库利-图基快速傅里叶变换算法

维基百科，自由的百科全书

库利－图基快速傅里叶变换算法（英語：Cooley–Tukey FFT algorithm）[1]是最常見的快速傅里葉變換算法。這一方法以分治法為策略遞歸地將長度為$N = N_1N_2$的DFT分解為長度分別為N_1和N_2的兩個較短序列的DFT，以及與旋轉因子的複數乘法。這種方法以及FFT的基本思路在1965年詹姆斯·庫利和約翰·圖基合作發表《*An algorithm for the machine calculation of complex Fourier series*》之後開始為人所知。但後來發現，實際上這兩位作者只是重新發明了高斯在1805年就已經提出的算法（此算法在歷史上數次以各種形式被再次提出）。

库利－图基算法最有名的應用，是將序列長為N的DFT分割為兩個長為N/2的子序列的DFT，因此這一應用只適用於序列長度為2的冪的DFT計算，即基2-FFT。實際上，如同高斯和库利與图基都指出的那樣，库利－图基算法也可以用於序列長度N為任意因數分解形式的DFT，即混合基FFT，而且還可以應用於其他諸如分裂基FFT等變種。儘管库利－图基算法的基本思路是採用遞歸的方法進行計算，大多數傳統的算法實現都將顯示的遞歸算法改寫為非遞歸的形式。另外，因為库利－图基算法是將DFT分解為較小長度的多個DFT，因此它可以同任一種其他的DFT算法聯合使用。

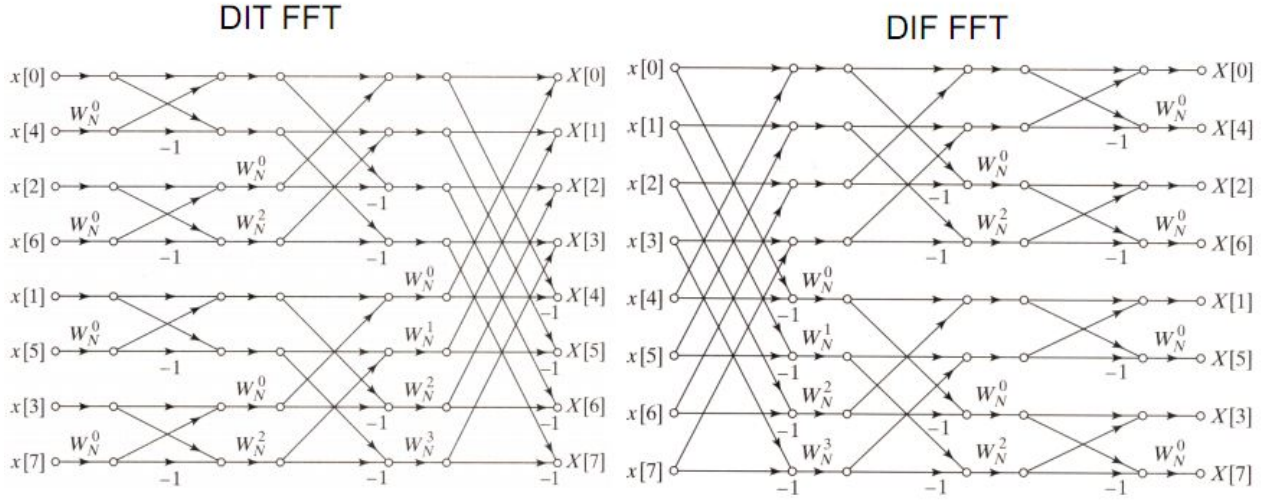
複雜度

離散傅立葉變換的複雜度為$\mathcal{O}(n^2)$（可参考大O符號）

快速傅立葉變換的複雜度為$\mathcal{O}(N \log N)$，分析可見下方架構圖部分，級數為$\log N$而每一級的複雜度為N，故複雜度為$\mathcal{O}(N \log N)$

時域/頻域抽取法

在FFT演算法中，針對輸入做不同方式的分組會造成輸出順序上的不同。如果我們使用時域抽取（Decimation-in-time），那麼輸入的順序將會是位元反轉排列（bit-reversed order），輸出將會依序排列。但若我們採取的是頻域抽取（Decimation-in-frequency），那麼輸出與輸出順序的情況將會完全相反，變為依序排列的輸入與位元反轉排列的輸出。



時域抽取法

我們可以將DFT公式中的項目在時域上重新分組，這樣就叫做時域抽取（Decimation-in-time），在這裡$e^{-j(2\pi\frac{nk}{N})}$將會被代換為旋轉因子（twiddle factor）W_N^{nk}。

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j(2\pi\frac{nk}{N})} \qquad k = 0, 1, \ldots, N-1$$

$$= \sum_{n\ even} x[n] W_N^{nk} + \sum_{n\ odd} x[n] W_N^{nk}$$

$$= \sum_{r=0}^{(N/2)-1} x[2r] W_N^{2rk} + \sum_{r=0}^{(N/2)-1} x[2r+1] W_N^{(2r+1)k}$$

$$= \sum_{r=0}^{(N/2)-1} x[2r] W_N^{2rk} + W_N^k \sum_{r=0}^{(N/2)-1} x[2r+1] W_N^{2rk}$$

$$= \sum_{r=0}^{(N/2)-1} x[2r]W_{N/2}^{rk} + W_N^k \sum_{r=0}^{(N/2)-1} x[2r+1]W_{N/2}^{rk}$$

$$= G[k] + W_N^k H[k]$$

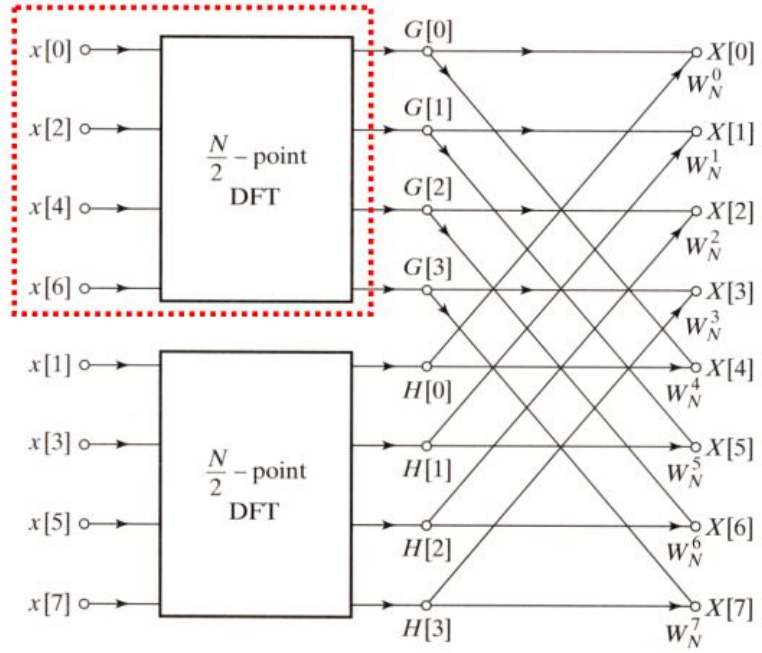
在這邊我們要注意的是，我們所替換的G[k]與H[k]具有週期性：

- $\begin{cases} G[k+\frac{N}{2}] = G[k] \\ H[k+\frac{N}{2}] = H[k] \end{cases}$

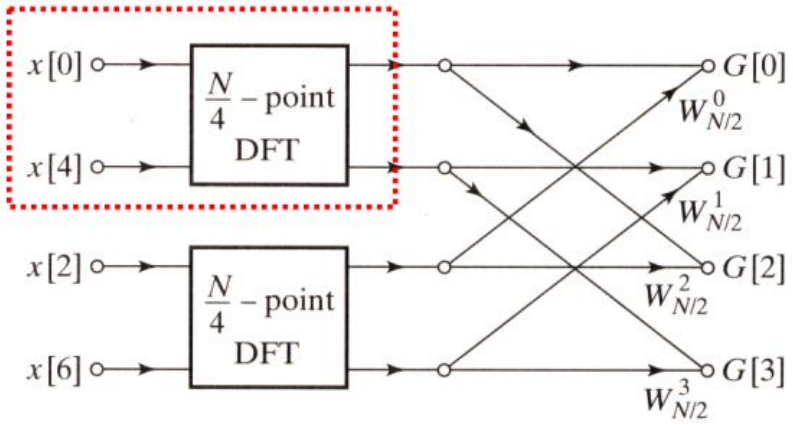
还注意到系数具有对称性：

- $W_N^{k+N/2} = -W_N^k$

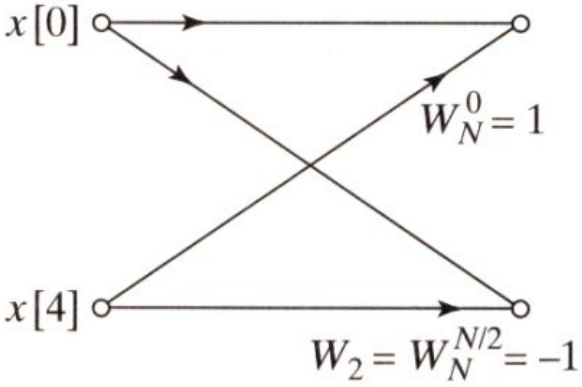
上述的推導可以劃成下面的圖：



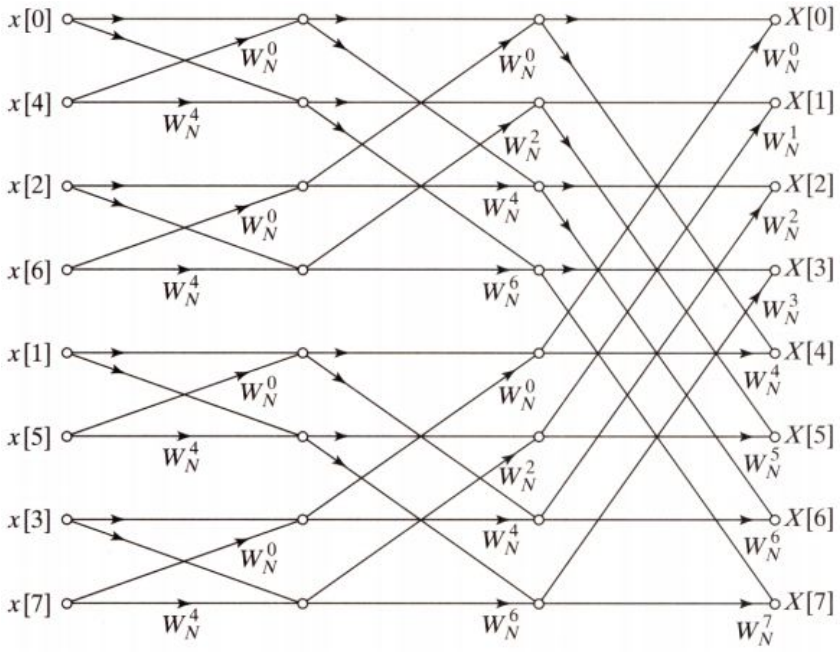
劃紅框處所示的$\frac{N}{2}$點DFT架構如下圖所示：



劃紅框處所示的$\frac{N}{4}$點DFT架構如下圖所示：



下圖是一個8點DIT FFT的完整架構圖。



頻域抽取法

我們可以將DFT公式中的項目在頻域上重新分組，這樣就叫做頻域抽取（Decimation-in-frequency）。

首先先觀察在頻域上偶數項的部分：

$$X[2r] = \sum_{n=0}^{N-1} x[n] W_N^{n(2r)} \quad r = 0, 1, \cdots, \frac{N}{2} - 1$$

$$= \sum_{n=0}^{\frac{N}{2}-1} x[n] W_N^{2nr} + \sum_{n=\frac{N}{2}}^{N-1} x[n] W_N^{2nr}$$

$$= \sum_{n=0}^{\frac{N}{2}-1} x[n] W_N^{2nr} + \sum_{n=0}^{\frac{N}{2}-1} x[n+\frac{N}{2}] W_N^{2r[n+\frac{N}{2}]}$$

$$\because W_N^{2r[n+\frac{N}{2}]} = W_N^{2rn} W_N^{rN} = W_N^{2rn}$$

$$= \sum_{n=0}^{\frac{N}{2}-1} x[n] W_N^{2rn} + \sum_{n=0}^{\frac{N}{2}-1} x[n+\frac{N}{2}] W_N^{2rn}$$

$$= \sum_{n=0}^{\frac{N}{2}-1} (x[n] + x[n+\frac{N}{2}]) W_{\frac{N}{2}}^{rn}$$

$$= \sum_{n=0}^{\frac{N}{2}-1} g[n] W_{\frac{N}{2}}^{rn}$$

再觀察在頻域上奇數項的部分：

$$X[2r+1] = \sum_{n=0}^{N-1} x[n] W_N^{n(2r+1)} \quad r = 0, 1, \cdots, \frac{N}{2} - 1$$

$$= \sum_{n=0}^{\frac{N}{2}-1} x[n] W_N^{n(2r+1)} + \sum_{n=\frac{N}{2}}^{N-1} x[n] W_N^{n(2r+1)}$$

$$= \sum_{n=0}^{\frac{N}{2}-1} x[n] W_N^{n(2r+1)} + \sum_{n=0}^{\frac{N}{2}-1} x[n+\frac{N}{2}] W_N^{(2r+1)[n+\frac{N}{2}]}$$

$$\because W_N^{(2r+1)[n+\frac{N}{2}]} = W_N^{(2r+1)n} W_N^{(2r+1)\frac{N}{2}} = -W_N^{(2r+1)n}$$

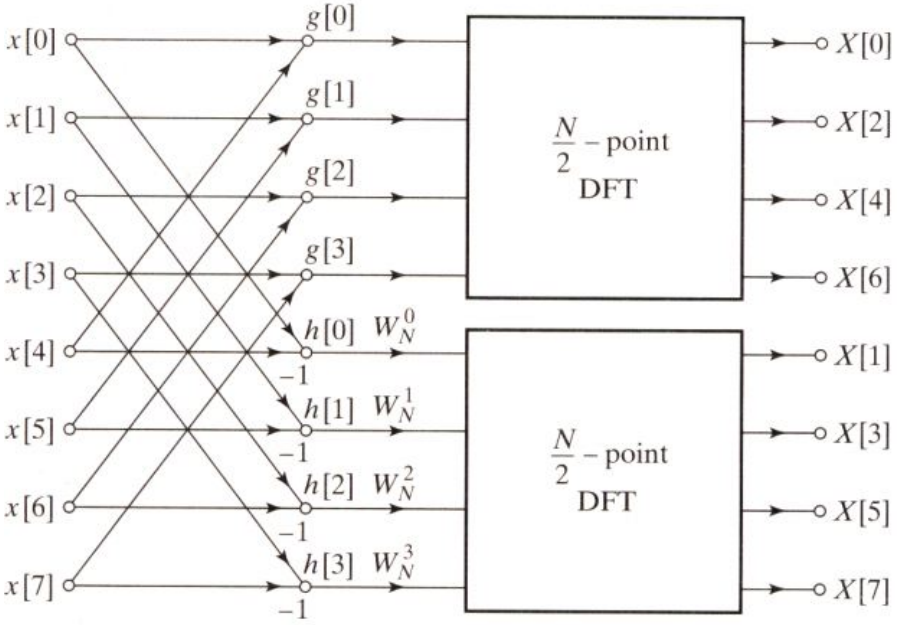
$$= \sum_{n=0}^{\frac{N}{2}-1} x[n] W_N^{(2r+1)n} - \sum_{n=0}^{\frac{N}{2}-1} x[n+\frac{N}{2}] W_N^{(2r+1)n}$$

$$= \sum_{n=0}^{\frac{N}{2}-1} (x[n] - x[n+\frac{N}{2}]) W_N^{n(2r+1)}$$

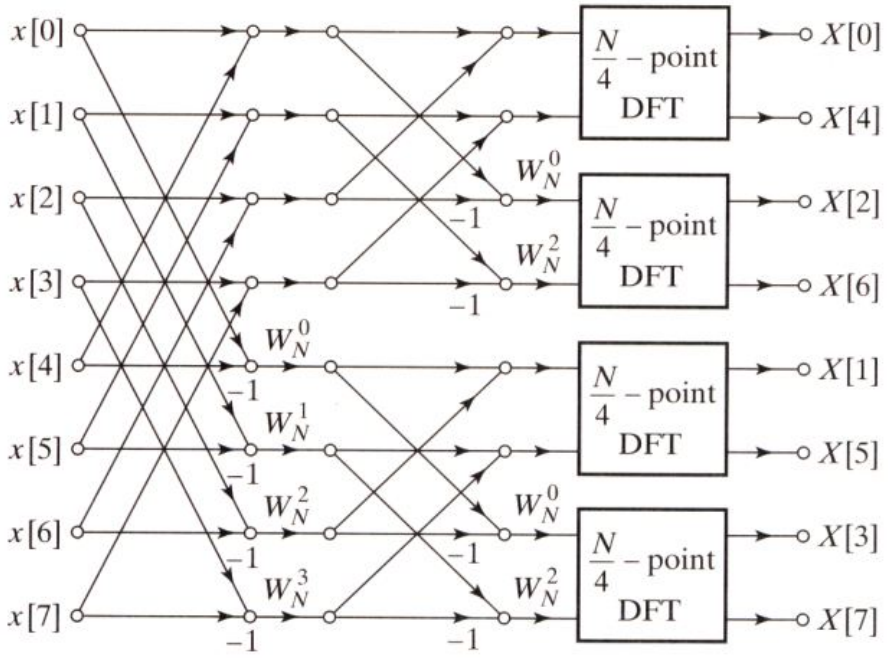
$$= \sum_{n=0}^{\frac{N}{2}-1} (x[n] - x[n+\frac{N}{2}]) W_N^{n} W_{\frac{N}{2}}^{nr}$$

$$= \sum_{n=0}^{\frac{N}{2}-1} (h[n] W_N^{n}) W_{\frac{N}{2}}^{rn}$$

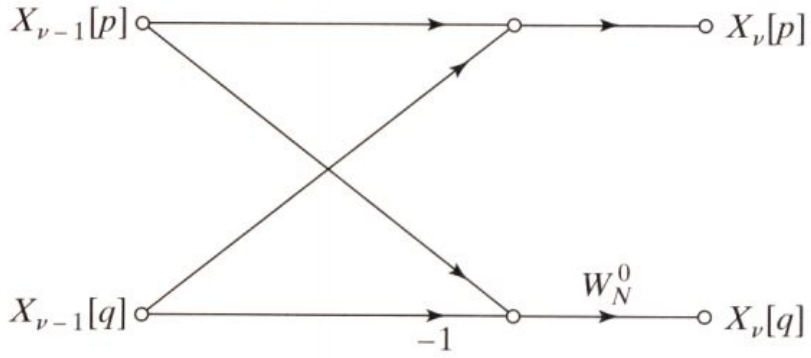
上述的推導可以畫成下面的圖：



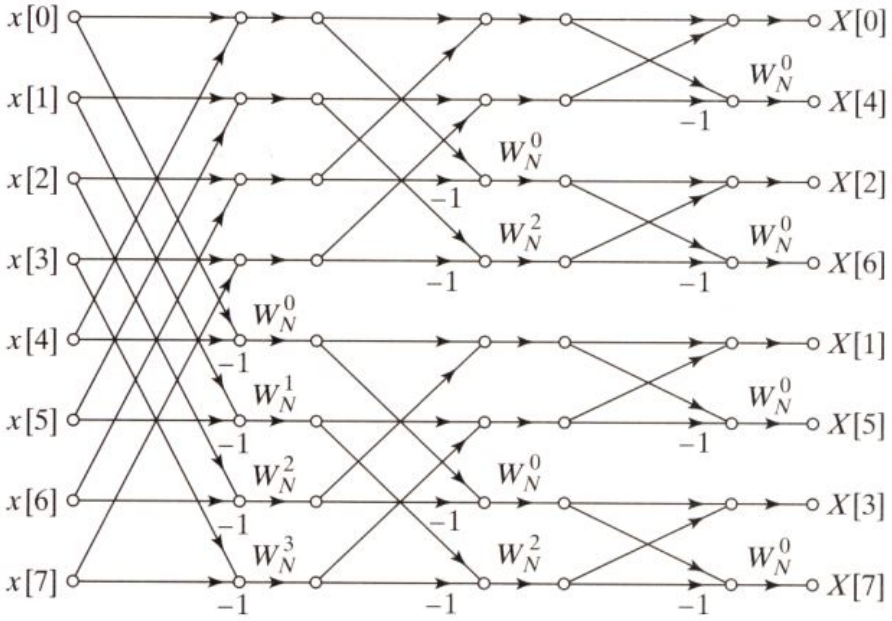
更進一步的拆解$\frac{N}{2}$-point DFT的架構



下圖為8點FFT下$\frac{N}{4}$-point DFT的架構



總結上述架構，完整的8點DIF FFT架構圖為



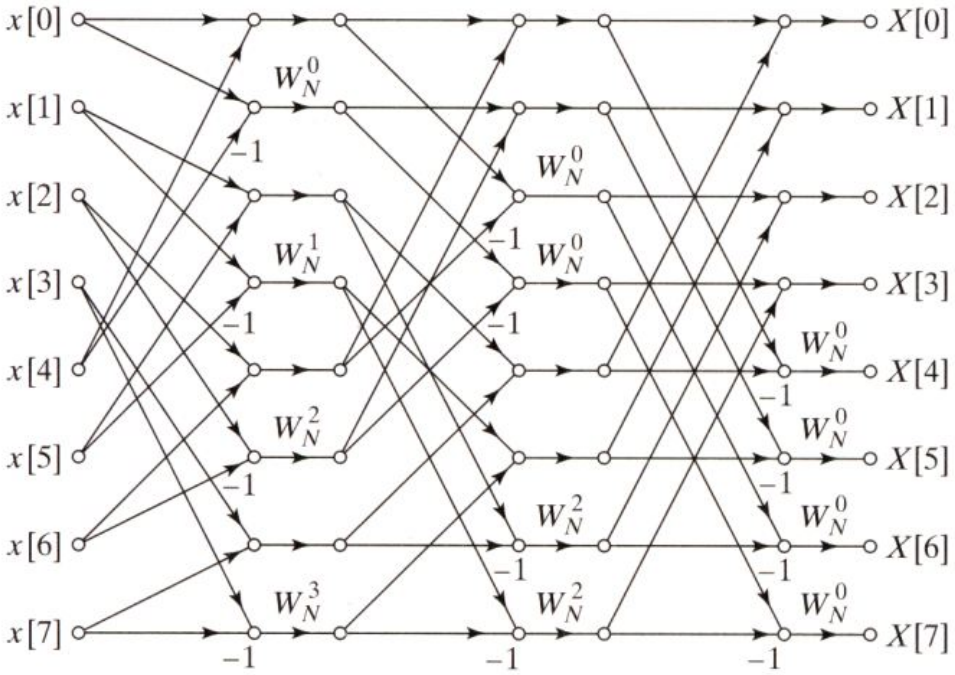
單一基底

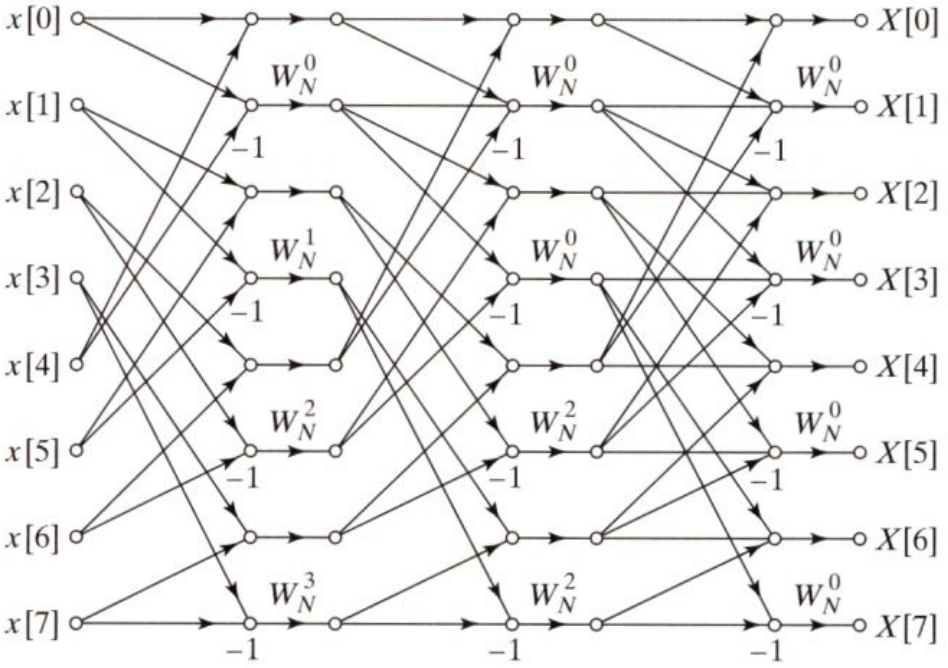
利用库利－图基演算法進行離散傅立葉拆解時，能夠依需求而以2, 4, 8…等2的冪次方為單位進行拆解，不同的拆解方法會產生不同層數快速傅里葉變換的架構，基底越大則層數越少，複數乘法器也越少，但是每級的蝴蝶形架構則會越複雜，因此常見的架構為2基底、4基底與8基底這三種設計。

2基底

2基底-快速傅立葉演算法（Radix-2 FFT）是最廣為人知的一種库利－图基快速傅立葉演算法分支。此方法不斷地將N點的FFT拆解成兩個'N/2'點的FFT，利用旋轉因子$W^{nk}, W^{\frac{N}{2}}$的對稱性藉此來降低DFT的計算複雜度，達到加速的功效。

而其實前述有關時域抽取或是頻域抽取的方法介紹，即為2基底的快速傅立葉轉換法。以下展示其他種2基底快速傅立葉演算法的連線方法，此種不規則的連線方法可以讓輸出與輸入都為循序排列，但是連線的複雜度卻大大的增加。





4基底

4基底快速傅立葉變換演算法則是承接2基底的概念，在此裡用**時域**抽取的技巧，將原本的DFT公式拆解為四個一組的形式：

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j(2\pi \frac{nk}{N})} \qquad k = 0, 1, \ldots, N-1$$

$$= \sum_{n=0}^{\frac{N}{4}-1} x[4n+0]W_N^{(4n+0)k} + \sum_{n=0}^{\frac{N}{4}-1} x[4n+1]W_N^{(4n+1)k}$$

$$+ \sum_{n=0}^{\frac{N}{4}-1} x[4n+2]W_N^{(4n+2)k} + \sum_{n=0}^{\frac{N}{4}-1} x[4n+3]W_N^{(4n+3)k}$$

$$= W_N^0 \sum_{n=0}^{\frac{N}{4}-1} x[4n+0]W_{\frac{N}{4}}^{nk} + W_N^{1k} \sum_{n=0}^{\frac{N}{4}-1} x[4n+1]W_{\frac{N}{4}}^{nk}$$

$$+W_N^{2k} \sum_{n=0}^{\frac{N}{4}-1} x[4n+2]W_{\frac{N}{4}}^{nk} + W_N^{3k} \sum_{n=0}^{\frac{N}{4}-1} x[4n+3]W_{\frac{N}{4}}^{nk}$$

$$W_N^0 F_0(k) + W_N^k F_1(k) + W_N^{2k} F_2(k) + W_N^{3k} F_3(k)$$

在這裡再利用$W^{nk+\frac{N}{4}} = -W^{nk+\frac{3N}{4}} = -jW^{nk}$的特性來進行與2基數FFT類似的化減方法，以降低演算法複雜度。

8基底

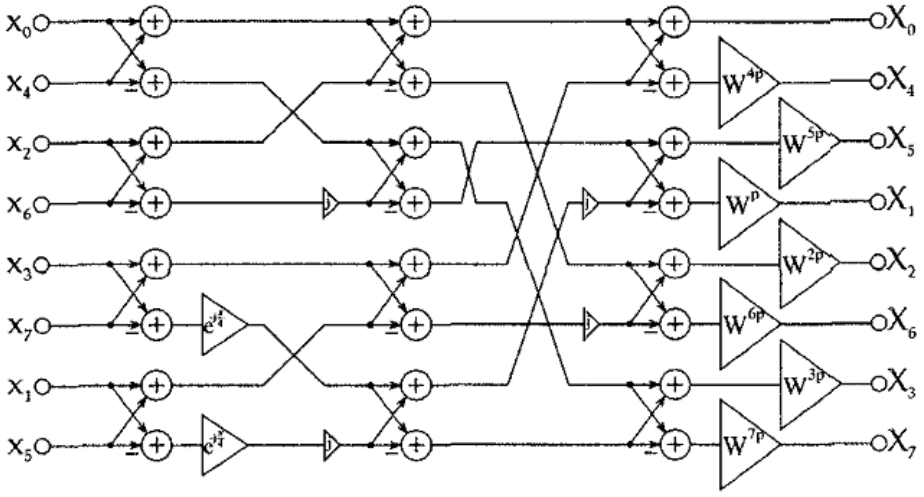
在库利－图基算法裡，使用的基底（radix）越大，複數的乘法與記憶體的存取就越少，所帶來的好處不言而喻。但是隨之而來的就是實數的乘法與實數的加法也會增加，尤其計算單元的設計也就越複雜，對於可應用FFT之點數限制也就越嚴格。在表中我們可以看到在不同基底下所需的計算成本。

使用4096點FFT在不同基底下的計算量

動作	2基底	4基底	8基底
複數乘法	22528	15360	10752
實數乘法	0	0	8192
複數加法	49152	49152	49152
實數加法	0	0	8192
記憶體存取	49152	24576	16384

在DFT的公式中，我們重新定義$x=[x(0),x(1),\ldots,x(N-1)]^T$, $X=[X(0),X(1),\ldots,X(N-1)]^T$，則DFT可重寫為$X=F_N \cdot x$。$F_N$是一個N×N的DFT矩陣，其元素定義為$[F_N]_{ij}=W_{N_{ij}}(i,j \in [0,N-1])$，當N=8時，我們可以得到以下的$F_8$矩陣並且進一步將其拆解。

在拆解成三個矩陣相乘之後，我們可以將複數運算的數量從56個降至24個複數的加法。底下是8基底的SFG。要注意的是所有的輸出與輸入都是複數的形式，而輸出與輸入的排序也並非規律排列，此種排列方式是為了要達到接線的最佳化。



混合基底

2/8基底

在2/8基底的演算法中，我們可以看到我們對於偶數項的輸出會使用2基底的分解法，對於奇數項的輸出採用8基底的分解法。這個做法充分利用了2基底與4基底擁有最少乘法數與加法數的特性。當使用了2基底的分解法後，偶數項的輸出如下所示。

$$C_{2k}=\sum_{n=0}^{\frac{N}{2}-1}(x_{2n}+x_{\frac{N}{2}+n})W^{2nk}$$

奇數項的輸出則交由8基底分解來處理，如下四式所述。

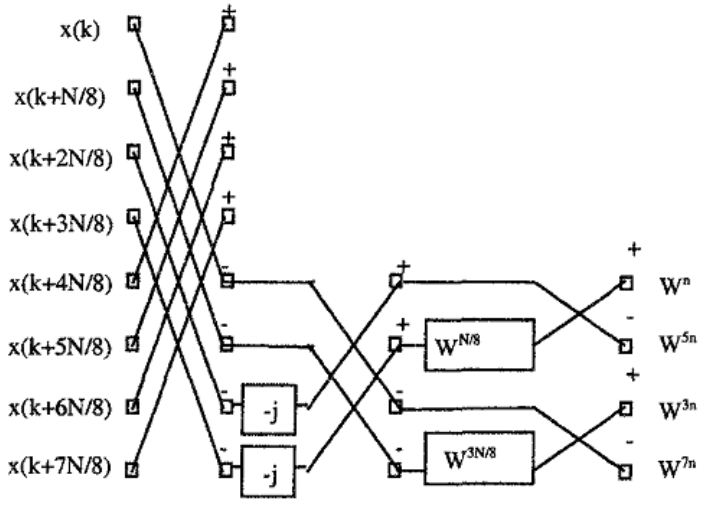
$$C_{8k+1}=\sum_{n=0}^{\frac{N}{8}-1}\left\{[(x_n-x_{n+\frac{N}{2}})-j(x_{n+\frac{N}{4}}-x_{n+\frac{3N}{4}})]+W^{\frac{N}{8}}[(x_{n+\frac{N}{8}}-x_{n+\frac{5N}{8}})-j(x_{n+\frac{3N}{8}}-x_{n+\frac{7N}{8}})]\right\}W^{8nk}W^{n}$$

$$C_{8k+5}=\sum_{n=0}^{\frac{N}{8}-1}\left\{[(x_n-x_{n+\frac{N}{2}})-j(x_{n+\frac{N}{4}}-x_{n+\frac{3N}{4}})]-W^{\frac{N}{8}}[(x_{n+\frac{N}{8}}-x_{n+\frac{5N}{8}})-j(x_{n+\frac{3N}{8}}-x_{n+\frac{7N}{8}})]\right\}W^{8nk}W^{5n}$$

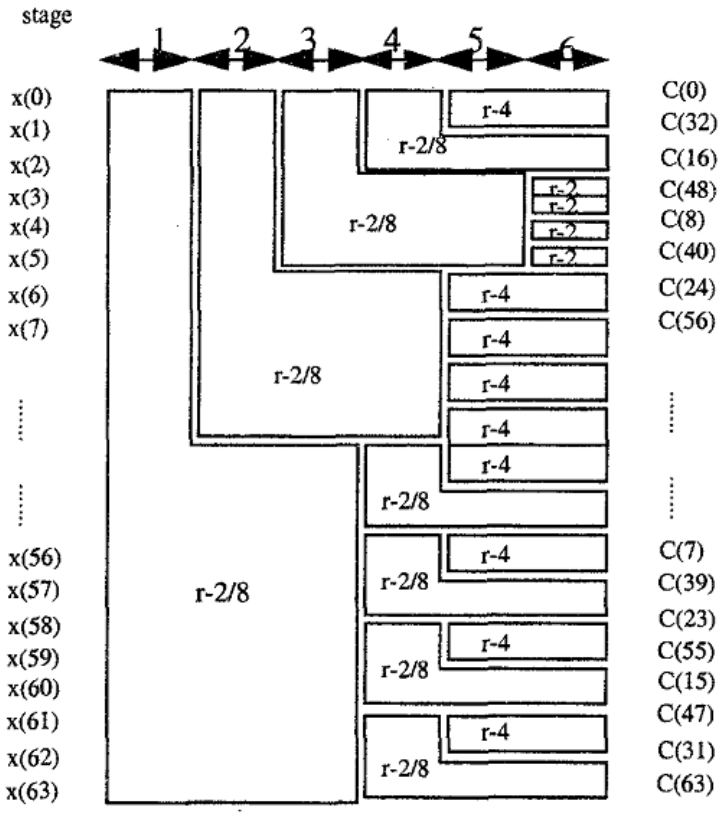
$$C_{8k+3}=\sum_{n=0}^{\frac{N}{8}-1}\left\{[(x_n-x_{n+\frac{N}{2}})+j(x_{n+\frac{N}{4}}-x_{n+\frac{3N}{4}})]+W^{\frac{3N}{8}}[(x_{n+\frac{N}{8}}-x_{n+\frac{5N}{8}})+j(x_{n+\frac{3N}{8}}-x_{n+\frac{7N}{8}})]\right\}W^{8nk}W^{3n}$$

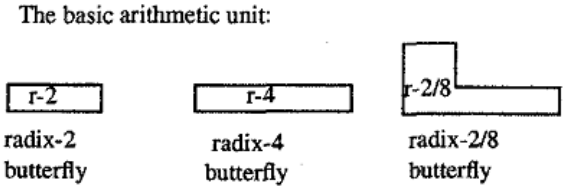
$$C_{8k+7}=\sum_{n=0}^{\frac{N}{8}-1}\left\{[(x_n-x_{n+\frac{N}{2}})+j(x_{n+\frac{N}{4}}-x_{n+\frac{3N}{4}})]-W^{\frac{3N}{8}}[(x_{n+\frac{N}{8}}-x_{n+\frac{5N}{8}})+j(x_{n+\frac{3N}{8}}-x_{n+\frac{7N}{8}})]\right\}W^{8nk}W^{7n}$$

以上式子就是2/8基底的FFT快速演算法。在架構圖上可化為L型的蝴蝶運算架構，如下圖所示。



而下圖表示的是一個64點的FFT使用2/8基底的架構圖。雖然2/8基底的演算法縮減了$W^{\frac{N}{8}},W^{\frac{3N}{8}}$的乘法量，但是這種演算法最大的缺點是跟其他固定基底或是混合基底比較起來，他的架構較為不規則。所以在硬體上比4基底或是2基底更難實現。





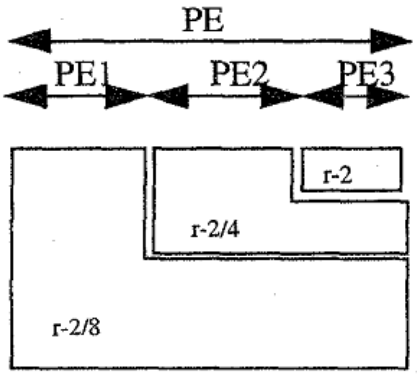
2/4/8基底

為了改進Radix 2/8在架構上的不規則性，我們在這裡做了一些修改，如下圖4.。此修改可讓架構更加規則，且所使用的加法器與乘法器數量更加減少，如下表所示。

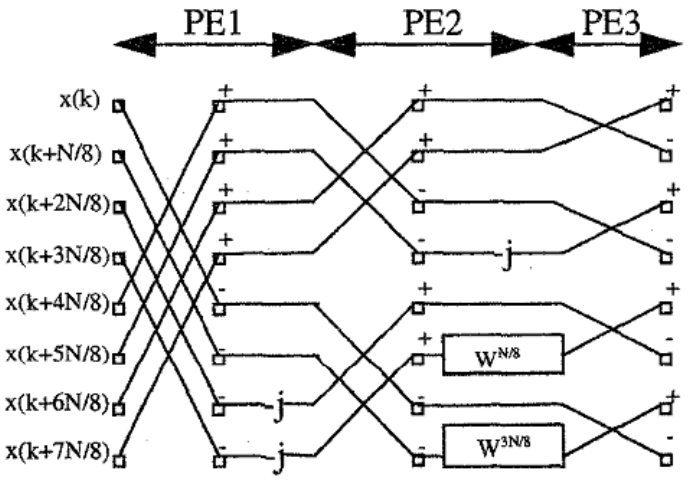
8^n點FFT在不同演算法下所需複數乘法量

$N=8^n$	2基底	4基底	2/4混合基底	2/4/8基底
8	2	-	2	0
64	98	76	72	48
512	1538	-	1082	824
4096	18434	13996	12774	10168

在這裡我們最小的運算單元稱為PE（Process Element）,PE內部包含了2/8基底、2/4基底、2基底的運算，簡化過的信號處理流程與蝴蝶型架構圖可見下圖



(a) Simplified signal-flow for a process element(PE)



(b) Butterflies for a process element(PE)

Fig. 7 The basic process element(PE) for radix-2/4/8 algorithm

2^2基底

基底的選擇越大會造成蝴蝶形架構更加複雜，控制電路也會複雜化。因此Shousheng He和Mats Torkelson在1996提出了一個2^2基底的FFT演算法，利用旋轉因子的特性：$W_{\frac{N}{4}} = -j$。而–j的乘法基本上只需要將被乘數的實部虛部對調，然後將虛部加上負號即可，這樣的負數乘法被定義為'簡單乘法'，因此可以用很簡單的硬體架構來實現。利用上面的特性，2^2基底FFT能用串接的方式將旋轉因子以4為單位對DFT公式進行拆解，將蝴蝶形架構層數降到log4N，大幅減少複數乘法器的用量，而同時卻維持了2基底蝴蝶形架構的簡單性，在效能上獲得改進。2^2基底DIF FFT演算法的拆解方法如下列公式所述：

N點DFT公式：$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk}, 0 \leqq k < N$

利用線性映射將n與k映射到三個維度上面

$$\begin{cases} n = < \frac{N}{2} n_1 + \frac{N}{4} n_2 + n_3 >_N \\ k = < k_1 + 2k_2 + 4k_3 >_N \end{cases}$$

然後套用Common Factor Algorithm（CFA）

$$X(k_1 + 2k_2 + 4k_3) = \sum_{n_3=0}^{\frac{N}{4}-1} \sum_{n_2=0}^{1} \sum_{n_1=0}^{1} x(\frac{N}{2} n_1 + \frac{N}{4} n_2 + n_3) W_N^{(\frac{N}{2} n_1 + \frac{N}{4} n_2 + n_3)(k_1 + 2k_2 + 4k_3)}$$

$$= \sum_{n_3=0}^{\frac{N}{4}-1} \sum_{n_2=0}^{1} \left\{ B_{\frac{N}{2}}^{k_1} W_N^{(\frac{N}{4}n_2+n_3)k_1} \right\} W_N^{(\frac{N}{4}n_2+n_3)(2k_2+4k_3)}$$

而蝴蝶型架構會變成以下形式

$$B_{\frac{N}{2}}^{k_1} = x(\frac{N}{4}n_2 + n_3) + (-1)^{k_1} x(\frac{N}{4}n_2 + n_3 + \frac{N}{2})$$

利用旋轉因子$W_{\frac{N}{4}} = -j$的特性，可以觀察出

$$W_N^{(\frac{N}{4}n_2+n_3)(k_1+2k_2+4k_3)} = W_N^{Nn_2n_3} W_N^{\frac{N}{4}n_2(k_1+2k_2)} W_N^{n_3(k_1+2k_2)} W_N^{4n_3k_3}$$

$$= (-j)^{n_2(k_1+2k_2)} W_N^{n_3(k_1+2k_2)} W_N^{4n_3k_3}$$

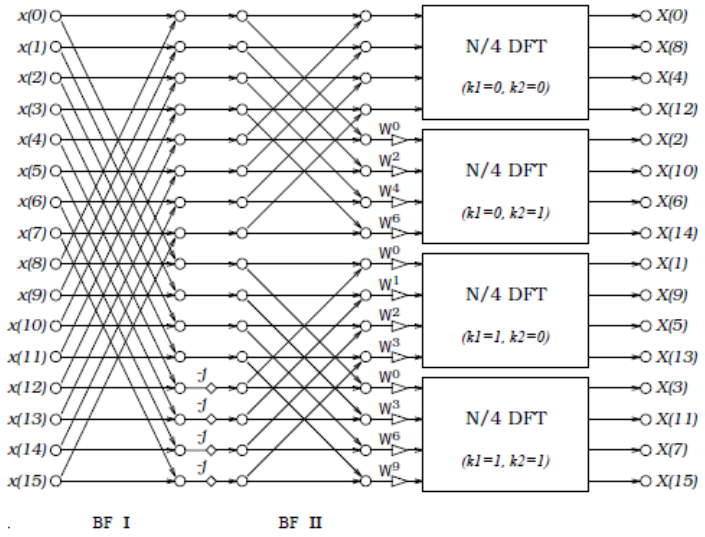
再將此公式帶入原式中可以得到

$$X(k_1 + 2k_2 + 4k_3) = \sum_{n_3=0}^{\frac{N}{4}-1} [H(k_1, k_2, n_3) W_N^{n_3(k_1+2k_2)}] W_N^{4n_3k_3}$$

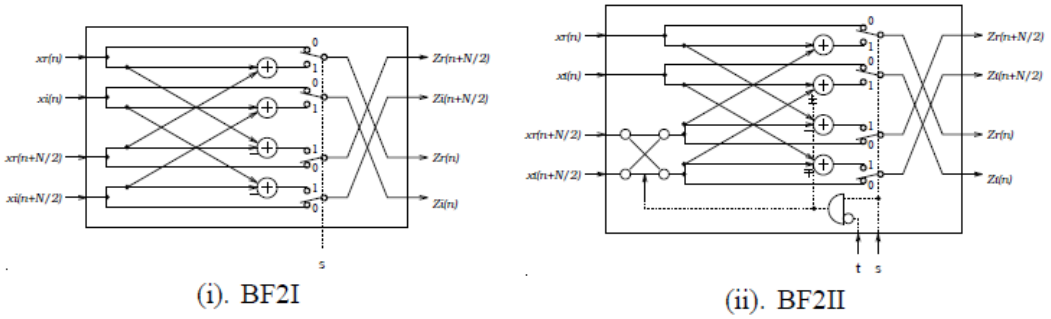
$$H(k_1, k_2, n_3) = \underbrace{\overbrace{[x(n_3) + (-1)^{k_1}(n_3 + \tfrac{N}{2})]}^{BF2I} + (-j)^{k_1+2k_2} \overbrace{[x(n_3 + \tfrac{N}{4}) + (-1)^{k_1}(n_3 + \tfrac{3N}{4})]}^{BF2I}}_{BF2II}$$

如上述公式所示，原本的DFT公式會被拆解成多個$H(k_1, k_2, n_3)$，而$H(k_1, k_2, n_3)$又可分為BF2I與BF2II兩個階層，分別會對應到之後所介紹的兩種硬體架構。

一個16點的DFT公式經過一次上面所述之拆解之後可得下面的FFT架構

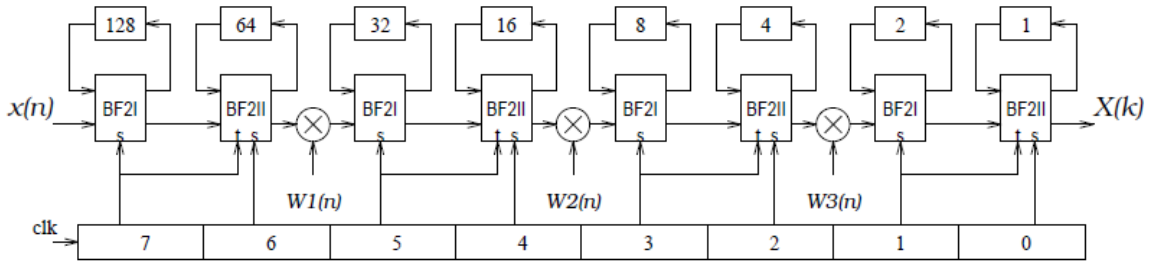


可以看出上圖的架構保留了2基底的簡單架構，然而複數乘法卻降到每兩級才出現一次，也就是$log_4 N$次。而BF2I以及BF2II所對應的硬體架構下圖：



其中BF2II硬體單元中左下角的交叉電路就是用來處理-j的乘法。

一個256點的FFT架構可以由下面的硬體來實現：



其中圖下方的為一7位元寬的計數器，而此架構的控制電路相當單純，只要將計數器的各個位元分別接上BF2I與BF2II單元即可。

下表將2基底、4基底與2^2基底演算法做一比較，可以發現2^2基底演算法所需要的乘法氣數量為2基底的一半，加法棄用量是4基底的一半，並維持一樣的記憶體用量和控制電路的簡單性。

乘法器與加法器數量比較

	乘法數	加法數	記憶體大小	控制電路
R2SDF	$2(\log_4 N-1)$	$4\log_4 N$	N-1	簡單
R4SDF	$\log_4 N$ -1	$8\log_4 N$	N-1	中等
R2^2SDF	$\log_4 N$ -1	$4\log_4 N$	N-1	簡單

2^3基底

如上所述，2^2演算法是將旋轉因子$W^{\frac{N}{4}} = -j$視為一個簡單乘法，進而由公式以及硬體上的化簡獲得硬體需求上的改進。而藉由相同的概念，Shousheng He和Mats Torkelson進一步將旋轉因子$W^{\frac{N}{8}} = \frac{\sqrt{2}}{2}(1-j)$的乘法化簡成一個簡單乘法，而化簡的方法將會在下面講解。

$\frac{\sqrt{2}}{2}$乘法化簡

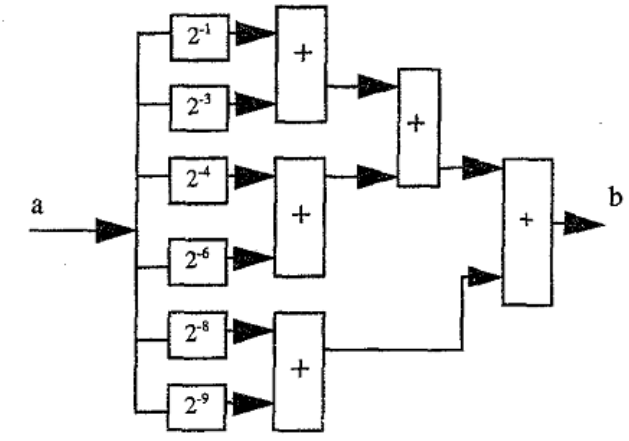
在2基數FFT演算法中的基本概念是利用旋轉因子$W^{nk}, W^{\frac{N}{2}}$的對稱性，4基數演算法則是利用$W^{nk+\frac{N}{4}} = -W^{nk+\frac{3N}{4}} = -jW^{nk}$的特性。但是我們會發現在這些旋轉因子的對稱特性中—

$$W^{\frac{N}{8}} = -W^{\frac{5N}{8}} = \frac{\sqrt{2}}{2}(1-j), W^{\frac{3N}{8}} = -W^{\frac{7N}{8}} = -\frac{\sqrt{2}}{2}(1+j)$$

—並沒有被利用到。主要是因為$\frac{\sqrt{2}}{2}(1-j)$的乘法運算會讓整個FFT變得複雜，但是如果藉由近似的方法，我們便可以將此一運算化簡為12個加法。

$$\frac{\sqrt{2}}{2} = 0.70710678 = 2^{-1} + 2^{-3} + 2^{-4} + 2^{-6} + 2^{-8} + 2^{-9}$$

我們可以從上式注意到，$\frac{\sqrt{2}}{2}$可以被近似為五個加法的結果，所以$\frac{\sqrt{2}}{2}(1+j)$就可以被簡化為只有六個加法，再算入實部與虛部的計算，總共只需12個加法器就可以運用到此一簡化特性。



經由與2^2基底類似的推導，並用串接的方式將旋轉因子以8為單位對DFT公式進行拆解，2^3基底FFT演算法進一步將複數乘法器的用量縮減到$\log_8 N$，並同時維持硬體架構的簡單性。　　推導的方法與2^2基底相當類似。藉由這樣的方法，2^3基底能將乘法器的用量縮減到2基底的1/3，並同時維持一樣的記憶體用量以及控制電路的簡單性。

一般性分解[2]

除了常在應用中見到與$\mathbf{2}$相關基底的拆解法，對於更加一般性的$N = N_1 N_2$點離散傅立葉變換問題， 我們也有辦法在理論上進行拆解，將問題化為數個N_1與N_2點離散傅立葉變換問題，並可對計算量進行估計。

而特別的是，透過互質在數論上的特性，對於N_1與N_2互質的情況，可以進一步節省一些運算， 在下面會特別分開討論。

$N_1 N_2$非互質

為了避免之後的符號混淆，我們先將$N_1 N_2$置換為$P_1 P_2$，也就是說接著要將$N = P_1 P_2$點離散傅立葉變換， 想辦法拆解為數個P_1與P_2點離散傅立葉變換問題。

接著定義要拆分的問題，要拆分的問題為N點離散傅立葉變換，將$f[n]$轉換至$F[m]$:

$$F[m] = \sum_{n=0}^{N-1} e^{-i\frac{2\pi}{N}mn} f[n] \qquad m, n = 0, 1, \dots, N-1$$

直觀地說，這個N點離散傅立葉變換，將由n作為參數的函數$f[n]$，轉換成由m作為參數的函數$F[m]$， 並且m, n都有N個可能的數值。

待定義好要拆分的問題，便可以開始討論如何進行拆分，基本概念是將有N個可能的數值的m, n， 分別化為個使用兩個參數進行描述的函數$m = [m_1, m_2], n = [n_1, n_2]$，並藉此將原問題化為二維度離散傅立葉變換， 便可使用數個較小的離散傅立葉變換問題描述整個過程。

而一種很直覺的轉換方式，便是透過將m, n分別除以P_2, P_1， 以商數與餘數來做為參數描述m, n的值:

$$\begin{aligned} &n = n_1 P_1 + n_2 \qquad m = m_1 + m_2 P_2 \\ &n_1, m_1 = 0, 1, \dots, P_2 - 1 \qquad n_2, m_2 = 0, 1, \dots, P_1 - 1 \end{aligned}$$

其中n_1作為將n除以P_1的商數，與作為m除以P_2的餘數的m_1相同， 具有P_2個可能數值，同理n_2與m_2有P_1個可能數值。

將上述的參數代換及$N = P_1 P_2$帶入原式，可以得到:

$$F[m_1 + m_2 P_2] = \sum_{n_1=0}^{P_2-1} \sum_{n_2=0}^{P_1-1} e^{-i\frac{2\pi}{P_1 P_2}(m_1 + m_2 P_2)(n_1 P_1 + n_2)} f[n_1 P_1 + n_2]$$

將右式的指數部分乘開並分項化簡可以得到:

$$F[m_1 + m_2 P_2] = \sum_{n_1=0}^{P_2-1} \sum_{n_2=0}^{P_1-1} f[n_1 P_1 + n_2] e^{-i\frac{2\pi}{P_2}m_1 n_1} e^{-i\frac{2\pi}{P_1}m_2 n_2} e^{-i\frac{2\pi}{P_1 P_2}m_1 n_2} e^{-i2\pi m_2 n_1}$$

最後透過$e^{-i2\pi} = 1$與$P_1 P_2 = N$，可以得到:

$$F[m_1 + m_2 P_2] = \sum_{n_2=0}^{P_1-1} \sum_{n_1=0}^{P_2-1} f[n_1 P_1 + n_2] e^{-i\frac{2\pi}{P_2}m_1 n_1} e^{-i\frac{2\pi}{N}m_1 n_2} e^{-i\frac{2\pi}{P_1}m_2 n_2}$$

觀察上式，並加上括號輔助釐清運算順序我們可以得到:

$$F[m_1 + m_2 P_2] = \sum_{n_2=0}^{P_1-1} \left\{ \left\{ \sum_{n_1=0}^{P_2-1} f[n_1 P_1 + n_2] e^{-i\frac{2\pi}{P_2}m_1 n_1} \right\} e^{-i\frac{2\pi}{N}m_1 n_2} \right\} e^{-i\frac{2\pi}{P_1}m_2 n_2}$$

最內層的運算可以視為將雙參數函數$f[n_1, n_2]$中的一個參數n_1，透過離散傅立葉變換取代為由m_1描述， 得到一個新函數$G_1[m_1, n_2]$(這步因為對每個不同n_2，都需要做一次將n_1取代為m_1的轉換， 共需要P_1個P_2點離散傅立葉變換):

$$F[m_1 + m_2 P_2] = \sum_{n_2=0}^{P_1-1} \left\{ G_1[m_1, n_2] e^{-i\frac{2\pi}{N}m_1 n_2} \right\} e^{-i\frac{2\pi}{P_1}m_2 n_2}$$

下一層的運算則可視為單純的乘法，將$G_1[m_1, n_2]$與$e^{-i\frac{2\pi}{N}m_1 n_2}$相乘，得到$G_2[m_1, n_2]$ (這步需要的計算量視$\frac{m_1 n_2}{N}$的特殊性而會有變化):

$$F[m_1 + m_2 P_2] = \sum_{n_2=0}^{P_1-1} G_2[m_1, n_2] e^{-i\frac{2\pi}{P_1}m_2 n_2}$$

最後的運算可以視為將函數$G_2[m_1,n_2]$中n_2，透過離散傅立葉變換取代為由m_2描述， 得到一個新函數$G_3[m_1,m_2]$(這步因為對每個不同m_1，都需要做一次將n_2取代為m_2的轉換， 共需要P_2個P_1點離散傅立葉變換)：

$$F[m(=m_1+m_2P_2)]=G_3[m_1,m_2]$$

就成功僅使用P_1與P_2點離散傅立葉變換，描述了原先的N點離散傅立葉變換。

而在這樣的分解下，我們使用了P_1個P_2點離散傅立葉變換與P_2個P_1點離散傅立葉變換與一些額外的乘法， 並且這些額外使用的複數乘法$G_1[m_1,n_2]\times e^{-i\frac{2\pi}{N}m_1n_2}$， 在電腦的運算架構中，如果$\frac{m_1n_2}{N}$是$\frac{1}{4}$的倍數則不需要使用乘法， 如果$\frac{m_1n_2}{N}$是$\frac{1}{8},\frac{1}{12}$的倍數則僅需兩個實數乘法， 其他則需三個實數乘法，所以總運算量可以如下方式表示：

$$P_2B_1+P_1B_2+3D_3+2D_2$$

其中B_1是P_1傅立葉所需乘法數，B_2是P_2傅立葉所需乘法數， D_3是需三個實數乘法m_1n_2組合個數，D_2是需兩個實數乘法m_1n_2組合個數。

而常見以2為基底的分解則是為了使離散傅立葉變換所需乘法數為零，這樣就僅需考慮上面提到的額外乘法，可以提高效率也有較簡單的結構。

N_1N_2互質

在N_1N_2互質的情況下，仍是採取和上面相近的思路來將問題進行拆分，首先，為了避免之後的符號混淆，我們同樣將N_1N_2置換為P_1P_2。

接著同樣定義要拆分的問題：

$$F[m]=\sum_{n=0}^{N-1}e^{-i\frac{2\pi}{N}mn}f[n]\qquad m,n=0,1,\dots,N-1$$

接著就是和上面的算法有最大差異的部分，在將m,n化為個使用兩個參數進行描述的函數$m=[m_1,m_2],n=[n_1,n_2]$時， 最直覺的作法便是使用商數和餘數，但在P_1,P_2互質的情況下，可以有一些其他更具技巧性的選擇。

當P_1,P_2互質，對所有$n=0,1,\dots,N-1$我們可以找到唯一且不重複的一組n_1,n_2使得：

$$n=((n_1P_1+n_2P_2))_N=n_1P_1+n_2P_2+c_1N\qquad 0\le n_1<P_2\qquad 0\le n_2<P_1$$

其中$((a))_N=a\mod N$，代表取餘數的意思，c_1是一個整數。

例如假設$N=15,P_1=3,P_2=5$，則$n=1$對應到的(n_1,n_2)就是$(2,2)$， 有$2\times3+2\times5\mod 15=16\mod 15=1$。

並且對所有n_1,n_2的組合(有$P_1\times P_2=N$組)，都對應到一個特定不重複的n。

同理我們可以把$m=0,1,\dots,N-1$表示為m_1,m_2的雙參數函數：

$$m=((m_1P_1+m_2P_2))_N=m_1P_1+m_2P_2+c_2N\qquad 0\le m_1<P_2\qquad 0\le m_2<P_1$$

將上述的參數代換及$N=P_1P_2$帶入原式，可以得到：

$$F[((m_1P_1+m_2P_2))_N]=\sum_{n_1=0}^{P_2-1}\sum_{n_2=0}^{P_1-1}e^{-i\frac{2\pi}{N}(m_1P_1+m_2P_2+c_2N)(n_1P_1+n_2P_2+c_1N)}f[((n_1P_1+n_2P_2))_N]$$

接著透過一次的展開化簡及應用$e^{-i2\pi}=1$可得：

$$\begin{aligned}F[((m_1P_1+m_2P_2))_N]&=\sum_{n_1=0}^{P_2-1}\sum_{n_2=0}^{P_1-1}e^{-i\frac{2\pi}{N}(m_1P_1+m_2P_2)(n_1P_1+n_2P_2)}e^{-i2\pi c_2(n_1P_1+n_2P_2+c_1N)}e^{-i2\pi c_1(m_1P_1+m_2P_2+c_2N)}e^{-i2\pi c_1c_2N}f[((n_1P_1+n_2P_2))_N]\\&=\sum_{n_1=0}^{P_2-1}\sum_{n_2=0}^{P_1-1}e^{-i\frac{2\pi}{N}(m_1P_1+m_2P_2)(n_1P_1+n_2P_2)}f[((n_1P_1+n_2P_2))_N]\end{aligned}$$

再將$N=P_1P_2$帶入並再透過一次的展開化簡及應用$e^{-i2\pi}=1$可得：

$$\begin{aligned}F[((m_1P_1+m_2P_2))_N]&=\sum_{n_1=0}^{P_2-1}\sum_{n_2=0}^{P_1-1}f[((n_1P_1+n_2P_2))_N]e^{-i\frac{2\pi}{P_2}m_1P_1n_1}e^{-i\frac{2\pi}{P_1}m_2P_2n_2}e^{-i2\pi m_1n_2}e^{-i2\pi m_2n_1}\\&=\sum_{n_1=0}^{P_2-1}\sum_{n_2=0}^{P_1-1}f[((n_1P_1+n_2P_2))_N]e^{-i\frac{2\pi}{P_2}m_1P_1n_1}e^{-i\frac{2\pi}{P_1}m_2P_2n_2}\end{aligned}$$

觀察上式，並加上括號輔助釐清運算順序我們可以得到：

$$F[((m_1P_1+m_2P_2))_N]=\sum_{n_2=0}^{P_1-1}\left\{\sum_{n_1=0}^{P_2-1}f[((n_1P_1+n_2P_2))_N]e^{-i\frac{2\pi}{P_2}m_1P_1n_1}\right\}e^{-i\frac{2\pi}{P_1}m_2P_2n_2}$$

內層的運算可以視為將雙參數函數$f[((n_1P_1+n_2P_2))_N]$中的一個參數n_1， 透過離散傅立葉變換取代為由一個與m_1有關的變數$m_3=((m_1P_1))_{P_2}$描述， 得到一個新函數$G_1[m_3,n_2]$(這步因為對每個不同n_2，都需要做一次將n_1取代為m_3的轉換， 共需要P_1個P_2點離散傅立葉變換)：

$$F[((m_1P_1+m_2P_2))_N]=\sum_{n_2=0}^{P_1-1}G_1[m_3,n_2]e^{-i\frac{2\pi}{P_1}m_2P_2n_2}$$

外層的運算可以視為將函數$G_1[m_3,n_2]$中的參數n_2， 透過離散傅立葉變換取代為由一個與m_2有關的變數$m_4=((m_2P_2))_{P_1}$描述， 得到一個新函數$G_2[m_3,m_4]$(這步因為對每個不同m_3，都需要做一次將n_2取代為m_4的轉換， 共需要P_2個P_1點離散傅立葉變換)：

$$F[m]=F[((m_1P_1+m_2P_2))_N]=G_2[m_3,m_4]=G_2[((m_1P_1))_{P_2},((m_2P_2))_{P_1}]$$

最後透過F與G_2在不同m_1,m_2時的點點數值對應關係， 就成功僅使用P_1與P_2點離散傅立葉變換，描述了原先的N點離散傅立葉變換。

而這個方法透過聰明的選取表達m,n的方式，使得拆解的過程中完全不需要多餘的乘法運算， 總運算量可以簡單表示為：

$$P_2B_1+P_1B_2+3D_3+2D_2$$

其中B_1是P_1傅立葉所需乘法數，B_2是P_2傅立葉所需乘法數。

雖然這個方法可以較上面的方法節省運算量， 但這個方法也牽涉較為複雜的m,n與m_1,m_2,n_1,n_2轉換，較為不直覺且不易理解， 也會遇到許多需要取餘數的運算，可能會需要較大的空間建表進行查表法。

最後關於實際上要如何求得n與n_1,n_2的轉換關係， 可以先透過輾轉相除法取得一對特定的n_{11},n_{21}使得：

$$((n_{11}P_1+n_{21}P_2))_N=1$$

然後我們可以知道對於任意整數$0\leq k<N$有：

$$((kn_{11}P_1+kn_{21}P_2))_N=k=((kn_{11}P_1-c_1N+kn_{21}P_2-c_2N))_N=(((kn_{11}-c_1P_2)P_1+(kn_{21}-c_2P_1)P_2))_N$$

然後就可以得到：

$$n_{1k}=((kn_{11}))_{P_2}\qquad n_{2k}=((kn_{21}))_{P_1}$$

滿足：

$$((n_{1k}P_1+n_{2k}P_2))_N=k\qquad 0\leq n_{1k}<P_2\qquad 0\leq n_{2k}<P_1$$

參考資料

1. 程佩青. 数字信号处理教程. 清华大学出版社. 2001. ISBN 9787900631671.
2. Jian-Jiun Ding. advanced digital signal processing lecture note, P375-387. 高等數位訊號處理課程網頁. [2022-06-16]. （原始內容存档于2020-05-08）.

1. Widhe, T., J. Melander, et al. (1997). Design of efficient radix-8 butterfly PEs for VLSI. Circuits and Systems, 1997. ISCAS '97., Proceedings of 1997 IEEE International Symposium on.
2. Lihong, J., G. Yonghong, et al. (1998). A new VLSI-oriented FFT algorithm and implementation. ASIC Conference 1998. Proceedings. Eleventh Annual IEEE International.
3. Duhamel, P. and H. Hollmann (1984). "Split-radix FFT algorithm." Electronics Letters 20(1): 14-16.
4. Vetterli, M. and P. Duhamel (1989). "Split-radix algorithms for length-pm DFT's." Acoustics, Speech and Signal Processing, IEEE Transactions on 37(1): 57-64.
5. Richards, M. A. (1988). "On hardware implementation of the split-radix FFT." Acoustics, Speech and Signal Processing, IEEE Transactions on 36(10): 1575-1581.
6. Shousheng, H. and M. Torkelson (1996). A new approach to pipeline FFT processor. Parallel Processing Symposium, 1996., Proceedings of IPPS '96, The 10th International.
7. Shousheng, H. and M. Torkelson (1998). Designing pipeline FFT processor for OFDM (de)modulation. Signals, Systems, and Electronics, 1998. ISSSE 98. 1998 URSI International Symposium on.