実装

以下で実装について述べる。データベクトル z を変換アルゴリズムに対する入力とすると、FFT は以下で表される。

$$
\begin{aligned}
x &= W_N z & (101)\\
&= T_{n_f} \ldots T_2 T_1 z & (102)
\end{aligned}
$$

ここで $T_i = (P_{q_i}^{f_i} D_{q_i}^{f_i} \otimes I_{p_{i-1}})(W_{f_i} \otimes I_{m_i})$ である。

アルゴリズムの大枠は、データ長を因数分解した各因数 n_f に関するループであり、ループ内で行列 T_i が適用され、各ループの結果がまとめられて最終的な出力となる。

for $(i = 1 \ldots n_f)$ **do**
  $v \Leftarrow T_i v$
end for

各因数がどの程度の大きさの値であるのかは重要な問題ではない。ここで、繰り返し計算 $v \Leftarrow T_i v$ を以下のように書くとする。

$$v' = (P_{q_i}^{f_i} D_{q_i}^{f_i} \otimes I_{p_{i-1}})\,(W_{f_i} \otimes I_{m_i}) v \tag{103}$$

この中には二つのクロネッカー積が含まれている。行列の積はもっとも右の行列から最初に計算されるが、最初はデータ長 f_i の DFT を N/f_i 個のより短い DFT に分解することである。これを t と置くと、以下のように書ける。これはアルゴリズム内で一時的に値を保存する領域となる。

$$t = (W_{f_i} \otimes I_{m_i}) v \tag{104}$$

二つ目の行列は置換行列と回転因子行列とにかけられる。これを v と置くと、以下のように書ける。これはアルゴリズム内で、v についての繰り返し計算の結果を保持する領域となる。

$$v' = (P_{q_i}^{f_i} D_{q_i}^{f_i} \otimes I_{p_{i-1}}) t \tag{105}$$

行列 $(W_{f_i} \otimes I_{m_i})$ の働きは、例を示すとよく分かる。$f_i = 3$ で FFT のデータ長が $N = 6$ の場合を考えると、クロネッカー積は以下のようになる。

$$t = (W_3 \otimes I_2) v \tag{106}$$

これを展開すると以下のようになる。

$$
\begin{pmatrix} t_0 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \end{pmatrix}
=
\begin{pmatrix}
W_3(1,1) & 0 & W_3(1,2) & 0 & W_3(1,3) & 0 \\
0 & W_3(1,1) & 0 & W_3(1,2) & 0 & W_3(1,3) \\
W_3(2,1) & 0 & W_3(2,2) & 0 & W_3(2,3) & 0 \\
0 & W_3(2,1) & 0 & W_3(2,2) & 0 & W_3(2,3) \\
W_3(3,1) & 0 & W_3(3,2) & 0 & W_3(3,3) & 0 \\
0 & W_3(3,1) & 0 & W_3(3,2) & 0 & W_3(3,3)
\end{pmatrix}
\begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{pmatrix}
\tag{107}
$$

ここで計算機での扱いがやりやすくなるように要素を並べ替える。

$$
\begin{pmatrix} t_0 \\ t_2 \\ t_4 \\ t_1 \\ t_3 \\ t_5 \end{pmatrix}
=
\begin{pmatrix}
W_3(1,1) & W_3(1,2) & W_3(1,3) & 0 & 0 & 0 \\
W_3(2,1) & W_3(2,2) & W_3(2,3) & 0 & 0 & 0 \\
W_3(3,1) & W_3(3,2) & W_3(3,3) & 0 & 0 & 0 \\
0 & 0 & 0 & W_3(1,1) & W_3(1,2) & W_3(1,3) \\
0 & 0 & 0 & W_3(2,1) & W_3(2,2) & W_3(2,3) \\
0 & 0 & 0 & W_3(3,1) & W_3(3,2) & W_3(3,3)
\end{pmatrix}
\begin{pmatrix} v_0 \\ v_2 \\ v_4 \\ v_1 \\ v_3 \\ v_5 \end{pmatrix}
\tag{108}
$$

すると、3×3 の DFT 行列 W_3 を二回かけるだけでよいのが分かる。一回目はデータベクトルの部分ベクトル (v_0, v_2, v_4)、二回目はデータベクトルの部分ベクトル (v_1, v_3, v_5) にかけられる。

一般的には、添え字 t が $t_k = t(\lambda, \mu) = t_{\lambda m+\mu}$ の場合 $\lambda = 0 \dots f-1$ が長さ f の各 FFT の添え字になり、各部分ベクトルには $\mu = 0 \dots m-1$ というラベルを付けることができる。全ての添え字を明示することでそれが分かる。元の式

$$t = (W_f \otimes I_m) z \tag{109}$$

は以下のようになる。

$$\begin{aligned} t_{\lambda m+\mu} &= \sum_{\lambda'=0}^{f-1} \sum_{\mu'=0}^{m-1} (W_f \otimes I_m)_{(\lambda m+\mu)(\lambda' m+\mu')} z_{\lambda' m+\mu} && (110) \\ &= \sum_{\lambda'\mu'} (W_f)_{\lambda\lambda'} \delta_{\mu\mu'} z_{\lambda' m+\mu'} && (111) \\ &= \sum_{\lambda'} (W_f)_{\lambda\lambda'} z_{\lambda' m+\mu} && (112) \end{aligned}$$

添え字 λ についての DFT は、各 f について最適化されたルーチンで行われる。

次のステージ

$$v' = (P_q^f D_q^f \otimes I_{p_{i-1}}) t \tag{113}$$

を計算するには、クロネッカー積を計算する際に、t を q_{i-1} 個に分割したそれぞれについて、行列の積 PD がそれぞれ独立に p_{i-1} 個、適切に計算されねばならない。$t(\lambda, \mu)$ の添え字 μ の範囲は 0 から m までだが、$m = p_{i-1} q$ なので、各 PD の計算について μ のうち q_i 個は同じものである。つまり μ はさらに $\mu = a p_{i-1} + b$ ($a = 0 \dots q-1$、$b = 0 \dots p_{i-1}$) と分解できると言うことである。

$$\begin{aligned} \lambda m + \mu &= \lambda m + a p_{i-1} + b && (114) \\ &= (\lambda q + a) p_{i-1} + b. && (115) \end{aligned}$$

すると第 2 ステージ v' は以下のように展開できる。

$$\begin{aligned} v' &= (P_q^f D_q^f \otimes I_{p_{i-1}}) t && (116) \\ v'_{\lambda m+\mu} &= \sum_{\lambda'\mu'} (P_q^f D_q^f \otimes I_{p_{i-1}})_{(\lambda m+\mu)(\lambda' m+\mu')} t_{\lambda' m+\mu'} && (117) \\ v'_{(\lambda q+a)p_{i-1}+b} &= \sum_{\lambda' a' b'} (P_q^f D_q^f \otimes I_{p_{i-1}})_{((\lambda q+a)p_{i-1}+b)((\lambda' q+a')p_{i-1}+b')} t_{(\lambda' q+a')p_{i-1}+b'} && (118) \end{aligned}$$

単位行列 I とクロネッカー積の部分空間をうまく利用して、冗長な添え字を取り除くことができる。

$$(P_q^f D_q^f \otimes I_{p_{i-1}})_{((\lambda q+a)p_{i-1}+b)((\lambda' q+a')p_{i-1}+b')} = (P_q^f D_q^f)_{(\lambda q+a)(\lambda' q+a')} \delta_{bb'} \tag{119}$$

これにより和記号を一つ省くことができ、以下のようになる。

$$v'_{(\lambda q+a)p_{i-1}+b} = \sum_{\lambda' a'} (P_q^f D_q^f)_{(\lambda q+a)(\lambda' q+a')} t_{(\lambda' q+a')p_{i-1}+b} \tag{120}$$

ここで D_q^f の定義を代入すると、以下のようになる。

$$= \sum_{\lambda' a'} (P_q^f)_{(\lambda q+a)(\lambda' q+a')} \omega_{q_{i-1}}^{\lambda' a'} t_{(\lambda' q+a')p_{i-1}+b} \tag{121}$$

P_q^f は $\lambda q + a$ と $af + \lambda$ を入れ替える置換行列だったが、その定義から最終的に以下の式を得ることができる。

$$v'_{(af+\lambda)p_{i-1}+b} = \omega_{q_{i-1}}^{\lambda a} t_{(\lambda q+a)p_{i-1}+b} \qquad (122)$$

実装せねばならないことは、一時的なベクトル t の各要素に回転因子をかけて、その結果を、逆順置換行列 P によって決められる場所に保存することである。

混合基数 FFT のアルゴリズムの実装は、以下のようになる。

for $i = 1 \ldots n_f$ **do**
  for $a = 0 \ldots q-1$ **do**
    for $b = 0 \ldots p_{i-1}-1$ **do**
      for $\lambda = 0 \ldots f-1$ **do**
        $t_\lambda \Leftarrow \sum_{\lambda'=0}^{f-1} W_f(\lambda,\lambda') v_{b+\lambda' m+ap_{i-1}}$ {DFT matrix-multiply module}
      end for
      for $\lambda = 0 \ldots f-1$ **do**
        $v'_{(af+\lambda)p_{i-1}+b} \Leftarrow \omega_{q_{i-1}}^{\lambda a} t_\lambda$
      end for
    end for
  end for
  $v \Leftarrow v'$
end for

5.6 実装の詳細

GSL の関数 `gsl_fft_complex_wavetable_alloc` はその内部でまず最初に計算用のメモリ領域として要素数 n の配列を 2 つ確保する。一つには繰り返し計算の各回でベクトル v' を保存し、もう一つは三角関数の値を計算して入れておき、回転因子の値の表とする。

次に n を因数分解する。そのための関数 `gsl_fft_factorize` が定義されており、これはできるだけ返して欲しい因数のリストを引数に取る。この関数はまず n を引数で渡されたリスト中の数で除し、次にそのリストにはない素数で除す。

回転因子の計算に使うアルゴリズムは以下である。

for $a = 1 \ldots n_f$ **do**
  for $b = 1 \ldots f_i - 1$ **do**
    for $c = 1 \ldots q_i$ **do**
      trig[k++] = $\exp(-2\pi ibcp_{a-1}/N)$
    end for
  end for
end for

ここで $\sum_1^{n_f}\sum_0^{f_i-1}\sum_1^{q_i} = \sum_1^{n_f}(f_i-1)q_i = n-1$ なので、回転因子の表の大きさは n でよいことになる。FFT 中で $\omega_{q_i-1}^{\lambda a} t_\lambda$ を計算する必要があるためにこのようにしている。この表を使うことで、

$$\begin{aligned}
\omega_{q_{i-1}}^{\lambda a} t_\lambda &= \exp(-2\pi i\lambda a/q_{i-1}) t_\lambda && (123)\\
&= \exp(-2\pi i\lambda a p_{i-1}/N) t_\lambda && (124)\\
&= \begin{cases} t_\lambda & a = 0 \\ \text{trig}[\text{twiddle[i]} + \lambda q + (a-1)] t_\lambda & a \neq 0 \end{cases} && (125)
\end{aligned}$$

とすることができる。ここで `twiddle[i]` には `trig` へのポインタが格納されており、外側のループはその値が初期値になる。実装の中心的な部分は関数 `gsl_fft_complex` である。この関数では、各因

数 N について繰り返し計算 $v' = T_i v$ が行われる。ある因数についての計算では、値の小さな N については、その値に特化して書かれたルーチン (gsl_fft_complex_pass_3、gsl_fft_complex_pass_5 など) が使われる。そういったルーチンのない N については gsl_fft_complex_pass_n が使われる。それぞれの特化ルーチンは前述した基礎アルゴリズムを簡潔に実装したものである。以下に gsl_fft_complex_pass_3 の例を示す。因数が 3 のときは、以下の計算をする必要がある。

$$v'_{(af+\lambda)p_{i-1}+b} = \sum_{\lambda'=0,1,2} \omega_{q_{i-1}}^{\lambda a} W_3^{\lambda\lambda'} v_{b+\lambda' m+ap_{i-1}} \tag{126}$$

ここで $b = 0 \ldots p_{i-1} - 1$, $a = 0 \ldots q_i - 1$ and $\lambda = 0, 1, 2$ である。この計算は以下のように実装されている。

for $a = 0 \ldots q - 1$ **do**
  for $b = 0 \ldots p_{i-1} - 1$ **do**
    $$\begin{pmatrix} t_0 \\ t_1 \\ t_2 \end{pmatrix} = \begin{pmatrix} W_3^0 & W_3^0 & W_3^0 \\ W_3^0 & W_3^1 & W_3^2 \\ W_3^0 & W_3^2 & W_3^4 \end{pmatrix} \begin{pmatrix} v_{b+ap_{i-1}} \\ v_{b+ap_{i-1}+m} \\ v_{b+ap_{i-1}+2m} \end{pmatrix}$$
    $v'_{ap_i+b} = t_0$
    $v'_{ap_i+b+p_{i-1}} = \omega_{q_{i-1}}^{a} t_1$
    $v'_{ap_i+b+2p_{i-1}} = \omega_{q_{i-1}}^{2a} t_2$
  end for
end for

このコードにおいて以下のように、入力となるデータを示す添え字を変数 from0、from1、from2 で与えるとする。

$$\texttt{from0} = b + ap_{i-1} \tag{127}$$
$$\texttt{from1} = b + ap_{i-1} + m \tag{128}$$
$$\texttt{from2} = b + ap_{i-1} + 2m \tag{129}$$

また v' の中で出力となる値がどれかを示す変数を to0、to1、to2 とする。

$$\texttt{to0} = b + ap_i \tag{130}$$
$$\texttt{to1} = b + ap_i + p_{i-1} \tag{131}$$
$$\texttt{to2} = b + ap_i + 2p_{i-1} \tag{132}$$

DFT における行列の乗算は、次の節で説明する、より小さな変換を行うルーチンで行われる。回転因子の値 $\omega_{q_{i-1}}^{a}$ は配列 trig にあるものを使う。

逆変換は、フーリエ変換の定義と、逆行列が順方向変換のものの複素共役 (を $1/N$ 倍したもの) であることから導かれる。

$$W_N^{-1} = W_N^{*}/N \tag{133}$$

これにより DFT 行列と回転因子行列の全ての要素の複素共役を使うことで逆変換ができることが分かる (または、入力となるデータを複素共役にして順方向変換を行い、出力の複素共役を得ればそれが逆変換となる)。

6 高速な小変換ルーチン

混合基数 FFT を実装するためには、いくつもの小さな因数 N にそれぞれ特化したルーチンを書かねばならない。しかし幸いにも優れた方法がいくつかある。そのなかから、ウィノグラッド (Winograd) による数論を応用した方法を以下に示す。

この節で説明するアルゴリズムは、

$$x_a = \sum_{b=0}^{N-1} W_N^{ab} z_b \tag{134}$$

を計算するものである。この「小変換ルーチン」は、同じくウィノグラッドによる正規法とは違っていて、テンパートン (Temperton) が推薦している方法である。彼の論文 [19] では、この方法は丸め誤差に強く、加算と乗算のバランスがよいとされている。$N = 2$ の DFT は以下の式で行われる。

$$x_0 = z_0 + z_1, \quad x_1 = z_0 - z_1 \tag{135}$$

また $N = 3$ の DFT は以下である。

$$t_1 = z_1 + z_2, \quad t_2 = z_0 - t_1/2, \quad t_3 = \sin(\pi/3)(z_1 - z_2), \tag{136}$$

$$x_0 = z_0 + t_1, \quad x_1 = t_2 + it_3, \quad x_2 = t_2 - it_3 \tag{137}$$

$N = 4$ の DFT は、加算と減算のみで行われる。

$$t_1 = z_0 + z_2, \quad t_2 = z_1 + z_3, \quad t_3 = z_0 - z_2, \quad t_4 = z_1 - z_3, \tag{138}$$

$$x_0 = t_1 + t_2, \quad x_1 = t_3 + it_4, \quad x_2 = t_1 - t_2, \quad x_3 = t_3 - it_4. \tag{139}$$

$N = 5$ の DFT は以下で行われる。

$$t_1 = z_1 + z_4, \quad t_2 = z_2 + z_3, \quad t_3 = z_1 - z_4, \quad t_4 = z_2 - z_3, \tag{140}$$

$$t_5 = t_1 + t_2, \quad t_6 = (\sqrt{5}/4)(t_1 - t_2), \quad t_7 = z_0 - t_5/4, \tag{141}$$

$$t_8 = t_7 + t_6, \quad t_9 = t_7 - t_6, \tag{142}$$

$$t_{10} = \sin(2\pi/5)t_3 + \sin(2\pi/10)t_4, \quad t_{11} = \sin(2\pi/10)t_3 - \sin(2\pi/5)t_4, \tag{143}$$

$$x_0 = z_0 + t_5, \tag{144}$$

$$x_1 = t_8 + it_{10}, \quad x_2 = t_9 + it_{11}, \tag{145}$$

$$x_3 = t_9 - it_{11}, \quad x_4 = t_8 - it_{10}. \tag{146}$$

$N = 6$ の DFT 行列は $N = 3$ のものと $N = 2$ のものを、要素をうまく並べ替えて組み合わせることで得られる。

$$\begin{pmatrix} x_0 \\ x_4 \\ x_2 \\ \hline x_3 \\ x_1 \\ x_5 \end{pmatrix} = \left(\begin{array}{c|c} W_3 & W_3 \\ \hline W_3 & -W_3 \end{array}\right) \begin{pmatrix} z_0 \\ z_2 \\ z_4 \\ \hline z_3 \\ z_5 \\ z_1 \end{pmatrix} \tag{147}$$

こうした簡略化は、因数の 2 と 3 が互いに素であるために可能であり、PFA (Prime Factor Algorithm 素因数法) と呼ばれる。詳細は FFT のための数論の書籍 [6, 7] を参照されたい。PFA による簡略化の利点を生かすと、$N = 6$ の場合の DFT は以下のようになる。

$$t_1 = z_2 + z_4, \quad t_2 = z_0 - t_1/2, \quad t_3 = \sin(\pi/3)(z_2 - z_4), \tag{148}$$

$$t_4 = z_5 + z_1, \quad t_5 = z_3 - t_4/2, \quad t_6 = \sin(\pi/3)(z_5 - z_1), \tag{149}$$

$$t_7 = z_0 + t_1, \quad t_8 = t_2 + it_3, \quad t_9 = t_2 - it_3, \tag{150}$$

$$t_{10} = z_3 + t_4, \quad t_{11} = t_5 + it_6, \quad t_{12} = t_5 - it_6, \tag{151}$$

$$x_0 = t_7 + t_{10}, \quad x_4 = t_8 + t_{11}, \quad x_2 = t_9 + t_{12}, \tag{152}$$

$$x_3 = t_7 - t_{10}, \quad x_1 = t_8 - t_{11}, \quad x_5 = t_9 - t_{12}. \tag{153}$$

他の任意の因数については、シングルトン (Singleton) の高効率法 [20] を使う。この方法の計算量は $O(N^2)$ だが、DFT の元の定義をそのまま計算する場合と比較すると、乗算の回数を 1/4 に減らすことができる。DFT 行列の全体をみてみると、以下のような形になっている。

$$\begin{pmatrix} h_0 \\ h_1 \\ h_2 \\ \vdots \\ h_{N-2} \\ h_{N-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 & 1 \\ 1 & W & W & \cdots & W & W \\ 1 & W & W & \cdots & W & W \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ 1 & W & W & \cdots & W & W \\ 1 & W & W & \cdots & W & W \end{pmatrix} \begin{pmatrix} g_0 \\ g_1 \\ g_2 \\ \vdots \\ g_{N-2} \\ g_{N-1} \end{pmatrix} \tag{154}$$

ここで、DFT 行列の第一行と第一列は、全ての要素が 1 であり、これを乗ずる演算は省くことができるが、それでも $(N-1) \times (N-1)$ の大きさの部分行列 (要素は複素数) が残っており、$(N-1)^2$ 回の複素数の乗算が必要である。シングルトンの方法では DFT 行列の対称性を利用して、複素数の乗算を等価な実数の乗算に置き換える。まず DFT の定義をもう一度見てみると、

$$a_k + ib_k = \sum_{j=0} (x_j + iy_j)(\cos(2\pi jk/f) - i\sin(2\pi jk/f)) \tag{155}$$

零番目の要素は、加算だけで計算できるのがわかる。

$$a_0 + ib_0 = \sum_{j=0}^{(f-1)} x_j + iy_j \tag{156}$$

他の要素については、以下のように書き換えられる。

$$a_k + ib_k = x_0 + iy_0 + \sum_{j=1}^{(f-1)/2} (x_j + x_{f-j}) \cos(2\pi jk/f) + (y_j - y_{f-j}) \sin(2\pi jk/f) \tag{157}$$

$$+i \sum_{j=1}^{(f-1)/2} (y_j + y_{f-j}) \cos(2\pi jk/f) - (x_j - x_{f-j}) \sin(2\pi jk/f) \tag{158}$$

これには、以下の等式を使っている。

$$\cos(2\pi(f-j)k/f) = \cos(2\pi jk/f) \tag{159}$$

$$\sin(2\pi(f-j)k/f) = -\sin(2\pi jk/f) \tag{160}$$

これらの要素は、下式で $k = 1, 2, \ldots, (f-1)/2$ について 4 つの部分和を使うことで全て計算することができる。

$$
\begin{aligned}
a_k + ib_k &= (a_k^+ - a_k^-) + i(b_k^+ + b_k^-) && (161) \\
a_{f-k} + ib_{f-k} &= (a_k^+ + a_k^-) + i(b_k^+ - b_k^-) && (162)
\end{aligned}
$$

ここで

$$
\begin{aligned}
a_k^+ &= x_0 + \sum_{j=1}^{(f-1)/2} (x_j + x_{f-j}) \cos(2\pi jk/f) && (163) \\
a_k^- &= \quad - \sum_{j=1}^{(f-1)/2} (y_j - y_{f-j}) \sin(2\pi jk/f) && (164) \\
b_k^+ &= y_0 + \sum_{j=1}^{(f-1)/2} (y_j + y_{f-j}) \cos(2\pi jk/f) && (165) \\
b_k^- &= \quad - \sum_{j=1}^{(f-1)/2} (x_j - x_{f-j}) \sin(2\pi jk/f) && (166)
\end{aligned}
$$

である。一度 a^+, a^-, b^+ and b^- の値が得られれば、それを使って直接的に $k' = f - k$ での値も得られる。計算せねばならない和の数は $4 \times (f-1)/2$ であり、各和の計算では $(f-1)/2$ 回の実数の乗算があるので、トータルでは $(f-1)^2$ 回の実数の乗算を、複素数の $(f-1)^2$ 回の乗算の代わりに行えばよい。

シングルトンの方法を実装するには、入力と出力のために v と v' という領域を用意し、最終的な結果を得るまでこれらを書き写していく必要がある。最初に v' にそれぞれ $x_j + x_{f-j}$ および $x_j - y_{f-j}$ の形の、対称な、および対称でないベクトルを保存する。次にこれらに回転因子をかけて a^+、a^-、b^+、b^- を計算し、計算結果 $a_k + ib_k$ および d $a_{f-k} + ib_{f-k}$ を v に書き戻す。最終的に必要な回転因子をかけて DFT を得て、v' に保存する。

7 実数に対する FFT

この章では、クライブ・テンパートン (Clive Temperton) の論文 *Fast Mixed-Radix Real Fourier Transforms*[21] およびソレンセン、ジョーンズ、ハイデマン、バラスの論文 *Real-Valued Fast Fourier Transform Algorithms*[22] を元に説明する。実数列に対する DFT は特徴的な対称性を持っている。それは複素共役 *(conjugate-complex)* 対称性、または半複素 *(half-complex)* 対称性と呼ばれており、以下のように表される。

$$h(a) = h(N-a)^* \qquad (167)$$

先頭の要素 $h(0)$ は実数になる。また N が偶数の時は $h(N/2)$ も実数になる。この共役対称性は、以下のようにして証明できる。

$$
\begin{aligned}
h(a) &= \sum W_N^{ab} g(b) && (168) \\
h(N-a)^* &= \sum W_N^{-(N-a)b} g(b)^* && (169) \\
&= \sum W_N^{-Nb} W_N^{ab} g(b) \qquad (W_N^N = 1) && (170) \\
&= \sum W_N^{ab} g(b) && (171)
\end{aligned}
$$

現実的な応用では、データが実数であることはごく普通である (たぶん複素数の場合よりも多い)。したがって実数に特化した FFT ルーチンは重要である。実数に対する FFT は、原理的には、等価な複素数データに対する FFT の半分の演算回数を必要とする (虚数部を 0 にして行うということ)。実数 FFT の実装には、二通りの考え方がある。

一つは長さ N の実数データを、長さ $N/2$ の複素数配列に「パック (pack)」することである。この場合、複素数の対する FFT ルーチンがそのまま使える。「パック」は配列の添え字を操作することで行われるが、複素数 FFT ルーチンの出力を実数 FFT として解釈する (展開 unpack) のも添え字の操作で行われる。同じ長さの二つのデータを同時に FFT することもできる。これは、複素数配列の実部に一方のデータを、虚部にもう一方のデータを格納して FFT することにより行われる。しかしこの「パック」と「展開」の操作にはそれぞれ $O(N)$ の計算量が必要となる。また長さ N の実数データを長さ $N/2$ の複素数配列に格納するためには、N が偶数でなくてはならない。さらに、数値の大きさが大きく異なる二つのデータを一つの複素数配列にパックして一度に FFT した場合、数値の「クロストーク (cross-talk)」が発生して精度が失われる可能性がある。

または、複素数に対する混合基数 FFT のようなルーチンを、入力データの虚部を 0 とおいて、より簡潔に書き直す方法がある。そうすればデータ長は任意でよくなり、不要な演算を省くことができる。また、半複素数列から元の実数データを得る逆変換のアルゴリズムも導出できる。

7.1 基数 2 の実数 FFT

混合基数の実数 FFT の前に、基数 2 の場合について述べる。任意のデータ長 N の FFT の本質的なことは、基数 2 の FFT に含まれているが、この二者の間の対応が分かりやすくなるように、以下では、因数の添え字は混合基数の場合と同じにする。まず、因数が全部 2 の場合、

$$f_1 = 2, f_2 = 2, \dots, f_{n_f} = 2 \tag{172}$$

と表される。そして因数同士の積 p_i は 2 のべき乗になる。

$$p_0 = 1 \tag{173}$$
$$p_1 = f_1 = 2 \tag{174}$$
$$p_2 = f_1 f_2 = 4 \tag{175}$$
$$\dots = \dots \tag{176}$$
$$p_i = f_1 f_2 \dots f_i = 2^i \tag{177}$$

このように表すと、基数 2 の時間間引き FFT アルゴリズムは以下のように表される。

bit-reverse ordering of g
for $i = 1 \dots n$ **do**
  for $a = 0 \dots p_{i-1} - 1$ **do**
    for $b = 0 \dots q_i - 1$ **do**
$$\begin{pmatrix} g(bp_i + a) \\ g(bp_i + p_{i-1} + a) \end{pmatrix} = \begin{pmatrix} g(bp_i + a) + W^a_{p_i} g(bp_i + p_{i-1} + a) \\ g(bp_i + a) - W^a_{p_i} g(bp_i + p_{i-1} + a) \end{pmatrix}$$
    end for
  end for
end for

ここでは、$p_i = 2\Delta$ であり、b の定義を前出の定義から変更し、$b \to bp_i$ としている。

元の再帰の式を見ると、共役対称性 (半複素対称性) をはっきりさせることによって各ステップでうまく変数を格納できるようにしている。ここでは、最初のパスで長さ 2 の実数 FFT への入力を以下のようにセットする。

$$g_1([b_0 b_1 b_2 a_0]) = \sum_{b_3} W_2^{a_0 b_3} g([b_0 b_1 b_2 b_3]) \tag{178}$$

ここで$FFT(x)_k = FFT(x)^*_{N-k}$という対称性を考慮すると、以下の条件が成り立つ。

$$
\begin{aligned}
g_1([b_0b_1b_20]) &= \text{real} &(179)\\
g_1([b_0b_1b_21]) &= \text{real'} &(180)
\end{aligned}
$$

次のパスでは、元データに対する長さ 4 の FFT への入力をセットする。

$$
\begin{aligned}
g_2([b_0b_1b_1a_0]) &= \sum_{b_2}\sum_{b_3} W_4^{[a_1a_0]b_2} W_2^{a_0b_3} g([b_0b_1b_2b_3]) &(181)\\
&= \sum_{b_2}\sum_{b_3} W_4^{[a_1a_0][b_3b_2]} g([b_0b_1b_2b_3]) &(182)
\end{aligned}
$$

ここでは、対称性により変換後のデータについていかが成り立つ。

$$
\begin{aligned}
g_2([b_0b_100]) &= \text{real} &(183)\\
g_2([b_0b_101]) &= x+iy &(184)\\
g_2([b_0b_110]) &= \text{real'} &(185)\\
g_2([b_0b_111]) &= x-iy &(186)
\end{aligned}
$$

上の式から、以下のように、i 番目のパスで元データの長さ 2^i の FFT をそれぞれ独立に計算しているのが分かる。

$$
g_i(bp_i+a) = \sum_{a'=0}^{p_i-1} W_{p_i}^{aa'} g(bp_i+a') \quad \text{for } b = 0\dots q_i-1 \qquad (187)
$$

つまり実数データに対しても共役対称性を利用して計算を進めることができるのがわかる。しかし、最後のパスだけは異なっている。i 段階のパスを終了したあと、データには以下のような対称性がある。

$$
\begin{aligned}
g_i(bp_i) &= \text{real} &(188)\\
g_i(bp_i+a) &= g_i(bp_i+p_i-a)^* \qquad a = 1\dots p_i/2-1 &(189)\\
g_i(bp_i+p_i/2) &= \text{real'} &(190)
\end{aligned}
$$

次の章では、上に示した時間間引きアルゴリズムにおける一般的な性質を改めて示す。周波数間引きアルゴリズムでは、このような簡潔な対称性が計算途中に現れないため、同様には計算できない。

$a > p_i/2$ での $g_i(bp_i+a)$ の値は、$a < p_i/2$ での値から計算できるため、複素数 FFT の場合に比べて、値を保持する変数は半分でよい。保持領域を半分にするのは難しくはなく、$a \le p_i/2$ についての項だけを扱うようにすればよいだけである。$a > p_i/2$ について $g_i(bp_i+a)$ を計算している箇所をその共役複素数 $g_i(bp_i+a')^*$ に置き換えればよい (ここで $a' = p_i - a$ である)。a、bp_i+a、$bp_i+p_{i-1}-a$ に対するバタフライ演算で二つの値が計算されるため、実質的には $a=0$ から $p_{i-1}/2$ までについて計算を行えばよい。アルゴリズムとしては以下のようになる。

for $a = 0\dots p_{i-1}/2$ **do**
  for $b = 0\dots q_i-1$ **do**

$$
\begin{pmatrix} g(bp_i+a) \\ g(bp_i+p_{i-1}-a)^* \end{pmatrix} = \begin{pmatrix} g(bp_i+a)+W_{p_i}^a g(bp_i+p_{i-1}+a) \\ g(bp_i+a)-W_{p_i}^a g(bp_i+p_{i-1}+a) \end{pmatrix}
$$

  end for
end for

上記で演算の数は半分になったが、必要とするメモリ領域の大きさも半減できる。上記のアルゴリズムは前出のものと同様、複素数配列 g を使っているが、入力として与えるデータを実数配列で受け取り、それを上書きして変換結果を返すようにできるとよい。

そのためには、複素数配列の実部、虚部の部分に自然な並びで実数を格納できるようなレイアウトを考え、添え字の決定がやたらと複雑にならないようにする必要がある。基数2のアルゴリズムでは、計算結果の一時的な保存場所を別途確保する必要はなかったが、これは二重ノード対の計算の際に変数を上書きする方法で行うからであった。

以下で、これらの条件を取り入れた方法を示す。つまり、i 段階目のパスで、$g(bp_i+a)$ の実部を bp_i+a の場所とする。虚部の場所は bp_i+p_i-a とする。複素共役の項 $g(bp_i+a)^* = g(bp_i+p_i-a)$ は冗長なので、計算、保持する必要はない。($a=0$ または $a=p_i/2$ のときは、虚部が 0 になるので、実部だけを保持しておけばよい)。

このやり方では、基数2のルーチン内で計算結果を保持するための変数などを確保しないので、入力に使われた変数の値を出力で置き換えることになる (in-place)。バタフライ演算において変数値が上書きされる様子を見てみるとき、非常に重要な点は、各パスの間には依存性があると言うことである。i 段階目のパスを計算する時、入力として $i-1$ 段階目のパスの出力を読み取る必要があり、またそのときに配列のどの要素が入力として想定されるベクトルのどの要素に対応するのかを考慮する必要がある。つまり $i-1$ 段階目のパスの出力が bp_i+a ではなく $bp_{i-1}+a$ として配列に格納されていて、共役対称性から $g_{i-1}(bp_{i-1}+a) = g_{i-1}(bp_{i-1}+p_{i-1}-a)^*$ となっている、ということである。これを考慮して実装するには、p_{i-1} についての繰り返し計算の右辺を g_{i-1} とする。たとえばつまり、$p_i = 2p_{i-1}$ であることから、$g_i(bp_i+a)$ の bp_i の代わりに $2bp_{i-1}$ とするということである。

バタフライ演算を $a=0$ について見てみると、

$$\begin{pmatrix} g(bp_i) \\ g(bp_i+p_{i-1})^* \end{pmatrix} = \begin{pmatrix} g(2bp_{i-1}) + g((2b+1)p_{i-1}) \\ g(2bp_{i-1}) - g((2b+1)p_{i-1}) \end{pmatrix} \tag{191}$$

$g_{i-1}(bp_{i-1}+a) = g_{i-1}(bp_{i-1}+p_{i-1}-a)^*$ なので、入力される値は全て実数である。入力の $g(2bp_{i-1})$ は $2bp_{i-1}$ から、$g((2b+1)p_{i-1})$ は $(2b+1)p_{i-1}$ から読まれる。つまり以下の表のようになる。

Term	Location		
$g(2bp_{i-1})$	real part	$2bp_{i-1}$	$=bp_i$
	imag part	—	
$g((2b+1)p_{i-1})$	real part	$(2b+1)p_{i-1}$	$=bp_i+p_{i-1}$
	imag part	—	
$g(bp_i)$	real part	bp_i	
	imag part	—	
$g(bp_i+p_{i-1})$	real part	bp_i+p_{i-1}	
	imag part	—	

出力される $g(bp_i+p_{i-1})^*$ は実数になるので、共役複素数は考えなくてよい。その実数値は入力値を上書きする形で bp_i および bp_i+p_{i-1} に書き込まれる。

$a = 1 \ldots p_{i-1}/2-1$ i ではバタフライ演算は以下のようになる。

$$\begin{pmatrix} g(bp_i+a) \\ g(bp_i+p_{i-1}-a)^* \end{pmatrix} = \begin{pmatrix} g(2bp_{i-1}+a) + W^a_{p_i} g((2b+1)p_{i-1}+a) \\ g(2bp_{i-1}+a) - W^a_{p_i} g((2b+1)p_{i-1}+a) \end{pmatrix} \tag{192}$$

ここでは、すべての項が複素数である。$a > p_i/2$ での $g(b'p_i + a')^*$ のような共役複素数は、実部を $b'p_i + a'$ に、虚部は打ち消した上で $b'p_i + p_i - a'$ に保存する。

つまりバタフライ演算は一般的には以下の表のようになる。

Term	Location		
$g(2bp_{i-1} + a)$	real part	$2bp_{i-1} + a$	$= bp_i + a$
	imag part	$2bp_{i-1} + p_{i-1} - a$	$= bp_i + p_{i-1} - a$
$g((2b+1)p_{i-1} + a)$	real part	$(2b+1)p_{i-1} + a$	$= bp_i + p_{i-1} + a$
	imag part	$(2b+1)p_{i-1} + p_{i-1} - a$	$= bp_i + p_i - a$
$g(bp_i + a)$	real part	$bp_i + a$	
	imag part	$bp_i + p_i - a$	
$g(bp_i + p_{i-1} - a)$	real part	$bp_i + p_{i-1} - a$	
	imag part	$bp_i + p_{i-1} + a$	

入力と出力の場所を見比べると、それぞれ置き換えられて上書きされていることがわかる。

最後の段階、$a = p_{i-1}/2$ でのバタフライ演算は以下のようになる。

$$\begin{pmatrix} g(bp_i + p_{i-1}/2) \\ g(bp_i + p_{i-1} - p_{i-1}/2)^* \end{pmatrix} = \begin{pmatrix} g(2bp_{i-1} + p_{i-1}/2) - ig((2b+1)p_{i-1} + p_{i-1}/2) \\ g(2bp_{i-1} + p_{i-1}/2) + ig((2b+1)p_{i-1} + p_{i-1}/2) \end{pmatrix} \tag{193}$$

ここで回転因子を $W^a_{p_i} = -i$ と置き換えている。これは以下のようにしてわかる。

$$W^{p_{i-1}/2}_{p_i} = \exp(-2\pi i p_{i-1}/2p_i) \tag{194}$$
$$= \exp(-2\pi i/4) \tag{195}$$
$$= -i \tag{196}$$

このバタフライ演算の２行目は、単に最初の複素数の共役 $p_{i-1} - p_{i-1}/2 = p_{i-1}/2$ を求めているだけである。そこで最初の行についてだけ考えてみると、これについての入力と出力は以下のようになる。

Term	Location		
$g(2bp_{i-1} + p_{i-1}/2)$	real part	$2bp_{i-1} + p_{i-1}/2$	$= bp_i + p_{i-1}/2$
	imag part	—	
$g((2b+1)p_{i-1} + p_{i-1}/2)$	real part	$(2b+1)p_{i-1} + p_{i-1}/2$	$= bp_i + p_i - p_{i-1}/2$
	imag part	—	
$g(bp_i + p_{i-1}/2)$	real part	$bp_i + p_{i-1}/2$	
	imag part	$bp_i + p_i - p_{i-1}/2$	

このバタフライ演算において入力と出力の保存される場所を見比べてみると、計算そのものは非常に簡潔であることが分かる。バタフライ演算により $bp_i + p_i - p_{i-1}/2$ で打ち消しが行われるが、他の部分は変更されない。演算対称の部分だけが上書きの対称になっている (in-place)。

以下に、基数２のアルゴリズムの全体を示す。$a = 0$、$a = 1 \ldots p_{i-1}/2 - 1$、$a = p_{i-1}/2$ の各部分がそれぞれ別のブロックになっている。

bit-reverse ordering of g
for $i = 1 \ldots n$ **do**
  for $b = 0 \ldots q_i - 1$ **do**
    $$\begin{pmatrix} g(bp_i) \\ g(bp_i + p_{i-1}) \end{pmatrix} \Leftarrow \begin{pmatrix} g(bp_i) + g(bp_i + p_{i-1}) \\ g(bp_i) - g(bp_i + p_{i-1}) \end{pmatrix}$$
  end for
  for $a = 1 \ldots p_{i-1}/2 - 1$ **do**
    for $b = 0 \ldots q_i - 1$ **do**
      $(\mathrm{Re}\, z_0, \mathrm{Im}\, z_0) \Leftarrow (g(bp_i + a), g(bp_i + p_{i-1} - a))$
      $(\mathrm{Re}\, z_1, \mathrm{Im}\, z_1) \Leftarrow (g(bp_i + p_{i-1} + a), g(bp_i + p_i - a))$
      $t_0 \Leftarrow z_0 + W_{p_i}^a z_1$
      $t_1 \Leftarrow z_0 - W_{p_i}^a z_1$
      $(g(bp_i + a), g(bp_i + p_i - a) \Leftarrow (\mathrm{Re}\, t_0, \mathrm{Im}\, t_0)$
      $(g(bp_i + p_{i-1} - a), g(bp_i + p_{i-1} + a)) \Leftarrow (\mathrm{Re}\, t_1, -\mathrm{Im}\, t_1)$
    end for
  end for
  for $b = 0 \ldots q_i - 1$ **do**
    $g(bp_i - p_{i-1}/2) \Leftarrow -g(bp_i - p_{i-1}/2)$
  end for
end for

a についてのループを $a = 0$、$a = 1 \ldots p_{i-1}/2 - 1$、$a = p_{i-1}/2$ の三つの場合に分けることで実行速度の向上をはかっている。$a = 0$ の場合は $W_{p_i}^a = 1$ なので、複素数の乗算を行うための b についてのループが必要なくなる。$a = p_{i-1}/2$ の場合は $W_{p_i}^a = -i$ なので、やはりループを回す必要はない。

7.1.1 逆変換の計算

複素数データの逆 FFT (inverse FFT) は、DFT 行列の複素共役を使うことで簡単に計算できる。入力と出力はそれぞれ複素数であり、なんら特別な対称性はない。実数データの逆 FFT は、FFT の出力に共役対称性 (半複素対称性) があり、入力となった実数データには対称性は一般にはなくてもよいことから、複素数の逆 FFT よりも複雑である。

逆変換は、順方向変換を逆にたどることで可能である。たどりやすくするために、順方向変換のバタフライ演算を行列演算の形で書くと、以下のようになる。

for $i = 1 \ldots n$ **do**
  for $a = 0 \ldots p_{i-1}/2$ **do**
    for $b = 0 \ldots q_i - 1$ **do**
      $$\begin{pmatrix} g(bp_i + a) \\ g(bp_i + p_{i-1} + a) \end{pmatrix} = \begin{pmatrix} 1 & W_{p_i}^a \\ 1 & -W_{p_i}^a \end{pmatrix} \begin{pmatrix} g(2bp_{i-1} + a) \\ g((2b+1)p_{i-1} + a) \end{pmatrix}$$
    end for
  end for
end for

これを逆にたどるには、繰り返し計算の順番を逆にし、もっとも内側のループでの行列の乗算を、逆行列の乗算にすればよい。

for $i = n \dots 1$ **do**
  for $a = 0 \dots p_{i-1}/2$ **do**
    for $b = 0 \dots q_i - 1$ **do**
$$\begin{pmatrix} g(2bp_{i-1}+a) \\ g((2b+1)p_{i-1}+a) \end{pmatrix} = \begin{pmatrix} 1 & W_{p_i}^a \\ 1 & -W_{p_i}^a \end{pmatrix}^{-1} \begin{pmatrix} g(bp_i+a) \\ g(bp_i+p_{i-1}+a) \end{pmatrix}$$
    end for
  end for
end for

a および b のループは、それぞれお互いに独立しているので、内側と外側を入れ替える必要はない。上記の逆行列は以下のようになる。

$$\begin{pmatrix} 1 & W_{p_i}^a \\ 1 & -W_{p_i}^a \end{pmatrix}^{-1} = \frac{1}{2} \begin{pmatrix} 1 & 1 \\ W_{p_i}^{-a} & -W_{p_i}^{-a} \end{pmatrix} \tag{197}$$

除算を少しでも減らすために、上のアルゴリズム中では逆行列にかかっている係数1/2 を無視している。これにより逆変換により得られるデータは、正規化しないと元のデータと一致しないことになる。正規化は、逆変換により得られるデータを $N = 2^n$ で除すことにより行われる。

以下に基数 2 の半複素データを実数データに逆 FFT するアルゴリズムを示す。基数 2 の場合の入力を出力で上書きするやり方である。

for $i = n \dots 1$ **do**
  for $b = 0 \dots q_i - 1$ **do**
$$\begin{pmatrix} g(bp_i) \\ g(bp_i+p_{i-1}) \end{pmatrix} \Leftarrow \begin{pmatrix} g(bp_i) + g(bp_i+p_{i-1}) \\ g(bp_i) - g(bp_i+p_{i-1}) \end{pmatrix}$$
  end for
  for $a = 1 \dots p_{i-1}/2 - 1$ **do**
    for $b = 0 \dots q_i - 1$ **do**
      $(\mathrm{Re}\, z_0, \mathrm{Im}\, z_0) \Leftarrow (g(bp_i+a), g(bp_i+p_i-a))$
      $(\mathrm{Re}\, z_1, \mathrm{Im}\, z_1) \Leftarrow (g(bp_i+p_{i-1}-a), -g(bp_i+p_{i-1}+a))$
      $t_0 \Leftarrow z_0 + z_1$
      $t_1 \Leftarrow z_0 - z_1$
      $(g(bp_i+a), g(bp_i+p_{i-1}-a)) \Leftarrow (\mathrm{Re}\, t_0, \mathrm{Im}\, t_0)$
      $(g(bp_i+p_{i-1}+a), g(bp_i+p_i-a)) \Leftarrow (\mathrm{Re}(W_{p_i}^a t_1), \mathrm{Im}(W_{p_i}^a t_1))$
    end for
  end for
  for $b = 0 \dots q_i - 1$ **do**
    $g(bp_i+p_{i-1}/2) \Leftarrow 2g(bp_i+p_{i-1}/2)$
    $g(bp_i+p_{i-1}+p_{i-1}/2) \Leftarrow -2g(bp_i+p_{i-1}+p_{i-1}/2)$
  end for
end for
bit-reverse ordering of g

7.2 実数データに対する混合基数 FFT

前述のように基数 2 の時間間引きアルゴリズムは、各段階のパスが元データの一部 (一部分の時間に対応するデータ) だけについてのフーリエ変換であった。これは一般の、複数の基数に対応するよう拡張できる。やはり前述の、混合基数の複素数 FFT は周波数間引き法であったが、そこで用

いた DFT 行列を転置させることで、以下のように時間間引き法を導出できる。

$$\begin{aligned} W_N &= W_N^T && (198)\\ &= (T_{n_f} \dots T_2 T_1)^T && (199)\\ &= T_1^T T_2^T \dots T_{n_f}^T && (200) \end{aligned}$$

ここで

$$\begin{aligned} T_i^T &= \left((P_{q_i}^{f_i} D_{q_i}^{f_i} \otimes I_{p_{i-1}})(W_{f_i} \otimes I_{m_i})\right)^T && (201)\\ &= (W_{f_i} \otimes I_{m_i})(D_{q_i}^{f_i} (P_{q_i}^{f_i})^T \otimes I_{p_{i-1}}). && (202) \end{aligned}$$

である。W、D、I はそれぞれ対称行列であり、置換行列 P は

$$(P_b^a)^T = P_a^b. \quad (203)$$

という性質を持っている。また D と P の定義から、以下の等式が成り立つことが分かる。

$$D_b^a P_a^b = P_a^b D_a^b. \quad (204)$$

これにより T_i^T は以下のように書き表される。

$$T_i^T = (W_{f_i} \otimes I_{m_i})(P_{f_i}^{q_i} D_{f_i}^{q_i} \otimes I_{p_{i-1}}). \quad (205)$$

この転置行列 T^T を DFT 行列 D の前に逆順置換行列 P に作用させれば時間間引き法になる。この転置行列を作用させる時、T_{n_f} が最初に、T_1 が最後に作用することになり、因数の並び順を逆にする演算となる。ここで、因数の順序を $f_1 \leftrightarrow f_{n_f}$、$f_2 \leftrightarrow f_{n_f-1}$、$\dots$ の用に逆順にし、その逆順にする演算を行うための置換 $p_{i-1} \leftrightarrow q_i$ を考える。この置換により周波数間引きアルゴリズムと同じ順序の時間間引きアルゴリズムを得ることができる。

$$W_N = T_{n_f} \dots T_2 T_1 \quad (206)$$

$$T_i = (W_{f_i} \otimes I_{m_i})(P_{f_i}^{p_{i-1}} D_{f_i}^{p_{i-1}} \otimes I_{q_i}) \quad (207)$$

ここで p_i、q_i、m_i はどれも前出の時と同じものであり、

$$\begin{aligned} p_i &= f_1 f_2 \dots f_i \quad (p_0 = 1) && (208)\\ q_i &= N/p_i && (209)\\ m_i &= N/f_i && (210) \end{aligned}$$

である。ここで、繰り返し計算 $x = Wz = T_{n_f} \dots T_2 T_1 z$ に「間引き」という性質があることを示す。ある段階のパス $v^{(i)}$ を

$$\begin{aligned} v^{(0)} &= z && (211)\\ v^{(1)} &= T_1 v^{(0)} && (212)\\ v^{(2)} &= T_2 v^{(1)} && (213)\\ \dots &= \dots && (214)\\ v^{(i)} &= T_i v^{(i-1)} && (215) \end{aligned}$$

と書くと、パスの演算結果 $v^{(i)}$ はどの段階のパスでも

$$v^{(i)} = (W_{p_i} \otimes I_{q_i}) z \quad (216)$$

と表すことができ、各段階において、長さがそれぞれ p_i の「間引いた」フーリエ変換を各 q_i について行うことになる。これを示すには、まず $v^{(i-1)}$ が正しい演算結果であると仮定して、

$$v^{(i-1)} = (W_{p_{i-1}} \otimes I_{q_{i-1}})z \qquad \text{(assumption)} \tag{217}$$

以下の等式が成り立つと仮定して、次の繰り返し計算を見てみればよい。

$$\begin{aligned} v^{(i)} &= T_i v^{(i-1)} && (218) \\ &= T_i (W_{p_{i-1}} \otimes I_{q_{i-1}})z && (219) \\ &= (W_{f_i} \otimes I_{m_i})(P_{f_i}^{p_{i-1}} D_{f_i}^{p_{i-1}} \otimes I_{q_i})(W_{p_{i-1}} \otimes I_{q_{i-1}})z && (220) \end{aligned}$$

$m_i = p_{i-1}q_i$ という関係から I_{m_i} は $I_{p_{i-1}q_i}$、$I_{q_{i-1}}$ は $I_{f_i q_i}$ と書くことができる。それに加えて単位行列についての以下の等式

$$I_{ab} = I_a \otimes I_b \tag{221}$$

を使うと、$v^{(i)}$ は以下の式で表される。

$$v^{(i)} = (((W_{f_i} \otimes I_{p_{i-1}})(P_{f_i}^{p_{i-1}} D_{f_i}^{p_{i-1}})(W_{p_{i-1}} \otimes I_{f_i})) \otimes I_{q_i})z. \tag{222}$$

最初の行列積は以下のように、$a = p_{i-1}$ および $b = f_i$ について W_{ab} を展開したものである。

$$W_{p_{i-1}f_i} = ((W_{f_i} \otimes I_{p_{i-1}})(P_{f_i}^{p_{i-1}} D_{f_i}^{p_{i-1}})(W_{p_{i-1}} \otimes I_{f_i})). \tag{223}$$

ここで $p_i = p_{i-1}f_i$ であることから、

$$v^{(i)} = (W_{p_i} \otimes I_{q_i})z \tag{224}$$

が導かれる。これがこのパスの演算結果である。$i = 1$ の場合はこの等式が成り立つことははっきりと示され、そこから帰納的にすべての i について正しいことが示される。基数 2 のアルゴリズムと同様に、元のデータ z が実数であれば変換結果は共役対称性を持つ (クロネッカー積の部分空間が適切に作用するため)。つまり、各パスでの変換結果および最終的な変換結果を保持するためには which is the desired result. The case $i = 1$ can be verified 長さ N の実数配列があればよいことになる。

7.3 実装

実数データに対する混合基数 FFT の実装は、複素数データの場合と同様だが、周波数間引き法ではなく時間間引き法を実装する場合は、いくつかのステップの順序を逆にせねばならない。ここでは DFT 行列 W_N がすでに入力ベクトル z のデータ長の因数に分解されているとする。

$$\begin{aligned} x &= W_N z && (225) \\ &= T_{n_f} \ldots T_2 T_1 z && (226) \end{aligned}$$

各因数 n_f についてループを回し、それぞれに対応する行列 x T_i を入力ベクトルに作用させることで、全体の変換を作っていく。

for $(i = 1 \ldots n_f)$ **do**
  $v \Leftarrow T_i v$
end for

ここでは、転置行列を以下のように考えたので、T_i の定義は前出のとは違っている。

$$T_i = (W_{f_i} \otimes I_{m_i})(P_{f_i}^{p_{i-1}} D_{f_i}^{p_{i-1}} \otimes I_{q_i}). \tag{227}$$

一時的に計算結果を保持するベクトル t を考え、もっとも右の行列を作用させた結果をこれに保持するとする。

$$t = (P_{f_i}^{p_{i-1}} D_{f_i}^{p_{i-1}} \otimes I_{q_i})v \tag{228}$$

これを、前出のように各要素について展開すると、以下のように簡略化される。

$$t_{aq+b} = \sum_{a'b'} (P_{f_i}^{p_{i-1}} D_{f_i}^{p_{i-1}} \otimes I_{q_i})_{(aq+b)(a'q+b')} v_{a'q+b'} \tag{229}$$

$$= \sum_{a'} (P_{f_i}^{p_{i-1}} D_{f_i}^{p_{i-1}})_{aa'} v_{a'q+b} \tag{230}$$

ここでは、添え字を $aq+b$ の形に分解して書いている。$0 \leq a < p_i$ かつ $0 \leq b < q$ である。周波数間引き法では添え字 a を分割することえで行列の乗算を省くことができた。ここでは a を $\mu f + \lambda$ $(0 \leq \mu < p_{i-1}$ かつ $0 \leq \lambda < f)$ と表さねばならない。

$$t_{(\mu f+\lambda)q+b} = \sum_{\mu'\lambda'} (P_{f_i}^{p_{i-1}} D_{f_i}^{p_{i-1}})_{(\mu f+\lambda)(\mu' f+\lambda')} v_{(\mu' f+\lambda')q+b} \tag{231}$$

$$= \sum_{\mu'\lambda'} (P_{f_i}^{p_{i-1}})_{(\mu f+\lambda)(\mu' f+\lambda')} \omega_{p_i}^{\mu'\lambda'} v_{(\mu' f+\lambda')q+b} \tag{232}$$

行列 $P_{f_i}^{p_{i-1}}$ を作用させると添え字 $(\mu f+\lambda)q+b$ が $(\lambda p_{i-1}+\mu)q+b$ に置き換わり、以下の式を得る。

$$t_{(\lambda p_{i-1}+\mu)q+b} = w_{p_i}^{\mu\lambda} v_{(\mu f+\lambda)q+b} \tag{233}$$

次のステージ

$$v' = (W_{f_i} \otimes I_{m_i})t, \tag{234}$$

を計算するには、一時的にクロネッカー積において添え字 t を m_i 個の独立した DFT に分割せねばならない。

$$v'_{(\lambda p_{i-1}+\mu)q+b} = \sum_{\lambda'\mu'b'} (W_{f_i} \otimes I_{m_i})_{((\lambda p_{i-1}+\mu)q+b)((\lambda' p_{i-1}+\mu')q+b')} t_{(\lambda' p_{i-1}+\mu')q+b'} \tag{235}$$

ここで $m = p_{i-1}q$ という関係から、t を以下のように書き換える。

$$t_{(\lambda p_{i-1}+\mu)q+b} = t_{\lambda m+(\mu q+b)} \tag{236}$$

クロネッカー積は以下のように分割できる。

$$v'_{(\lambda p_{i-1}+\mu)q+b} = \sum_{\lambda'\mu'b'} (W_{f_i} \otimes I_{m_i})_{((\lambda p_{i-1}+\mu)q+b)((\lambda' p_{i-1}+\mu')q+b')} t_{(\lambda' p_{i-1}+\mu')q+b'} \tag{237}$$

$$= \sum_{\lambda'} (W_{f_i})_{\lambda\lambda'} t_{\lambda' m_i+(\mu q+b)} \tag{238}$$

ここで添え字の表記を元に戻すと、最後の行は以下のようになる。

$$= \sum_{\lambda'} (W_{f_i})_{\lambda\lambda'} t_{(\lambda p_{i-1}+\mu)q+b} \tag{239}$$

これに前出の t の式を代入すると、

$$v'_{(\lambda p_{i-1}+\mu)q+b} = \sum_{\lambda'} (W_{f_i})_{\lambda\lambda'} w_{p_i}^{\mu\lambda'} v_{(\mu f+\lambda')q+b} \tag{240}$$

となる。これにより、複素数データに対する基本的な時間間引きの混合基数 FFT を、実数データに特化させることができる。

for $i = 1 \ldots n_f$ **do**
  for $\mu = 0 \ldots p_{i-1} - 1$ **do**
    for $b = 0 \ldots q - 1$ **do**
      for $\lambda = 0 \ldots f - 1$ **do**
        $t_\lambda \Leftarrow \omega_{p_i}^{\mu\lambda'} v_{(\mu f+\lambda')q+b}$
      end for
      for $\lambda = 0 \ldots f - 1$ **do**
        $v'_{(\lambda p_{i-1}+\mu)q+b} = \sum_{\lambda'=0}^{f-1} W_f(\lambda, \lambda') t_{\lambda'}$ {DFT matrix-multiply module}
      end for
    end for
  end for
  $v \Leftarrow v'$
end for

ここまでの展開で、変数の保持方法を変えることで複素変数の FFT アルゴリズムを実数用のアルゴリズムに書き換える用意ができた。ここでは FFTPACK での方法を採用する。FFTPACK では、基数の異なる各 FFT は飛び石のようになっているのではなく連続しており、実部と虚部が隣接した場所に入れられている。基数２の場合、これは都合が良い。

クロネッカー積 $W_p \otimes I_q$ から、FFT の各ステップでの変数の配置が決まる。各 FFT は、各ステップでのクロネッカー積をうまく並べ替えることで、隣接させることができる。

$$W_p \otimes I_q \Rightarrow I_q \otimes W_p \tag{241}$$

これは添え字のアクセスの仕方を少しか得ることで実装できる。i 番目のパスで要素 v_{aq_i+b} を v_{bp_i+a} の場所にいれ、データを読み出す時にそれを考慮して読み出せばよい。この方法なら、添え字のアクセスの仕方だけを変えればすむ。0 番目のステップでは $p_0 = 1$ なので、何も変わらず、また最後のステップでも $q_{n_f} = 1$ なので同様である。これを反映すると、上述のアルゴリズムは以下のようになる。

for $i = 1 \ldots n_f$ **do**
  for $\mu = 0 \ldots p_{i-1} - 1$ **do**
    for $b = 0 \ldots q - 1$ **do**
      for $\lambda = 0 \ldots f - 1$ **do**
        $t_\lambda \Leftarrow \omega_{p_i}^{\mu\lambda'} v_{(\lambda' q+b)p_{i-1}+\mu}$
      end for
      for $\lambda = 0 \ldots f - 1$ **do**
        $v'_{bp+(\lambda p_{i-1}+\mu)} = \sum_{\lambda'=0}^{f-1} W_f(\lambda, \lambda') t_{\lambda'}$ {DFT matrix-multiply module}
      end for
    end for
  end for
  $v \Leftarrow v'$
end for

入力の項の添え字を $aq_{i-1} + b$ と書くことでアクセスの順序を変更している。これにより各パスでの依存関係は以下のようになる。

$$v_{(\mu f+\lambda')q+b} = v_{\mu q_{i-1}+\lambda' q+b} \tag{242}$$

ここで $q_{i-1} = fq$ であることを利用している。b は 0 から $q - 1$ まで、λ' は 0 から $f - 1$ なので、$\lambda' q + b$ は 0 から $q_{i-1} - 1$ までである。したがって入力データの配置は以下のようになる。

$$v_{\mu q_{i-1}+\lambda' q+b} \Rightarrow v_{(\lambda' q+b)p_{i-1}+\mu} \tag{243}$$

FFTPACK のやり方では、各ステップの出力データは基数 2 の場合と同様、以下のような共役対称性を持っている。

$$v^{(i)}_{bp+a} = v^{(i)*}_{bp+(p-a)} \tag{244}$$

また前のステップからの入力データには以下のような対称性がある。

$$v^{(i-1)}_{(\lambda' q+b)p_{i-1}+\mu} = v^{(i-1)*}_{(\lambda' q+b)p_{i-1}+(p_{i-1}-\mu)} \tag{245}$$

この二種類の対称性を利用することで、各ステップでの変数領域のサイズを半減することができ、また計算量も半減できる。また入力を出力で置き換えるわけではないので、基数 2 の場合よりもやり方に自由が利く。FFTPACK での変数配置では、変数の実部と虚部を隣接させているが、値が 0 であることが分かっている虚部は保持しなくてもよい。すると変数は結局、以下のように配置すればよい。

Term	Location		
$g(bp_i)$	real part	bp_i	
	imag part	—	
$g(bp_i + a)$	real part	$bp_i + 2a - 1$	for $a = 1 \ldots p_i/2 - 1$
	imag part	$bp_i + 2a$	
$g(bp_i + p_i/2)$	real part	$bp_i + p_i - 1$	if p_i is even
	imag part	—	

$a = 0$ での実数は bp に保持される。$a = 1 \ldots p/2 - 1$ の実部は $bp + 2a - 1$ に、虚部は $bp + 2a$ に保持される。p が偶数の時は $a = p/2$ は実数になり $bp + p - 1$ に保持される。0 である虚部はどこにも保持されない。

基本的な演算は以下である。

$$v^{(i)}_{bp+(\lambda p_{i-1}+\mu)} = \sum_{\lambda'} W_f^{\lambda\lambda'} \omega_{p_i}^{\mu\lambda'} v^{(i-1)}_{(\lambda' q+b)p_{i-1}+\mu} \tag{246}$$

これを計算する時、共役対称性で表される $a > p_i/2$ の項は、基数 2 のアルゴリズムでは無視していた。これらの項を保持されるはずの値は、$a' = p - a$ などとして共役複素数を考えればよい。同様に $\mu > p_{i-1}/2$ の項も $\mu' = p_{i-1} - \mu$ などとして考えればよい。

各ステージでの入力は共役対称性を持っているため、$\mu = 0 \ldots p_{i-1}/2$ の範囲だけを考えればよい。各パスでの演算も、$\mu = 0 \ldots p_{i-1}/2$ についてだけ考えればよい。

最初のパスでは $\mu = 0$ であり、$W_{p_{i-1}} \otimes I_{q_{i-1}}$ の 0 番目の要素にアクセスすることになる。この要素は虚部が 0 で、実数である。$\mu = 0$ のパスは、入力が実数のみで、出力が共役対称性のある複素数列として実装する。$\mu = 0$ のとき回転因子 $\omega_{p_i}^{\mu\lambda'}$ はすべて 1 であり、計算を省略できる。小さな N に対する実数 DFT は、複素数の場合のプログラムから不要な部分を取り除けばよい。たとえば $N = 3$ の場合は以下のようになる。

$$t_1 = z_1 + z_2, \quad t_2 = z_0 - t_1/2, \quad t_3 = \sin(\pi/3)(z_1 - z_2), \tag{247}$$

$$x_0 = z_0 + t_1, \quad x_1 = t_2 + it_3, \quad x_2 = t_2 - it_3. \tag{248}$$

入力データ z_0、z_1、z_2 が複素数の場合には演算は全て複素数で行われる。実数の場合には z_0、z_1、z_2 はどれも実数であり、したがって t_1、t_2、t_3 も実数になり、$x_1 = t_1 + it_2 = x_2^*$ という対称性のあることを考えると x_1 を計算すれば x_2 を計算する必要がないことが分かる。

続くパス $\mu = 1 \ldots p_{i-1}/2 - 1$ では入力は複素数であり、複素数データのためのアルゴリズムと同じ方法で DFT を計算せねばならない。入力は $v_{(\lambda q+b)p_{i-1}+\mu}$ の形で配置されており $(\mu < p_{i-1}/2)$、したがって $\mu > p_{i-1}/2$ の共役複素数の要素にアクセスする必要はない。

p_{i-1} が偶数の時は、データの中間点 $\mu = p_{i-1}/2$ が実数になる。$\mu = p_{i-1}/2$ かつ $\mu = p_{i-1} - \mu$ だからである。この場合は $\mu = 0$ のときと同様、実数入力の実数 DFT を用いることができる。しかし回転因子は 1 ではなく、以下のようになる。

$$\begin{aligned} \omega_{p_i}^{\mu\lambda'} &= \omega_{p_i}^{(p_{i-1}/2)\lambda'} && (249) \\ &= \exp(-i\pi\lambda'/f_i) && (250) \end{aligned}$$

これにより、出力の対称性を操作するステップが新たに必要になる。$\mu = 0$ のときの共役対称性の代わりに、以下で示される、ずれのある共役対称性がある。

$$t_\lambda = t^*_{f-(\lambda+1)} \quad (251)$$

このことは以下で容易に示される。

$$\begin{aligned} t_\lambda &= \sum e^{-2\pi i\lambda\lambda'/f_i} e^{-i\pi\lambda'/f_i} r_{\lambda'} && (252) \\ t_{f-(\lambda+1)} &= \sum e^{-2\pi i(f-\lambda-1)\lambda'/f_i} e^{-i\pi\lambda'/f_i} r_{\lambda'} && (253) \\ &= \sum e^{2\pi i\lambda\lambda'/f_i} e^{i\pi\lambda'/f_i} r_{\lambda'} && (254) \\ &= t^*_\lambda && (255) \end{aligned}$$

この対称性にはズレがあるが、出力のうち半分だけを計算すればよく、残りは共役対称性から計算できることには変わりはない。$f = 4$ の場合、出力は $(x_0 + iy_0, x_1 + iy_1, x_1 - iy_1, x_0 - iy_0)$ に、$f = 5$ なら $(x_0 + iy_0, x_1 + iy_1, x_2, x_1 - iy_1, x_0 - iy_0)$ になる。回転因子と DFT 行列の積を考えることで、この 2 つを同時に作用させるルーチンを書くことができる。複素数データに対する DFT プログラムから回転因子の部分を取り出して修正すればよい。x が実数、z が共役対称性を持つ場合に $z = W\Omega x$ に対してテンパートンが導出した方法を以下に示す。回転因子の行列 Ω は以下のようになる。

$$\Omega = \mathrm{diag}(1, e^{-i\pi/f}, e^{-2\pi i/f}, \ldots, e^{-i\pi(f-1)/f}) \quad (256)$$

ここで z を $z = a + ib$ とし、二つの実数からなるベクトルと考える。すると、以下の結果が得られる。$N = 2$ の場合、

$$a_0 = x_0, \quad b_0 = -x_1. \quad (257)$$

$N = 3$ の場合、

$$t_1 = x_1 - x_2, \quad (258)$$

$$a_0 = x_0 + t_1/2, \quad b_0 = x_0 - t_1, \quad (259)$$

$$a_1 = -\sin(\pi/3)(x_1 + x_2) \quad (260)$$

$N = 4$ の場合、

$$t_1 = (x_1 - x_3)/\sqrt{2}, \quad t_2 = (x_1 + x_3)/\sqrt{2}, \quad (261)$$

$$a_0 = x_0 + t_1, \quad b_0 = -x_2 - t_2, \quad (262)$$

$$a_1 = x_0 - t_1, \quad b_1 = x_2 - t_2. \quad (263)$$

$N=5$ の場合、

$$t_1 = x_1 - x_4, \quad t_2 = x_1 + x_4, \tag{264}$$

$$t_3 = x_2 - x_3, \quad t_4 = x_2 + x_3, \tag{265}$$

$$t_5 = t_1 - t_3, \quad t_6 = x_0 + t_5/4, \tag{266}$$

$$t_7 = (\sqrt{5}/4)(t_1 + t_3) \tag{267}$$

$$a_0 = t_6 + t_7, \quad b_0 = -\sin(2\pi/10)t_2 - \sin(2\pi/5)t_4, \tag{268}$$

$$a_1 = t_6 - t_7, \quad b_1 = -\sin(2\pi/5)t_2 + \sin(2\pi/10)t_4, \tag{269}$$

$$a_2 = x_0 - t_5 \tag{270}$$

$N=6$ の場合、

$$t_1 = \sin(\pi/3)(x_5 - x_1), \quad t_2 = \sin(\pi/3)(x_2 + x_4), \tag{271}$$

$$t_3 = x_2 - x_4, \quad t_4 = x_1 + x_5, \tag{272}$$

$$t_5 = x_0 + t_3/2, \quad t_6 = -x_3 - t_4/2, \tag{273}$$

$$a_0 = t_5 - t_1, \quad b_0 = t_6 - t_2, \tag{274}$$

$$a_1 = x_0 - t_3, \quad b_1 = x_3 - t_4, \tag{275}$$

$$a_2 = t_5 + t_1, \quad b_2 = t_6 + t_2 \tag{276}$$

8 実数データに対する混合基数の逆 FFT

混合基数の場合に実数データの逆 FFT を行うには、順方向変換のアルゴリズムを逆にたどればよい。

以下に FFTPACK での変数配置のやり方を使ったアルゴリズムを示す。

for $i = n_f \ldots 1$ **do**
  for $\mu = 0 \ldots p_{i-1} - 1$ **do**
    for $b = 0 \ldots q - 1$ **do**
      for $\lambda = 0 \ldots f - 1$ **do**
        $t_{\lambda'} = \sum_{\lambda'=0}^{f-1} W_f(\lambda, \lambda') v_{bp+(\lambda p_{i-1}+\mu)}$ {DFT matrix-multiply module}
      end for
      for $\lambda = 0 \ldots f - 1$ **do**
        $v'_{(\lambda' q+b)p_{i-1}+\mu} \Leftarrow \omega_{p_i}^{-\mu\lambda'} t_\lambda$
      end for
    end for
  end for
  $v \Leftarrow v'$
end for

$\mu = 0$ のとき、半複素対称性のあるデータに対する逆 DFT を行い、実数の出力を得る。回転因子はこのときすべて 1 である。このときには順方向変換のときと同様に、実数用のルーチンが使える。まず最初は複素数データのルーチンを使い、無駄な項を省く。この場合、最終的には虚部が 0 になり、半複素対称性から無駄な項を省くことができる。

$\mu = 1 \ldots p_{i-1}/2 - 1$ に対しては、複素数データに対する逆変換を行わねばならないため、簡略化できることはない。

$\mu = p_{i-1}/2$ の場合 (p_{i-1} が偶数の時のみ) はズレのある半複素対称性を持つデータに対して回転因子と DFT 行列を同時に作用させるアルゴリズムを上述した。これの出力は実数である。これについては、順方向変換で示した数式を逆にたどればよい。ここでは、テンパートンが示したアルゴリズム [21] を、いくつかの項の表記を若干修正して示す。

$N = 2$ の場合、

$$x_0 = 2a_0, \quad x_1 = -2b_0. \tag{277}$$

$N = 3$ の場合、

$$t_1 = a_0 - a_1, \quad t_2 = \sqrt{3}b_1, \tag{278}$$

$$x_0 = 2a_0 + a_1, \quad x_1 = t_1 - t_2, \quad x_2 = -t_1 - t_2 \tag{279}$$

$N = 4$ の場合、

$$t_1 = \sqrt{2}(b_0 + b_1), \quad t_2 = \sqrt{2}(a_0 - a_1), \tag{280}$$

$$x_0 = 2(a_0 + a_1), \quad x_1 = t_2 - t_1, \tag{281}$$

$$x_2 = 2(b_1 - b_0), \quad x_3 = -(t_2 + t_1). \tag{282}$$

$N = 5$ の場合、

$$t_1 = 2(a_0 + a_1), \quad t_2 = t_1/4 - a_2, \tag{283}$$

$$t_3 = (\sqrt{5}/2)(a_0 - a_1), \tag{284}$$

$$t_4 = 2(\sin(2\pi/10)b_0 + \sin(2\pi/5)b_1), \tag{285}$$

$$t_5 = 2(\sin(2\pi/10)b_0 - \sin(2\pi/5)b_1), \tag{286}$$

$$t_6 = t_3 + t_2, \quad t_7 = t_3 - t_2, \tag{287}$$

$$x_0 = t_1 + a_2, \quad x_1 = t_6 - t_4, \tag{288}$$

$$x_2 = t_7 - t_5, \quad x_3 = -t_7 - t_5, \tag{289}$$

$$x_4 = -t_6 - t_4. \tag{290}$$

9 Conclusions

ここでは一次元の基数 2 および混合基数の FFT について述べた。ここで示した単純な基数 2 のアルゴリズムよりも計算の効率がよい、基数分割 FFT アルゴリズムについても学ぶとよりよいだろう。また次元数が 2 以上の FFT についても学ぶべきである。

References

[1] P. Duhamel and M. Vetterli. Fast fourier transforms: A tutorial review and a state of the art. *Signal Processing*, 19:259–299, 1990.

[2] E. Oran Brigham. *The Fast Fourier Transform.* Prentice Hall, 1974.

[3] C. S. Burrus and T. W. Parks. *DFT/FFT and Convolution Algorithms.* Wiley, 1984.

[4] Digital Signal Processing Committee and IEEE Acoustics, Speech, and Signal Processing Committee, editors. *Programs for Digital Signal Processing.* IEEE Press, 1979.

[5] Winthrop W. Smith and Joanne M. Smith. *Handbook of Real-Time Fast Fourier Transforms.* IEEE Press, 1995.

[6] Douglas F. Elliott and K. Ramamohan Rao. *Fast transforms: algorithms, analyses, applications.* Academic Press, 1982. This book does not contain actual code, but covers the more advanced mathematics and number theory needed in the derivation of fast transforms.

[7] Richard E. Blahut. *Fast Algorithms for Digital Signal Processing.* Addison-Wesley, 1984.

[8] James H. McClellan and Charles M. Rader. *Number Theory in Digital Signal Processing.* Prentice-Hall, 1979.

[9] C. S. Burrus. Notes on the FFT. Available from http://www-dsp.rice.edu/res/fft/fftnote.asc.

[10] Henrik V. Sorenson, Michael T. Heideman, and C. Sidney Burrus. On computing the split-radix FFT. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-34(1):152–156, 1986.

[11] Pierre Duhamel. Implementation of split-radix FFT algorithms for complex, real, and real-symmetric data. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-34(2):285–295, 1986.

[12] Clive Temperton. A note on prime factor FFT algorithms. *Journal of Computational Physics*, 58:198–204, 1983.

[13] Clive Temperton. Implementation of a self-sorting in-place prime factor FFT algorithm. *Journal of Computational Physics*, 58:283–299, 1985.

[14] Charles M. Rader. Discrete fourier transform when the number of data samples is prime. *IEEE Proceedings*, 56(6):1107–1108, 1968.

[15] Mehalic, Rustan, and Route. Effects of architecture implementation on DFT algorithm performance. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASP-33:684–693, 1985.

[16] P. B. Visscher. The FFT: Fourier transforming one bit at a time. *Computers in Physics*, 10(5):438–443, Sep/Oct 1996.

[17] Jeffrey J. Rodriguez. An improved FFT digit-reversal algorithm. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(8):1298–1300, 1989.

[18] Petr Rösel. Timing of some bit reversal algorithms. *Signal Processing*, 18:425–433, 1989.

[19] Clive Temperton. Self-sorting mixed-radix fast fourier transforms. *Journal of Computational Physics*, 52(1):1–23, 1983.

[20] Richard C. Singleton. An algorithm for computing the mixed radix fast fourier transform. *IEEE Transactions on Audio and Electroacoustics*, AU-17(2):93–103, June 1969.

[21] Clive Temperton. Fast mixed-radix real fourier transforms. *Journal of Computational Physics*, 52:340–350, 1983.

[22] Henrik V. Sorenson, Douglas L. Jones, Michael T. Heideman, and C. Sidney Burrus. Real-valued fast fourier transform algorithms. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-35(6):849–863, 1987.