國立交通大學電資學院
電信工程學系碩士班
碩士論文

# 渦輪乘積碼解碼演算法之解碼效能與設計複雜度比較

# The Study of Turbo Product Decoding Algorithms and Their Performance/Complexity Evaluation

研 究 生：陳 昱 竹
指導教授：紀 翔 峰 博士

中華民國九十四年十月

# 渦輪乘積碼解碼演算法之解碼效能與設計複雜度比較

# The Study of Turbo Product Decoding Algorithms and Their Performance/Complexity Evaluation

研 究 生：陳昱竹　　Student：Iu-Zu Chen

指導教授：紀翔峰博士　　Advisor：Dr. Hsiang-Feng Chi

國 立 交 通 大 學

電信工程學系碩士班

碩 士 論 文

A Thesis

Submitted to Department of Communication Engineering

College of Electrical Engineering and Computer Science

National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of Master

in

Communication Engineering

October 2005

Hsinchu, Taiwan, Republic of China

中華民國九十四年十月

# 渦輪乘積碼解碼演算法之
# 解碼效能與設計複雜度比較

研究生: 陳昱竹　　指導教授: 紀翔峰 博士

國立交通大學

電信工程學系碩士班

## 摘　要

隨著近年來通訊技術的發展，許多新穎的無線通訊系統需要強力的通道編碼(Channel Coding)來達到低抗通道的衰減。各種通道編碼的技術中，渦輪乘積碼(Turbo Product Code)提供了優越的錯誤更正能力，故相當適合運用在無線通訊系統。在解碼方法中，最直接的方式是將線性區塊碼轉換為格子圖(trellis diagram)表示，再用最大後驗概率(Maximum A-Posteriori，MAP)演算法尋找所有狀態點的機率(State Probability)以找回其傳輸序列；另一個解碼方法是利用卻斯(Chase)演算法和最大相似度(Maximum likelihood)的觀念來找到軟式解碼的方法。除了這兩種慣用的解碼演算法，本論文中另外提出了一個利用編碼後的碼字元(code word)互相間的相關性來找到額外的資訊(Extrisic information)，再利用此資訊加以解回傳送端傳輸的序列。論文中將針對這三種解碼的方式所需的運算複雜度及其解碼效能做出比較，以求能達到針對不同硬體複雜度和解碼效能需求的應用中採用最合適的解碼技術之目的。

# The Study of Turbo Product Decoding Algorithms and Their Performance/Complexity Evaluation

Student: Iu-Zu Chen  Advisor: Dr. Hsiang-Feng Chi

Department of Communication Engineering

National Chiao Tung University

## ABSTRACT

In order to reduce the performance loss caused by channel fading, many state-of-the-art channel coding techniques have been used in modern wireless communication systems in recent years. Among these channel coding techniques, turbo product codes have good error correction capability and have been widely used in many wireless communication systems. The most direct way of decoding turbo product codes is to express the linear block codes as trellis diagrams, and then ultilize the Maximum A-Posteriori (MAP) algorithm to conduct the sequence decoding. Another method is to use Chase algorithm and utilize the maximum likelihood approach to calculate soft output for sequence decoding. In this thesis, besides the conventional decoding algorithms, we develop a decoding method in which the relationship of the data bits and the parity bits are used to calculate extrinsic information for interative decoding. We compare the complexity and the

performance of these decoding methods. The goal is to provide a guideline of choosing the suitable decoding method for applications with different complexity and performance requirements.

# 誌　謝

本篇論文得以順利完成，第一個要特別感謝的是我的指導教授紀翔峰博士。在這兩年研究過程中，紀老師給予教導了我研究的態度與方法，在有問題或瓶頸時，更適時的提供建議與方向，讓我在獨立思考和分析問題方面，有了很好的培養與指導。另外在語言表達上，老師同樣給予我充分的訓練，同時讓我學習了主動積極的研究精神和學習方法，在此致上最高的謝意。

其次，我還要感謝實驗室的同窗同學、學長和學弟，大家在課業上或生活上一起的努力奮鬥和互相切磋鼓勵，也給了我諸多的幫助，造就了這篇論文。

最後我要由衷感謝家人長久以來對我的支持與關心，也才能使我在努力求學，深入研究的過程中無後顧之憂。

民國九十四年十月

研究生陳昱竹謹識於交通大學

# 目　錄

# 圖 目 錄

# 表 目 錄



# 第一章　緒論

近年無線通訊技術、數位通訊一直不停的發展，各個系統的需求與日俱增；而未來數據傳輸的涵蓋範圍勢必更加廣泛，傳輸速率也日益加快，使得資料在傳輸過程中受到雜訊干擾影響的機率變大。這將導致接受的資料訊息無法正確的辨視，故如何降低通道的干擾，使得錯誤得以校正將越來越受到重視，所以通道編碼的方法以校正效果將成為系統設計的重要課題。

本論文的目的主要是對於一個具有優秀錯誤更正能力，並且不會太過複雜的通道編碼—渦輪乘積碼(Turbo Product Code)的實現與評估。針對此編碼方法，提出三種不同的解碼方式，以求能達到高效能解碼、硬體與效能平衡解碼或是簡單硬體需求解碼三種不同要求。

本章節將首先介紹通道編碼的發展，然後針對本論文主要討論的編碼方法，區塊碼以及區塊乘積碼做簡單的介紹，最後概述一下本論文的結構與章節。

## 1.1 研究動機

隨著數據通訊和訊號處理技術的日新月異，以及 VLSI 製程技術的快速進步，使得以往一些較難以實現的編碼方法及通訊理論得以重新被拿來應用，因而帶動了通訊系統的相關產業蓬勃發展。因此，通訊產品對於抗雜訊的能力也越來越要求，而需求高運算的編碼方式也重新受到重視，其中以渦輪碼(Turbo Code)最能提供優越的錯誤更正能力。而本論文研究的編碼-渦輪乘積碼(Turbo Product Code，TPC)則是由渦輪碼(Turbo Code)改進而來，使其不謹能有更好的錯誤更正能力，還更適合

應用在高傳輸的系統中。

在解渦輪乘積碼(TPC)的方法主要分為格子圖解碼和以卻斯(Chase)演算法延伸軟式輸出解碼為主，格子圖由於受到狀態數的影響而需要龐大的運算資料，但其有接近夏儂(Shannon)極限的好效能；卻斯演算法雖然較格子圖演算法簡單，但由於需要反複的運用硬式和軟式解碼以及大量的測試樣本，同樣有相當的複雜度；論文中將另外提出一種可以加快解碼處理速度的方法，以節省硬體負荷量，但相對的需要以較表的解碼效能做為代價。本論文會對此三種方法一一介紹，同時在論文最後會對三種方法之中做一個效能與硬體需求的比較，

## 1.2 通道編碼(Channel Coding)簡介

通道編碼(Channel Coding)的主要目的是在於抵抗通道雜訊對訊號的影響。較簡單的對抗方法是利用週期性循環檢查法(Cyclical Redundancy Check，CRC)來檢查接收資料是否有錯誤，當接受端發覺有錯誤時再要求重新發送。此種方法既占用頻寬又增加延遲。現在的通道編碼則是由傳送端將資料先行經過重新隨機排序(Randomization)、向前錯誤更正碼(Forward Error Correction Encoding，FEC encoding)、交錯(Interleaver)資料最後再經過調變(Modulation)後送出，整個過程如(圖 1.1)所示，其中以向前錯誤更正碼為主要的編碼動作。

向前錯誤更正碼(FEC)的運作方法是先將要傳送的資料進行編碼，在接收端接受後即可對編碼過的字元(Code Word)實行解碼的動作，如此即可在一定的錯誤範圍內進行即時的更正，而不需要要求傳送端再一次的傳送資料。

有了向前錯誤更正碼(FEC)的觀念後，自 1950 年由漢明(Hamming)所提出的單錯誤更正漢明碼(Single error correcting Hamming code)開始，陸續就有各式的編碼方法被提出。如 1955 年 Elias 提出的迴旋碼(Convolutional FEC Code)，和 1959 年被提出的多錯誤更正(Multiple

error correcting)碼 Bose-Chaudhuri-Hocquenghem 區塊碼(BCH block code)，以及由 Berrou、Glavieux 和 Thitimajshima 三人所提出具優越錯誤更正能力的渦輪碼(Turbo Code)。

## 1.3 區塊編碼(Block Coding)簡介

在各個向前錯誤更正(FEC)碼的方法中，區塊碼(Block Code)和迴旋碼(Convolutional Code)是較為基本的通道碼(Channel Code)，在之後發展出來的渦輪碼(Turbo Code)亦或是乘積碼(Product Code)都是取兩種方法中的一種做為基本的編碼方法。其中區塊碼和迴旋碼最大的不同在於區塊碼(Block Code)的編碼過程中，不會影響到以後的資料，也就是不像迴旋碼(Convolutional Code)一樣具有時間相關性，資料的影響是一個區塊一個區塊為主，不會有連續的影響。

區塊碼如此分段的好處主要顯現在解碼的過程中，由於其沒有時間上的連續性，故解碼時亦能夠一個區塊一個區塊的解碼，這樣就不會像是傳統迴旋碼一樣受到時間連續性的限制，而能採用數學代數的方式來解碼。使用數學代數解碼的好處在於其解碼過程較為簡單，不像傳統解迴旋碼時需要追蹤很長的一段時間的資料，才能將原本的資料還原，而解法也相對的較不複雜。

## 1.4 乘積碼(Product Code)的介紹

乘積碼(Product Code)是在 1954 年由 Elias 所提出的一種編碼方法，其想法是將兩種現有的區塊碼(Block Code)加以組合，將訊息做縱向及橫向的兩次編碼，藉此來達到對通道雜訊有更高抵抗力的編碼方式也因此其亦稱之為區塊乘積碼(Block Product Code)。

乘積碼縱橫交錯的編碼方式，跟渦輪碼(Turbo Code)在編碼上利用交錯器(interleaver)來進行兩種不同排列的訊號做編碼的方式相當的類似，不同的是乘積碼就像是用來一個簡單而單純的區塊交錯器(block

interleaver)來做渦輪碼,並且對第一次編碼之後的碼字元(Code word)同樣的再進行編，詳細的情形會在第二章加以說明。也由於乘積碼有著和渦輪碼同樣的交錯特性，故解碼時亦可用解渦輪碼的反覆解碼觀念來解乘積碼，以求能達到更好的解碼效果。

## 1.5 論文組織

本論文在第一章中主要是在介紹通道編碼的發展情形，並了解研究主題中主要使用的編碼方法，藉此說明研究題目的動機和目標。在第二章中，主要是簡介區塊乘積碼系統及其設計架構，以了解區塊乘積碼的編碼方法，如此才能針對乘積碼進行解碼。第三、四、五章便是解碼的部分。在這三個章節中，我們將介紹三種以不同演算法為基礎的解碼方式。第三章是利用格子圖(Trellis Diagram)的觀念來進行解碼，使用的是最大後驗概率(Maximum A-Posteriori，MAP)演算法來進行解碼。第四章則是以卻斯(Chase)演算法為主，配合者硬式的解碼方法，再以 Chase 演算法來達到軟式解碼，以達到更好的解碼效果。第五章是為了達到低複雜度而設計的解碼方法，其主要是利用 BCH 編碼後其碼字元(Code Word)是由資料做互斥運算得來的這個關係(此關係可以從同位檢查矩陣(parity-check matrix)中得到)，運用數學的方法將額外的資訊(Extrinsic information)計算出來再加以解碼。在講解完各個解碼方法後，將在第六章中說明這些方法解碼出來的模擬解果，以及將其設計為硬體時的運算需求考量。最後在第七章中提出本論文的結論。

# 第二章　BCH 乘積碼系統

第一章中，我們已經大略的介紹了各式的通道編碼，而本論文所探討的編碼方法則是以 BCH 乘積碼(BCH Product Code)為主，如何能找到一個低複雜度的解碼方法正是本論文的目的。

在本章節中，將會介紹 BCH 乘積碼的編碼系統，以至於其如何達到反覆解碼運作的基本解碼器。開頭我們會先介紹 BCH 乘積碼中的組成碼(Component Code)-BCH 碼的編碼，之後再解說乘積碼是如何從問題的起因開始說起，接著是演算法的推導，而最後是硬體的設計與實現，這些都會在本章節中有著詳細的解說。

## 2.1 延伸 BCH 碼(Extendd BCH Codes)之編碼演算法

在乘積碼的編碼中，其編碼方法主要為將資料分區塊進行縱向及橫向的編碼。因此，其基本的組成碼(Component Code)必須是區塊編碼(Block Code)才行，而區塊編碼中，則是以系統 BCH 碼所需要的複雜度較低，同時能擁有相當的糾錯能力。故我們採取了系統 BCH 碼做為乘積碼的組合碼，所以在瞭解乘積碼的運作情形之前，我們要先行探討 BCH 碼的編碼方法。

### 2.1.1 有限元素場(Finite Field)

BCH 碼是運作在有限元素場裡面，所以要瞭解 BCH 碼的基本知識，必須先要有限元素場的概念。有限元素場是由 Evariste Galois 所發現提出的，故其亦稱之為 Galois field。有限元素場的記號為 GF(q)，其中 q 為此有限元素場中的元素個數。

### 2.1.1.1 有限元素場與多項式的結合

一個元素場(Field，F)代表一個集合上定義了加法和乘法兩種運算，並且需滿足下列 3 個公設：

1. 此集合加法運算滿足交換性
2. 此集合的乘法運算要有封閉性，且所有非零元素在乘法運算下形成一個具交換性的體。
3. 在元素場 F 裡的元素滿足分配性，如 F 裡的 a,b,c 三元素需滿足 (a + b) c = ac + bc 的運算。

在建立有限元素場時，我們可以利用多項式來建構。在 GF(q)上多項式的數學表示法為

$$f(x) = f_{n-1}x^{n-1} + f_{n-2}x^{n-2} + \cdots + f_0 \qquad (2.1\text{-}1)$$

其中 $f_{n-1}, f_{n-2}, \ldots, f_0$ 屬於 GF(q)。若在一個 GF(q)中所有的多項式形成了一個多項式循環體(polynomial ring)，且其多項式的加法和乘法運算如同一般多項式的運算，我們將此多項式循環體記為 GF(q)[x]。

若給定一個多項式循環體 F[x]，我們可以很容易在 F[x]上建立一個商數循環體(quotient ring)。還定任意的非零次多項式(nonzero degree) p(x)為模數(modulus)，則將 F[x]對 p(x)做模數運算可得一商數循環體，記為 F[x]/(p(x))，此循環體除了不一定符合公設 2 的條件之外，對於元素場的其他條件都是符合的。

當我們選定的p(x)為一個質多項式(prime polynomial)，也就是在F[x]上為不可分解多項式(irreducible polynomial)時，則F[x]的商數多項式F[x]/(p(x))將會形成一個有限元素場。因此，我們可以在GF(q)上選定一個n次質多項式，則可以建立一個具有$q^n$個元素的有限元素場。在這元素場裡的元素是以GF(q)上次數低於n的多項式來表示，其中共有$q^n$個不同的多項式，因此能建構一個元素場GF($q^n$)。

例如要建立一個 GF(4)的元素場時，我們可以從 GF(2)上的多項式來建立，首先在 GF(2)上還擇一個 2 次的質多項式 $p(x)=x^2+x+1$，則可形成一個含{0，1，x，x+1}四個元素的有限元素場 GF(4)。

### 2.1.1.2 最小多項式(minimal polynomial)

若GF($q^m$)為GF(q)的擴展場，則我們稱為在GF($q^m$)中的下列元素

$$\left\{\beta,\beta^q,\beta^{q^2},\beta^{q^3},\cdots,\beta^{q^{r-1}}\right\}$$

為相對於GF(q)上的共軛元素(conjugates)。而以這些共軛元素為根的多項式稱為最小多項式。所以要得到GF($q^m$)中某元素$\beta$的最小多項式就取其共軛元素為根所構成之多項式，其表示如下：

$$f(x)=(x-\beta)(x-\beta^q)\cdots(x-\beta^{q^{r-1}}) \qquad (2.1\text{-}2)$$

最小多項式的特性是其多項式係數都是在 GF(q)中，例如取$\alpha$ 為 GF(256)的基本元素，其相對於 GF(2)的共軛元素為

$$\left\{\alpha,\alpha^2,\alpha^4,\alpha^8,\alpha^{16},\alpha^{32},\alpha^{64},\alpha^{128}\right\}$$

則$\alpha$ 的最小多項式為

$$f(x)=(x-\alpha)(x-\alpha^2)(x-\alpha^4)(x-\alpha^8)(x-\alpha^{16})(x-\alpha^{32})(x-\alpha^{64})(x-\alpha^{128})$$

若將上式乘開，可發現其多項式的係數將會屬於 GF(2)之上。

## 2.1.2 BCH 碼的編碼

有了元素場的基本觀念後，我們就較容易瞭解 BCH 碼的運作方法，BCH 碼是由 Hocquenghem 所發現提出的，之後在 1959 和 1960 年由 Bose 和 Chaudhuri 整理發展出來，而 BCH 碼編碼方式可分為系統(systematic)與非系統(non-systematic)兩類，系統的編碼是在經過編碼後，資料位元(information bits)仍會成為輸出的編碼位元(coded bits)中的一部分，非系統的編碼則不是如此。由於系統編碼會有較好的抵抗能力，故較常被各界使用，也因此，我們只討論系統編碼。

### 2.1.2.1 BCH 碼的參數

若一個BCH碼是接收k個資料位元後將其轉為n個碼位元的編碼方法，而最小的漢明距離(Hamming distance)為$d_{min}$，這樣的一個BCH碼我們將其標示為BCH(n，k，$d_{min}$)。而$d_{min}$的大小決定了此BCH碼可更正的錯誤符元數目。讓t等於可更正錯誤符元數目的大小，則我們可以得到$t=\left\lfloor \frac{d_{\min}-1}{2} \right\rfloor$的關係。

### 2.1.2.2 BCH 碼的生產多項式

BCH碼是由訊息經過生產多項式(Generator Polynomial)的編碼後產生的。在BCH碼中，生產多項式是由最小多項式(Minimal Polynomail)組合而成。若指定$\alpha$為GF($2^m$)中的基本元素(Primitive element)，令$\phi_{a^i}(D)$為$\alpha^i$的最小多項式，$i=1,2,...,2t$，則對於能更正t個錯誤符元的BCH碼之產生多項式g(D)定義為

$$g(D)=LCM\left(\phi_{\alpha^1}(D),\phi_{\alpha^2}(D),\cdots,\phi_{\alpha^{2t}}(D)\right) \quad (2.1\text{-}3)$$

其中，LCM 表示最小公倍式(Least Common Multiple)。也因此，$\alpha^i$必定是g(D)中的一個根。

系統化的編碼方法是將原本輸入的k次多項式資料先行提高至n次多項式，再經由產生多項式做出一個商數多項式，而此多項式即表現出編碼後的結果。其編碼的過程可由(式2.1-4)來表示，在式中，x(D)為碼多項式(Code Polynomial)、u(D)則是資料多項式(Information Polynomial)。

$$x(D)=D^{N-K}u(D)+D^{N-K}u(D)\bmod g(D) \quad (2.1\text{-}4)$$

從式中可以看出碼多項式一定會被生產多項式整除，因此$\alpha^i$同樣會成為碼多項式的根，而這樣的關係就可以應用在解碼的處理上。

### 2.1.2.3 BCH 碼的同位檢查矩陣(Parity-Check Matrix)

若我們有一個碼多項式 $x(D)=x_{N-1}D^{N-1}+x_{N-2}D^{N-2}+\cdots+x_1D+x_0$ 且 $\alpha^i$ 是其生產多項式 g(D)的根，因為碼多項式是生產多項式的倍式，所以我們知道 $\alpha^i$ 也將是碼多項式的根。即

$$x(\alpha^i)=x_{N-1}\alpha^{(N-1)i}+x_{N-2}\alpha^{(N-2)i}+\cdots+x_1\alpha^i+x_0=0,\qquad 1\le i\le 2t \qquad (2.1\text{-}5)$$

若將上述 2t 個式子轉為矩陣的形態，我們可以得到

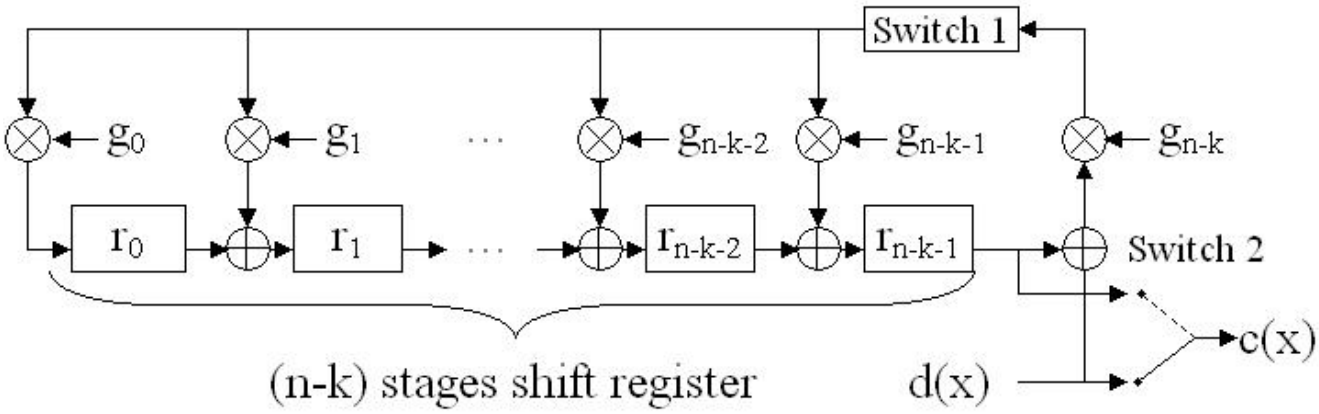
$$\begin{bmatrix} x_{N-1} & x_{N-2} & \cdots & x_0 \end{bmatrix}\begin{bmatrix} \alpha^{(N-1)} & \alpha^{(N-2)} & \cdots & 1 \\ \alpha^{2(N-1)} & \alpha^{2(N-2)} & \cdots & 1 \\ \alpha^{3(N-1)} & \alpha^{3(N-2)} & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ \alpha^{2t(N-1)} & \alpha^{2t(N-2)} & \cdots & 1 \end{bmatrix}=0 \qquad (2.1\text{-}6)$$

上面的式子我們可以看成 $xH^T=0$，其中 H 就稱之為同位檢查矩陣(Parity-Check Matrix)。由於BCH碼是作用在GF($2^m$)之中，我們可以得知 $\alpha^i$ 和 $(\alpha^i)^2$ 是同一個最小多項式的根，也就是說當 $x(\alpha^i)=0$ 時，$x[(\alpha^i)^2]=x(\alpha^{2i})=0$。所以我們可以將同位檢查矩陣簡化為

$$H=\begin{bmatrix} \alpha^{(N-1)} & \alpha^{(N-2)} & \cdots & \alpha & 1 \\ \alpha^{3(N-1)} & \alpha^{3(N-2)} & \cdots & \alpha^3 & 1 \\ \alpha^{5(N-1)} & \alpha^{5(N-2)} & \cdots & \alpha^5 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \alpha^{(2t-1)(N-1)} & \alpha^{(2t-1)(N-2)} & \cdots & \alpha^{(2t-1)} & 1 \end{bmatrix} \qquad (2.1\text{-}7)$$

### 2.1.3 BCH 碼的硬體架構

BCH 碼的編碼器架構是和環狀碼相當類似的，所以在設計上我們可以利用多個階段(stage) 的位移暫存器( Shift register)來組成編碼器，如(圖 2.1-1)所示




圖(2.1-1) BCH 碼編碼器

在圖中，前k個資料傳送時開關1(switch 1)是連通(close)的，而開關2(switch 2)則是在調到和下方的輸入資料(information data)d(x)相連接。如此即連產生而輸入資料相同的k個輸出；在k個資料的位移後，開關1則轉為斷路(open)，開關2也轉為連接最後一個位移暫存器，以藉此將之後的(n-k)個編碼符元輸出。利用如此的架構，只要決定好生產多項式，將生產多項式的係數填入$g_0, g_1, \cdots, g_k$則可以利用此編碼器來完成系統BCH編碼。

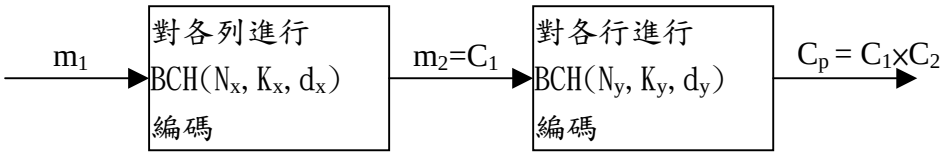
### 2.1.4 延伸 BCH 碼

在本論文中，我們所採用的組合碼是延伸 BCH 碼(Extended BCH Codes)，所謂的延伸 BCH 碼和原本的 BCH 碼最大的不同在於延伸 BCH 碼是原本的 BCH 碼完成之後，在補上一個同位檢查位元(parity check bit)，而這樣的一個位元可以使得及漢明距離得以加 1。即原本的 BCH (N，K，d)碼在加上一個同位檢查位元後可成為 eBCH(N+1，K，d+1)的碼，這樣在乘積碼之中，可以使其更正錯誤能力再加強，而我們只需要附出一個位元的代價。

## 2.2 BCH 乘積碼的架構及編碼器

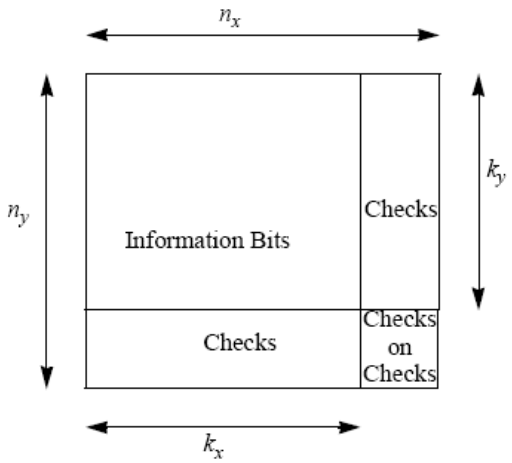
在瞭解了延伸 BCH 碼的編碼方法後，我們接著將介紹如何利用組成碼-BCH 碼來做縱向及橫向編碼以達到 BCH 乘積碼的編碼。另外，乘積碼可以利用不同碼率的組成碼結合來達成不同碼率及更正能力的編碼，這種設計可以輕易而有效率的編成長度較長的碼。

### 2.2.1 BCH 乘積碼的編碼




圖(2.2-1) 乘積編碼過程圖

在進行乘積碼的編碼時，其過程如(圖 2.2-1)先決定好所要進行編碼的組成碼BCH($N_x$, $K_x$, $d_x$)及BCH($N_y$, $K_y$, $d_y$)，然後將資料分成$K_x$×$K_y$大小的區塊$m_1$，此時再對$m_1$第一列的資料進行BCH($N_x$, $K_x$, $d_x$)編碼的動作，接著依序完成$K_y$行的編碼，得到$m_2$。經過第一段的編碼後，區塊大小應為$N_x$×$K_y$。第二段則是將第一段完成的編碼$m_2$第一行的資料當做輸入，進行BCH($N_y$, $K_y$, $d_y$)的編碼動作，從第一行一直做到$N_x$行為止，這樣就可以產生一個$N_x$×$N_y$經過編碼的區塊，如(圖 2.2-2)所示。此即為乘積碼的編碼過程。




圖(2.2-2) 乘積編碼架構圖

### 2.2.2 BCH 乘積碼的性質

乘積碼的更錯能力及字碼長度和其組成碼相關，舉例來說，若我們選用$C_1$=BCH($N_x$，$K_x$，$d_x$)與$C_2$=BCH($N_y$，$K_y$，$d_y$)為乘積碼CP的兩個組合碼，即$C_P=C_1\times C_2$。則我們可以得到$C_P$的長度將為$N_x\times N_y$個位元，最小漢明距離則為$d_x\times d_y$，所以$C_P$能夠更正$\left\lfloor \frac{d_x\times d_y-1}{2} \right\rfloor$個錯誤位元，而其基本碼率為$R=\frac{K_x\times K_y}{N_x\times N_y}$。

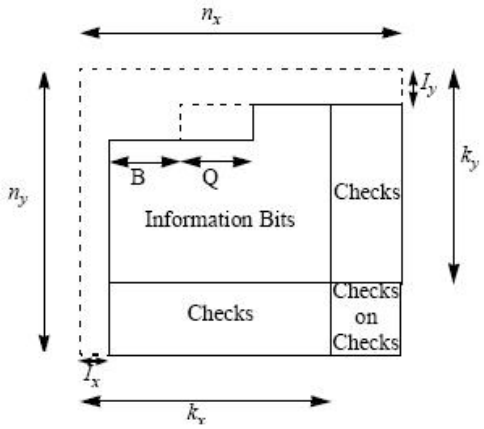
乘積碼的長度可以有很大的變動空間，其不僅可以利用組成碼的選擇來決定基本的長度外，更能以補零、省去的方法來調整碼率，以此來達到各個要求的區塊長度。如(圖 2.2-3)所示，我們可以利用 3 個步驟將資料調整到所要的長度。

1. 將 2 維方陣中的$I_x$行和$I_y$列的資料填零後編碼，如此即可在編碼後將$I_x$行和$I_y$列的資料移去。
2. 將省去行列後的區塊中第一列的B+Q個位元填零，則其編碼前的B+Q個位元都是無效的，所以實際上進行編碼的資料長度為($K_x$-$I_x$) × ($K_y$-$I_y$)-(B+Q)個位元。
3. 通常在設計上總希望能以一個Byte為單位，故在輸出和輸入上很難同時減少一定位元達到兩邊都是 8 的倍數的值。因此，我們需要Q這個值來表示輸入部分多捨去的位元數，也就是說，編碼後的輸出資料中，第一列的前Q筆資料是 0 即輸出的長度為($N_x$-$I_x$)($N_y$-$I_y$)-B個位元。

利用 Ix，Iy，B，Q 這四個值來控制位元長度，如此即可達到我們所需要的編碼長度，而此時的碼率將變為

$$R=\frac{(K_x-I_x)(K_y-I_y)-B-Q}{(N_x-I_x)(N_y-I_y)-B} \qquad (2.2\text{-}1)$$

其編碼後的情形如（圖 2.2-3）所示，而在(表 2.2-1)中，我們列出了在 IEEE 802.16a 中所使用到的一些編碼長度：




圖(2.2-3) 乘積碼調節長度運作圖

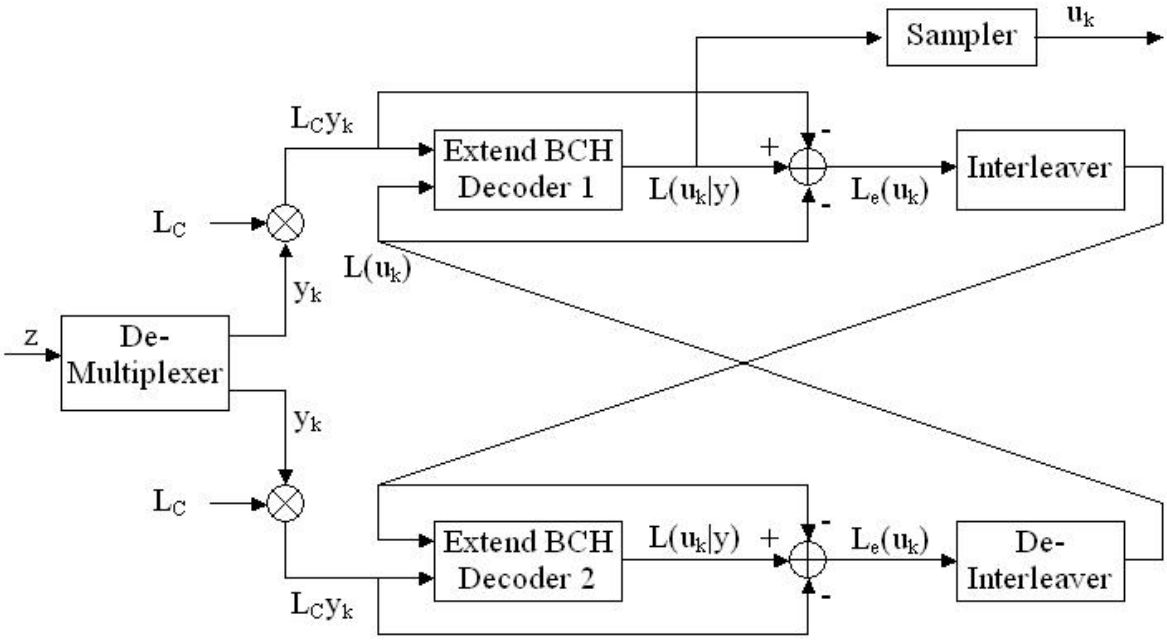
| Modulation | Data Block Size (Bytes) | Coded Block Size (Bytes) | Overall Code Rate | Efficiency (bit/s/Hz) | Constituent Codes | Code Parameter |
|---|---|---|---|---|---|---|
| QPSK | 23 | 48 | ~1/2 | 1.0 | (32,26)(16,11) | $I_x$=4,$I_y$=2,B=8,Q=6 |
| QPSK | 35 | 48 | ~3/4 | 1.5 | (32,26)(16,15) | $I_x$=0, $I_y$=4,B=0,Q=6 |
| 16 QAM | 58 | 96 | ~3/5 | 2.4 | (32,26)(32,26) | $I_x$=0, $I_y$=8, B=0,Q=4 |
| 16 QAM | 77 | 96 | ~4/5 | 3.3 | (64,57)(16,15) | $I_x$=7, $I_y$=2, B=30,Q=4 |
| 64 QAM | 96 | 144 | ~2/3 | 3.8 | (64,63)(32,26) | $I_x$=3, $I_y$=13, B=7,Q=5 |
| 64 QAM | 120 | 144 | ~5/6 | 5.0 | (32,31)(64,57) | $I_x$=13, $I_y$=3, B=7,Q=5 |

表(2.2-1) IEEE 802.16a 取用編碼長度表

## 2.3 乘積碼的解碼器簡介

乘積碼和渦輪碼的編碼方法相當的類似，不同處在於乘積碼的組成碼必須是區塊碼，同時其交錯器(interleaver)必須是區塊交錯器(block interleaver)。因此，其解碼器的架構和渦輪碼解碼器架構大略相同。

乘積碼解碼器的典型架構如(圖 2.3-1)所示，主要由兩個軟式輸入輸出(Soft in Soft Out，SISO)的組成碼解碼器及反向交錯器(de-interleaver)所構成，主要是利用遞迴(interation)的觀念來進行解碼，把組成碼 eBCH 碼的解碼器所輸出之位元資料進行反覆解碼，進而達到減少解碼後位元錯誤機率(Bit Error Rate，BER)的目的。




圖(2.3-1) 乘積碼的典型硬體架構圖

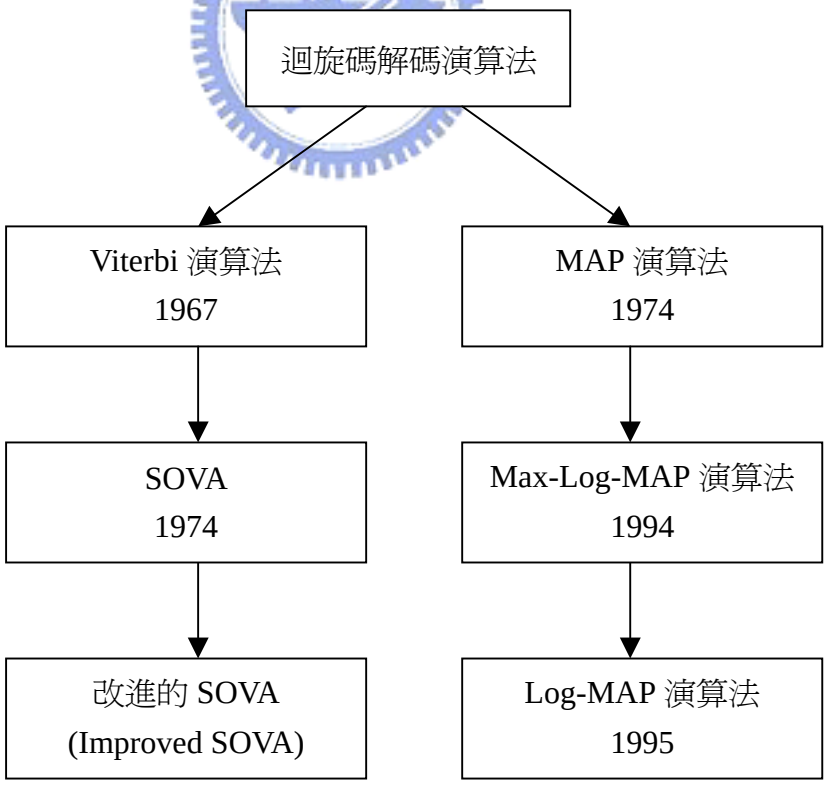
如圖所示，乘積碼的運作是先行將資料分為縱向及橫向的資料$y_K$，在乘上通道的預估大小$L_C$得到$L_C y_K$，再將此資料配合另一方向解碼後提供的資料$L(U_k)$一起進行解碼的動作，將解碼後得到的資訊$L(U_k|y)$去除原本已知的通道訊息$L_C y_K$及事前資訊$L(U_k)$即可取得由解碼動作得來的額外資訊$L_e(U_k)$，再經過交錯器使其排列由縱向變為橫向、橫向改為縱向後，送到另一個解碼器做為事前已知的本質訊息(a-priori information)。如此反覆解碼，在經過一定次數的遞迴後即將解碼後的結果輸出，經過比較器來還原原本訊息。

# 第三章　以格子圖(Trellis Diagram)的觀念來解碼

接下來的三個章節都是用來解碼的方法，其主要是以討論組成碼-延伸 BCH 碼的解碼情形，最後在看其在乘積碼的運作下是否合宜。本章節將討論的是最常用來解渦輪碼的方法，利用格子圖的觀念來解碼。

以格子圖來解碼的方法主要有兩種，一種是軟式輸出維特比演算法(soft-output viterbi algorithm，SOVA) 另一種則是我們本章要討論的重點，最大後驗概率(Maximum-a-posteriori，MAP)演算法。

這兩種演算法的歷史如下(圖 3-1)所示：




圖(3-1) 格子圖觀點演算法的演進

## 3.1 最大後驗概率（Maximum-a-posteriori，MAP）演算法

在這節裡，我們將描述最大後驗概率的運算方法，藉此了解在實際做解碼動作時，使用此演算法所能得到的效益以及必須附出的硬體複雜度。

### 3.1.1 對數相似比(Log Likelihood Ratios，LLR)

假設傳輸的未編碼資料位元為$u_k$，則對數相似比(LLR)$L(u_k)$的定義為對$u_k$的兩種可能值(±1)發生機率的比值之對數值

$$L(u_k) \equiv \ln\left(\frac{P(u_k=+1)}{P(u_k=-1)}\right) \qquad (3.1\text{-}1)$$

其中$P(u_k=\pm 1)$為資料位元$u_k$值為的機率。

令輸入資料$u_k$編碼後，將其傳輸到接收端，中間經過通道的衰減後，所接收到的資料序列(received data sequence)為$\underline{y}$。則我們可以定義出條件對數相似比(conditional LLR)

$$L(u_k|\underline{y}) \equiv \ln\left(\frac{P(u_k=+1|\underline{y})}{P(u_k=-1|\underline{y})}\right) \qquad (3.1\text{-}2)$$

其中條件機率$P(u_k=\pm 1|\underline{y})$即為經過解碼後的資料位元$u_k$的後驗機率(a-posteriori probabilities)。

MAP解碼法的目的即是在得知接收到的資料序列$\underline{y}$值的條件之下，找到各個位元解碼後的序列並且提供各位元值為±1的機率。而且其要盡量使得位元錯誤率(BER)達到最小。這表示MAP演算法中主要要能找到並算出後驗對數相似比(a-posteriori LLR)$L(u_k|y)$的值。

利用貝爾定律(Bayes' rule )中所知的兩個條件：

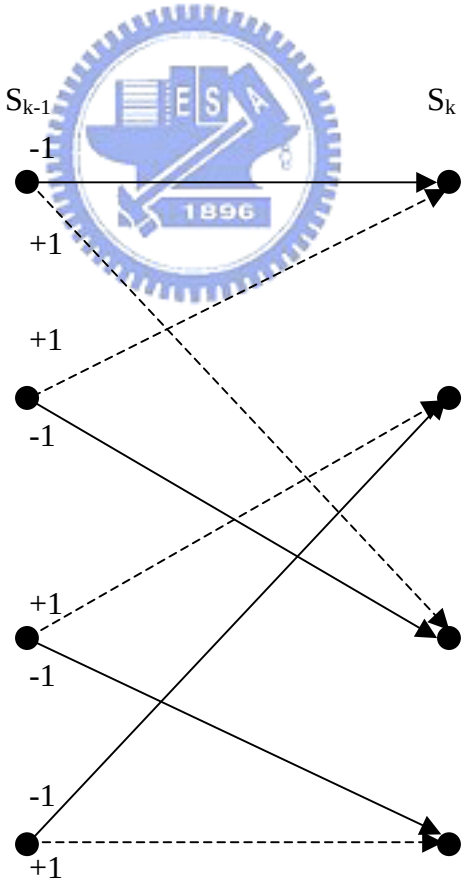
$$P(a \wedge b) = P(a|b) \cdot P(b) \qquad (3.1\text{-}3)$$

$$P(\{a \wedge b\}|c) = P(a|\{b \wedge c\}) \cdot P(b|c) \qquad (3.1\text{-}4)$$

我們可以將後驗對數相似比的值重寫為：

$$L(u_k|\underline{y}) \equiv \ln\left(\frac{P(u_k = +1 \wedge \underline{y})}{P(u_k = -1 \wedge \underline{y})}\right) \qquad (3.1\text{-}5)$$

接著我們從試著找出資料位元$u_k$和格子圖中的狀態(state)的關係，這裡以簡單的K=3 RSC編碼為例。圖(3.1-1)為K=3 RSC編碼的所對應格子圖的一部分。




圖(3.1-1) K=3 RSC 編碼的部分格子圖

當現在所處的狀態$S_k = s$以及前一個狀態$S_{k-1} = s'$被知道時，我們就能從狀態的變化得知輸入位元$u_k$的值。由於每個輸入位元$u_k$的值是彼此獨立變化的，所以我們可以利用相加的方法，將各個狀態變化中，使得$u_k$值為+1 或-1 的機率相加起來，以求得輸入位元$u_k$的後驗機率。式(3.1-5)可改寫為：

$$L\left(u_k \middle| \underline{y}\right) \equiv \ln\left(\frac{\sum_{\substack{(s',s)\Rightarrow \\ u_k=+1}} P\left(S_{k-1}=s' \wedge S_k = s \wedge \underline{y}\right)}{\sum_{\substack{(s',s)\Rightarrow \\ u_k=-1}} P\left(S_{k-1}=s' \wedge S_k = s \wedge \underline{y}\right)}\right) \qquad (3.1\text{-}6)$$

上式中$(s',s)\Rightarrow u_k = +1$是指狀態變化的集合，在此集合裡，從狀態$s'$進入到狀態$s$是由於輸入位元$u_k$=+1 的關係。同樣的，$(s',s)\Rightarrow u_k = -1$則是指$u_k$=-1 時的集合。我們將$P\left(S_{k-1}=s' \wedge S_k = s \wedge \underline{y}\right)$簡化記為$P\left(s' \wedge s \wedge \underline{y}\right)$。

接著利用通道的無記憶性(memoryless)以及貝爾定律(Bayes' rule)，我們可以將$P\left(s' \wedge s \wedge \underline{y}\right)$重新整理如下

$$\begin{aligned}
P\left(s' \wedge s \wedge \underline{y}\right) &= P\left(s' \wedge s \wedge \underline{y}_{j<k} \wedge \underline{y}_k \wedge \underline{y}_{j>k}\right) \\
&= P\left(s' \wedge s \wedge \underline{y}_{j<k} \wedge \underline{y}_k\right) \cdot P\left(\underline{y}_{j>k} \middle| \left\{s' \wedge s \wedge \underline{y}_{j<k} \wedge \underline{y}_k\right\}\right) \\
&= P\left(s' \wedge s \wedge \underline{y}_{j<k} \wedge \underline{y}_k\right) \cdot P\left(\underline{y}_{j>k} \middle| s\right) \\
&= P\left(s' \wedge \underline{y}_{j<k}\right) \cdot P\left(\left\{\underline{y}_k \wedge s\right\} \middle| \left\{s' \wedge \underline{y}_{j<k}\right\}\right) \cdot P\left(\underline{y}_{j>k} \middle| s\right) \\
&= P\left(s' \wedge \underline{y}_{j<k}\right) \cdot P\left(\left\{\underline{y}_k \wedge s\right\} \middle| s'\right) \cdot P\left(\underline{y}_{j>k} \middle| s\right) \\
&= \alpha_{k-1}(s') \cdot \gamma_k(s',s) \cdot \beta_k(s) \qquad (3.1\text{-}7)
\end{aligned}$$

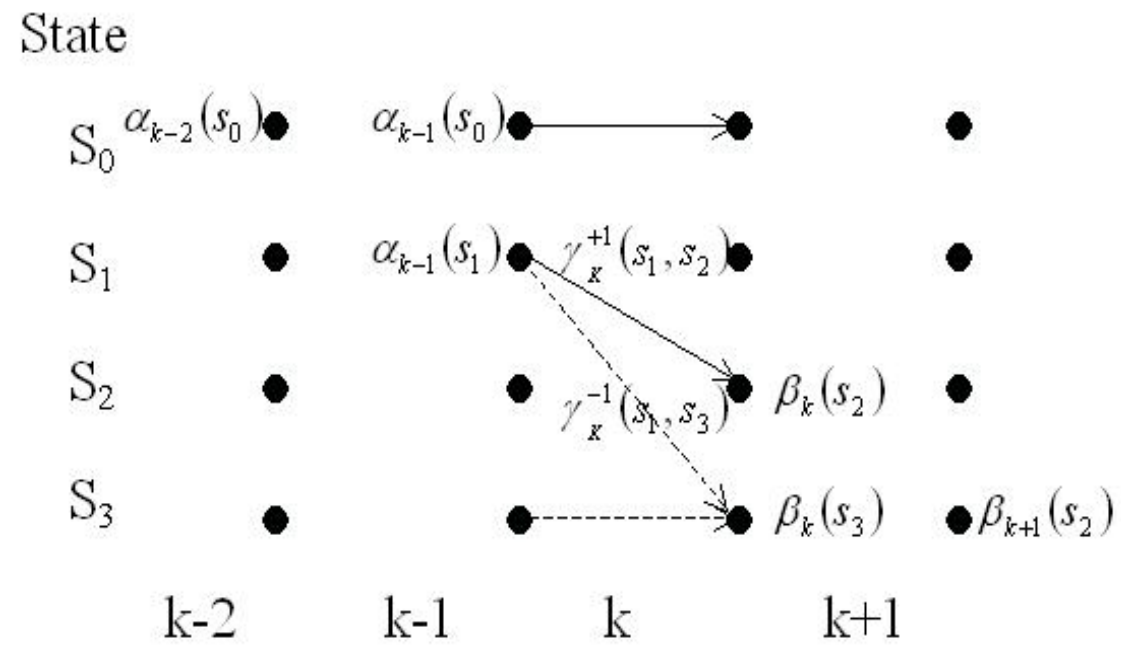
在式(3.1-7)中，我們訂出了$\alpha_{k-1}(s')$、$\beta_k(s)$、$\gamma_k(s',s)$，其和各個狀態的關係如下：

$$\alpha_{k-1}(s') = P\left(S_{k-1} = s' \wedge \underline{y}_{j<k}\right) \qquad (3.1\text{-}8)$$

$$\beta_k(s) = P\left(\underline{y}_{j>k} | S_k = s\right) \qquad (3.1\text{-}9)$$

$$\gamma_k(s',s) = P\left(\left\{\underline{y}_k \wedge S_k = s\right\} | S_{k-1} = s'\right) \qquad (3.1\text{-}10)$$

如(圖 3.1-2)所示，$\alpha_{k-1}(s')$表示著在時間 k 點時，從頭進行到狀態 s′發生的機率，而$\beta_k(s)$則表示著時間 k 點時，由結尾推算回來，接著會進入狀態 s 的機率。$\gamma_k(s',s)$則是代表有時間 k 點時，從狀態 s′進入到狀態 s 的機率。只要能求得$\alpha_{k-1}(s')$、$\beta_k(s)$、$\gamma_k(s',s)$的值，就可以得到後驗機率值。所以接下來我們將描述如何求得這三個值。




圖(3.1-2) MAP 演算法在格子圖中的表現

### 3.1.2 計算路徑值$\gamma_k(s',s)$

利用貝爾定律以及式(3.1-10)中提到$\gamma_k(s',s)$的定義，我們可以將路徑值寫成：

$$\begin{aligned}\gamma_k(s',s) &= P\left(\{\underline{y}_k \wedge s\}|s'\right)\\ &= P\left(\underline{y}_k|\{s'\wedge s\}\right)\cdot P\left(s|s'\right)\\ &= P\left(\underline{y}_k|\{s'\wedge s\}\right)\cdot P(u_k)\\ &= P\left(\underline{y}_k|\underline{x}_k\right)\cdot P(u_k) \qquad (3.1\text{-}11)\end{aligned}$$

式中$u_k$是會讓狀態$S_{k-1}=s'$進入到狀態$S_k=s$的資料位元，而$P(u_k)$即表示資料位元$u_k$的事前機率(a-prior probability)；$\underline{x}_k$則是由$u_k$編碼後相對應的輸入序列。由於BCH的編碼中，一個狀態進入到另一個狀態時所影響的輸入或多出資料都只有一位元，所以其相對應的$\underline{x}_k$、$\underline{y}_k$都只是一位元的資料，可記為$x_k$、$y_k$。

假設訊號所經過的通道是高斯通道(Gaussian channel)並以二階相移鍵控調變(BPSK)傳輸資料時，我們可以得到$P(u_k)$以及$P\left(\underline{y}_k|\underline{x}_k\right)$的值。

$$\begin{aligned}\gamma_k(s',s) &= P(u_k)\cdot P\left(y_k|x_k\right)\\ &= C\cdot e^{(u_k L(u_k)/2)}\cdot\exp\left(\frac{E_b}{2\sigma^2}2a(y_k\cdot x_k)\right)\\ &= C\cdot e^{(u_k L(u_k)/2)}\cdot\exp\left(\frac{L_C}{2}(y_k\cdot x_k)\right)\\ &= C\cdot e^{(u_k L(u_k)/2)}\cdot\exp\left(\frac{L_C}{2}(y_k\cdot u_k)\right) \qquad (3.1\text{-}12)\end{aligned}$$

上列的最後一個等式是基於系統BCH編碼中，前n個位元的編碼是輸入和輸出相同的，也就是有這$x_k=u_k$的關係。若我們只針對有需要解碼的位元解碼，則可以得到此關係而將路徑值$\gamma_k(s',s)$寫成上式(3.1-12)最後的

型式。另外，在式中C表示計算中不會因為輸入位元$u_k$的正負值或是編碼後資料$x_k$的值改變的常數項，而$L_C$則是跟通道有關的一個值，其定義為：

$$L_C = \frac{E_b}{2\sigma^2} \cdot 4a \qquad (3.1\text{-}13)$$

式(3.1-13)中的$E_b$是傳送每位元的能量大小，$\sigma^2$是雜訊變異數(noise variance)而$a$則是通道衰減大小(fading amplitude)(在沒有衰減的可加性高斯白雜訊通道(non-fading AWGN channels)其值為1)。

由式(3.1-12)中可看出，只要對傳輸的通道性質了解以得到$L_C$的值，並得到事前的發生機率L(u)值，我們就可以計算出路徑值。

### 3.1.3 遞迴計算求出前行狀態值$\alpha_k(s)$及逆行狀態值$\beta_k(s)$

同樣利用貝爾定律以及通道的無記憶性可得：

$$\begin{aligned}
\alpha_k(s) &= P\left(S_k = s \wedge \underline{y}_{j<k+1}\right) \\
&= P\left(s \wedge \underline{y}_{j<k} \wedge \underline{y}_k\right) \\
&= \sum_{all\ s'} P\left(s \wedge s' \wedge \underline{y}_{j<k} \wedge \underline{y}_k\right) \\
&= \sum_{all\ s'} P\left(\left\{s \wedge \underline{y}_k\right\} \middle| \left\{s' \wedge \underline{y}_{j<k}\right\}\right) \cdot P\left(s' \wedge \underline{y}_{j<k}\right) \\
&= \sum_{all\ s'} P\left(\left\{s \wedge \underline{y}_k\right\} \middle| s'\right) \cdot P\left(s' \wedge \underline{y}_{j<k}\right) \\
&= \sum_{all\ s'} \gamma_k(s', s) \cdot \alpha_{k-1}(s') \qquad (3.1\text{-}14)
\end{aligned}$$

由式中可以看出，只要我們有路徑值$\gamma_k(s', s)$以及初始值，就可以利用遞迴計算從$\alpha_0(s)$算出所有的$\alpha_k(s)$值。在 BCH 的編碼中，初始的狀態是有給定的，其階是從狀態 0($S_0 = 0$)開始出發，故我們可以得到初始值：

$$\alpha_0(S_0=0) = 1$$

$$\alpha_0(S_0=s) = 0 \qquad \text{for all } s\neq 0 \qquad (3.1\text{-}15)$$

$\beta_k(s)$的計算方法和之前所述$\alpha_k(s)$值相當的相似：

$$\begin{aligned}
\beta_{k-1}(s') &= P\left(\underline{y}_{j>k-1}\middle|s'\right) \\
&= \sum_{all\ s'} P\left(\left\{\underline{y}_{j>k-1}\wedge s\right\}\middle|s'\right) \\
&= \sum_{all\ s'} P\left(\left\{\underline{y}_k\wedge\underline{y}_{j>k}\wedge s\right\}\middle|s'\right) \\
&= \sum_{all\ s'} P\left(\underline{y}_{j>k}\middle|\left\{s'\wedge s\wedge\underline{y}_k\right\}\right)\cdot P\left(\left\{\underline{y}_k\wedge s\right\}\middle|s'\right) \\
&= \sum_{all\ s'} P\left(\underline{y}_{j>k}\middle|s\right)\cdot P\left(\left\{\underline{y}_k\wedge s\right\}\middle|s'\right) \\
&= \sum_{all\ s'}\beta_k(s)\cdot\gamma_k(s',s) \qquad (3.1\text{-}16)
\end{aligned}$$

由於 BCH 的編碼同樣的有終結狀態(terminated state)的設定，故其初始條件同樣可以得到：

$$\beta_N(S_N=0) = 1$$

$$\beta_N(S_N=s) = 0 \qquad \text{for all } s\neq 0 \qquad (3.1\text{-}17)$$

只要計算出所有的路徑值，我們就可以利用遞迴計算的方法將各個狀態的$\alpha_k(s)$和$\beta_k(s)$值計算出來。

### 3.1.4 整體計算方法及結論

在了解了各個機率值的計算方法後，我們可以將式(3.1-6)改寫為：

$$L\left(u_k\middle|\underline{y}\right)=\ln\left(\frac{\sum_{\substack{(s',s)\Rightarrow\\ u_k=+1}} a_{k-1}(s')\cdot\gamma_k(s',s)\cdot\beta_k(s)}{\sum_{\substack{(s',s)\Rightarrow\\ u_k=-1}} a_{k-1}(s')\cdot\gamma_k(s',s)\cdot\beta_k(s)}\right) \qquad (3.1\text{-}18)$$

而這個條件機率就是 MAP 演算法中要找出來的值。

將式(3.1-12) $\gamma_k(s',s)$的值代入(3.1-18)式中，再將常數項消去，我們可以得到：

$$L(u_k|\underline{y}) = \ln\left(\frac{\sum_{\substack{(s',s)\Rightarrow \\ u_k=+1}} a_{k-1}(s')\cdot e^{(+L(u_k)/2)}\cdot e^{(+L_C y_k/2)}\cdot \beta_k(s)}{\sum_{\substack{(s',s)\Rightarrow \\ u_k=-1}} a_{k-1}(s')\cdot e^{(-L(u_k)/2)}\cdot e^{(-L_C y_k/2)}\cdot \beta_k(s)}\right)$$

$$= L(u_k) + L_C y_k + \ln\left(\frac{\sum_{\substack{(s',s)\Rightarrow \\ u_k=+1}} a_{k-1}(s')\cdot \beta_k(s)}{\sum_{\substack{(s',s)\Rightarrow \\ u_k=-1}} a_{k-1}(s')\cdot \beta_k(s)}\right)$$

$$= L(u_k) + L_C y_k + L_e(u_k) \qquad (3.1\text{-}19)$$

其中

$$L_e(u_k) = \ln\left(\frac{\sum_{\substack{(s',s)\Rightarrow \\ u_k=+1}} a_{k-1}(s')\cdot \beta_k(s)}{\sum_{\substack{(s',s)\Rightarrow \\ u_k=-1}} a_{k-1}(s')\cdot \beta_k(s)}\right) \qquad (3.1\text{-}20)$$

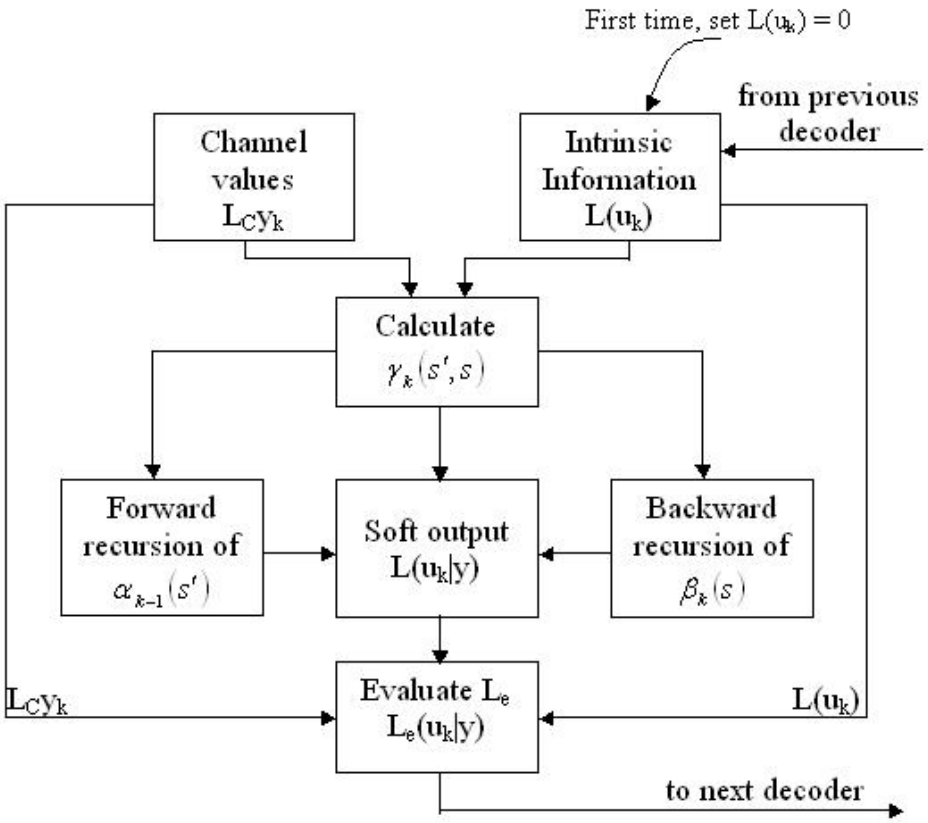
在式(3.1-19)中，第一個項目 $L(u_k)$是事前機率的對數相似比(a-prior LLR)。此項目的值是由資料位元的事前機率 $P(u_k)$所決定的，而解碼器通常無法得知此值。在實際的解碼過程中，我們通常先假設資料位元 uk 為+1 或-1 的機率是相同的，也就是說 $P(u_k=\pm 1)=0.5$，因此在第一次的解碼中，$L(u_k)$的值將被設為 0。但隨著縱橫反覆的解碼過程，每一次縱向解碼將提供給橫向解碼新的 $L(u_k)$值；同樣的橫向解碼也將提

供新的$L(u_k)$值。藉此獲得更強的錯誤更正能力。

第二個項目$L_C y_k$則是由通道來影響的，$y_k$的值是輸入資料位元$u_k$經過通道影響後所決定的，而$L_C$則是和通道的SNR有直接的關係。透過$L_C$的大小，決定了$y_k$值所能帶來的影響，當通道的SNR越高時，$y_k$相對的也就對後驗對數相對比(a-posteriori LLR)$L(u_k|\underline{y})$有越大的影響力。

第三個項目$L_e(u_k)$則是資料位元$u_k$的額外對數相似比(extrinsic LLR)。因為其值是由除了自己時間k的路徑外，其它的路徑值($\gamma_{k'}(s',s)$，for all $k' \neq k$)所決定。這個項目的值將會傳給下一個BCH解碼路做為下一次解碼運算中事前機率對數相對比的參考值。

整個 MAP 演算法及其重複運算的運作法可由下圖(圖 3.1-3)表示：




圖(3.1-3) MAP 重複解碼運作圖

## 3.2 Max-Log-MAP 演算法以及 Log-MAP 演算法

Max-Log-MAP 演算法是由 MAP 演算法做簡化之後得到的，其簡化了在 MAP 演算法中計算所需的計算量。此演算法的原理在於它將所有的計算對應到對數領域(log arithmetic domain)裡，此外還應用了 Jacobian 對數近似定理：

$$\ln\left(\sum_i e^{x_i}\right) \approx \max_i(x_i) \qquad (3.2\text{-}1)$$

式中的 $\max_i(x_i)$ 表示 $x_i$ 的最大值。

定義 $\mathrm{A}_k(s)$、$\mathrm{B}_k(s)$、$\Gamma_k(s',s)$ 為 $\alpha_{k-1}(s')$、$\beta_k(s)$、$\gamma_k(s',s)$ 對應到對數領域後的值，我們可以得到：

$$\begin{aligned}
\mathrm{A}_k(s) &\equiv \ln(\alpha_k(s)) \\
&= \ln\left(\sum_{all\ s'} \alpha_{k-1}(s')\gamma_k(s',s)\right) \\
&= \ln\left(\sum_{all\ s'} \exp[\mathrm{A}_{k-1}(s') + \Gamma_k(s',s)]\right) \\
&\approx \max_{s'}(\mathrm{A}_{k-1}(s') + \Gamma_k(s',s)) \qquad (3.2\text{-}2)
\end{aligned}$$

$$\begin{aligned}
\mathrm{B}_{k-1}(s') &\equiv \ln(\beta_{k-1}(s')) \\
&= \ln\left(\sum_{all\ s} \beta_k(s)\gamma_k(s',s)\right) \\
&= \ln\left(\sum_{all\ s} \exp[\mathrm{B}_k(s) + \Gamma_k(s',s)]\right) \\
&\approx \max_{s}(\mathrm{B}_k(s) + \Gamma_k(s',s)) \qquad (3.2\text{-}3)
\end{aligned}$$

有了上述兩個式子的關係，我們就可以利用向前遞迴計算和向後遞

迴計算的方法將$A_k(s)$、$B_k(s)$計算出來，而其中只需要用到加法器和比較器，較原本的 MAP 演算化簡化許多。

重新定義的路徑值為：

$$\begin{aligned}\Gamma_k(s',s) &\equiv \ln(\gamma_k(s',s)) \\ &= \ln\left(C\cdot e^{(u_k\cdot L(u_k)/2)}\cdot\exp\left[\frac{E_b}{2\sigma^2}2a(y_k\cdot x_k)\right]\right) \\ &= \ln\left(C\cdot e^{(u_k\cdot L(u_k)/2)}\cdot\exp\left[\frac{L_C}{2}(y_k\cdot x_k)\right]\right) \\ &= \hat{C}+\frac{1}{2}u_k\cdot L(u_k)+\frac{L_C}{2}(y_k\cdot x_k) \qquad (3.2\text{-}4)\end{aligned}$$

式中$\hat{C}=\ln(C)$和輸入資料位元$u_k$或是編碼後的資料$x_k$都沒有關係，故其可視為常數，並將在以後的式子中被消去。

將式(3.1-18)中的$\alpha_{k-1}(s')$、$\beta_k(s)$、$\gamma_k(s',s)$改為以$A_k(s)$、$B_k(s)$、$\Gamma_k(s',s)$表示，即可得到 Max-Log-MAP 演算法中的後驗對數相似比$L(u_k|\underline{y})$：

$$\begin{aligned}L(u_k|\underline{y}) &= \ln\left(\frac{\sum_{\substack{(s',s)\Rightarrow\\ u_k=+1}} a_{k-1}(s')\cdot\gamma_k(s',s)\cdot\beta_k(s)}{\sum_{\substack{(s',s)\Rightarrow\\ u_k=-1}} a_{k-1}(s')\cdot\gamma_k(s',s)\cdot\beta_k(s)}\right) \\ &= \ln\left(\frac{\sum_{\substack{(s',s)\Rightarrow\\ u_k=+1}}\exp(A_{k-1}(s')+\Gamma_k(s',s)+B_k(s))}{\sum_{\substack{(s',s)\Rightarrow\\ u_k=-1}}\exp(A_{k-1}(s')+\Gamma_k(s',s)+B_k(s))}\right) \\ &\approx \max_{\substack{(s',s)\Rightarrow\\ u_k=+1}}(A_{k-1}(s')+\Gamma_k(s',s)+B_k(s)) \\ &\quad -\max_{\substack{(s',s)\Rightarrow\\ u_k=-1}}(A_{k-1}(s')+\Gamma_k(s',s)+B_k(s)) \qquad (3.2\text{-}4)\end{aligned}$$

在處理的過程中，我們將格子圖中從狀態$S_{k-1}$進入到狀態$S_k$的轉換分為兩個群組，其中一個是當$u_k$=+1 時會帶動的轉換；另一個則是$u_k$=−1 時將會發生的轉換，在這兩個群組中，只要找到$(A_{k-1}(s')+\Gamma_k(s',s)+B_k(s))$的最大

值，即可利用兩組中最大值的差來得到我們要的LLR $L(u_k|\underline{y})$。

在使用 MAX-Log-MAP 演算法時，由於用到了式(3.2-1)的近似，故其效能會比原本的 MAP 演算法較為下降。為了將此效能差異補正回來，我們可以利用 Jacobian logaritm 做補正的動作，其運作如式(3.2-5)所示：

$$\begin{aligned}\ln\left(e^{x_1}+e^{x_2}\right) &= \max(x_1,x_2)+\ln\left(1+e^{-|x_1-x_2|}\right)\\ &= \max(x_1,x_2)+f_c\left(|x_1-x_2|\right)\\ &= g(x_1,x_2)\end{aligned} \qquad (3.2\text{-}5)$$

式中 $f_c(x)$即為Log-MAP演算法中所需要的修正項目(correction term)。

在 Log-MAP 演算法裡，其跟 MAX-Log-MAP 演算法一樣，同樣是將所有的運算對應到對數領擊之中，故其同樣有著 $\mathrm{A}_k(s)\equiv\ln(\alpha_k(s))$ 以及 $\mathrm{B}_{k-1}(s')\equiv\ln(\beta_{k-1}(s'))$ 的定義。並且同樣的利用遞迴運算將所有時間上的各個狀態機率值推算出來；唯一不同的是，取在計算過程中不只是取最大值，同時還要加上後面的修正項 $f_c(\delta)$使得計算更為精準，其中$\delta$為兩個比較路徑值的大小差距。我們可以將式(3.2-5)通用化，在累加很多值時，其式子可如下表示：

$$\ln\left(\sum_{i=1}^{I}e^{x_i}\right)=g(x_I,g(x_{I-1},\cdots,g(x_3,g(x_2,x_1)))\cdots) \qquad (3.2\text{-}6)$$

利用此式，再計算最後的後驗概率值時，同樣可以進行每個比較的修正，以達到和 MAP 演算法相同的結果。雖然在式子上觀察下，此演算法需要在每次對不同的$|x_i-x_j|$做計算，但在實際的應用上可將此值以查表的方式來完成，只針對一些不同的$|x_i-x_j|$值做表，在需要用到時再利用查表的方法，以接近的值來做為修正項，如此即可更加的省卻硬體的需求，同時能達到更好的解碼效能。

## 3.3 對延伸 BCH 碼的特殊處理

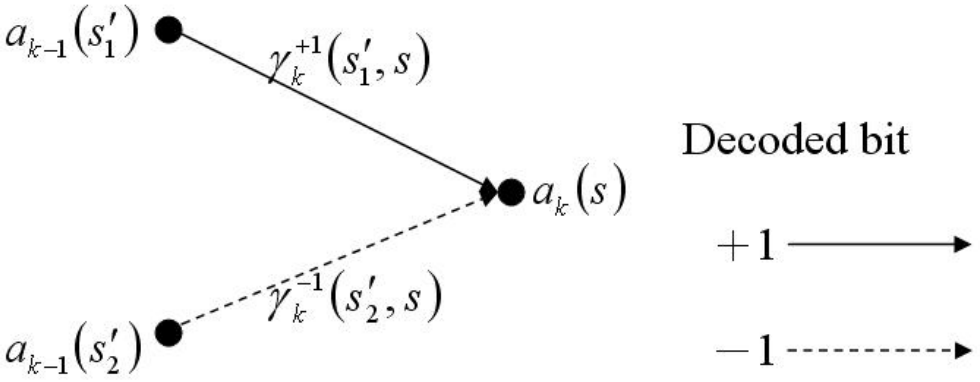
在這節中將介紹最大後驗概率(MAP)演算法如何應用在以延伸 BCH 碼為組成碼的乘積碼上。首先針對 BCH 碼的應用上，觀察 MAP 演算法的運作情形，之後再討論在延伸 BCH 碼上改變。由於延伸 BCH 碼多了一個同位檢查位元，我們必須重新定義 MAP 演算法來使其可以應用到延伸 BCH 碼中。

### 3.3.1 MAP 演算法在 BCH 碼上的運作

由於BCH碼中，一個狀態的變化只會影響一個$x_k$位元，而在系統編碼中，前n個位元的$x_k$值和輸入的資料位元$u_k$值是相同的，所以路徑值可寫為：

$$\gamma_k(s',s) = C \cdot \exp\left\{u_k \frac{L(u_k)}{2}\right\} \cdot \exp\left\{\frac{L_C}{2} u_k y_k\right\} \qquad (3.3\text{-}1)$$

另外，因為BCH碼是二位元的格子圖(binary trellis)，所以對任一個狀態$S_k$=s中，只會有兩個路徑可以從前一個狀態$S_{k-1}$=s´，如(圖 3.3-1)所示。而且這兩個路徑中，一定一個對應到$u_k$=+1 時所進行的轉換，另




圖(3.3-1) BCH 碼計算$\alpha_k(s)$示意圖

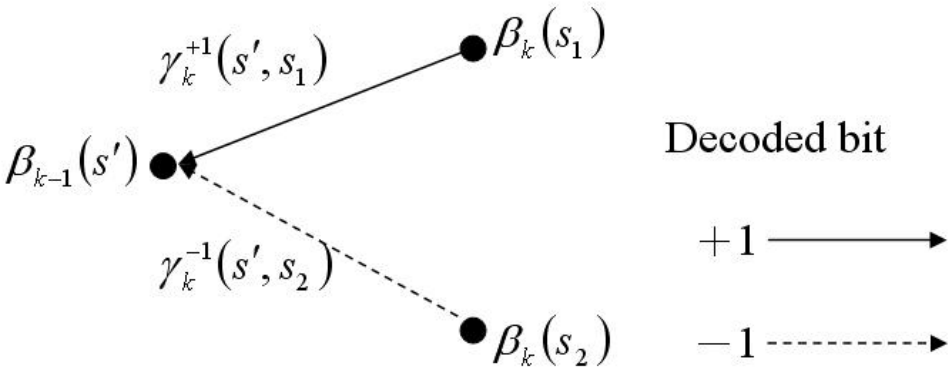
一個則對應到$u_k$=−1 所造成的轉換，因此我們可以將式(3.1-14)展開如

下：

$$\alpha_k(s)=\alpha_{k-1}(s_1')\cdot\gamma_k^{+1}(s_1',s)+\alpha_{k-1}(s_2')\cdot\gamma_k^{-1}(s_2',s) \qquad (3.3\text{-}2)$$

式中$\gamma_k^{u_k}(s',s)$代表在狀態轉換過程中$\gamma_k(s',s)$的值，其中$u_k$的值為+1 或-1 就如(圖 3.3-1)所示。

基於同樣的理由，在計算$\beta_{k-1}(s')$的值時，一樣可以利用之後的$\beta_k(s)$倒算回來，其相關圖如(圖 3.3-2)所示。像之前所描述的一樣，即使是




圖(3.3-2) BCH 碼計算$\beta_k(s)$示意圖

在倒回計算的方向裡，同樣只會有兩個狀態分別對應著$u_k=\pm1$，因此就像是展開$\alpha_k(s)$一樣，可以將式(3.1-16)展開如下：

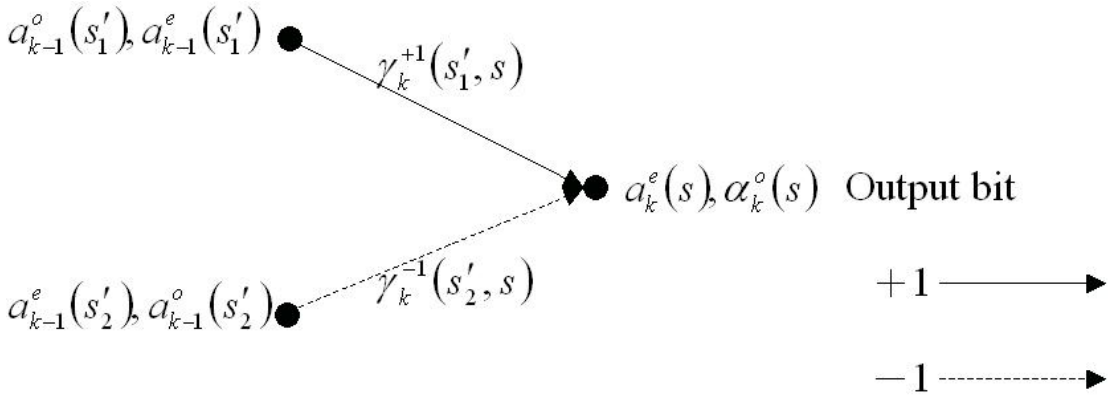
$$\beta_{k-1}(s')=\beta_k(s_1)\cdot\gamma_k^{+1}(s',s_1)+\beta_k(s_2)\cdot\gamma_k^{-1}(s',s_2) \qquad (3.3\text{-}3)$$

利用式(3.1-18)，(3.3-1)，(3.3-2)和(3.3-3)即可對 BCH 編碼做解碼的動作，找出每個位元的$L(u_k|\underline{y})$值。

### 3.3.2 重新定義的 MAP 演算法

對於延伸BCH碼，為了能讓同位檢查位元(parity check bit)有作用，我們定義$\alpha_k^e(s)$表示在格子圖時間k點時的狀態$S_k$=s並且從一開始跑

格子圖到時間k點時總共經過偶數(even)次傳送位元為+1 這種情形發生的機率。而這個值的決定是由接收到的序列 $\underline{y}_{j\le k}$ 所提供的。同樣的，$\alpha_k^o(s)$ 定義成在格子圖時間k點時的狀態$S_k$=s並且從一開始跑格子圖到時間k點時總共經過奇數(odd)次傳送位元為+1 這種情形發生的機率。其值的決定同樣是由接收到的序列 $\underline{y}_{j\le k}$ 所提供。這兩個值在格子圖的表現情形如(圖 3.3-3)所示。




圖(3.3-3) eBCH 碼計算 $\alpha_k^o(s), \alpha_k^e(s)$ 示意圖

如(圖 3.3-3)所表現出來的情形，我們考慮兩個狀況，第一個情形是時間 k-1 點時總共已經過奇數個+1 時通過資料位元為+1 的路徑在時間 k 點達到狀態 s。另一個情形則是時間 k-1 點時總共已經過偶數個+1 此時再通過資料位元為-1 的路徑在時間 k 點轉換到狀態 s 而達成條件。因此，我們可將式(3.3-2)改寫如下：

$$\alpha_k^e(s) = \alpha_{k-1}^o(s_1') \cdot \gamma_k^{+1}(s_1', s) + \alpha_{k-1}^e(s_2') \cdot \gamma_k^{-1}(s_2', s) \qquad (3.3\text{-}4)$$

在式中，$\gamma_k^{+1}(s_1', s)$ 為從狀態 $s_1'$ 轉換到狀態 $s$ 經由資料位元為+1 時的機率，所以配合者之前經過奇數個+1 到達狀態 $s_1'$ 的機率 $\alpha_{k-1}^o(s_1')$，即表示經過了奇數加一個+1，達到偶數的情形。而另外一種情形則是由 $\gamma_k^{-1}(s_2', s)$ 和 $\alpha_{k-1}^e(s_2')$ 配合，因為最後並沒有經過+1，所以同樣達到偶數的要求。

以奇數相同的，利用圖(3.3-3)以及式(3.3-2)我們得以求得偶數的前行狀態值$\alpha_k^o(s)$如下：

$$\alpha_k^o(s)=\alpha_{k-1}^e(s_1')\cdot\gamma_k^{+1}(s_1',s)+\alpha_{k-1}^o(s_2')\cdot\gamma_k^{-1}(s_2',s) \qquad (3.3\text{-}5)$$

利用相似的原理，我們同樣可以將逆向遞迴計算的式(3.3-3)改寫為下列二式：

$$\beta_{k-1}^e(s')=\beta_k^o(s_1)\cdot\gamma_k^{+1}(s',s_1)+\beta_k^e(s_2)\cdot\gamma_k^{-1}(s',s_2) \qquad (3.3\text{-}6)$$

以及

$$\beta_{k-1}^o(s')=\beta_k^e(s_1)\cdot\gamma_k^{+1}(s',s_1)+\beta_k^o(s_2)\cdot\gamma_k^{-1}(s',s_2) \qquad (3.3\text{-}7)$$

在延伸 BCH 碼的兩個狀態傳送傳換中，不在是只靠簡單的$\alpha_{k-1}(s')$、$\beta_k(s)$以及$\gamma_k(s',s)$可以表現出來的，其還必須考慮到整條路徑中，總共經過了偶數次或奇數次+1 的影響。所以以狀態$s'$轉換到狀態$s$為例，其轉換機率就需要考慮到整條路徑所經過+1 的次數為奇數時的情形：

$$P(s'\to s)=\alpha_{k-1}(s')\cdot\gamma_k(s',s)\cdot\beta_k(s)\cdot P(y_n|x_n=+1) \qquad (3.3\text{-}8)$$

以及整條路徑所經過+1 的次數偶數時的情形：

$$P(s'\to s)=\alpha_{k-1}(s')\cdot\gamma_k(s',s)\cdot\beta_k(s)\cdot P(y_n|x_n=-1) \qquad (3.3\text{-}9)$$

式中$x_n$表示傳送的同位檢查位元(parity bit)而$y_n$則表示接送端所對應接收到同位檢查位元的軟式輸出值。

在有了上述的各個條件之後，我們可以利用遞迴計算將$\alpha_k^o(s)$、$\alpha_k^e(s)$、$\beta_k^o(s)$和$\beta_k^e(s)$計算出來，同時同位檢查位元的機率值$P(y_n|x_n)$亦可計算得到，故最後可將事後機率的 LLR 從式(3.1-18)變化後，整理如下：

$$L(u_k\,|\,y)=\ln\frac{\sum\limits_{\substack{(s,s)\Rightarrow\\ u_k=+1}}\gamma_k^{+1}(s',s)\cdot\left[\left(\alpha^e\cdot\beta^e+\alpha^o\cdot\beta^o\right)\cdot P(y_n\,|\,x_n=+1)+\left(\alpha^e\cdot\beta^o+\alpha^o\cdot\beta^e\right)\cdot P(y_n\,|\,x_n=-1)\right]}{\sum\limits_{\substack{(s,s)\Rightarrow\\ u_k=-1}}\gamma_k^{-1}(s',s)\cdot\left[\left(\alpha^e\cdot\beta^o+\alpha^o\cdot\beta^e\right)\cdot P(y_n\,|\,x_n=+1)+\left(\alpha^e\cdot\beta^e+\alpha^o\cdot\beta^o\right)\cdot P(y_n\,|\,x_n=-1)\right]} \qquad (3.3\text{-}10)$$

式中$\alpha^o$、$\alpha^e$、$\beta^o$和$\beta^e$是$\alpha_{k-1}^o(s')$、$\alpha_{k-1}^e(s')$、$\beta_k^o(s)$及$\beta_k^e(s)$的簡化記法。

在式的分子中，其所累加的對象一定是在狀態轉換中會經過+1 的路徑，因此，為了能達到整個路徑經過+1 的次數為偶數的條件，當最後的同位檢查位元為+1 時，$\alpha$與$\beta$經過+1 的次數就必須是偶數，故只有$\alpha^e \cdot \beta^e$以及$\alpha^o \cdot \beta^o$符合此條件。同樣在分子中，若最後的同位檢查位元為-1 時，則$\alpha$與$\beta$經過+1 的次數就必須是奇數，也只有$\alpha^e \cdot \beta^o$以及$\alpha^o \cdot \beta^e$符合此條件。利用類似的方法，即可將分母的式子也導出來，因此最後能得到式(3.3-10)的結果。

### 3.3.3 應用在延伸 BCH 碼的 Max-Log-MAP 及 Log-MAP 演算法

在這節中，我們將介紹如何將 3.2 節提出的 Max-Log-MAP 及 Log-MAP 演算法應用到延伸 BCH 碼中，其主要的基礎和 MAP 演算法應用到延伸 BCH 碼相同，故其要求的條件亦相同，如式(3.3-8)所示，以下的值都是要知道的：

I. $\alpha_{k-1}^e(s')$以及$\alpha_{k-1}^o(s')$的值，如式(3.3-4)及式(3.3-5)所示，此為遞迴計算前行狀態值所必須要有的。

II. $\beta_k^e(s)$以及$\beta_k^o(s)$的值，此為遞迴計算逆行狀態值所需要用的，如式(3.3-6)及式(3.3-7)所示。

III. 傳送路徑轉換機率$\gamma_k(s',s)$。

IV. 傳送中+1 值為偶數或奇數的機率，$P(y_n|x_n=+1)$和$P(y_n|x_n=-1)$。

在 Max-Log-MAP 演算法中，如之前所提過的，需要利用到式(3.2-1)的近似定律，另外我們同樣的定義在對數領域中的值如下：

$$A_k^e(s) \equiv \ln[\alpha_k^e(s)] \qquad (3.3\text{-}11)$$

$$A_k^o(s) \equiv \ln[\alpha_k^o(s)] \qquad (3.3\text{-}12)$$

$$B_k^e(s) \equiv \ln[\beta_k^e(s)] \qquad (3.3\text{-}13)$$

$$\mathrm{B}_k^o(s) \equiv \ln\left[\beta_k^o(s)\right] \tag{3.3-14}$$

$$\Gamma_k^{x_k}(s',s) \equiv \ln\left[\gamma_k^{x_k}(s',s)\right] \tag{3.3-15}$$

以及

$$\varepsilon^{x_n} \equiv \ln\left[P\left(y_n|x_n\right)\right] \tag{3.3-16}$$

將式(3.3-4)對應到對數領域後，將其代入式(3.3-11)再利用式(3.2-1)的近似定律可得到：

$$\begin{aligned}
\mathrm{A}_k^e(s) &= \ln\left[\alpha_{k-1}^e(s')\cdot\gamma_k^{-1}(s',s)+\alpha_{k-1}^o(s')\cdot\gamma_k^{+1}(s',s)\right] \\
&= \ln\left[\exp\left\{\mathrm{A}_{k-1}^e(s')+\Gamma_k^{-1}(s',s)\right\}+\exp\left\{\mathrm{A}_{k-1}^o(s')+\Gamma_k^{+1}(s',s)\right\}\right] \\
&\approx \max\left[\left\{\mathrm{A}_{k-1}^e(s')+\Gamma_k^{-1}(s',s)\right\},\left\{\mathrm{A}_{k-1}^o(s')+\Gamma_k^{+1}(s',s)\right\}\right]
\end{aligned} \tag{3.3-17}$$

利用同樣的近似方法，將式(3.3-5)代入式(3.3-12)我們可推出：

$$\begin{aligned}
\mathrm{A}_k^o(s) &= \ln\left[\alpha_{k-1}^o(s')\cdot\gamma_k^{-1}(s',s)+\alpha_{k-1}^e(s')\cdot\gamma_k^{+1}(s',s)\right] \\
&= \ln\left[\exp\left\{\mathrm{A}_{k-1}^o(s')+\Gamma_k^{-1}(s',s)\right\}+\exp\left\{\mathrm{A}_{k-1}^e(s')+\Gamma_k^{+1}(s',s)\right\}\right] \\
&\approx \max\left[\left\{\mathrm{A}_{k-1}^o(s')+\Gamma_k^{-1}(s',s)\right\},\left\{\mathrm{A}_{k-1}^e(s')+\Gamma_k^{+1}(s',s)\right\}\right]
\end{aligned} \tag{3.3-18}$$

像之前的式(3.3-17)和(3.3-18)一樣，我們可將式(3.3-13)改寫為：

$$\begin{aligned}
\mathrm{B}_{k-1}^e(s) &= \ln\left[\beta_k^e(s)\cdot\gamma_k^{-1}(s',s)+\beta_k^o(s)\cdot\gamma_k^{+1}(s',s)\right] \\
&= \ln\left[\exp\left\{\mathrm{B}_k^e(s)+\Gamma_k^{-1}(s',s)\right\}+\exp\left\{\mathrm{B}_k^o(s)+\Gamma_k^{+1}(s',s)\right\}\right] \\
&\approx \max\left[\left\{\mathrm{B}_k^e(s)+\Gamma_k^{-1}(s',s)\right\},\left\{\mathrm{B}_k^o(s)+\Gamma_k^{+1}(s',s)\right\}\right]
\end{aligned} \tag{3.3-19}$$

以及將式(3.3-14)改為：

$$\begin{aligned}
\mathrm{B}_{k-1}^o(s) &= \ln\left[\beta_k^o(s)\cdot\gamma_k^{-1}(s',s)+\beta_k^e(s)\cdot\gamma_k^{+1}(s',s)\right] \\
&= \ln\left[\exp\left\{\mathrm{B}_k^o(s)+\Gamma_k^{-1}(s',s)\right\}+\exp\left\{\mathrm{B}_k^e(s)+\Gamma_k^{+1}(s',s)\right\}\right] \\
&\approx \max\left[\left\{\mathrm{B}_k^o(s)+\Gamma_k^{-1}(s',s)\right\},\left\{\mathrm{B}_k^e(s)+\Gamma_k^{+1}(s',s)\right\}\right]
\end{aligned} \tag{3.3-20}$$

而將式(3.1-12)對應到對數領域後，可得：

$$\Gamma_k^{x_k}(s',s)=\frac{u_k}{2}L(u_k)+\frac{L_C}{2}y_k x_k \qquad (3.3\text{-}21)$$

式中$L(u_k)$為$u_k$事前機率值，而$L_C$則是和通道性質有關。

而同位檢查位元的傳送機率可以以照傳送通道的不同而計算出來，假設我們傳送的通道為高斯通道並以 BPSK 調變，則其機率如下：

$$P(y_n|x_n=\pm 1)=\frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{E_b}{2\sigma^2}(y_n\mp a)^2} \qquad (3.3\text{-}22)$$

式中$E_b$為傳送每一個位元的能量， $\sigma^2$是雜訊變異數，而$a$則是通道衰減大小，在沒有衰減的可加性高斯白雜訊通道(non-fading AWGN channels)其值為 1。

利用式(3.3-22)，我們可將式(3.3-16)重寫如下：

$$\begin{aligned}\varepsilon^{x_n=\pm 1} &= \ln\left[\frac{1}{\sigma\sqrt{2\pi}}\right]-\frac{E_b}{2\sigma^2}(y_n\mp a)^2\\ &= C-\frac{E_b}{2\sigma^2}\left(y_n^2\mp 2ay_n+a^2\right)\\ &= C-\frac{E_b}{2\sigma^2}\left(y_n^2+a^2\right)\pm\frac{E_b}{2\sigma^2}2ay_n\\ &= C-C'\pm\frac{L_C}{2}y_n \qquad (3.3\text{-}23)\end{aligned}$$

式中$C=\ln\left[\frac{1}{\sigma\sqrt{2\pi}}\right]$，$C'=\frac{E_b}{2\sigma^2}\left(y_n^2+a^2\right)$而$L_C=\frac{E_b\cdot 2a}{\sigma^2}$。不論是$C$還是$C'$其值和傳送的同位檢查位元$x_n$都沒有關係，故在計算中會相互消去，因此在解碼時可以省略不計。

最後，利用上述的式子(從式(3.3-17)到式(3.3-23))，我們可將 MAP 演算法中用於延伸 BCH 碼的後驗機率對數相似度$L(u_k|\underline{y})$從式(3.3-10)延伸到 Max-Log-MAP 演算法中，其結果表示如下：

$$L(u_k|y) \approx \max_{\substack{(s,s)\Rightarrow \\ u_k=+1}}[P(s',s)] - \max_{\substack{(s,s)\Rightarrow \\ u_k=-1}}[P(s',s)] \text{ ,} \qquad (3.3\text{-}24)$$

式中

$$\begin{aligned} P(s',s) \quad = \quad & \max\Big[\Big\{\Gamma_k^{u_k}(s',s) + \mathrm{A}_{k-1}^{e}(s') + \mathrm{B}_k^{e}(s) + \varepsilon^{x_n=u_k}\Big\}, \\ & \Big\{\Gamma_k^{u_k}(s',s) + \mathrm{A}_{k-1}^{o}(s') + \mathrm{B}_k^{o}(s) + \varepsilon^{x_n=u_k}\Big\}, \\ & \Big\{\Gamma_k^{u_k}(s',s) + \mathrm{A}_{k-1}^{e}(s') + \mathrm{B}_k^{o}(s) + \varepsilon^{x_n=-u_k}\Big\}, \\ & \Big\{\Gamma_k^{u_k}(s',s) + \mathrm{A}_{k-1}^{o}(s') + \mathrm{B}_k^{e}(s) + \varepsilon^{x_n=-u_k}\Big\}\Big] \qquad (3.3\text{-}25) \end{aligned}$$

至於 Log-MAP 演算法則和 MAX-Log-MAP 演算法相當的相似，只要在所有比較的部分以查表的方法加入修正項的值，即可得到 Log-MAP 演算法的結果。

# 第四章　以卻斯(Chase)理論為基礎的解碼方法

本章同樣是解說解碼的方法，但本章是以卻斯的理論為基礎，先利用代數演算得到硬式輸入輸出(Hard in hard out)的解碼方法，取得硬式解碼的資料後卻斯提出了配合軟式輸入(Soft in)的方法取得更佳的解碼效果，而為了能用在乘積碼的解碼上，最後我們利用最大相似度(Maximum likelihood)的觀念，以卻斯演算法為基礎推算出軟式輸出(Soft out)結果。

## 4.1 以硬式輸入(Hard In)導算硬式輸出(Hard Out)的解碼方法

在這節中將介紹兩個不同的方法來進行硬式解碼，一開始的 BM 演算法可以針對各種 BCH 碼來解碼，但相對的其複雜度較高，而之後介紹的症狀(Syndrome)解碼方法則可以利用查表的方法快速的解碼，只是此種方法每換一個編碼就得重作一次，可攜性較差。

### 4.1.1 Berlekamp-Massey (BM)演算法

BM 演算法是一種利用線性回饋位移暫存器(Linear Feedback Shift Register，LFSR)配合求得的症狀值來找到接受到的資料中何處為錯誤位置的方法。此方法的運算是作用在有限元素場裡，並且需要反覆運算以求得錯誤位置。在這節中，所有的輸入及輸出資料都是硬式的，也就是說其資料都是已經經過比較器(slicer)判斷，其值皆為 0 或 1。

#### 4.1.1.1 症狀(Syndrome)計算及錯誤位置多項式(Error locator polynomial)

接收端接受到的訊息我們可以將其化為多項式 y(D)表示如下：

$$y(D)=x(D)+e(D) \qquad (4.1\text{-}1)$$

式中 x(D)為輸入位元經過編碼後資料 x 所轉化的多項式，而 e(D)則是經過通道影響而產生的錯誤多項式(error polynomial)。在 2.1.2 節中，我們知道生產多項式的根$\alpha^i$將會同時是 y(D)以及 x(D)的根。若 e(D)的值不為 0 時，則 y(D)即使代入生產多項式的根$\alpha^i$同樣不會為 0。症狀$S_i$， $i=1,2,\cdots,2t$的定義如下：

$$\begin{aligned} S_i &= y(\alpha^i) \\ &= x(\alpha^i)+e(\alpha^i) \\ &= e(\alpha^i), \quad i=1,2,\cdots,2t \end{aligned} \qquad (4.1\text{-}2)$$

由於$\alpha^i$為碼多項式 x(D)的根，故$x(\alpha^i)$的值為 0。由於 BCH 碼中能更正 t 個錯誤的碼共有 2t 個根，所以症狀也是有 2t 個。

假設在經過通道後有 v 個錯誤的判斷發生，並且其錯誤發生位置定在$j_1, j_2, \cdots, j_v$，而且$0 \le j_1 < j_2 < \cdots < j_v < N$，則錯誤多項式可改寫為：

$$e(D)=D^{j_v}+D^{j_{v-1}}+\cdots+D^{j_1} \qquad (4.1\text{-}3)$$

我們定義錯誤指標(error locator)$\chi_l$為$\chi_l=\alpha^{j_l}$，$l=1,2,\cdots,v$將此關係代入式(4.1-3)以及式(4.1-2)可得下式：

$$\begin{aligned} S_i = e(\alpha^i) &= (\alpha^i)^{j_v}+(\alpha^i)^{j_{v-1}}+\cdots+(\alpha^i)^{j_1} \\ &= \chi_v^i+\chi_{v-1}^i+\cdots+\chi_1^i, \quad i=1,2,\cdots,2t \end{aligned} \qquad (4.1\text{-}4)$$

另外，我們定義錯誤位置多項式(Error locator polynomial)為：

$$\begin{aligned} \sigma(D) &\equiv \prod_{i=1}^{v}(1+\chi_i D) \\ &= \sigma_v D^v+\sigma_{v-1}D^{v-1}+\cdots+\sigma_1 D+1 \end{aligned} \qquad (4.1\text{-}5)$$

觀察式(4.1-5)我們可以發現，$\{\chi_l\}$是$\sigma(D)$的根的倒數值，也就是說不論$l$的值為何$\sigma(\chi_l^{-1})$的值必定為 0。因此可得下式：

$$\chi_l^v \sigma(\chi_l^{-1}) = \chi_l^v + \sigma_1 \chi_l^{v-1} + \cdots + \sigma_{v-1}\chi_l + \sigma_v = 0, \quad l = 1,2,\cdots,v \qquad (4.1\text{-}6)$$

將上式加以乘上$\chi_l^j$的值，我們可以得到：

$$\chi_l^{v+j} + \sigma_1 \chi_l^{v+j-1} + \cdots + \sigma_{v-1}\chi_l^{j-1} + \sigma_v \chi_l^j = 0, \quad l, j = 1,2,\cdots,v \qquad (4.1\text{-}7)$$

將上式中$l$值從 1 一直代入到$v$後，再相加起來，並且和式(4.1-4)相比較，我們可以得到：

$$S_{v+j} + \sigma_1 S_{v+j-1} + \cdots + \sigma_{v-1} S_{j-1} + \sigma_v S_j = 0, \quad j = 1,2,\cdots,v \qquad (4.1\text{-}7)$$

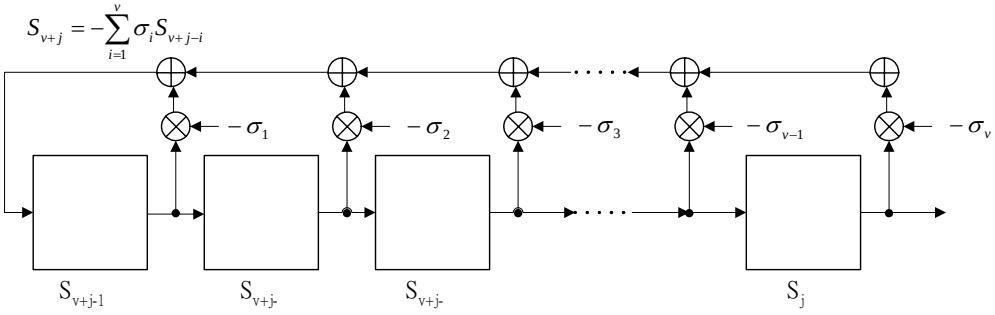
利用此式子，只要我們能解出$\{\sigma_i\}$的值，就可以找出錯誤位置的所在，而這正是下一節 BM 演算法中所要介紹的主要原理。

### 4.1.1.2 BM 演算法的架構

利用線性回饋位移暫存器(LFSR)以及上一節所描述的症狀和錯誤位置多項式的關係，我們試著利用症狀值將錯誤的位置表示出來，回顧式(4.1-7)，此式子可改寫如下：

$$S_{v+j} = -\sum_{i=1}^{v} \sigma_i S_{v+j-i}, \quad j = 1,2,\cdots,v \qquad (4.1\text{-}7)$$

現在我們考慮一個線性回饋位移暫存器如圖(4.1-1) 所示：




圖(4.1-1) 產生症狀$S_1, S_2, S_3, \cdots$的線性回饋位移暫存器圖

如圖所示，當我們將$-\{\sigma_i\}$的值填入位移暫存器的係數後，其最後輸出的值將會是$S_{v+1}$若是暫存器裡存放$S_v, S_{v-1}, \cdots, S_2, S_1$。而若是位移後，當暫存器裡存放$S_{v+1}, S_v, \cdots, S_3, S_2$時，則會輸出$S_{v+2}$，如此循還下去。利用這樣的暫存器，BM 演算法的想法即是先行假定一個錯誤位置多項式$\sigma^{(\gamma-1)}(D)$，而利用此錯誤多項式求得估計的症狀$S'_\gamma$，再以比較估計症狀$S'_\gamma$和實際症狀$S_\gamma$的差別，藉此修正錯誤位置多項式，在經過多次重複的運作下，找出能符合所有症狀的錯誤位置多項式，其中$\gamma$表示第$\gamma$次的運算操作，同時也代表了估計了多少個症狀出來。

此方法一開始設定$\sigma^{(0)}(D)=1$然後經由反覆(iterative)的運算來建立最小長度的位移暫存器來產生症狀序列。我們可以假設在第$\gamma-1$次時，此位移暫存器產生了估計症狀為：

$$S'_\gamma = -\sum_{i=1}^{L_{\gamma-1}} \sigma_i^{(\gamma-1)} S_{\gamma-i} \qquad (4.1\text{-}8)$$

此時第$\gamma$個由位移暫存器$\sigma^{(\gamma-1)}(D)$所產生的估計症狀和實際症狀的差異定為$\Delta_\gamma$，其值為：

$$\begin{aligned}\Delta_\gamma &= S_\gamma - S'_\gamma \\ &= S_\gamma + \sum_{i=1}^{L_{\gamma-1}} \sigma_i^{(\gamma-1)} S_{\gamma-i} \qquad (4.1\text{-}9)\end{aligned}$$

有了此關係式後，我們接下來要試著讓下次的錯誤位置多項式$\sigma^{(\gamma)}(D)$能夠將此差異補足，必且讓位移暫存器的長度達到最小。為了達到此目的，我們定義了 m 這個值。m 表示了最接近第$\gamma$次運算並且滿足了其差異值必須由改變位移暫存器長度來補足的反覆運算(iteration)時間，也就是說在第 m 次之前的估計症狀都可以利用$\sigma^{(m-1)}(D)$來完美達成，此情形可以下式表達：

$$\Delta_j = S_j + \sum_{i=1}^{L_{m-1}} \sigma_i^{(m-1)} S_{j-i} \qquad (4.1\text{-}10)$$

此式滿足

$$\Delta_j = \begin{cases} 0, & for \quad j = 1,2,\cdots,m-1 \\ \text{非零數}, & for \quad j = m \end{cases}$$

我們將錯誤位置多項式的更新值設定如下：

$$\sigma^{(\gamma)}(D) = \sigma^{(\gamma-1)}(D) - \frac{\Delta_\gamma}{\Delta_m} D^{\gamma-m}\sigma^{(m-1)}(D) \qquad (4.1\text{-}11)$$

有了上述錯誤位置多項式，根據式(4.1-8)，式(4.1-9)利用新的$\sigma^{(\gamma)}(D)$產生的估計症狀和實際症狀的差異比較如下所示：

$$S_j + \sum_{i=1}^{L_\gamma} \sigma_i^{(\gamma)} S_{j-i} = S_j + \sum_{i=1}^{L_{\gamma-1}} \sigma_i^{(\gamma-1)} S_{j-i} - \frac{\Delta_\gamma}{\Delta_m}\left[ S_{j-r+m} + \sum_{i=1}^{L_{m-1}} \sigma_i^{(m-1)} S_{j-i-(\gamma-m)} \right] \qquad (4.1\text{-}12)$$

在式(4.1-12)中，我們可以觀察到：

$$S_j + \sum_{i=1}^{L_{\gamma-1}} \sigma_i^{(\gamma-1)} S_{j-i} = \begin{cases} 0, & for \quad j \le \gamma - 1 \\ \Delta_\gamma, & for \quad j = \gamma \end{cases}$$

以及

$$S_{j-r+m} + \sum_{i=1}^{L_{m-1}} \sigma_i^{(m-1)} S_{j-i-(\gamma-m)} = \begin{cases} 0, & for \quad j \le \gamma - 1 \\ \Delta_m, & for \quad j = \gamma \end{cases}$$

所以式(4.1-12)等號右方的式子可化簡如下：

$$\begin{cases} 0, & for \quad j \le \gamma - 1 \\ \Delta_\gamma - \frac{\Delta_\gamma}{\Delta_m} \times \Delta_m = 0, & for \quad j = \gamma \end{cases}$$

結果將為 0，此即表示$\sigma^{(\gamma)}(D)$可以確實的產生$S_1, S_2, \cdots, S_\gamma$共$\gamma$個的症狀。

在有了更新$\sigma^{(\gamma)}(D)$的方法後，接著是有關$\sigma^{(\gamma)}(D)$的長度應該要多少才足以達到能確實產生$\gamma$個症狀的最小長度，在本論文中，我們只提出其長度值以及如何應用，而其証明則可以從 Massey 所提出的論文中找到。當位移暫存器(LFSR)有著長度$L_{\gamma-1}$可以確實的產生$S_1, S_2, \cdots, S_{\gamma-1}$，並且同樣能產生$S_1, S_2, \cdots, S_\gamma$時，其長度自然不用更新，即$L_\gamma = L_{\gamma-1}$。但當其能確實產生$S_1, S_2, \cdots, S_{\gamma-1}$，卻不能產生出$S_1, S_2, \cdots, S_\gamma$的序列時，其值則必

須更新，其更新值如下，：

$$L_\gamma \equiv \max\left[L_{\gamma-1}, \gamma - L_{\gamma-1}\right] \qquad (4.1\text{-}13)$$

所以，$L_\gamma$ 的值只有在以下的情況會更新：

$$\gamma - L_{\gamma-1} > L_{\gamma-1} \qquad (4.1\text{-}14)$$

即

$$\gamma > 2 \cdot L_{\gamma-1} \qquad (4.1\text{-}15)$$

接下來將會列出 BM 演算法的步驟，在步驟的描述中，$\sigma(D)$為錯誤位置多項式，$\beta(D)$則是儲存了上一次改變長度時的錯誤位置多項式，並且已先除去了Δ值，如此即可在更新時用上。在式(4.1-11 中)原須要做$D^{(\gamma-m)}$次的位移，在步驟中，每一次$\beta(D)$的運算中都會做位移，不論Δ值是否為 0，如此可剛好達到所需次數的位移。最後要注意的是$\sigma(D)$的位階(degree)必須少於可更正的錯誤數量 t 值，若其位階比 t 值還大，則表示有過多的錯誤發生，無法以此演算法將錯誤更正回來。以下即是演算法步驟：

第一步(初始化)

$$\gamma = 0, \quad \sigma(D) = 1, \quad L = 0, \quad \beta(D) = 1$$

第二步(重複運算)

for $\gamma = 1$ to $2t$

計算錯誤差異Δ　　$\Delta = \sum_{i=0}^{L} \sigma_i S_{\gamma-i}$

if $\Delta \neq 0$ then（更新$\sigma(D)$）

$\sigma'(D) = \sigma(D) - \Delta D \beta(D)$

if $2L < \gamma$ then（更新位移暫存器的長度）

$\beta(D) = \Delta^{-1}\sigma(D)$

$L = \gamma - L$

else $\beta(D) = D\beta(D)$

$\sigma(D) = \sigma'(D)$

else $\beta(D) = D\beta(D)$

第三步(錯誤判斷)

若 $\deg\sigma(D)\neq L$ 表示有超過 t 個錯誤產生,則無法解碼。若 $\deg\sigma(D)=L$ 則最後的 $\sigma(D)$ 即為錯誤位置多項式，只要找到它的根，再將其倒置(inverse)後，即能找到錯誤的位置。

### 4.1.1.3 秦式搜尋(Chien´s search)

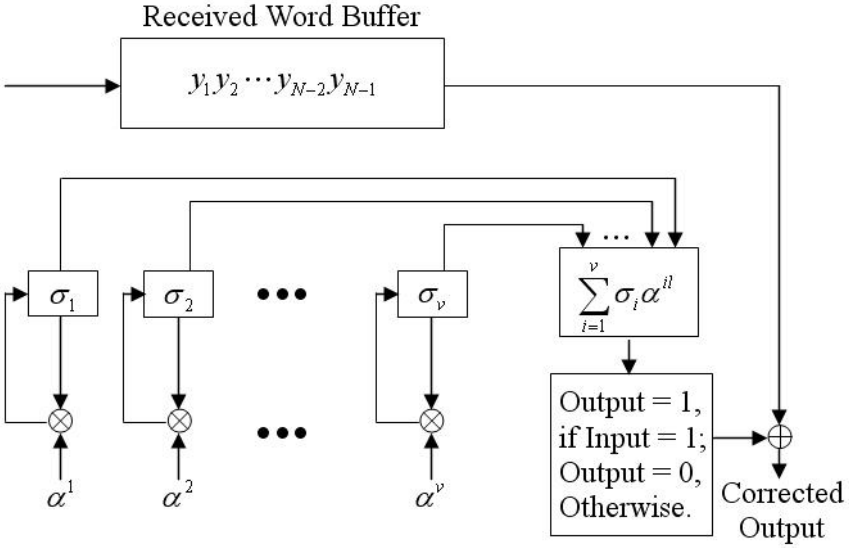
在有了錯誤位置多項式之後，只要能取得式子的根，即可解出錯誤的所在，此節將介紹由秦式(Chien)所發現的尋根方法，並解出其根的倒置值，此方法稱之為秦式搜尋(Chien′ s search)。此搜尋方法是將有限元素場的所有值 $\alpha,\alpha^2,\cdots,\alpha^{N-1}$ 一一代入 $\sigma(D)$ 中去驗證其值是否為 0。若 $\alpha^l$ 為 $\sigma(D)$ 其中的一根則 $\alpha^{-l}=\alpha^{N-l}$ 即為錯誤位置，也就表示接收到的符元(symbol) $y_{N-l}$ 為錯誤的。由於 $\alpha^l$ 為根，我們可以得到下式

$$1+\sigma_1\alpha^{l}+\sigma_2\alpha^{2l}+\cdots+\sigma_v\alpha^{vl}=0$$

即

$$\sigma_1\alpha^{l}+\sigma_2\alpha^{2l}+\cdots+\sigma_v\alpha^{vl}=1 \qquad (4.1\text{-}16)$$

因此，只要 $\sum_{i=1}^{v}\sigma_i\alpha^{il}$ 的值為 1，即表示 $y_{N-l}$ 為錯誤符元，而其更正法即為將其加 1。而若和的值不為 1，即表示 $y_{N-l}$ 為正確的值。下圖為實現秦式搜尋的線路圖。




圖(4.1-2) 秦式搜尋(chien′ s search)的線路圖

此線路圖將接收到的序列存在一個緩衝區(Buffer)裡，並且先行將 $v$ 個暫存器裡存放的值初始化為 $\sigma_1,\sigma_2,\cdots,\sigma_v$。每個暫存器都接上一個乘法器乘上對應的 $\alpha,\alpha^2,\cdots,\alpha^v$ 的值，而相乘後的結果都連回到暫存器以便能儲存此值。要測試 $\alpha$ 是否為 $\sigma(D)$ 的根，則將一次相乘後的結果 $\sigma_1\alpha,\sigma_2\alpha^2,\cdots,\sigma_v\alpha^v$ 存到暫存器後相加起來，其結果 $\sum_{i=1}^{v}\sigma_i\alpha^i$ 可來拿來測試是否為 1。而之後在接上一個邏輯電路其輸入為 1 時輸出亦為 1，而其餘輸入的輸出為 0。將此邏輯電路的輸出和序列 $y_{N-1}$ 相加後，即可將 $y_{N-1}$ 更正為正確的輸出。在第二輪中，暫存器的值會在乘上相對應的值，使得其和變為 $\sum_{i=1}^{v}\sigma_i\alpha^{2i}$ 可以用來測試 $\alpha^2$ 是否為根，如此循還下去，即可測試所有的元素，並將接受的序列更正回來。

### 4.1.2 利用症狀(Syndrome)解碼的解碼方法

在這節中將介紹另一種硬式的解碼方法，其利用輸入的序列找出症狀來，並將症狀和對應的編碼方法的檢查矩陣，相互比較，藉此判斷錯誤的位元在那，而達到解碼的目的。此節所提到的症狀和上節所提的定義不同，上節中是以多項式的觀點來看，而這節中則是以序列來表示，但兩節中皆是以經過通道的錯誤做為基礎的資訊。

接受端接受到的序列可以以序列$\mathbf{y}$來表示，$\mathbf{y}=\mathbf{x}+\mathbf{e}$，$\mathbf{x}$為傳送端經過編碼的序列，而$\mathbf{e}=(e_0, e_1, \cdots, e_{N-1})$則為錯誤向量。當$e_i=1$ 時，表示在第$i$個位置接受到的訊息發生了錯誤；而$e_i=0$ 則表示沒有錯誤發生。

對應於序列 $\mathbf{y}$ 的症狀 $\mathbf{s}$ 定義如下：

$$\mathbf{s} = \mathbf{y}\mathbf{H}^T = (\mathbf{x} + \mathbf{e})\mathbf{H}^T = \mathbf{0} + \mathbf{e}\mathbf{H}^T = \mathbf{e}\mathbf{H}^T \qquad (4.1\text{-}17)$$

在有了症狀之後，就可以藉由其只和錯誤有關的條件之下，去解出錯誤的所在。但即使是如此，由症狀解回錯誤序列並不是唯一的，我們可以以以下的例子來証明。假設錯誤序列$\mathbf{e}_i$會一個已定的錯誤序列$\mathbf{e}$加上編碼過後的碼序列$\mathbf{x}_i$，即$\mathbf{e}_i = \mathbf{e} + \mathbf{x}_i$，則我們可以發現：

$$e_iH^T = eH^T + 0 = eH^T \qquad (4.1\text{-}18)$$

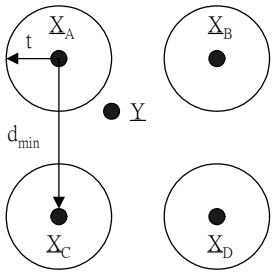
所以總共有多少的編碼，就會有多少個錯誤序列$e_i$產生一樣的症狀序列。我們產用最大相似度的方法，取錯誤序列裡 1 值最少的，也就是說發生錯誤最少的錯誤序列當成真正的錯誤序列。

在找到錯誤症狀後，若是錯誤只有一個時，即錯誤向量為一個含有N個元素(N-tuple)的序列(0,0,...,1,...0)，而 1 處在第$l_{th}$個位置，則症狀即為$S = eH^T = \sum_{i=0}^{N-1} e_ih_i = h_l$，式中$h_l$表示$H^T$的第$l_{th}$列。若錯誤有兩個時，則$S = h_l + h_m$，其中l和m表示錯誤序列中為 1 的位置。所以症狀其實就是將檢查矩陣中對應到錯誤序列中為為 1 的列相加後所得的結果。最直覺的解碼方法就是將症狀和檢查矩陣相比較，觀察症狀是由那幾列相加而成，藉此找到錯誤的所在。

當編碼只能解一個錯誤時，我們可以利用簡單的查表方法，找出症狀和檢查矩陣的那一列相對應，藉此得知錯誤位置的所在。而當採用的碼所能更正的錯誤位元較多時，則可以以錯誤設陷(error-trapping)解碼器來進行解碼。

## 4.2 卻斯(Chase)演算法

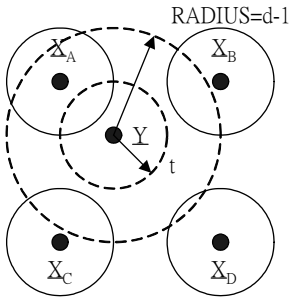
在傳統的硬式代數解碼方法中，其解碼情形有如(圖 4.2-1)所示：




圖(4.2-1) 傳統代數解碼示意圖

圖中$\underline{X}_A, \underline{X}_B, \underline{X}_C, \underline{X}_D$表示四個有效編碼，而兩個碼間最小距離為$d_{min}$，$\underline{Y}$則表示接收到的序列所在位置，t所代表距離為每個碼能將錯誤解回來的範圍。則由圖可看出，$\underline{Y}$座落的位置已經在四個有效編碼的範圍外，因

此無法正確的解碼回去，而卻斯(Chase)演算法則解決了此問題。




圖(4.2-2) 卻斯(Chase)演算法解碼示意圖

在上圖(4.2-2)中，顯示了卻斯演算法解碼時的情形，卻斯利用傳輸時通道影響所給的訊息，試著讓解碼的過程更加的精確。卻斯的想法主要是利用將1散步在各個位置的二位元序列來組成一組測試模組來測試。其方法為將接受到的序列$\underline{Y}$以二位元的加上加上測試樣本TP，以得到新的測試序列$\underline{Y}'$，即：

$$\underline{Y}' = \underline{Y} \oplus TP \qquad (4.2\text{-}1)$$

在(圖 4.2-2)中可看出，原本的序列$\underline{Y}$沒有處在可解碼的有效範圍中，但在加上了測試樣本後，其有效的解碼範圍將會隨著測試樣本的多寡而增加，以使得新的測試序列$\underline{Y}'$落在$\underline{X}_A$的有效解碼範圍中，而能正確的解碼。

在對新的測試序列進行硬式解碼時，我們會將新的測試樣本解為有效碼$\underline{X}_A$，此時會得到一個新的錯誤序列$\underline{e}'$，其和原本的接受到的序列應有的錯誤序列$\underline{e}$的關係式如下所示：

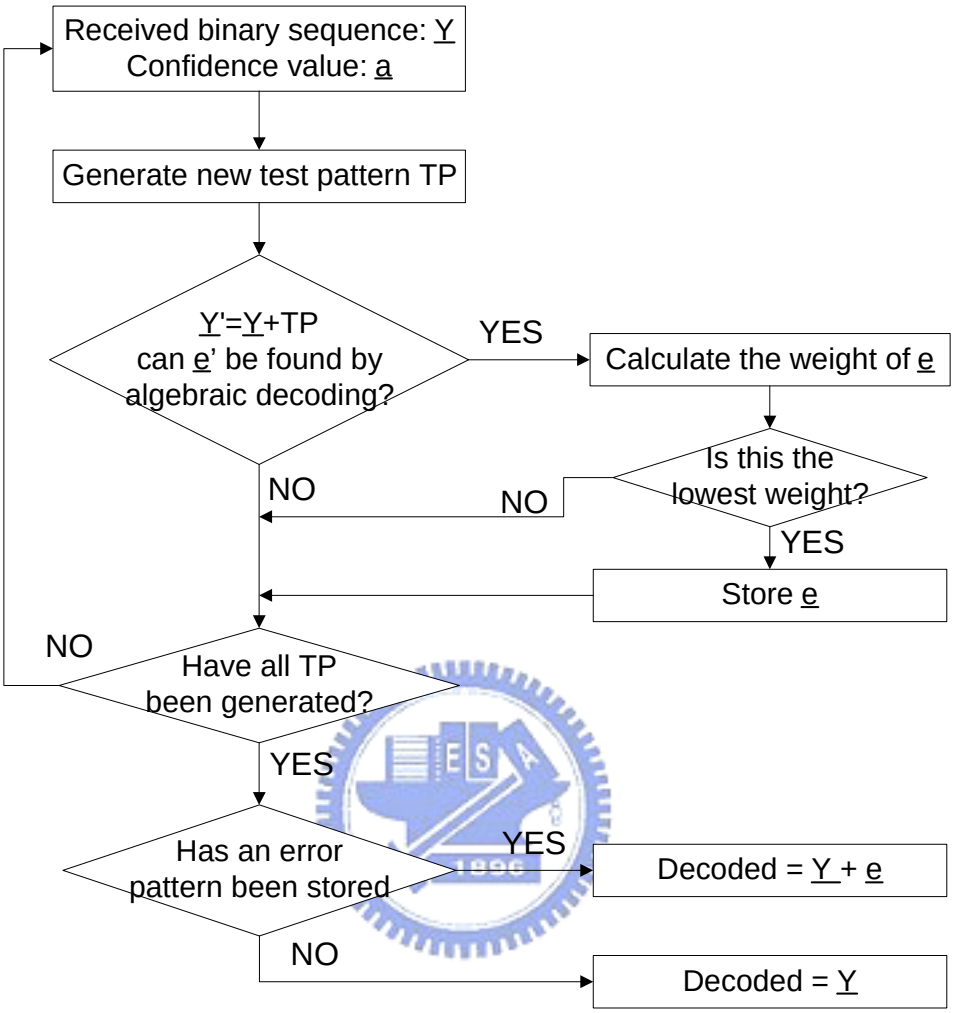
$$\underline{e} = \underline{e}' \oplus TP \qquad (4.2\text{-}2)$$

藉由此關係式，我們即可找到實際的錯誤序列，並由此判斷此解碼是否會最接近原接受訊號的解碼。由最大相似解碼(Maximum likelihood decoder)的觀念，我們所要找到最接近的解碼可根據錯誤序列來加以判斷，而對於一個錯誤序列的重量定義如下：

$$W(\underline{e}) = \sum_{i=1}^{n} e_i |a_i| \qquad (4.2\text{-}3)$$

此式中的$a_i$值為序列$\underline{X}$經過通道影響後的軟式輸入值，而重量的值越小，表示其錯誤的值較小，也就是說其可信度越高。

卻斯演算法的過程可由下圖(4.2-3)來表示，




圖(4.2-3) 卻斯(Chase)演算法流程圖

在每次的解碼過程中，此演算法將一個含有n個元素的集合用來考慮解碼，此集合包含了接收到的二位元序列$\underline{Y}$以及其對應的可靠度軟式參考值$a_i$，此即會流程圖中的第一步。接著產生測試樣本TP的組合，對於每個樣本利用二位元的加法產生新的測試序列$\underline{Y}'$，之後再對每個新的測試序列進行傳統的硬式解碼。若硬式解碼出來的解果產生了一個非零的錯誤序列$\underline{e}'$，則我們可以利用式(4.2-2)將實際的錯誤序列$\underline{e}$找回來。在找回錯誤序列後，利用式(4.2-3)即可得知此錯誤所帶來的錯誤重量W,若W的值為最小時，即表示此解碼最為可能，故將當時的錯誤序列$\underline{e}$儲存下來。在測試完所有的測試樣本後，將檢查是否有錯誤序列$\underline{e}$被儲存下來，

若有即可將Y序列解碼為序列；若沒有錯誤序列儲存下來，則將原本接受到的序列Y值輸出。

測試樣本的取捨同樣也會影響著更錯的效能，卻斯提出了三個取捨的方法，而在本論文中所採用的為第二種方法。此方法的精神是依照接受序列的可靠度參考值$a_i$來決定錯誤樣本的組合，其將選擇 $p=\lfloor d_{\min}/2 \rfloor$ 個最不可靠的位元做為測試樣本更動的地方，經由 0 和 1 在這$\lfloor d_{\min}/2 \rfloor$個位置做排列組合而產生$2^{\lfloor d_{\min}/2 \rfloor}$個測試樣本。以BCH(7,4,3)為例，若接收到的情形如(表 4.2-1)所示，$y_i$為其判斷序列，$a_i$為可靠度參考值，則

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $a_i$ | -0.9 | -1.3 | -0.4 | -2.2 | +0.6 | -1,2 | +0.8 |
| $y_i$ | 0 | 0 | 0 | 0 | 1 | 0 | 1 |

表(4.2-1) 軟式輸入訊號的範例表

可看出第 2，4，6 的位置可靠性是較低的，因此所產生的測試樣本將如(表 4.2-2)所示。產生測試樣本之後，再以(圖 4.2-3)所示的流程，即可得到硬式的解碼輸出。

| | |
|---|---|
| $TP_0$ | 0 0 0 0 0 0 0 |
| $TP_1$ | 0 0 0 0 0 0 1 |
| $TP_2$ | 0 0 0 0 1 0 0 |
| $TP_3$ | 0 0 0 0 1 0 1 |
| $TP_4$ | 0 0 1 0 0 0 0 |
| $TP_5$ | 0 0 1 0 0 0 1 |
| $TP_6$ | 0 0 1 0 1 0 0 |
| $TP_7$ | 0 0 1 0 1 0 1 |

表(4.2-2) 測試樣本範例表

## 4.3 以卻斯(Chase)理論為基礎推算軟式輸出(Soft Output)結果的方法

在這節中，我們將討論如何在做卻斯演算法時，經由數學的傳換，找出軟式的輸出結果，以配合在做乘積碼解碼時反複運作的輸式輸出需要。

在上節的卻斯演算法的解碼過程中，其主要的觀念在於從所有的碼字元之中挑出一個最接近接收到的序列$\underline{y}$的碼字元做為解碼序列，也就是說若解碼得到的序列關係式如下：

$$D = C^i \quad \text{if} \quad P\{\underline{x} = C^i | \underline{y}\} > P\{\underline{x} = C^j | \underline{y}\} \quad \forall j \neq i \qquad (4.3\text{-}1)$$

其中$C^i = (c_1^i, c_2^i, \cdots, c_n^i)$為第$i_{th}$個BCH(n, k, $d_{min}$)碼字元，$D = (d_1, d_2, \cdots, d_n)$則是最後決定最接近傳送端接受到序列$\underline{y}$的解碼後序列。

同樣的關係將會被用在軟式輸出的建立上，但在軟式輸出上，我們將採用對數相似度來估計其可靠度。對一個解碼後的序列$D$其中的一個位元$d_j$的對數相似度我們計為$(LLR)_j$，其定義如下：

$$(LLR)_j = \ln\left(\frac{P\{x_j = +1 | \underline{y}\}}{P\{x_j = -1 | \underline{y}\}}\right) \qquad (4.3\text{-}2)$$

若將此定義由各個碼字元做成的碼集合為觀點來觀察時，上式(4.3-2)分子的部分可以以下式表示：

$$P\{x_j = +1 | \underline{y}\} = \sum_{C^i \in S_j^{+1}} P\{\underline{x} = C^i | \underline{y}\} \qquad (4.3\text{-}3)$$

式中$S_j^{+1}$表示由$c_j^i = +1$的碼字元所組成的集合，同樣的觀點可將式(4.3-2)分母以下式表示：

$$P\{x_j = -1 | \underline{y}\} = \sum_{C^i \in S_j^{-1}} P\{\underline{x} = C^i | \underline{y}\} \qquad (4.3\text{-}4)$$

式中$S_j^{-1}$表示由$c_j^i = -1$的碼字元所組成的集合。將貝爾定律應用到式(4.3-3)以及式(4.3-4)並假設所有不同的碼字元出現的機率為常態分

配(uniformly distributed)，則我們可以將式(4.3-2)中對$d_j$的對數相似度$LLR$改寫如下：

$$(LLR)_j = \ln\left(\frac{\sum_{C^i \in S_j^{+1}} P\left\{\underline{y}\middle|\underline{x}=C^i\right\}}{\sum_{C^i \in S_j^{-1}} P\left\{\underline{y}\middle|\underline{x}=C^i\right\}}\right) \qquad (4.3\text{-}5)$$

而式中的條件機率值可以展開如下：

$$P\left\{\underline{y}\middle|\underline{x}=C^i\right\}=\left(\frac{1}{\sqrt{2\pi}\sigma}\right)^n \exp\left(-\frac{\left|\underline{y}-C^i\right|^2}{2\sigma^2}\right) \qquad (4.3\text{-}6)$$

式中

$$\left|\underline{y}-C^i\right|^2 = \sum_{l=1}^{n}\left(y_l - c_l^i\right)^2 \qquad (4.3\text{-}7)$$

將式(4.3-6)代入式(4.3-5)中，我們可以得到：

$$(LLR)_j = \ln\left(\frac{\sum_{C^i \in S_j^{+1}} \exp\left(-\frac{\left|\underline{y}-C^i\right|^2}{2\sigma^2}\right)}{\sum_{C^i \in S_j^{-1}} \exp\left(-\frac{\left|\underline{y}-C^i\right|^2}{2\sigma^2}\right)}\right) \qquad (4.3\text{-}8)$$

利用上式即可找出每個bit的對數相似度值，只是在實際的應用上，此式太過於複雜，故需要對其做簡化的動作。我們以$C^{\min(+1)}$表示一個碼字元其第$j_{th}$個位元值為+1(即$c_j^{\min(+1)}=+1$)，並且和接收序列$\underline{y}$的歐基理德距離(Euclidian distance)最小；以$C^{\min(-1)}$表示一個碼字元其第$j_{th}$個位元值為-1(即$c_j^{\min(-1)}=-1$)，並且和接收序列$\underline{y}$的歐基理德距離最小。則第$j_{th}$個位元的對數相似度值可近似為：

$$(LLR)_j = \frac{1}{2\sigma^2}\left(\left|R-C^{\min(-1)}\right|^2 - \left|R-C^{\min(+1)}\right|^2\right) \qquad (4.3\text{-}9)$$

利用此式子，只要能找到$C^{\min(+1)}$以及$C^{\min(-1)}$就可以得到每個位元的對數相似度值，再以此值做為軟式輸出，即可應用到反複運算的解碼法當中，藉此將乘積碼解碼。

在實際的應用上我們不會真的把所有的碼字元找過一遍，這樣太過於複雜，所以我們利用在做卻斯演算法中，會以不同的測試樣本來做解碼，而此時對於每一個解碼出來的碼字元，將會計算其歐基理德距離：

$$M^q = \left| \underline{y} - C^q \right|^2 \qquad (4.3\text{-}10)$$

在卻斯演算法中，我們一定可以找到一個和接收序列$\underline{y}$歐式距離最小的碼字元$C^d$，但同時也會從測試樣本群的解碼過程中，試著找出另一碼字元$C^c$其符合$c_j^c \neq c_j^d$的條件，並且為符合此條件中和接收序列$\underline{y}$歐基理德距離最小的碼字元。當我們在測試樣本中找到此種碼字元時，可以用下式來代表$d_j$的對數相似度值：

$$y'_j = \left( \frac{M^c - M^d}{4} \right) c_j^d \qquad (4.3\text{-}11)$$

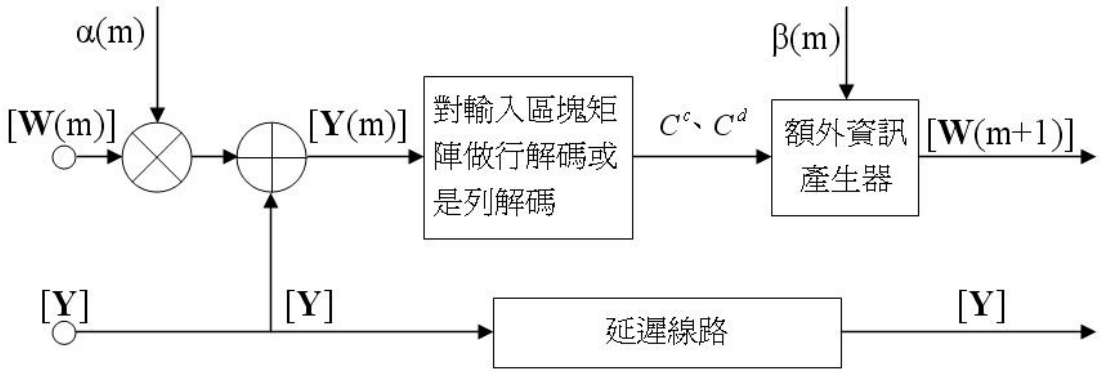
若從中找不到符合的碼字元$C^c$則以下式表示其值：

$$y'_j = \beta(m) \times c_j^d \qquad (4.3\text{-}12)$$

式中$\beta(m)$為可靠度係數(Reliable Factor)，m 代表疊代次數，為一個和疊代次數有關的係數，會隨著疊代次數的增加而加大。

## 4.4 乘積碼的軟式輸出(Soft Output)渦輪解碼疊代法

在有了每一個位元的對數相似度值後，我們要將其應用在區塊乘積




圖(4.4-1) 渦輪解碼疊代法方塊圖

碼的解碼之中，而其架構圖如圖(4.4-1)所示，我們需要將收到的資料先做列向量的解碼，再將解碼後的資料提供給行向量做解碼，如此縱橫交錯一次，才算是一個完整的解碼。

在圖(4.4-1)中，[Y]代表了接收的區塊矩陣，此矩陣經由卻斯演算法以及式(4.3-11)以及式(4.3-12)的解碼計算而得到軟式輸出值，將此輸出值和原本的輸式輸入值相減可得到額外資訊[W(m)]，其中 m 為疊代次數。其中一列或一行中的資訊可以表示如下：

$$w_j = y'_j - y_j, \quad 1 \le j \le n \qquad (4.4\text{-}1)$$

[Y(m)]則是在第 m 次的疊代計算中，經過額外資訊加權後解碼器的輸入，其和接收到的區塊矩陣[Y]可用以下的關係式來表示：

$$[Y(m)] = [Y] + \alpha(m)[W(m)] \qquad (4.4\text{-}2)$$

式中$\alpha(m)$為[W(m)]的權重係數(Weighting Factor)，關係著額外資訊對第 m 次輸入[Y(m)]的影響程度，隨著疊代次數 m 的增加，其值也漸漸增加。如圖(4.4-1)所示，如此疊代足夠的次數 M 後，將最後得到的[Y(M)]做正負的判斷後即可得到最後的解碼輸出。

# 第五章　利用碼字元(Code Wode)間的相關性解碼

在本章中將介紹不同於前兩章的解碼方法，我們希望找出一個擁有較低複雜度的解碼方法。此章提出利用碼字元相互之間的相關係來做解碼的依據，在此種方法下的解碼只需要用到比較器(Compare element)以及互斥器(Exclusive-OR element)即可得到額外資訊，藉此低複雜度的運算來做解碼。如此一來，在每次疊代處理下都可省卻運算的複雜度。

## 5.1　碼字元相關性解碼演算法

此節主要在說明如何利用碼字元的相關性來做解碼的處理，一開始會先對位元間關係作用在對數相似度(LLR)的影響做個介紹，之後在從BCH 碼的檢查矩陣中找出碼字元相互之間的關係，由此導出我們的解碼方法。在最後會用簡單的例子來觀察此演算法的運作情形。

### 5.1.1 對數相似度的數學式子

在第三章中，我們已經提過對數相似度的定義，在本節中將會介紹其本身數值做二位元的加法時會有什麼影響。由之前式(3.1-1)以及式(3.1-2)配合貝爾定律，可以得到事後資訊如下式：

$$\begin{aligned} L\left(u_k|\underline{y}\right) &= \ln\left(\frac{P\left(u_k=+1|\underline{y}\right)}{P\left(u_k=-1|\underline{y}\right)}\right) \\ &= \ln\left(\frac{P\left(u_k=+1\right)}{P\left(u_k=-1\right)}\right)+\ln\left(\frac{P\left(\underline{y}|u_k=+1\right)}{P\left(\underline{y}|u_k=-1\right)}\right) \\ &= L\left(u_k\right)+L\left(\underline{y}|u_k\right) \end{aligned} \qquad (5.1\text{-}1)$$

而軟式輸出的結果大都由此式計算得來。現在來看一個位元單位的對數相似度$L(u_k)$之性質，由定義中我們知道：

$$L(u_k)=\ln\left(\frac{P(u_k=+1)}{P(u_k=-1)}\right)=\ln\left(\frac{P(u_k=+1)}{1-P(u_k=+1)}\right) \qquad (5.1\text{-}2)$$

將上式做為自然對數(natural logarithm,e)的次方項可得：

$$e^{L(u_k)}=\left(\frac{P(u_k=+1)}{1-P(u_k=+1)}\right) \qquad (5.1\text{-}3)$$

再將上式做交叉相乘後，經過整理可得下式：

$$P(u_k=+1)=\left(\frac{e^{L(u_k)}}{1+e^{L(u_k)}}\right) \qquad (5.1\text{-}4)$$

以及

$$P(u_k=-1)=1-P(u_k=+1)=\left(\frac{1}{1+e^{L(u_k)}}\right) \qquad (5.1\text{-}5)$$

利用這兩個式子，以及互斥的性質，若將-1的值設為編碼上的單位零時，我們可以得到：

$$L(u_1\oplus u_2)=\ln\left(\frac{P(u_1=+1)\times P(u_2=-1)+[1-P(u_1=+1)][1-P(u_2=-1)]}{P(u_1=+1)\times P(u_2=+1)+[1-P(u_1=+1)][1-P(u_2=+1)]}\right)$$

$$=\ln\left[\frac{\left(\frac{e^{L(u_1)}}{1+e^{L(u_1)}}\right)\left(\frac{1}{1+e^{L(u_2)}}\right)+\left(\frac{1}{1+e^{L(u_1)}}\right)\left(\frac{e^{L(u_2)}}{1+e^{L(u_2)}}\right)}{\left(\frac{e^{L(u_1)}}{1+e^{L(u_1)}}\right)\left(\frac{e^{L(u_2)}}{1+e^{L(u_2)}}\right)+\left(\frac{1}{1+e^{L(u_1)}}\right)\left(\frac{1}{1+e^{L(u_2)}}\right)}\right]$$

$$=\ln\left(\frac{e^{L(u_1)}+e^{L(u_2)}}{1+e^{L(u_1)}e^{L(u_2)}}\right) \qquad (5.1\text{-}6)$$

$$\approx(-1)\times sign(L(u_1))\cdot sign(L(u_2))\cdot\min(|L(u_1)|,|L(u_2)|) \qquad (5.1\text{-}7)$$

接著我們用⊞符號表示對數相似度的二位元加法，其定義如下：

$$L(u_1)\boxplus L(u_2)\equiv L(u_1\oplus u_2) \qquad (5.1\text{-}8)$$

從式(5.1-6)或式(5.1-7)中，可以得到以下的結果：

$$L(u)\boxplus\infty=-L(u)\text{，}\qquad L(u)\boxplus-\infty=L(u)\tag{5.1-9}$$

以及

$$L(u)\boxplus 0=0\tag{5.1-10}$$

若我們對很多的單位進行對數相似度的相加，可以從兩個位元的加法推導出多個位元的加法如下：

$$\begin{aligned}\sum_{j=1}^{J}\boxplus L(u_j)&\equiv L\left(\sum_{j=1}^{J}\oplus u_j\right)\\&=\ln\left(\frac{\prod_{j=1}^{J}\left(e^{L(u_k)}+1\right)-\prod_{j=1}^{J}\left(e^{L(u_k)}-1\right)}{\prod_{j=1}^{J}\left(e^{L(u_k)}+1\right)+\prod_{j=1}^{J}\left(e^{L(u_k)}-1\right)}\right)\end{aligned}\tag{5.1-11}$$

最後可以將其近似於類似式(5.1-7)的形勢，其近似值如下：

$$\begin{aligned}\sum_{j=1}^{J}\boxplus L(u_j)&=L\left(\sum_{j=1}^{J}\oplus u_j\right)\\&\approx(-1)^{J-1}\left(\prod_{j=1}^{J}sign(L(u_j))\right)\cdot\min_{j=1,\cdots,J}\left(\left|L(u_j)\right|\right)\end{aligned}\tag{5.1-12}$$

## 5.1.2 從檢查矩陣中看碼字元的關係式

在第二章中，我們介紹了同位檢查矩陣H，其和碼字元$\underline{x}$的關係可從式(2.1-6)得知，即$\underline{x}\mathbf{H}^{T}=0$。而由此關係可得知，碼字元中對應的各個位元做二進位的加法後，其值應為 0。從式(2,1-7)可以得到檢查矩陣：

$$\mathbf{H}=\begin{bmatrix}\alpha^{(N-1)} & \alpha^{(N-2)} & \cdots & \alpha & 1\\ \alpha^{3(N-1)} & \alpha^{3(N-2)} & \cdots & \alpha^{3} & 1\\ \alpha^{5(N-1)} & \alpha^{5(N-2)} & \cdots & \alpha^{5} & 1\\ \vdots & \vdots & \ddots & \vdots & \vdots\\ \alpha^{(2t-1)(N-1)} & \alpha^{(2t-1)(N-2)} & \cdots & \alpha^{(2t-1)} & 1\end{bmatrix}$$

若將作用在GF($2^n$)中的$\alpha^k$值用向量表示法（以n個 0，1 值表示，即

n-tuple）展開後，即可將上述矩陣中的一列展開為n列。此時配合$\underline{x}\mathbf{H}^{T}=0$這個關係，假設同位檢查矩陣$\mathbf{H}$如下：

$$\mathbf{H}=\begin{bmatrix} h_{0,(N-1)} & h_{0,(N-2)} & \cdots & h_{0,j} & \cdots & h_{0,0} \\ h_{1,(N-1)} & h_{1,(N-2)} & \cdots & h_{1,j} & \cdots & h_{1,0} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ h_{i,(N-1)} & h_{i,(N-2)} & \cdots & h_{i,j} & \cdots & h_{i,0} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ h_{(n\times t-1),(N-1)} & h_{(n\times t-1),(N-2)} & \cdots & h_{(n\times t-1),j} & \cdots & h_{(n\times t-1),0} \end{bmatrix} \quad (5.1\text{-}13)$$

式中$h_{i,j}$，$0\le i\le(n\times t-1);0\le j\le N-1$為二位元值；我們可以得到以下關係式：

$$x_{N-1}h_{i,(N-1)}+x_{N-2}h_{i,(N-2)}+\cdots+x_1h_{i,1}+x_0h_{i,0}=0,\quad 0\le i\le(n\times t-1) \quad (5.1\text{-}14)$$

從此關係式中可得知，將矩陣$\mathbf{H}$中任一列所有值為1所對應到的$x_m$值做二位元加法後，最後所得的值為0。即：

$$\sum_{m\in S_i^{h_m=1}}\oplus x_m=0,\quad 0\le i\le(n\times t-1) \quad (5.1\text{-}15)$$

式中$S_i^{h_m=1}$表示矩陣中第i列裡序號m值使得$h_{i,m}=1$所組成的集合。而將上式中的一個$x_{m'}$移到等式的另一邊，可得到下列關係式：

$$x_{m'}=\sum_{m\in S_i^{h_m=1},m\ne m'}\oplus x_m,\quad 0\le i\le(n\times t-1) \quad (5.1\text{-}16)$$

這樣的關係說明了$x_{m'}$的值可以由另外的幾個位元來加以決定，若配合式(5.1-12)則可以得到額外的對數相似度訊息。

### 5.1.3 解碼演算過程

由上一節中，我們得知了碼字元之間有相關性的存在，利用此相關性可以找到由其它位元提供的額外訊息，藉此來得到軟式輸出，再經由反覆運算來完成解碼的目的。此節將會對其過程詳細解說一遍。

在上一節的介紹中可得知，同位檢查矩陣的大小為$(nt \times N)$，其中編碼是作用在$GF(2^n)$的元素場裡，而此編碼能更正的錯誤量為 t，編碼後長度為 N。由式(5.1-16)可得到 nt 個關係式，每個關係式都能為自己所屬之$S_i^{h_m=1}$集合中的其它位元提供額外的訊息。因此，同一個位元可能會有不止一個式子提供額外的資訊。由於此關係式是由多個位元的正負號來決定的，若其中一個位元出錯，則此關係式將會帶來錯誤的結果，而造成不正確的額外資訊。因此，我們需要一個能判斷採用那一個關係式的方法。

在選擇關係式的過程中，一定要盡量避免選到錯誤的位元，故我們可以合理假設錯誤會發生在可靠度最小的值之位元。因此，在計算一個位元時選擇關係式的原則是從所有關係式中，比較取採用的位元可靠度值，從所有關係式中，找到一個可靠度最大的值來做為計算此位元額外資訊所用的式子。

由式(5.1-12)中可看出，關係式的值是由在$S_i^{h_m=1}$集合中各個位元的正負號做二位元的加法來決定最後額外資訊的正負號，而其絕對值是由集合$S_i^{h_m=1}$中裡絕對值最小的位元來決定的。因此我們採用關係式所得到的資料中最大的做為結果，如此自然可以減少選到錯誤位元所在式子的機會。因此，我們可得到其額外資訊如下式所示：

$$L_e(x_{m'}) = \max_i \left( L\left( \sum_{m \in S_i^{h_m=1}, m \neq m'}^{\oplus} x_m \right) \right)$$

$$\approx \max_i \left( (-1)^{\left(n\left(S_i^{h_m=1}\right)-1\right)-1} \left( \prod_{m \in S_i^{h_m=1}, m \neq m'} sign(L(x_m)) \right) \cdot \min_{m \in S_i^{h_m=1}, m \neq m'} \left( |L(x_m)| \right) \right) \quad (5.1\text{-}17)$$

式中$n\left(S_i^{h_m=1}\right)$為集合$S_i^{h_m=1}$中元素的個數。

如式(5.1-17)所示，此運算所需要的值為集合$S_i^{h_m=1}$中除了自己的所有元素符號位元(sign bit)做連乘後乘上$(-1)^{\left(n\left(S_i^{h_m=1}\right)-1\right)-1}$再乘上除了自己之外絕對值最小的值。在實作中，可以利用先行計算正負值以及找到一列中的最小值和第二小的值來加快運算速度。我們可以先行計算出第 i 列中$(-1)^{n\left(S_i^{h_m=1}\right)-1}\left(\prod_{m\in S_i^{h_m=1}} sign(L(x_m))\right)$的值，如此在計算每一個$x_{m'}$時，只要乘上$x_{m'}$的符號位元的負數，即可得到式(5.1-17)中一列值所求得的正負號。之後，再檢查$x_{m'}$的絕對值是否為 i 列中最小，當其為最小值時，由於要取除了自己之外的最小值，故其值將是第二小值的絕對值；若其不是第 i 列的最小值，則其額外資訊的絕對值將等於之前計算所得的最小值。其演算過程如下所示：

第一步(初始化)

$$m=0,\quad i=0,\quad L_i=0,\quad Lext(x_m)=0,\quad signbits_i=-1,$$

$\text{firstmin}_i$ = 100(任意的大數值), $\text{secondmin}_i$ = 100(任意的大數值),

第二步(先行計算整列的預備值)

for $i=0$ to $nt-1$

  for $m=0$ to $N-1$

    if $x_m \in S_i^{h_m=1}$

      if $|L(x_m)| < \text{secondmin}_i$ then

        if $|L(x_m)| < \text{firstmin}_i$ then (更新最小及第二小值)

          $\text{firstmin}_i = |L(x_m)|$

        else $\text{secondmin}_i = |L(x_m)|$

      endif

      $signbits_i = -1 \times sign(signbits \times L(x_m))$,

    endif

  endfor

endfor

第三步(分別計算各位元值)

for $m=0$ to $N-1$

for $i=0$ to $nt-1$

if $x_m \in S_i^{h_m=1}$

if $|L(x_m)| \neq \text{firstmin}_i$ then（取最小的值為輸出結果）

$L_i = -sign(L(x_m)) \times signbits \times \text{firstmin}_i$

else （取第二小的值為輸出結果）

$L_i = -sign(L(x_m)) \times signbits \times \text{secondmin}_i$

endif

endif

endfor

$Lext(x_m) = \max_i(L_i)$ （取 i 列裡計算結果最大的做為輸出）

endfor

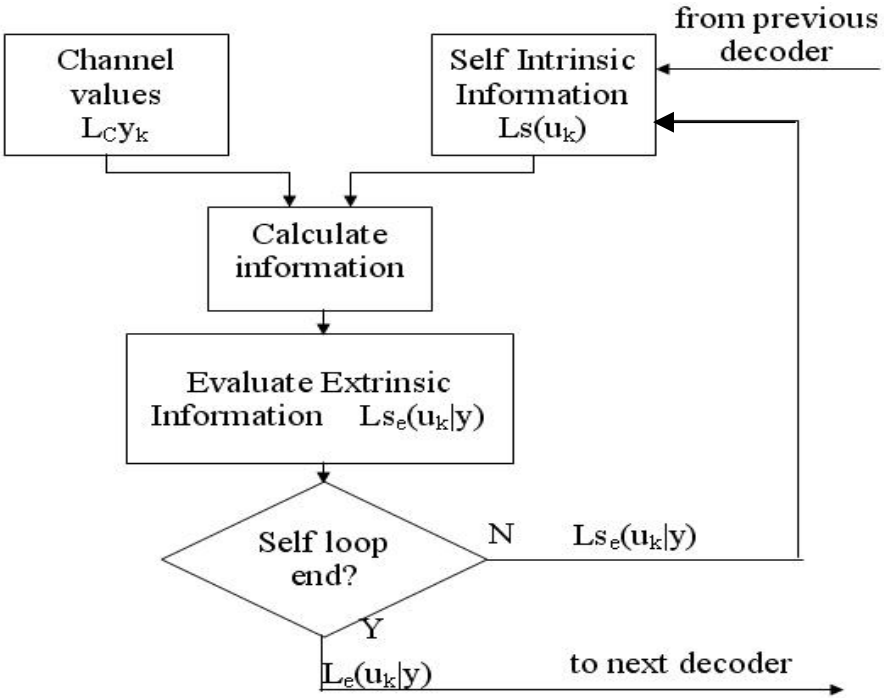
上述的運算過程中，$\text{firstmin}_i$值為儲存第i列中最小值的變數、$\text{secondmin}_i$值為儲存第i列中第二小值的變數、$signbits_i$值是儲存第i列裡進行符號運算後的結果、$L_i$則是運算第i列所給的額外資訊，$Lext(x_m)$為最後輸出的結果。

## 5.2 利用遞迴處理加強解碼效益

此節將介紹如何利用遞迴處理方法來增加此演算法的效能，此方法能幫助碼字元關係解碼的演算法在較小的遞迴次數下即達到較好的解碼效能，而在較低的訊號雜訊比時，更有加強更正錯誤的能力。

此方法是先對同一個方向的解碼動作先行重複遞迴運算三次，將以碼字元之間關係式所解碼得到的額外資訊$Ls_e(U_k)$當成下一次遞迴運算的事前機率資訊$Ls(U_k)$，將其加上經過通道後所得到的本身訊息$Lc y_K$做為解碼器的輸入資訊。如此反複的運算三次，再將第三次所得到的額外資訊$L_e(U_k)$當做另一個方向的事前機率值，而藉此再進行另一方向的解碼。其中$Ls_e(U_k)$為同一方向自行解碼後得到的額外資訊，而$Ls(U_k)$為輸入給同方向做事

前資訊的解碼資料,$L_e(U_k)$則為計算給另一方向的額外事後機率而$L(U_k)$為提供給另一外向做事前資訊的值，此運作情形如圖(5.2-1)所示：



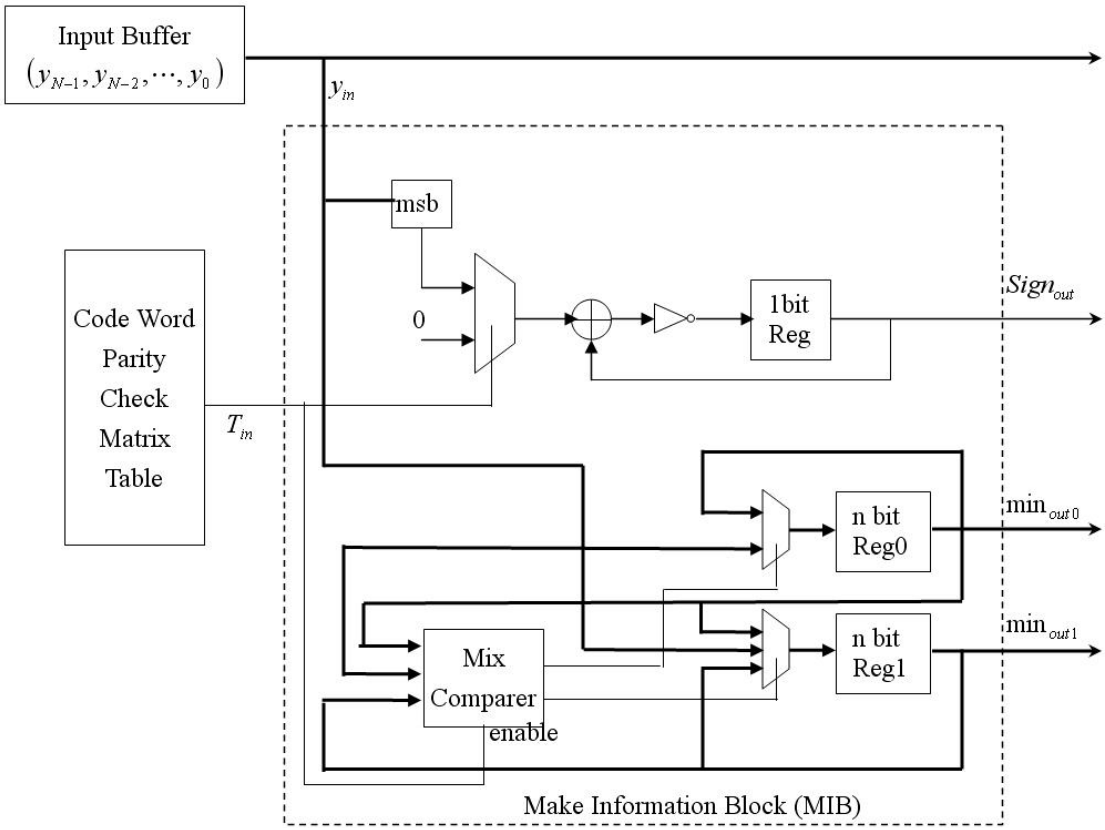

圖(5.2-1) 自我遞迴處理演算流程圖

## 5.3 硬體設計建議

此演算法最大的好處在於其可以利用簡單的硬體設計來實現此演算法，在這節中將介紹硬體設計的方法，用低複雜度的硬體來完成此演算法，達到快速解碼的目的。

在 5.1.3 節裡,我們對其演算法已經仔細介紹過,正如演算法所示，實現硬體時同樣可分為兩個部分，第一個部分為計算位於同位檢查矩陣中同一列裡面其值為 1 的元素所組成的集合$s_i^{h=1}$中每個元素做二位元加法後的資訊，其中包括尋找可靠度最小以及第二小的值，以及將屬於集合$s_i^{h=1}$中的每個元素的符號位元做互斥運算的倒數(inverter)來得到額外資訊的符號資料。此部分可用圖(5.3-1)來表示，圖中上半段是利用輸入資料的符號位元來做運算，其中碼字元同位檢查矩陣表(Code word parity check matrix table)可由隨機存取記憶體(RAM)輸入相對應的

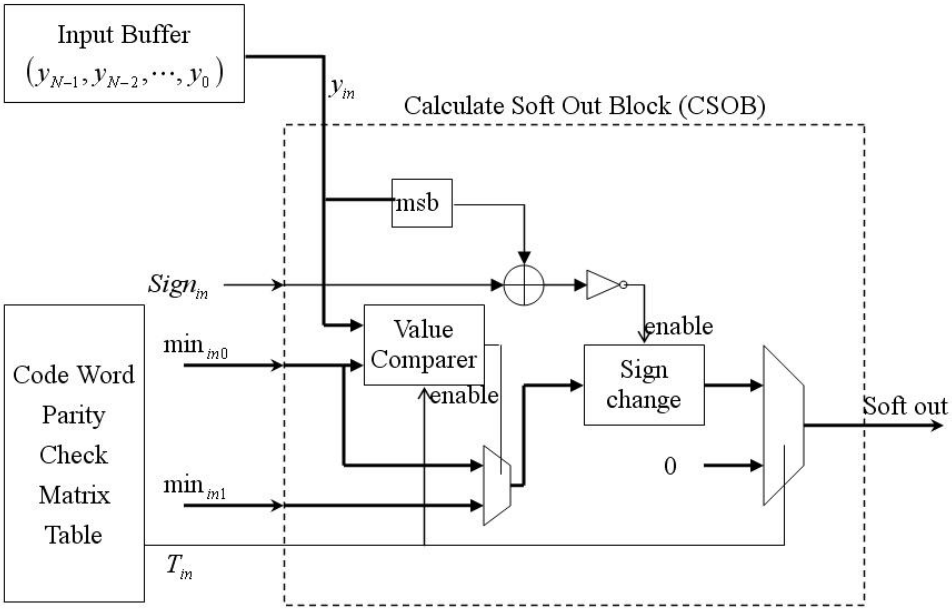
資訊以完成。利用此記憶體輸出值來控制是否進行所需運算。




圖(5.3-1) 計算同位矩陣中同一列的訊息總合硬體架構圖

圖(5.3-1)下半段則是比較可靠度值大小的硬體架構圖，同樣利用由同位檢查矩陣轉化而來的表來控制是否要進行比較的動作。其中 Reg0 是為了能儲存最小值，而 Reg1 則是為了儲存第二小值。此硬體在執行，即將最後所需要的符號位元由 $sign_{out}$ 輸出、最小值由 $\min_{out0}$ 輸出而第二小值由 $\min_{out1}$ 輸出。

在執行 N 次的時間後，我們需要另一個硬體來做計算輸式輸出的動作，此硬體主要是將之前的硬體區塊(Make information block, MIB)所計算出的資訊，配合輸入的資料做處理。如 5.1.3 節裡所介紹的一樣，此計算主要為找出軟式輸出的正負號；此外，在輸入資訊是最小值時，以第二小值為輸出，否則以最小值做為輸出。此部分的硬體架構如圖(5.3-2)所示，主要利用之前一個區塊所算好的資訊做為輸入，$Sign_{in}$ 為符號位元的輸入，$\min_{in0}$ 為之前計算的最小值，$\min_{in1}$ 為次小值。

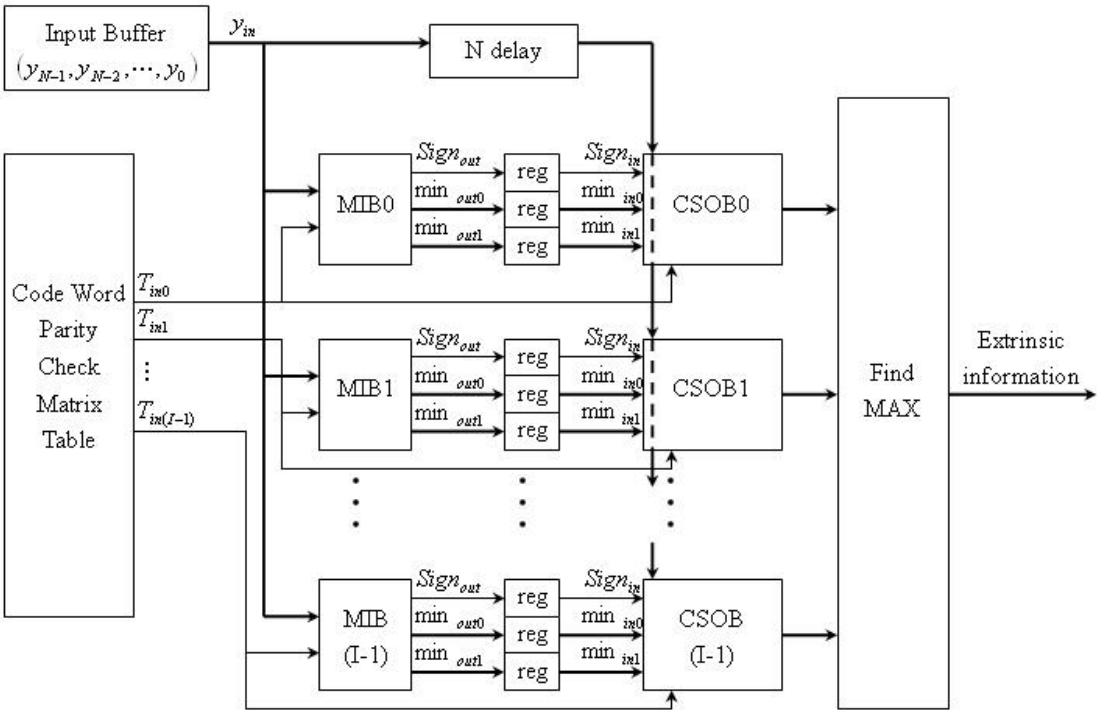



圖(5.3-2) 計算同一列中各個位元所帶訊息之硬體架構圖

此處同樣會用到碼字元同位檢查矩陣表，藉此來判斷此是否屬於集合 $s_i^{h=1}$ 之中，若計算的位元不屬於集合裡，則輸出為 0。

由於同位檢查矩陣的列數不止一個，若我們希望能在 N 的時間之內，將所有的資訊先行計算完成，而在下一個 N 的時間把所有的輸式訊息計算出來，就需要有和同位檢查矩陣一樣多列數的硬體。圖(5.3-3)為實際實現此種演算方法的硬體架構圖，其所對應的同位檢查矩陣有 I 列，圖中 MIB 區塊為圖(5.3-1)的硬體架構，用來先行計算同一列的訊息總合，而 CSOB 則是圖(5.3-2)的硬體架構，用來將同一列中各個位元的資訊計算出來。最後所有計算得到的軟式資訊需要再經過一個比較器，以找出不同列所得的資訊中，絕對值最大的資訊做為最後的輸出。

在實際的應用上，我們還可以以管線(pipe-line)的方式來分段處理，當區塊 CSOB 在解第一個碼字元的各位元軟式輸出時，區塊 MIB 則可以對第二個碼字元做先行處理的動作。如此可節省計算所需的時間，以達到更快速的解碼。




圖(5.3-3) 利用碼字元間相關性解碼方法之硬體架構圖

# 第六章　模擬結果與硬體設計考量

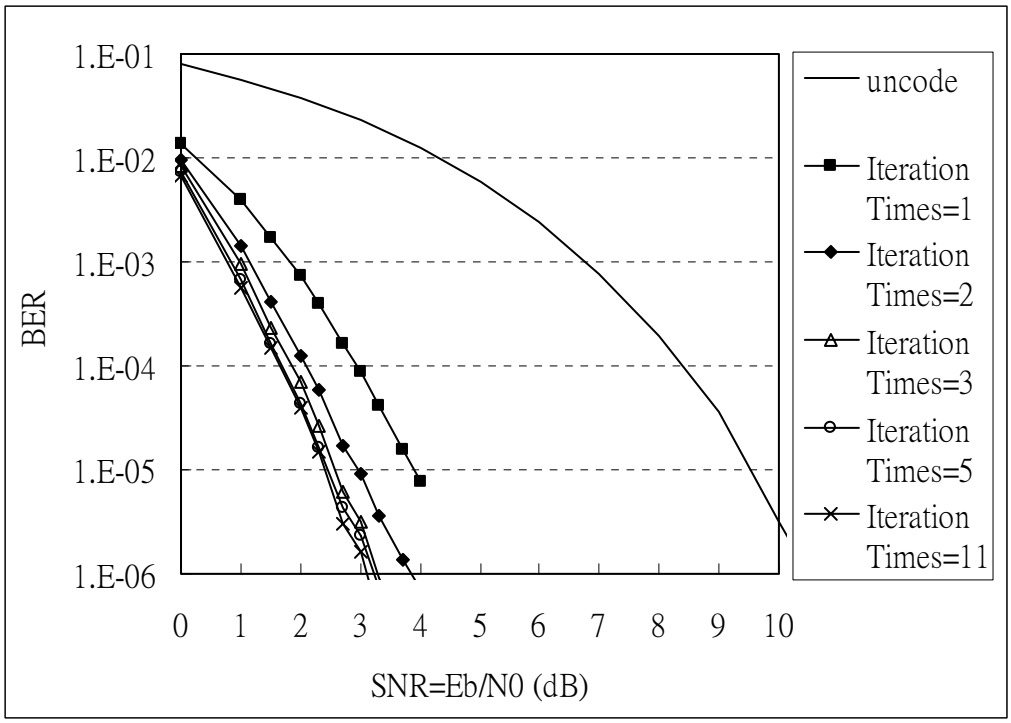
在之前的第三、四、五章，我們已經分別對 MAP 演算法解碼、卻斯(Chase)演算法延伸解碼以及碼字元相關性解碼，三種不同的解碼方法一一介紹，因此本章將會將上述的理論模擬出來，藉由觀察模擬的結果以更加了解各個方法的效能的不同；另外將對各個方法做實現硬體的考量，如此可更加的了解不同方法的優缺點，及其適用的場合。

## 6.1　各個方法的模擬結果

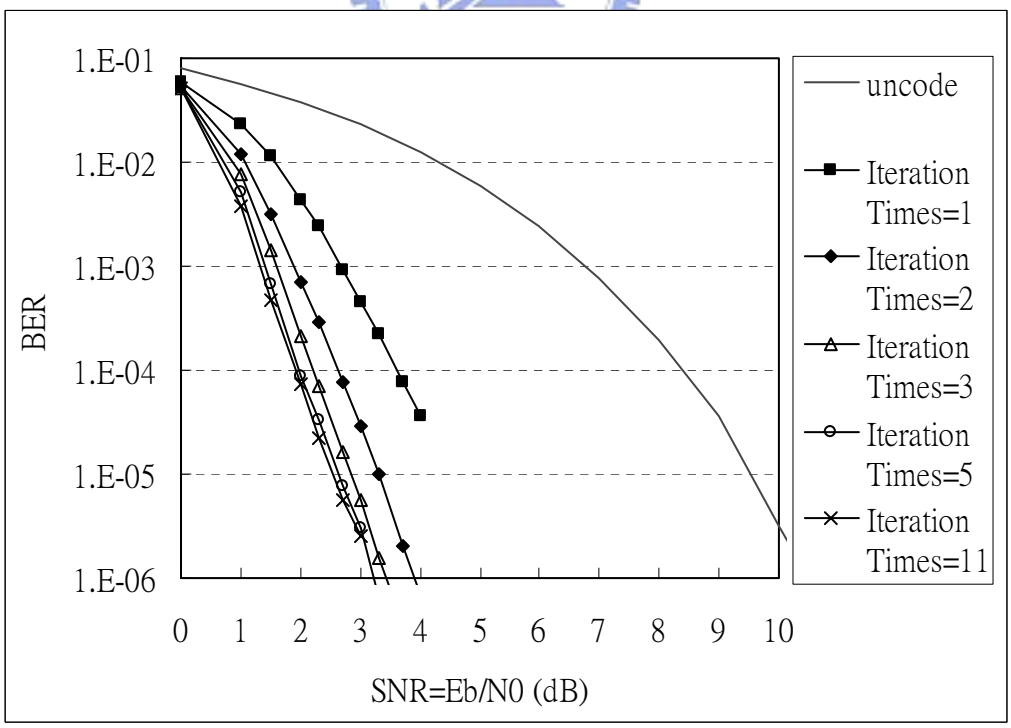
在此章中，我們將以 IEEE802.16a 中所使用到的編碼規格做測試，如表(2.2-1)所示，IEEE802.16a 所用到的乘積編碼有許多不同的構成組合，在本論文中將會取用 eBCH(32,26,4)×eBCH(16,11,4)乘積碼，以及 eBCH(32,26,4)×eBCH(32,26,4)這兩種不同的構成架構來做模擬，藉此觀察不同的解碼方式下的效能。在所有模擬中，使用資料量皆為 50000 個區塊乘積碼。

### 6.1.1 以格子圖觀念解碼方法的模擬結果

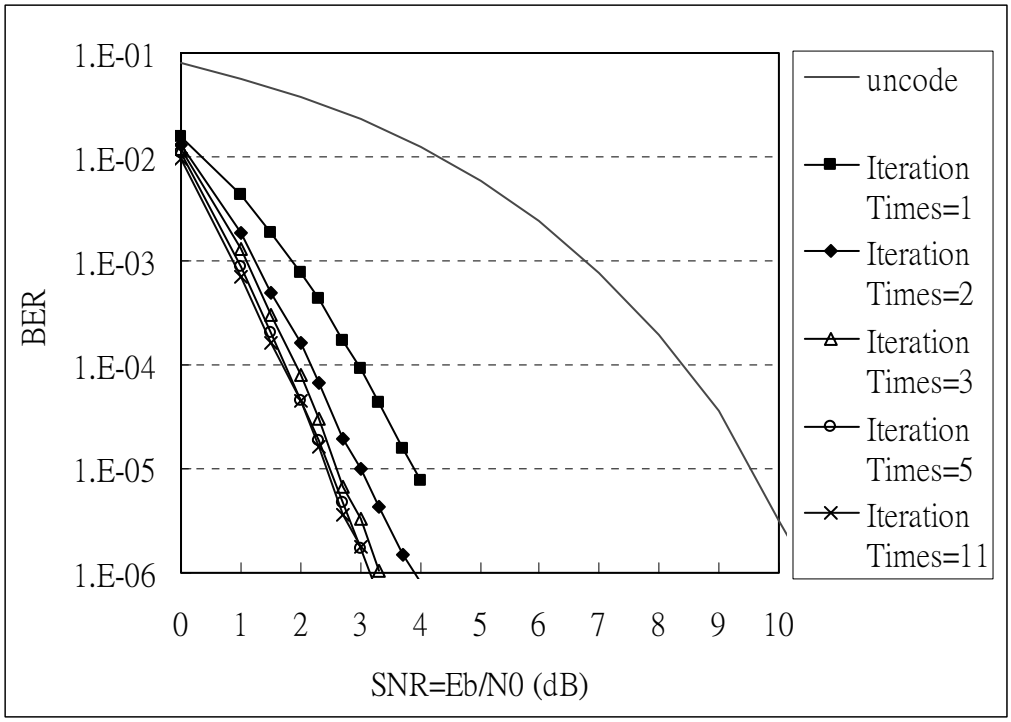
在以格子圖為觀念的解碼方法中，我們採用的是 Log-MAP 演算法，其演算法是輸入的資料做為資訊，藉此推算出格子圖中各個狀態變化發生的機率，再由此推算出每個資料位元的額外資訊。下圖(6.1-1)為 eBCH(32,26)×eBCH(16,11)乘積碼在不同次數的遞迴次數下所模擬出來的位元錯誤率(Bit Error Rate,BER)對照訊號雜訊比(Signal to Noise Ratio,SNR)結果。而圖(6.1-2)則是 eBCH(32,26)×eBCH(32,26)乘積碼的模擬結果。



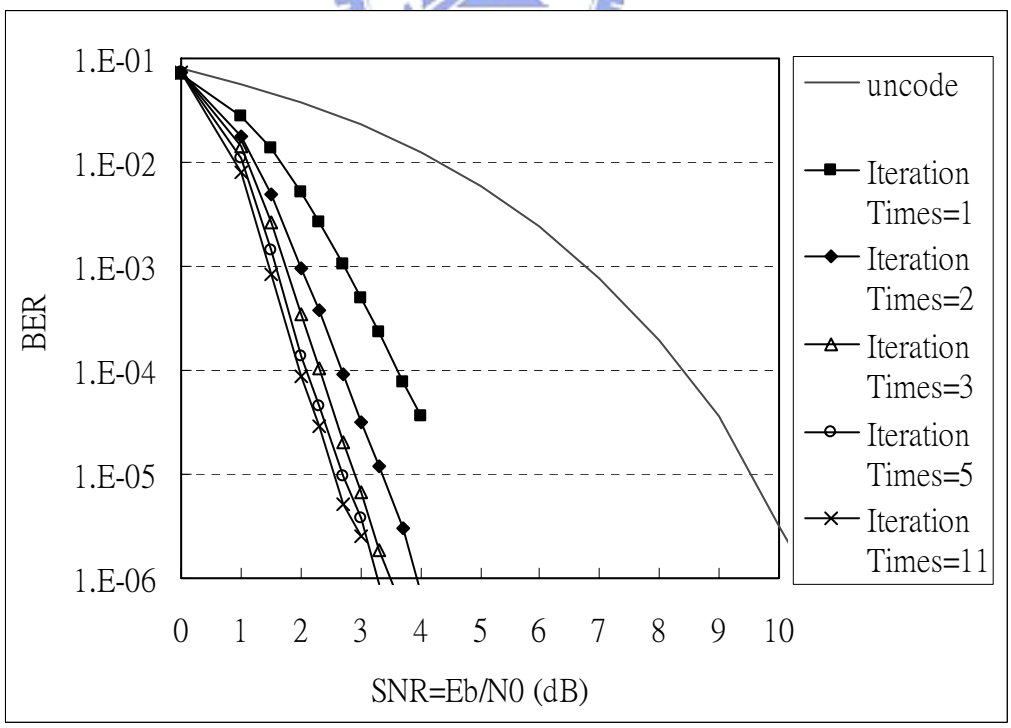

圖(6.1-1) 利用 Log-MAP 演算法對 eBCH(31,26)×
eBCH(15,11)乘積碼解碼的模擬結果




圖(6.1-2) 利用 Log-MAP 演算法對 eBCH(31,26)×
eBCH(31,26)乘積碼解碼的模擬結果




圖(6.1-3) 利用 MAX-Log-MAP 演算法對 eBCH(31,26)× eBCH(15,11)乘積碼解碼的模擬結果




圖(6.1-4) 利用 MAX-Log-MAP 演算法對 eBCH(31,26)× eBCH(31,26)乘積碼解碼的模擬結果

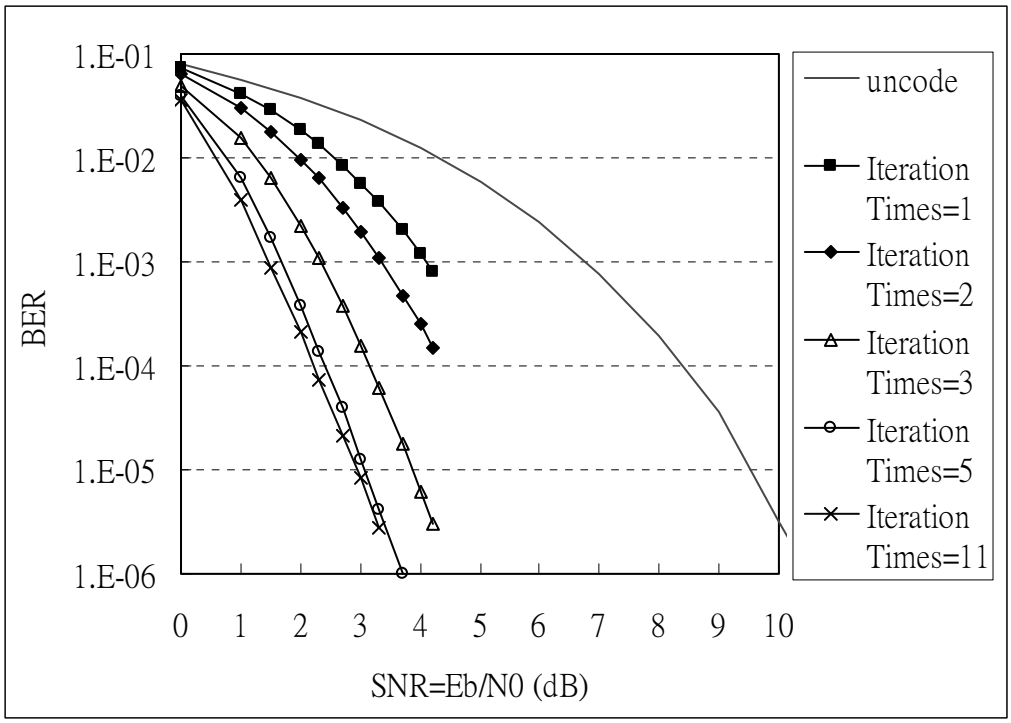
從圖中可看出，Log-MAP 演算法在前幾次的遞迴計算中，可以使得解碼效能增進不少，而大概 3 次的遞迴計算後，其增加的效益就不是很大。做 5 次遞迴和 11 次遞迴所產生的位元錯誤率是差不多的，而由於此演算法較為複雜，故實施應用上會建議以取 3 次遞迴演算來實現。

為了簡化 MAP 演算法的複雜度，在第三章中我們另外提出了將其對應到對數領域後，在採用最大值近似的 MAX-Log-MAP 演算法，其效能雖如圖(6.1-3)及所示，其效能雖較 Log-MAP 演算法為差，但差異並不是很大，而計算複雜度卻相對減少了，這使得 MAP 演算法得以在實際的應用上實現。其中圖(6.1-3)是 eBCH(32,26)×eBCH(16,11)乘積碼在不同次數的遞迴次數下所模擬出來的位元錯誤率(BER)對照訊號雜訊比(SNR)的結果，而圖(6.1-4)則是 eBCH(32,26)×eBCH(32,26)乘積碼所模擬的結果。從圖中同樣可看出我們只需採用 3 次的遞迴計算後，即可達到很高的解碼效益。而此兩種不同編碼所得到的解碼效果以 eBCH(32,26)×eBCH(16,11)較好，此可由兩種編碼的碼率推算出此結果。

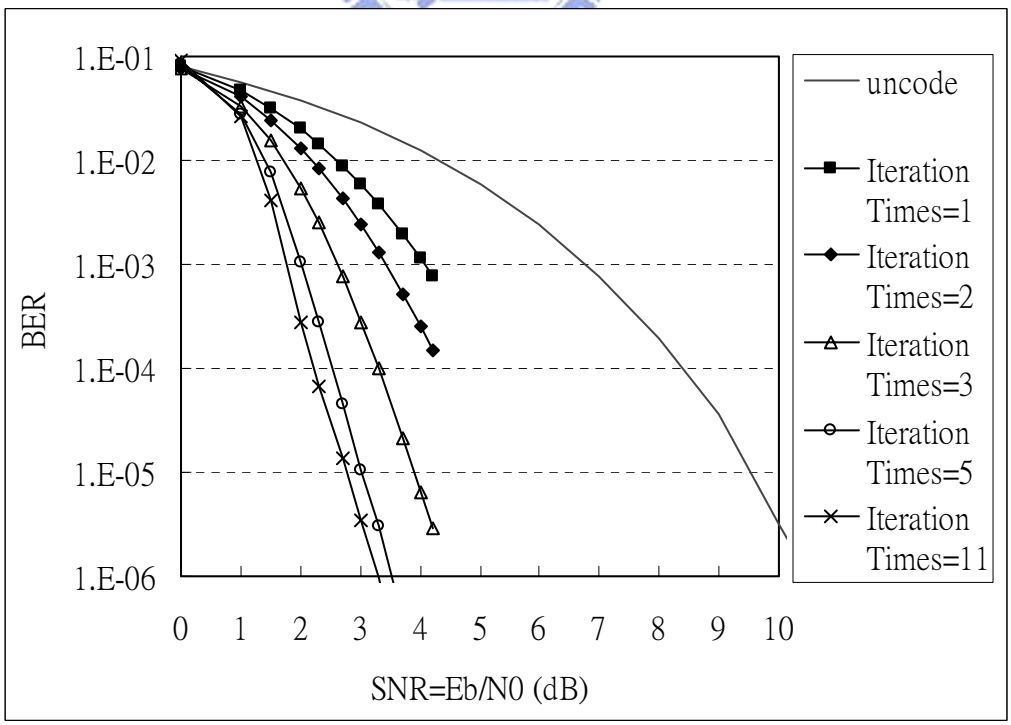
### 6.1.2 卻斯演算法延伸軟式輸出解碼方法的模擬結果

在第四章中，我們另外介紹了一個由卻斯(Chase)演算法延伸的軟式輸出解碼的方法，此方法主要是利用最大相似度的概念來從一些較可能的碼字元中找出各個字元的額外資訊，其演算的複雜度就比 MAP 演算法少了許多。同樣的，我們以兩個不同結構的乘積碼來觀察其解碼效能。圖(6.1-5) 是 eBCH(32,26)×eBCH(16,11)乘積碼以卻斯演算法延伸的軟式輸出解碼方法在不同次數的遞迴處理下所模擬出來結果，其中縱軸為位元錯誤率(BER)而橫軸是訊號雜訊比(SNR)。圖(6.1-6)則是對 eBCH(32,26)×eBCH(32,26)乘積碼解碼後的模擬結果。這兩張圖的測試樣本值皆是設定在 $p=2$ 的情形。

由圖中可以得知，卻斯演算法所延伸的軟式輸出解碼沒有像 MAP 演算法一樣能以少數的遞迴次數即達到最大的解碼效能，此演算法要到 5 次的遞迴計算後才能得到較完整的效能，若只採用 1 或 2 次的遞迴計算




圖(6.1-5) 利用 Chase 演算法延伸的軟式輸出解碼法對 eBCH(31,26)×eBCH(15,11)乘積碼解碼的模擬結果(p=2)




圖(6.1-6) 利用 Chase 演算法延伸的軟式輸出解碼法對 eBCH(31,26)×eBCH(31,26)乘積碼解碼的模擬結果(p=2)

時，則會有很大要效能損失。同樣的，此解碼方法仍是以 eBCH(31,26)xeBCH(15,11)乘積碼有較好的效果，但此兩種編碼的差異跟 MAP 演算法時相比就明顯較小了。

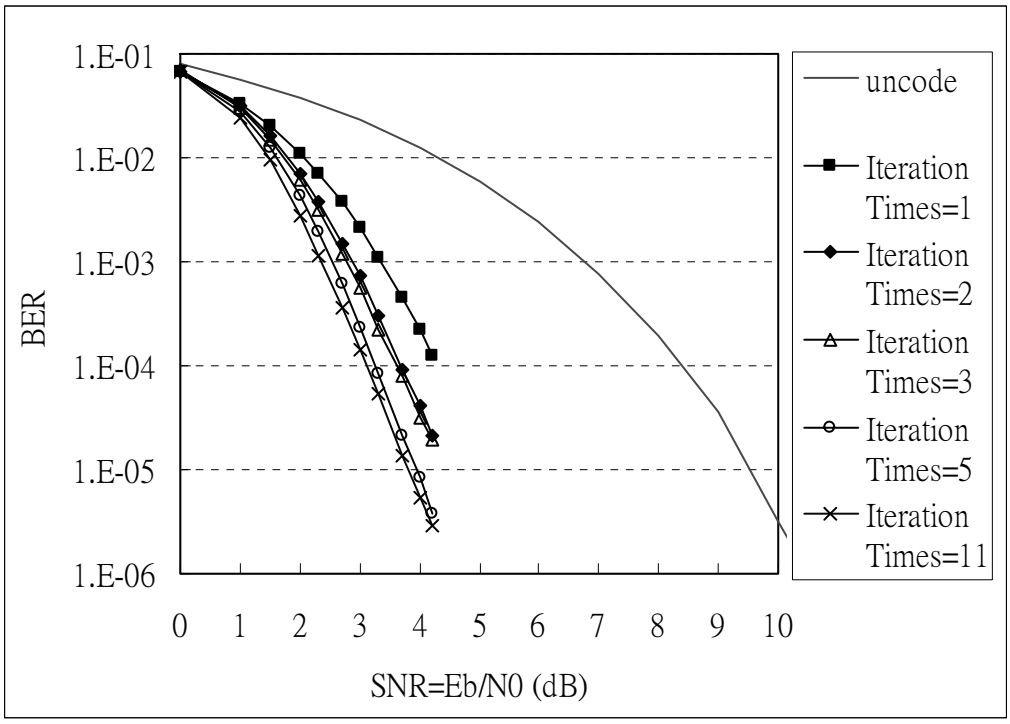
### 6.1.3 碼字元間相關性解碼方法的模擬結果

在第五章中，我們提出了一個更為簡化的解碼方法，利用編碼後的碼字元之間的相關性，只需要簡單的運算位元即可得到各個位元的額外資訊，藉此達到解碼的目的。但相對的其解碼效能就比另外兩種演算法差。

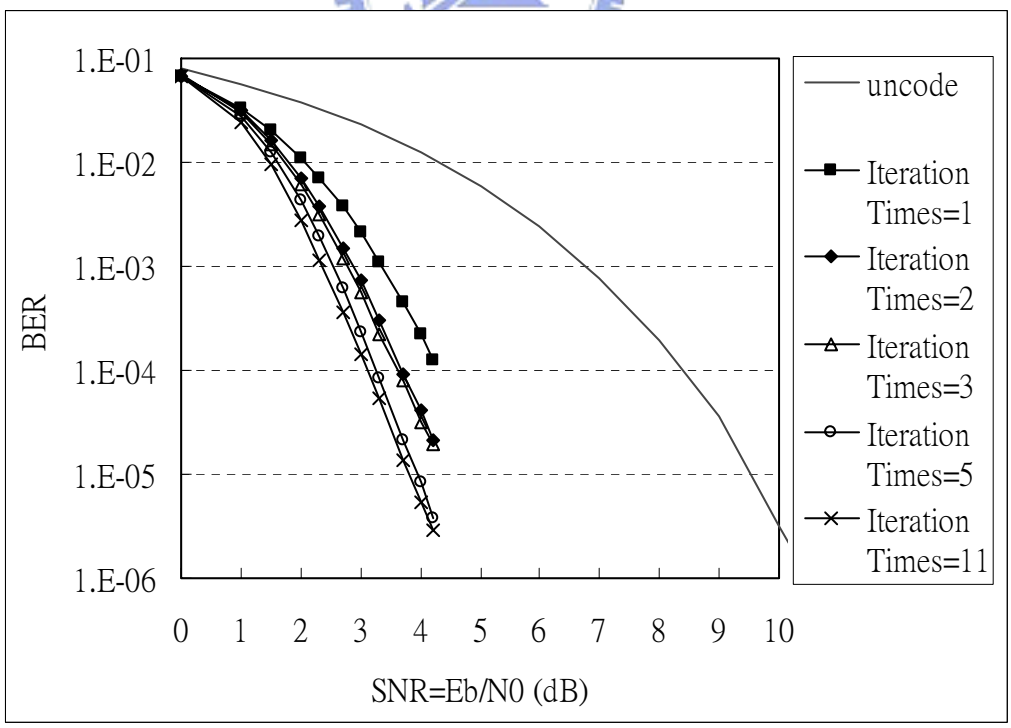
圖(6.1-7) 是將 eBCH(32,26)xeBCH(16,11)乘積碼以第五章所提的方法做解碼後，在不同的遞迴次數下所模擬出來結果圖，其中縱軸為位元錯誤率(BER)而橫軸是訊號雜訊比(SNR)。圖(6.1-8)則是以同樣方法對 eBCH(32,26)xeBCH(32,26)乘積碼解碼的模擬結果。

利用碼字元相關性的解碼方法時，可由圖和圖(6.1-8)中看出其同樣需要較多的遞迴次數才能達到較高的效能。但在 5 次的遞迴之後，其效能的增進同樣有限，故在實際的應用上同樣會建議以5次的遞迴為主。

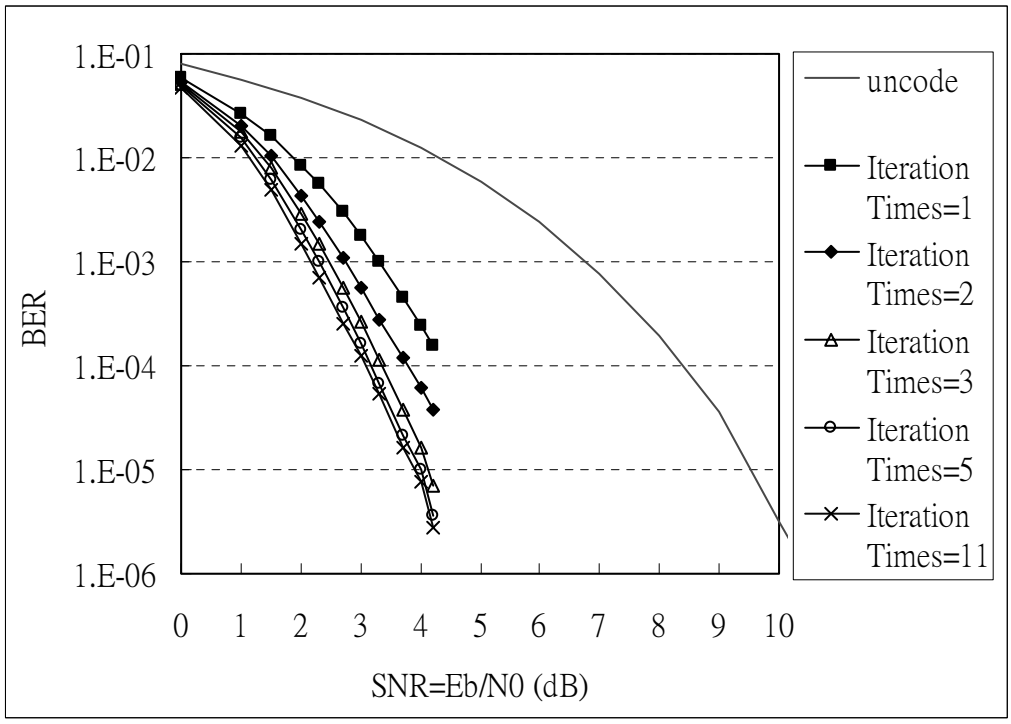
在第五章，我們另外提出了利用自身遞迴來增進碼字元相關解碼方法的效能，圖(6.1-9)和圖(6.1-10)即是在自身遞迴 3 次下，再進行不同方向訊息傳遞的解碼情形，其中圖(6.1-9)是針對 eBCH(32,26)xeBCH(16,11)乘積碼做解碼的模擬結果，而圖(6.1-10)則是對eBCH(32,26)xeBCH(32,26)乘積碼解碼的模擬結果。在有了自身的幫忙下，我們可看出只要較少的整體遞迴次數，其效能就已經接近高遞迴次數時的效能，故以此方法可以只用 3 次的遞迴來做解碼即可接近此解碼方法的效能上限。



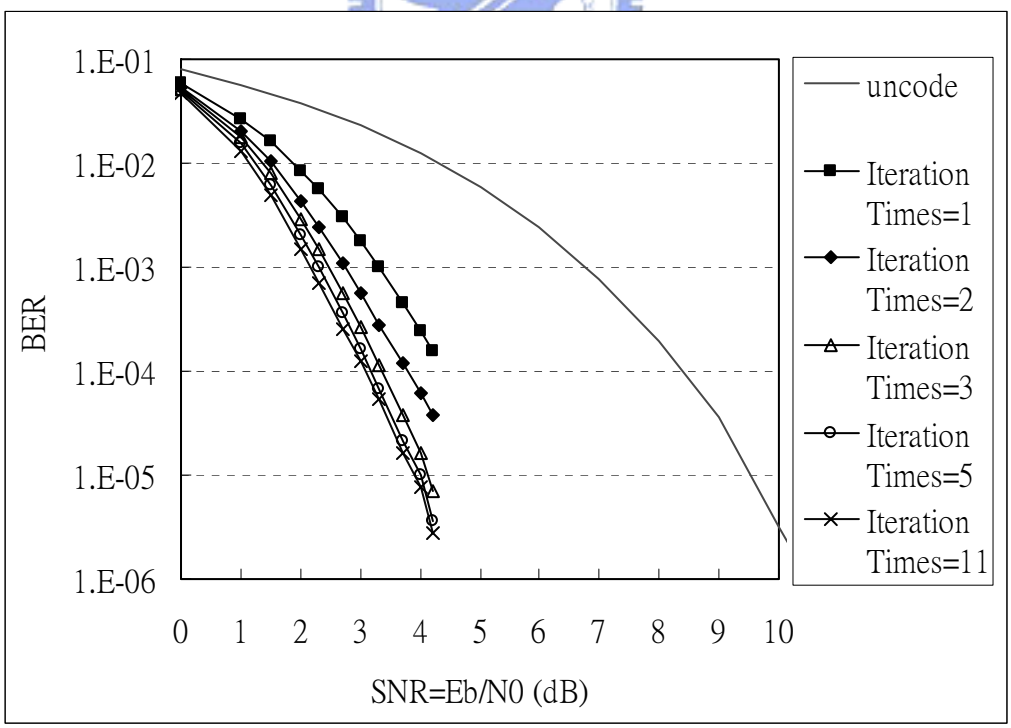

圖(6.1-7) 利用碼字元相關性解碼方法對 eBCH(31,26)× eBCH(15,11)乘積碼解碼的模擬結果(無自身遞迴)




圖(6.1-8) 利用碼字元相關性解碼方法對 eBCH(31,26)× eBCH(31,26)乘積碼解碼的模擬結果(無自身遞迴)




圖(6.1-9) 利用碼字元相關性解碼方法對 eBCH(31,26)× eBCH(15,11)乘積碼解碼的模擬結果(自身遞迴 3 次)




圖(6.1-10) 利用碼字元相關性解碼方法對 eBCH(31,26)× eBCH(31,26)乘積碼解碼的模擬結果(自身遞迴 3 次)

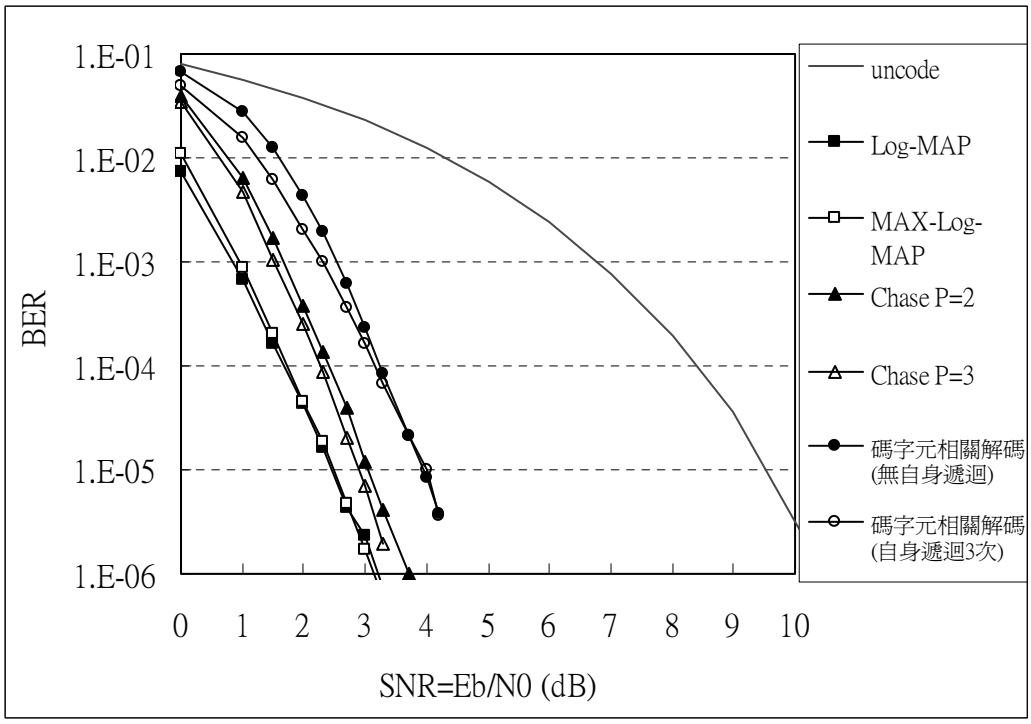
### 6.1.4 三種方法模擬結果的分析與比較

在圖(6.1-11)和圖(6.1-12)中，我們將論文中介紹的三大方法放在一起做比較，如此可以更能容易看出各種方法的好壞。在圖中不論何種方法，皆是採用了 5 次不同方向的遞迴演算後所模擬出來的結果。在格子圖的解碼方法上，可看出 Log-MAP 演算法和 MAX-Log-MAP 演算法所得到的結果差異不會太大，但在三大方法裡，以格子圖為觀念的解碼方法所能達到的效能是最好的，此能從圖(6.1-11)和圖(6.1-12)中輕易的發現。

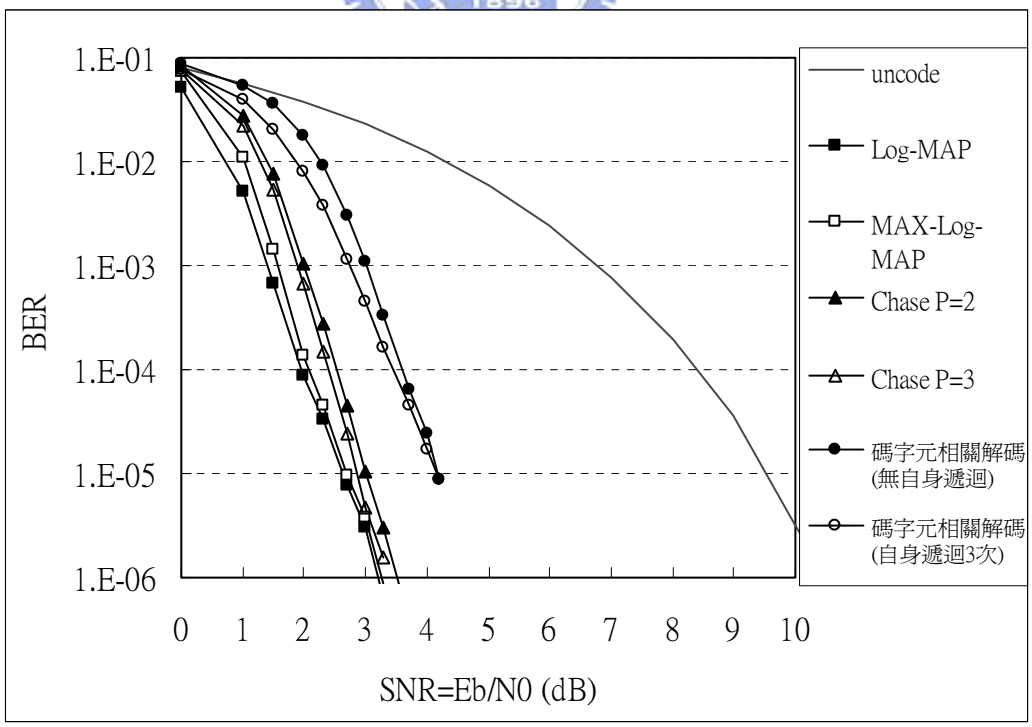
第二個方法是以卻斯演算法延伸解碼，此方法解碼的能力雖然比不上以格子圖為觀念的解碼方法，但其解碼效能仍是很高。圖中另外對測試樣本數的多寡也做了比較，在測試樣本數較高時，其解碼效能同樣的也在增進，在高樣本的條件之下，其和格子圖觀念的 MAP 演算法效能差異就更加的少；但相對的必須附出更多的運算量。

最後則是利用碼字元間相關性來加以解碼的模擬結果，其效能明顯的較另外兩種方法為差，但相對的，其硬體設計上就有較強的優勢。圖中同樣有將加入自身遞迴後的解碼方法做為考量，從模擬結果可看出加入自身遞迴後所得到的效益在信號雜訊比(SNR)加高時即漸漸變小。

由此模擬的結果可看出，當我們需要相當良好的解碼效能時，解碼的方法就必須要採用以格子圖為觀念的方法，若是要求解碼能力的錯誤範圍較廣時，就可以視硬體限制的程度再卻斯演算法或是碼字元相關性解碼中做選擇。




圖(6.1-11) 各種不同方法在 5 次遞迴下對 eBCH(31,26)× eBCH(15,11)乘積碼解碼的模擬結果




圖(6.1-12) 各種不同方法在 5 次遞迴下對 eBCH(31,26)× eBCH(31,26)乘積碼解碼的模擬結果

## 6.2 各個方法的硬體設計考量

在前面一節裡，我們已經看到了各個不同方法的解碼效能，從模擬結果中可以明顯的看出 MAP 演算法擁有最好的解碼能力，Chase 演算法延伸的解碼方法同樣有不錯的解碼效果，而最後提出的碼字元相關解碼的能力是較差的。而在這一節中，我們將對各個方法在實現時所需要的硬體做考量，以了解各個演算法實現上的難度，也藉此能更加了解各種演算法所適合的領域。

### 6.2.1 以格子圖為觀念的解碼運算需求

在第三章裡介紹的 MAP 演算法、Log-MAP 演算法和 MAX-Log-MAP 演算法都是以格子圖為觀念的解碼方法，此方法受到格子圖的影響，必須要對格子圖中的各個狀態(state)做運算。因此，隨著編碼的不同，其狀態數量也不相同，而在區塊 eBCH(n, k, d)碼中，其狀態數就有 $2^{n-k-1}$ 種，因此最少的狀態數也有 $2^3=8$ 個，而若是採用高碼率(Code rate)的編碼方法，則其狀態數也會跟者增加，以 IEEE802.16a 為例，其狀態度最多可以達到 $2^6=64$ 個，跟一般傳統的迴旋碼(Convolutional Code)狀態數目相差許多。此外，區塊 eBCH 碼的解碼是以一個區塊一個區塊來做解碼，故其一個編碼只會有 n 的長度不會像傳統迴旋碼一樣有相當長的連續長度，所以利用此種方式來做區塊 eBCH 解碼是較為沒有效率而且複雜度很高。

由於Log-MAP演算法和MAP演算法的效能是相同的，而Log-MAP演算法明顯降低了許多複雜度，因此本論文只對Log-MAP和MAX- Log_MAP演算法所需的運算要求做考量。首先以Log-MAP演算法來做考量，在對一個eBCH($n_1, k_1, d_1$)×eBCH($n_2, k_2, d_2$)的乘積碼做解碼時，需要對各別的eBCH(n, k)碼計算 $\alpha$ 、 $\beta$ 和 $\gamma$ 三個值，在計算 $\gamma$ 值時，需要用到 $2\times 2n=4n$ 次的常數乘法， $2\times n=2n$ 次的變數乘法以及 $2\times n=2n$ 次的加法。而計算 $\alpha$ 、 $\beta$ 值時，分別需要 $2\times n\times 2^{n-k-1}$ 次比較， $6\times n\times 2^{n-k-1}$ 次的加法。最後在

計算$\alpha$、$\beta$和$\gamma$所給予的額外資訊值時，還需要再做$6\times n\times 2^{n-k-1}$次比較，$10\times n\times 2^{n-k-1}+3$次的加法。故在解一個eBCH(n, k, d)編碼時，以Log-MAP演算法解碼需要$10n\times 2^{n-k-1}$次的比較，$22n\times 2^{n-k-1}+2n+3$次的加法，$4n$次的常數乘法以及$2n$次的變數乘法。而若是以MAX-Log-MAP演算法來解碼，則在計算$\alpha$、$\beta$值以及額外資訊時，可以省略查表的動作以及補正所需要的加法，所以其在解一次eBCH(n, k, d)碼時所需要的運算為：$10n\times 2^{n-k-1}$次的比較，$12n\times 2^{n-k-1}+2n+3$次的加法，$4n$次的常數乘法以及$2n$次的變數乘法。

在了解解一個eBCH碼時所需要的運算後，我們在對做整個乘積碼做考量，其做一個方向的解碼過程中，總共要解$n_2$個eBCH($n_1$, $k_1$, $d_1$)碼，所以當解碼採用Log-MAP演算法時，總共會用到的運算單位為$10n_1\times n_2\times 2^{n_1-k_1-1}$次的比較，$22n_1n_2\times 2^{n_1-k_1-1}+2n_1n_2+3n_2$次的加法，$4n_1n_2$次的常數乘法以及$2n_1n_2$次的變數乘法。同樣的，我們還必須在解$n_1$個eBCH($n_2$, $k_2$, $d_2$)碼以達到另外一個方向解碼後提供的資訊。因此還需要再用到$10n_1n_2\times 2^{n_2-k_2-1}$次的比較，$22n_1n_2\times 2^{n_2-k_2-1}+2n_1n_2+3n_1$次的加法，$4n_1n_2$次的常數乘法以及$2n_1n_2$次的變數乘法。若將演算法改為MAX-Log-MAP演算法時，再減去查表動作的影響後，做兩個不同方向的資訊運算總共所需的運算量除了加法總和會減為$12n_1n_2\times\left(2^{n_1-k_1-1}+2^{n_2-k_2-1}\right)+4n_1n_2+3n_1+3n_2$次，其它的不變。

除此之外，乘積碼解碼時的訊息轉換過程中也需要額外的運算，為了得到額外的資訊，演算過程中需要將事前機率值及本身值的影響扣除，所以還需要用到$2\times 2\times n_1n_2$次的加法。這地方為乘積碼在傳遞訊息時所需要做到的加法，故其所需要的次數不論是對 Log-MAP 演算法還是MAX-Log-MAP 演算法都是相同的數量。在將其數量總合除於我們總共得到的資訊位元量$n_1n_2$，則最後可整理出在一次的縱橫遞迴運算中每個位元總共所需的運算單位數量，其結果如表(6.2-1)所示。

| 演算法 | 比較運算 | 加法運算 | 常數乘法運算 | 變數乘法運算 |
|---|---|---|---|---|
| Log-MAP | $10\left(2^{n_1-k_1-1}+2^{n_2-k_2-1}\right)$ | $22\left(2^{n_1-k_1-1}+2^{n_2-k_2-1}\right)+8+3(n_1+n_2)/n_1n_2$ | 8 | 4 |
| MAX-Log-MAP | $10\left(2^{n_1-k_1-1}+2^{n_2-k_2-1}\right)$ | $12\left(2^{n_1-k_1-1}+2^{n_2-k_2-1}\right)+8+3(n_1+n_2)/n_1n_2$ | 8 | 4 |

表(6.2-1) Log-MAP 演算法和 MAX-Log-MAP 演算法
實現所需的單位元運算量比較表

### 6.2.2 卻斯演算法延伸軟式輸出的解碼運算需求

在第四章介紹了另一個解碼的方法，其是以最大相似度以及卻斯演算法相互應用之下來求得解碼所需要的資訊，其演算過程中會應用到硬式解碼器(hard-decoder)來幫助解碼。就如 4.1 節所提到的一樣，我們可採用不同的硬式解碼器來幫助解碼，其解碼效能並不會受到影響，主要不同的地方再於其可攜性。但以 IEEE802.16a 為例，其所用到的最大編碼為 eBCH(64,57,4)，其碼長度還不會太長；而且實際應用到的編碼如表(2.2-1)所示，只有 3 種不同的 eBCH 編碼。因此建議採用症狀解碼的方法，只要利用隨機存取記憶體(RAM)建立一個可更改內容的編碼表，再以此表格來做查表解碼即可省去許多硬體複雜度，而其中也不會有太多運算的產生。以一個 eBCH(n, k, d)的症狀硬式解碼器為例，其中所需要的運算為$n\times(n-k)$個互斥運算，以及$n$個比較運算。

而在卻斯演算法所延伸的解碼過程中，在運算 eBCH(n, k, d)的歐基理德距離時，需要用到$3\times n$個加法以及$n$個變數乘法。在卻斯演算法裡，還需要產生$2^{\left\lfloor\frac{d-1}{2}\right\rfloor}=2^t$個測試樣本，其中$t=\left\lfloor\frac{d-1}{2}\right\rfloor$為 eBCH(n, k, d)碼的更正錯誤能力。在尋找$2^t$個可靠度最小的位元時，需要用到$n\times 2^t$個比較運算。在找到$2^t$個可靠度最小的位元之後，其需要使用個$t\times 2^t$互斥運算來產生測試樣本，對於每個測試樣本都需要經過硬式解碼器的運作，運算硬式結果的歐基理德距離，並找出何者的歐基理德距離最小。加上測試樣本的產生，這四個行為總共會用到個$n\times(n-k)\times 2^t+t\times 2^t$互斥運算，$n\times 2^t+2^t+n$個比較，$2\times n\times 2^t$個加法，以及$n\times 2^t$個變數乘法。

在得到各個樣本的歐基理德距離後，還需要計算軟式輸出值，在進行此運算時，會需要用到$n\times 2^t$個比較，$n$個加法以及$n$個常數乘法。將上述的運算整理後，我們可得知再對一個eBCH($n_1$, $k_1$, $d_1$)解碼時所需要的運算量為$n_1\times(n_1-k_1)\times 2^t+t\times 2^t$互斥運算，$2\times n_1\times 2^t+2^t+n_1$個比較，$2\times n_1\times 2^t+n_1$個加法，$n_1$個常數乘法以及$n_1\times 2^t$個變數乘法。

在實際的乘積碼運作時，我們不論對橫向還是縱向的eBCH碼都採用同樣多的測試樣本來處理，其數目定為$2^p$個。在做乘積碼的解碼時，需要對$n_2$個eBCH($n_1$, $k_1$, $d_1$)解碼以得到一個方向的資訊，之後再對$n_1$個eBCH($n_2$, $k_2$, $d_2$)解碼來得到另一個方向的資訊，這兩個動作將會需要個$n_1n_2\times 2^p\times((n_1-k_1)+(n_2-k_2))+p\times 2^p(n_1+n_2)$互斥運算，$2\times(2\times 2^t+1)\times n_1n_2$個加法，$2\times(2\times 2^t+1)\times n_1n_2+2^t(n_1+n_2)$個比較，$2\times n_1n_2$個常數乘法以及$2\times n_1n_2\times 2^t$個變數乘法。

在一次的遞迴運算中，同樣會使用到兩次的訊息傳遞動作，在卻斯演算法延伸的軟式輸出中，其需要做到常數的乘法及加法來處理更新的額外資訊，故這兩次的傳遞總共會花費$2\times n_1n_2$次的加法及$2\times n_1n_2$次的常數乘法。將所有的動作整理後，在定下了測試樣本數量值$p$後，我們可將此演算法所需要的每個位元運算動作量列在表(6.2-2)。

| 測試樣本數量值（$2^p$） | 互斥運算 | 比較運算 | 加法運算 | 常數乘法運算 | 變數乘法運算 |
|---|---|---|---|---|---|
| $p=2$ | $4((n_1-k_1)+(n_2-k_2))+8\frac{n_1+n_2}{n_1n_2}$ | $18+4\frac{n_1+n_2}{n_1n_2}$ | 20 | 4 | 8 |
| $p=3$ | $8((n_1-k_1)+(n_2-k_2))+24\frac{n_1+n_2}{n_1n_2}$ | $34+8\frac{n_1+n_2}{n_1n_2}$ | 36 | 4 | 16 |
| $p=4$ | $16((n_1-k_1)+(n_2-k_2))+64\frac{n_1+n_2}{n_1n_2}$ | $66+16\frac{n_1+n_2}{n_1n_2}$ | 68 | 4 | 32 |

表(6.2-2) 卻斯(Chase)演算法延伸軟式輸出解碼方法在不同測試樣本數的每位元運算需求表

### 6.2.3 碼字元間相關性的解碼運算需求

第五章所介紹的碼字元間相關性解碼方法所需要的硬體需求是最少的，其只利用了簡單的比較和互斥運算即可求得解碼所需要的額外資訊，而藉此交互傳遞訊息來加以解碼。

在此解碼方法中，同樣會用到一個隨機存取記憶體(RAM)做為不同編碼的解碼表，此記憶體將視解碼需要而輸入不同 eBCH 編碼的同步位元矩陣值，藉此來判斷是否進行互斥運算。如之前所說的，在第二章所提到的 IEEE802.16a 規格下，其所使用的編碼方法不會太多，同時也不會有高長度的編碼，故此方法所需要的記憶體不會太大。

在運算過程中，如第五章裡所提到的，此方法可以分為兩個部分，一個是先行計算位於同位檢查矩陣中同一列中裡所有 1 值對應的位元做運算的結果，另一部分則是在針對每一個位元做額外資訊的運算。在解 eBCH(n, k, d)碼時，第一個部分需花費 $2\times n\times(n-k)$ 次的比較以及 $n\times(n-k)$ 次的互斥運算；第二個部分則需要花費 $3\times n\times(n-k)$ 次的比較，$n$ 次的互斥運算以及 $n$ 次的變號運算，而變號運算我們在此為了方便比較，將其當做加法運算來計量。故在解一個方同裡 $n_2$ 個 eBCH($n_1$, $k_1$, $d_1$)碼的動作裡，總共會需要 $5\times n_1n_2\times(n_1-k_1)$ 次的比較，$n_1n_2\times(n_1-k_1)+n_1n_2$ 的互斥運算和 $n_1n_2$ 次的加法運算。若是我們採用本身的遞迴處理來加快每個遞迴所得到的效果時，則運算量將乘上自身遞迴的次數。而在解乘積碼時，其資料的傳遞是較為直接的，而不需要做另外的處理，因此利用此演算法所需要的每位元運算量將可列出如表(6.2-3)所示：

| 自身遞迴處理次數 | 互斥運算 | 比較運算 | 加法運算 | 常數乘法運算 | 變數乘法運算 |
|---|---|---|---|---|---|
| 無遞迴處理 | $((n_1-k_1)+(n_2-k_2))+2$ | $5((n_1-k_1)+(n_2-k_2))$ | 2 | 0 | 0 |
| 遞迴處理 3 次 | $3((n_1-k_1)+(n_2-k_2))+6$ | $15((n_1-k_1)+(n_2-k_2))$ | 6 | 0 | 0 |

表(6.2-3) 以碼字元間相關解碼演算法在不同自身遞迴處理次數下的每位元運算需求表

### 6.2.4 三種方法的分析與比較

在了解了各別演算法所需要的每位元運算量，我們可以看出格子圖(Trellis)觀念延伸出的演算法由於受到狀態數量的影響而需要龐大的運算量。而在卻斯演算法中，也會因為測試樣本數量的多寡而使得運算數量增加不少，只有第五章所提出的方法能以最少的運算量來進行解碼，當相對的其需要附出效能較差的代價。表(6.2-4)針對我們所模擬的兩種乘積碼eBCH(32,26)×eBCH(32,26)以及eBCH(32,26)×eBCH(16,11)計算出得到每個位元所需要的資訊所需要的運算量。

| 編碼 | 演算法 | 互斥運算 | 比較運算 | 加法運算 | 常數乘法運算 | 變數乘法運算 |
|---|---|---|---|---|---|---|
| 乘積碼 eBCH(32,26) × eBCH(16,11) | MAX-Log-MAP | 0 | 480 | 1064.28 | 8 | 4 |
| | Log-MAP | 0 | 480 | 584.28 | 8 | 4 |
| | Chase( $p=2$ ) | 44.75 | 18.38 | 20 | 4 | 8 |
| | Chase( $p=4$ ) | 182 | 67.5 | 68 | 4 | 32 |
| | 碼字元相關解碼(無自身遞迴) | 13 | 55 | 2 | 0 | 0 |
| | 碼字元相關解碼(自身遞迴3次) | 39 | 165 | 6 | 0 | 0 |
| 乘積碼 eBCH(32,26) × eBCH(32,26) | MAX-Log-MAP | 0 | 640 | 1416.56 | 8 | 4 |
| | Log-MAP | 0 | 640 | 776.56 | 8 | 4 |
| | Chase( $p=2$ ) | 48.62 | 18.31 | 20 | 4 | 8 |
| | Chase( $p=4$ ) | 196.92 | 67.23 | 68 | 4 | 32 |
| | 碼字元相關解碼(無自身遞迴) | 14 | 60 | 2 | 0 | 0 |
| | 碼字元相關解碼(自身遞迴3次) | 42 | 180 | 6 | 0 | 0 |

表(6.2-4) 不同解碼演算法對乘積碼eBCH(32,26)×eBCH(16,11)及乘積碼eBCH(32,26)×eBCH(32,26)的每位元運算需求表

從表(6.2-4)中可看出，當不同的編碼進行解碼動作時，由於格子圖觀念解碼會受到編碼狀態數的影響，故其值會隨組成碼的編碼長度而大幅增加。卻斯演算法延伸的軟式解碼則是以測試樣本數為影響運算量的主要原因，因此在不同編碼下的運算量不會相差太多，而且其值和 MAP 演算法相比下降許多，只要採用的 p 值不要太大，其所需的運算量就不會太大。而第五章所介紹的利用碼字元相關性解碼方法則確實只需要相當少的運算量即可達到解碼的目的，其主要的運算量在於比較運算，而加法運算相當的少，至於乘法則是完全不需要，因此此方法相當的適合用在低硬體需求的應用。

# 第七章　結論

在本論文中，我們已經介紹了幾種不同的解碼方法，其主要解碼過程皆是以軟式輸出和軟式輸入來進行兩個不同方向的遞迴處理，藉此來完成乘積碼的解碼。在經過模擬各個不同的方法並對其運算量加以改量以後，可發現各個方法都有其優缺點。

在以格子圖觀念來解碼時，會受到編碼狀態數的影響，其值會隨組成碼的編碼長度而大幅增加，這使得 Log-MAP 以及 MAX-Log-MAP 演算法對於高碼率(Code Rate)的編碼所需要的運算量相當的大，因此其在硬體的實現上較為複雜。但相對的，其解碼更正能力也相當的強，因此對於要求傳輸正確率高並有足夠解碼時間的應用上是相當適合的。

以卻斯演算法解碼時，則需要對測試樣本群進行硬式解碼並計算出其軟式輸出，中間的運算過程和測試樣本數目的多寡有關，當測試樣本數越多時，其所需要的運算量也越大，因此其運作所需要的硬體雖然不像 MAP 演算法龐大，但還是需要配合測試樣本數而提供其解碼複雜度。不同於利用碼字元關係式解碼，其解碼效能而 MAP 演算法之間的差異不會太大，同時其可利用測試樣本數來調整其運算所需要的複雜度，因此在硬體複雜度和解碼更正能力較為平衡，故卻斯演算法延伸的解碼方式所能適用的範圍較廣。

利用碼字元關係式解碼的方法則是以硬體的考量為走向，故其擁有只需要簡單的硬體架構即可實現解碼的優點；但相對的，其解碼後更正錯誤效能也較另外兩種方法為差。所以此方法較適合應用在廣泛銷售這類型需要低成本的硬體。

在實際的應用上，只要針對各個應用的需求來決定解碼的方法，在解碼效能與硬體複雜度上做適度的取捨，即可有效的為乘積碼做解碼，如此即使在不同的要求下，區塊乘積碼都能做為有效的通道編碼，將通道雜訊的干擾確實降低。

# 參考文獻


[1] A. Hocquenghem, “Codes correcteurs d, erreurs,” *Chiffres (Paris),* vol. 2, September 1959, pp.147-156.

[2] R. Bose and D. Ray-Chaudhuri, “On a class of error correcting binary group codes,” *Information and Control,* vol. 3, March 1960, pp. 68-79.

[3] R. Bose and D. Ray-Chaudhuri, “Further results on error correcting binary group codes,” *Information and Control,* vol. 3, Sept. 1960, pp. 279-290.

[4] C. Berrou, A. Glavieux, and P. Thitimajshima, “Near Shannon Limit Error-Correcting Coding and Decoding: Turbo Codes,” in *Proceedings of the International Conference on Communications,* (Geneva, Switzerland), May 1993, pp. 1064-1070.

[5] C. Berrou and A. Glavieux, “Near optimum error correcting coding and decoding: Turbo codes,” *IEEE Transactions on Communications,* vol. 44, October 1996, pp.1261-1271.

[6] E. Berlekamp, “On decoding binary Bose-Chaudhuri-Hocquenghem codes,” *IEEE Trans. Inform. Theory,* vol. 11, pp. 577-579, 1965.

[7] J. Massey, “Step-by-step decoding of the Bose-Chaudhuri-Hocquenghem codes,” *IEEE Trans. Inform. Theory,* vol. 11, pp. 580-585, 1965.

[8] J. Massey,“Shift-register synthesis and BCH decoding,”*IEEE Transactions on Information Theory,* vol. IT-15, January 1969, pp. 122-127.

[9] R. Pydiah, “Near-optimum decoding of product codes: Block turbo codes,” *IEEE Trans. Commun.*, vol. 46, Aug. 1998, pp. 1003-1010.

[10] R. Pydiah, A. Glavieux, A. Picart, and S. Jacq, “Near optimum decoding of product codes,”in *proc. IEEE GLOBECOM’94 Conference*., vol. 1/3, San Francisco, Nov.-Dec. 1994, pp. 339-343.

[11] A. Goalic, and R. Pyndiah, “Real-time turbo-decoding of product codes on a digital signal processor” in *Proc. IEEE Int. Symp. Turbo Codes & Related Topics*, Brest France, Sept. 1997, pp. 267-270.

[12] D. Chase, “A class of algorithms for decoding block codes with channel measurement information,”*IEEE Trans. Inform.Theory*, vol. IT-4, Sept. 1954, pp.29-37.

[13] J. Hagenauer, E. Offer, and L. Papke, “Iterative Decoding of Binary Block and Convoolutional Codes,” *IEEE Trans. Inform. Theory,* vol. 42, No. 2, Mar. 1996, pp. 429-445.

[14] B. Skler, “A Primer on Turbo Code Concepts,” *IEEE Communications Magazine*. Dec. 1997, pp. 94-102

[15] P. Adde, R. Pyndiah, O. Raoul, and J. Inisan, “Block turbo decoder design,” in *Proc. IEEE Int. Symp. Turbo Codes & Related Topics,* vol. 1/1, Brest, France, Sept. 1997, pp. 166-169.

[16] R. Pyndiah, “Iterative decoding of product codes: Block turbo codes,” in *proc. IEEE Int. Symp. Turbo Codes & Related Topics*, vol. 1/1, Brest, France, Sept. 1997, pp. 71-79.

[17] P. Adde, R. Pyndiah, and O. Raoul, “Performance and complexity of block turbo decoder circuits,” *Third International Conference on Electronics, Circuits and System ICECS’96*, 13-16 Oct. 1996 –Rodos, Greece, pp. 172-175.

*[18]* J. K. Wolf, “Efficient maximum likelihoood decoding of linear Block codes using a trellis,” *IEEE Trans. Inform. Theory*, vol. IT-22, Sept. 1976. pp. 514-517.