

极化码讲义

A First Course in Polar Codes

于永润

MATLAB代码在附录

前言

我不知道这个讲义的反响怎么样，有没有人读，所以我先仅写主干内容。

一些非主干内容，例如极化码的BP译码器、球形译码器、打孔和缩短、编码调制和HARQ等内容我没写。

我注意到Arikan Erdal的论文这两年的阅读量暴增，所以我认为有大量的人开始研究极化码，我觉得这个讲义应该有人看。

注意，这个讲义里几乎没有一处是我自己的成果，我只是编纂已有的结果，几乎所有的参考文献都来自IEEE。如果你认为我漏了一些引用，告诉我，我引上即可。

本讲义内容对于一个极化码资深研究者而言没什么用，但是对于一个初学者还是有用的。

本资料的`LATEX`模板是在网上找的清华大学吕同富老师数值分析教材的模板。

有问题、发现错误，联系我。

于永润

498699845@qq.com

目 录

第1章 背景知识	1
第2章 长度为2的幂的极化码	10
2.1 长度为2的极化码	10
2.2 长度为 2^n 的极化码	15
2.3 长度为 2^n 的极化码可达信道容量	20
第3章 极化码编码	28
3.1 极化码构造	28
3.2 极化码生成矩阵的性质	38
3.3 非系统极化码编码过程	53
3.4 系统极化码编码过程	60
第4章 串行抵消译码	66
4.1 长度为2的极化码的SC译码过程	66
4.2 长度为 2^n 的极化码的SC译码过程	69
4.3 快速SC译码算法	78
4.4 比特翻转SC译码器	95
4.5 SC译码器在编码中的应用	100
4.5.1 用SC译码器进行系统极化码的编码	100
4.5.2 使用SC译码器进行Monte-Carlo码构造	107

第5章 串行抵消列表译码	108
5.1 基于LLR的路径度量	108
5.2 基于LLR的SCL译码器的执行过程	110
5.3 快速SCL译码	131
5.3.1 路径度量的另一种计算方式	132
5.3.2 路径度量的无损简化计算公式	134
5.3.3 R0节点的快速SCL译码方法	135
5.3.4 Rep节点的快速译码	135
5.3.5 R1节点的快速译码	137
5.3.6 SPC节点的快速译码算法	141
5.4 SCL译码的其它改进算法	156
5.4.1 循环冗余校验协助下的SCL译码器	156
5.4.2 自适应CA-SCL译码器	157
5.4.3 剪枝CA-SCL译码器	157
第6章 简单MATLAB代码	159
6.1 代码运行速度	159
6.2 传统CA-SCL译码器	160
6.3 FastSCL译码器	161
6.4 BP译码器	162
参考文献	163

第1章 背景知识

本章介绍本资料中的最基本的概念，这些概念将在后续章节中频繁用到。本章定理的证明虽未详细写出，但给了足够的提示。

离散信道指的是输入符号集合和输出符号集合都是离散集合的信道，本资料考虑的离散信道的输入符号集合和输出符号集合的元素数量是有限值。

信道无记忆是指信道的任意两次传输是独立的，即信道对每个输入符号添加的噪声是独立的。

离散无记忆信道 W 可以用转移概率 $W(y|x)$ 完全刻画，其中 $y \in \mathcal{Y}$ 是输出符号集合中的元素， $x \in \mathcal{X}$ 是输入符号集合中的元素， $W(y|x)$ 表示在输入符号为 x 的条件下，输出符号是 y 的概率。在给定信源分布的情况下，即 $x \in \mathcal{X}$ 的离散分布律已知，整个传输系统 (X, Y) 的联合概率 $\Pr(X = x, Y = y) = p(x)W(y|x)$ 。

定义 1.1 对数似然比

二进制输入信道 W 的输出符号 $y \in \mathcal{Y}$ 的对数似然比（Log-Likelihood Ratio, LLR）的定义如下：

$$L(y) = \ln \frac{W(y|0)}{W(y|1)}. \quad (1.1)$$

如果 W 的输出符号集合 \mathcal{Y} 是离散的，上式中的分子分母取条件概率质量；如果 W 的输出符号集合 \mathcal{Y} 是连续的，上式中的分子分母取条件概率密度。因为 y 和 $L(y)$ 一一对应¹，因此这时 $L(y)$ 是 y 的平凡充分统计量。 $L(y)$ 将在极化码的译码中使用。

定义 1.2 二进制输入离散无记忆对称信道

¹有时会出现若干个LLR相同的情况： $L(y_i) = L(y_j), y_i, y_j \in \mathcal{Y}$ 。这说明 $\frac{W(y_i|0)}{W(y_i|1)} = \frac{W(y_j|0)}{W(y_j|1)}$ ，这时合并符号 y_i 和 y_j 即可。

说 W 是二进制输入离散无记忆对称信道（Binary-Input Discrete Memoryless output-Symmetric Channel, BI-DMSC），乃是指 W 的转移概率具有如下性质：

- (i) 存在一个置换 π , 满足 $\pi^2 = 1$;
- (ii) $W(y|0) = W(\pi(y)|1)$

定义1.2中置换 π 的定义域是输出符号集合 \mathcal{Y} , 1表示恒等置换。

例 1.1 BI-DMSC的抽象例子。

$$W = \begin{bmatrix} p_1 & p_2 & \dots & p_N & , & q_1 & q_2 & \dots & q_N \\ q_1 & q_2 & \dots & q_N & , & p_1 & p_2 & \dots & p_N \end{bmatrix}. \quad (1.2)$$

上式中 W 是大小为 $2 \times 2N$ 的矩阵, W 对应置换 $\pi = (1, N+1)(2, N+2), \dots, (N, 2N)$, 其中数对 (a, b) 表示对换位于索引 a 和 b 上的元素。

定义 1.3 二进制输入离散信道的最大似然判决

二进制输入离散信道 W 的最大似然（Maximum Likelihood, ML）判决指的是当收到符号 y 时, x 的值按下式判决:

$$\hat{x}_{\text{ML}} = \arg \max_{x \in \{0,1\}} W(y|x). \quad (1.3)$$

定理 1.1 二进制输入离散信道 W 在ML判决规则下的错误率是:

$$P_e^{\text{ML}}(W) = \frac{1}{2} \sum_{y \in \mathcal{Y}} \min \{W(y|0), W(y|1)\}. \quad (1.4)$$

证明 所谓计算错误率, 即是寻找单点事件 $(X = x, Y = y)$ 满足 y 在ML判决规则下的结果不是 x , 并把所有这样的单点事件的概率求和。由此, 依据定义1.3易得上述定理。

定义 1.4 二进制输入离散信道的巴特查理亚系数

W 是二进制输入离散信道, W 的巴特查理亚系数 $Z(W)$ 如下:

$$Z(W) = \sum_{y \in \mathcal{Y}} \sqrt{W(y|0)W(y|1)}. \quad (1.5)$$

例 1.2 例1.2中的信道 W 的巴特查理亚系数如下：

$$Z(W) = 2 \sum_{i=1}^N \sqrt{p_i q_i}. \quad (1.6)$$

定理 1.2

$$\begin{aligned} 0 &\leq Z(W) \leq 1, \\ P_e^{\text{ML}}(W) &\leq Z(W). \end{aligned} \quad (1.7)$$

证明 第一个式子中第一个不等号是显然的，第二个不等号用基本不等式证明。由 $\min\{W(y|0), W(y|1)\} \leq \sqrt{W(y|0)W(y|1)}$ 得到第二个式子。

由 $P_e^{\text{ML}}(W) \leq Z(W)$ 可见， $Z(W)$ 是 $P_e^{\text{ML}}(W)$ 的上界，因此，在没有 $P_e^{\text{ML}}(W)$ 具体值的情况下， $Z(W)$ 可以粗略表示 W 的可靠性。

推论 1.1

$$\begin{aligned} Z(W) = 1 &\Rightarrow P_e^{\text{ML}}(W) = 0.5, \\ Z(W) = 0 &\Rightarrow P_e^{\text{ML}}(W) = 0. \end{aligned} \quad (1.8)$$

证明 注意到 $Z(W) = 0$ 或 1 的条件和 $P_e^{\text{ML}}(W)$ 的计算公式立即得证。

定义 1.5 二进制输入离散信道的对称容量

W 是二进制输入离散信道， W 的对称 $I(W)$ 如下：

$$I(W) = \frac{1}{2} \sum_{x \in \{0,1\}} \sum_{y \in \mathcal{Y}} W(y|x) \log_2 \frac{W(y|x)}{\Pr(y)}. \quad (1.9)$$

如果 W 是BI-DMSC，则上式中的对称容量等于信道容量。

例 1.3 例1.2中信道 W 的 $I(W)$ 为 $\sum_{i=1}^N p_i \log_2 \frac{p_i}{\frac{1}{2}(p_i+q_i)} + \sum_{i=1}^N q_i \log_2 \frac{q_i}{\frac{1}{2}(p_i+q_i)}$ 。

定理 1.3

$$\log_2 \frac{2}{1 + Z(W)} \leq I(W) \leq \sqrt{1 - Z(W)^2}. \quad (1.10)$$

证明 参见文献[3]中的命题1，此定理将在证明极化码可以到达信道容量时使用。

定理 1.4

$$I(X_1, \dots, X_N; Y) = \sum_{i=1}^N I(Y; X_i | X_1, \dots, X_{i-1}). \quad (1.11)$$

证明 这是著名的互信息链式法则，利用条件互信息 $I(X; Y|Z)$ 的定义易得。

定义 1.6 离散信道的统计退化

$W(y|x), Q(z|x)$ 是离散信道，如果存在离散信道 $P(y|z)$ ，使得 $W(y|x) = \sum_{z \in \mathcal{Z}} Q(z|x)P(y|z)$ ，则称 W 是 Q 的统计退化（Stochastic degrading），记作 $W \preceq Q$ 。

定理 1.5 如果 $W \preceq Q$ ，则下式成立：

$$\begin{aligned} I(W) &\leq I(Q), \\ Z(W) &\geq Z(Q), \\ P_e^{\text{ML}}(W) &\geq P_e^{\text{ML}}(Q). \end{aligned} \quad (1.12)$$

证明 第一个式子就是信息处理不等式。第二个式子的证明需要使用柯西-布尼雅可夫斯基不等式、 Z 的定义以及统计退化的定义。第三个式子用 P_e^{ML} 的定义以及统计退化的定义易得。

推论 1.2 如果 $W \preceq Q, Q \preceq R$ ，则 $W \preceq R$ 。

证明 由定义1.6易得。

例 1.4 对于任意一个信道 Q , 合并符号 $z_1, z_2 \in \mathcal{Z}$ (等价于把 z_1, z_2 对应的转移矩阵的列相加) 得到信道 W , 则 $W \preceq Q$ 。

下面的定理是寻常的结论, 但是很多时候只是被作者们随手写出来[1], 而证明的过程需要到处找。尽管这个定理含义简单, 但是对于初学者往往具有致命的混淆。

定理 1.6 设 \mathcal{X} 是某个长度为 N 、码率为 R 的二进制线性分组码的码字集合, $\mathbf{x}_1^N = \{x_1, x_2, \dots, x_N\} \in \mathcal{X}$ 是一个码字。该线性分组码的码字重量的取值自然是离散的, 记为 $\{d_0, d_1, d_2, \dots, d_s\}, d_0 = 0, 1 \leq d_i \leq N, d_i < d_{i+1}$, 且重量为 $d_i, 0 \leq i \leq s$ 的码字共有 A_{d_i} 个, $A_{d_0} = 1$ 。

假设使用BPSK传输码字比特: $s_i = 1 - 2x_i$, s_i 是 x_i 对应的BPSK调制符号, $s_i \in \{-1, 1\}$ 。

假设传输 $\mathbf{s}_1^N = (s_1, s_2, \dots, s_N)$ 的信道为AWGN信道, 即 s_i 对应的接收信号 $y_i = s_i + n_i, 1 \leq i \leq N$, 其中 n_i 是i.i.d.零均值且方差为 σ^2 的高斯随机变量。

上述传输过程可以表示为 $\mathbf{x}_1^N \xrightarrow{\text{BPSK}} \mathbf{s}_1^N \xrightarrow{\text{AWGN}} \mathbf{y}_1^N$ 。

假设每个 $\mathbf{x}_1^N \in \mathcal{X}$ 是等概率出现的, 每个 \mathbf{x}_1^N 出现的概率为 2^{-NR} , 那么该线性分组码在ML译码下误组率(BLER)的并上界(Union Upper bound, UUB)为:

$$P_e^{\text{ML}} \leq P_e^{\text{ML,UUB}} = \sum_{d=d_1}^{d_s} A_d Q\left(\frac{\sqrt{d}}{\sigma}\right) = \sum_{d=d_1}^{d_s} A_d Q\left(\sqrt{\frac{2RdE_b}{N_0}}\right). \quad (1.13)$$

其中 $N_0 = 2\sigma^2$ 是高斯加性白噪声的单边功率谱密度, $\frac{E_b}{N_0}$ 是信息比特信噪比, $\frac{E_s}{N_0}$ 是发送符号信噪比, 其关系为 $\frac{E_s}{RN_0} = \frac{E_b}{RN_0}$, 意思是所有能量都用于传输信息。本定理中 $E_s = 1$, 这是通用做法: 星座图能量归一化。

证明过程中我们考虑的联合分布为 $(\mathbf{X}_1^N, \mathbf{Y}_1^N)$, 对于每个实现值 $(\mathbf{x}_1^N, \mathbf{y}_1^N)$, 有 $\Pr(\mathbf{x}_1^N, \mathbf{y}_1^N) = \Pr(\mathbf{x}_1^N) \Pr(\mathbf{y}_1^N | \mathbf{x}_1^N) = \frac{2^{-NR}}{2\pi\sigma^2 N/2} \exp\{-\sum_{i=1}^N (y_i - s_i)^2 / 2\sigma^2\}$, 其中 $s_i = 1 - 2x_i$ 。在下面的叙述中, 为了记号的简明, 向量的上下标都被扔掉了, 例如, \mathbf{x} 就是 \mathbf{x}_1^N 。

证明

$$\begin{aligned} P_e^{\text{ML}} &\stackrel{(a)}{=} \sum_{\mathbf{x} \in \mathcal{X}} \sum_{\mathbf{y} \in \mathbb{R}^N} \Pr(\mathbf{x}, \mathbf{y}) \mathbb{I}(\text{Dec}_{\text{ML}}(\mathbf{y}) \neq \mathbf{x}) \\ &\stackrel{(b)}{=} 2^{-NR} \sum_{\mathbf{x} \in \mathcal{X}} \sum_{\mathbf{y} \in \cup_{\mathbf{x}_i \neq \mathbf{x}} \Omega_{\mathbf{x}_i}} \Pr(\mathbf{y} | \mathbf{x}) \\ &\stackrel{(c)}{=} \sum_{d=d_1}^{d_s} A_d Q\left(\frac{\sqrt{d}}{\sigma}\right). \end{aligned} \quad (1.14)$$

(a): \mathbb{R}^N 是接收信号的取值集合。 \mathbb{I} (logic value)是指示函数, $\mathbb{I}(\text{true}) = 1$, 这时乘指示函数等于什么也没做; $\mathbb{I}(\text{false}) = 0$, 这时乘指示函数等于让这一项消失。 $\text{Dec}_{\text{ML}}(\mathbf{y})$ 表示接收信号 \mathbf{y} 的最大似然译码结果。

(b): $\Omega_{\mathbf{x}_i} = \{\mathbf{y} | \Pr(\mathbf{y}|\mathbf{x}_i) > \Pr(\mathbf{y}|\mathbf{x}_j), i \neq j, 1 \leq i, j \leq 2^{NR}\} \subset \mathbb{R}^N$ 是码字 \mathbf{x}_i 的最大似然判决区域, 即如果 $\mathbf{y} \in \Omega_{\mathbf{x}_i}$, 则 $\text{Dec}_{\text{ML}}(\mathbf{y}) = \mathbf{x}_i$ 。当发送的码字是 \mathbf{x}_i , 接收信号是 \mathbf{y} , 且 $\mathbf{y} \in \Omega_{\mathbf{x}_j}, i \neq j$ 时, 则最大似然译码发生错误, \mathbf{x}_i 被错判为 \mathbf{x}_j , 这个错误译码器自身当然是意识不到的。

(c): **0**表示长度为 N 的全零码字。任选一个 $\mathbf{x} \in \mathcal{X}$, 有下式成立:

$$\begin{aligned}
 & \sum_{\mathbf{y} \in \cup_{\mathbf{x}_i \neq \mathbf{x}} \Omega_{\mathbf{x}_i}} \Pr(\mathbf{y}|\mathbf{x}) \stackrel{(i)}{=} \sum_{\mathbf{y} \in \Omega_{\mathbf{x}_1}} \Pr(\mathbf{y}|\mathbf{x}) + \dots + \sum_{\mathbf{y} \in \Omega_{\mathbf{x}_{2^{NR}-1}}} \Pr(\mathbf{y}|\mathbf{x}) \\
 & \stackrel{(ii)}{\leq} \sum_{\mathbf{y} \in \Omega_{(\mathbf{x}_1, \mathbf{x})}} \Pr(\mathbf{y}|\mathbf{x}) + \dots + \sum_{\mathbf{y} \in \Omega_{(\mathbf{x}_{2^{NR}-1}, \mathbf{x})}} \Pr(\mathbf{y}|\mathbf{x}) \\
 & \stackrel{(iii)}{=} \left[\sum_{\mathbf{y} \in \Omega_{(\mathbf{x}_{i_1}, \mathbf{x})}} \Pr(\mathbf{y}|\mathbf{x}) + \dots + \sum_{\mathbf{y} \in \Omega_{(\mathbf{x}_{i_{A_{d_1}}, \mathbf{x})}}} \Pr(\mathbf{y}|\mathbf{x}) \right] \\
 & \quad + \left[\sum_{\mathbf{y} \in \Omega_{(\mathbf{x}_{i_{A_{d_1}+1}}, \mathbf{x})}} \Pr(\mathbf{y}|\mathbf{x}) + \dots + \sum_{\mathbf{y} \in \Omega_{(\mathbf{x}_{i_{A_{d_2}}, \mathbf{x})}}} \Pr(\mathbf{y}|\mathbf{x}) \right] \tag{1.15} \\
 & \quad + \dots \\
 & \quad + \left[\sum_{\mathbf{y} \in \Omega_{(\mathbf{x}_{i_{A_{d_{s-1}}+1}}, \mathbf{x})}} \Pr(\mathbf{y}|\mathbf{x}) + \dots + \sum_{\mathbf{y} \in \Omega_{(\mathbf{x}_{i_{A_{d_s}}, \mathbf{x})}}} \Pr(\mathbf{y}|\mathbf{x}) \right] \\
 & \stackrel{(iv)}{=} A_{d_1} Q\left(\frac{\sqrt{d_1}}{\sigma}\right) + A_{d_2} Q\left(\frac{\sqrt{d_2}}{\sigma}\right) + \dots + A_{d_s} Q\left(\frac{\sqrt{d_s}}{\sigma}\right).
 \end{aligned}$$

其中等号(i)是因为 $\Omega_{\mathbf{x}_i}, 1 \leq i \leq 2^{NR}$ 两两不相交。

不等号(ii)是因为我们把每一个求和区域都变大了: $\Omega_{\mathbf{x}_i} \subset \Omega_{(\mathbf{x}_i, \mathbf{x})} = \{\mathbf{y} | \Pr(\mathbf{y}|\mathbf{x}_i) > \Pr(\mathbf{y}|\mathbf{x}), \mathbf{x}_i \neq \mathbf{x}\}$ 。

等号(iii)把不等号(ii)后的 $2^{NR} - 1$ 项分了类, 每一类用中括号圈了出来。分类的方法是: 与所有 \mathbf{x} 的汉明距离相同的码字是一类, 一个中括号内的所有码字 $\mathbf{x}_{A_{d_i}+1}, \dots, \mathbf{x}_{A_{d_{i+1}}}$ 与 \mathbf{x} 的汉明距离是 d_{i+1} , 自然地, 第*i*个中括号中有 A_{d_i} 项。

等号(iv)是因为: 设某个码字比特 \mathbf{x}_i 与 \mathbf{x} 的汉明距离是 d , 那么它们两个BPSK调制信号之差为 $\delta = (1 - 2\mathbf{x}_i) - (1 - 2\mathbf{x})$, 显然 $\|\delta\| = 2\sqrt{d}$, 意思是 \mathbf{x}_i 与 \mathbf{x} 的欧式距离是 $2\sqrt{d}$ 。由此, 不妨假设 \mathbf{x} 位于一维数轴的 \sqrt{d} 处, 而 \mathbf{x}_i 位于 $-\sqrt{d}$ 处, 则在AWGN信道中, 有 $\sum_{\mathbf{y} \in \Omega_{(\mathbf{x}_i, \mathbf{x})}} \Pr(\mathbf{y}|\mathbf{x}) =$

$\int_{\sqrt{d}}^{+\infty} \frac{1}{\sigma\sqrt{2\pi}} e^{-\alpha^2/(2\sigma^2)} d\alpha = Q(\sqrt{d}/\sigma)$ 。按分类规则, (iv)后每个中括号中的 $\mathbf{x}_{i_{A_{d_i+1}}}, \dots, \mathbf{x}_{i_{A_{d_i+1}}}$ 与 \mathbf{x} 的距离都是 d_{i+1} , 因此一个中括号中的每一项都等于 $Q(\sqrt{d_{i+1}}/\sigma)$, 从而等号(iv)成立。

既然对任意一个 $\mathbf{x} \in \mathcal{X}$ 有上式成立¹, 则每个 $\sum_{\mathbf{y} \in \cup_{\mathbf{x}_i \neq \mathbf{x}} \Omega_{\mathbf{x}_i}} \Pr(\mathbf{y}|\mathbf{x})$ 都是上式的计算结果。因为共有 2^{NR} 个 $\sum_{\mathbf{y} \in \cup_{\mathbf{x}_i \neq \mathbf{x}} \Omega_{\mathbf{x}_i}} \Pr(\mathbf{y}|\mathbf{x})$, 所以与式(1.14)中的系数 2^{-NR} 抵消, 式(1.14)成立。证毕。

式(1.13)中第二个等号是因为 $\frac{\sqrt{d}}{\sigma} = \sqrt{\frac{2d}{N_0}} \stackrel{(*)}{=} \sqrt{\frac{2dE_s}{N_0}} = \sqrt{\frac{2RdE_b}{N_0}}$, 等号(*)是因为 $E_s = 1$ 。

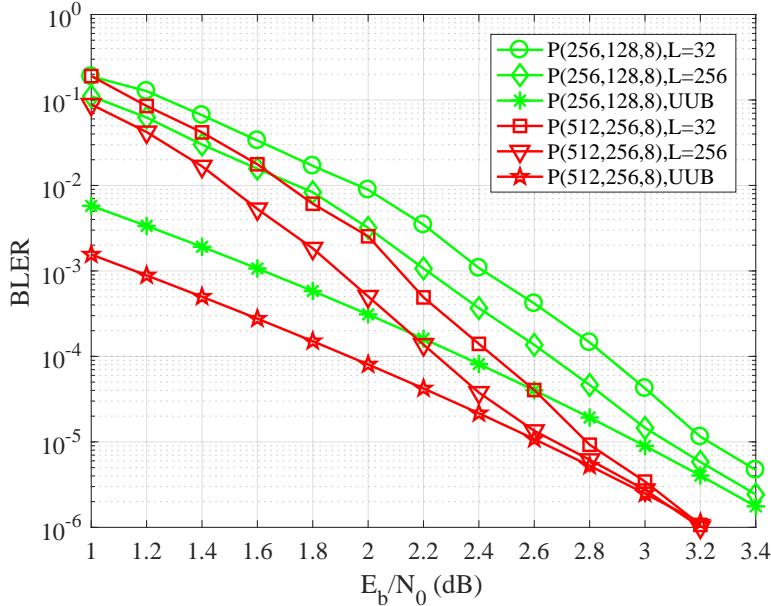


图 1.1 $P_e^{\text{ML,UUB}}$ 与仿真结果的对比。

$P_e^{\text{ML,UUB}}$ 与仿真结果的对比如图1.1所示, 其中 $P(N, K, r)$ 表示码长为 N 、信息比特数为 K 以及使用校验位长度为 r 的CRC的极化码。图1.1中的极化码由高斯近似在 $E_b/N_0 = 2.5$ dB 处构造, 8位CRC的生成多项式16进制形式为A6。图1.1使用的译码器是CRC辅助下的串行抵消列表译码器, L 是列表数。对于 $P(256, 128, 8)$, 有 $d_1 = 16, d_2 = 20, A_{d_1} = 972, A_{d_2} = 8740$; 对于 $P(512, 256, 8)$, 有 $d_1 = 16, d_2 = 24, A_{d_1} = 303, A_{d_2} = 24157$ 。 d 和 A_d 的搜索算法取自文献[2]。由图1.1可见, 当信噪比较低时, $P_e^{\text{ML,UUB}}$ 与仿真结果差距较大; 当信噪比较高时, $P_e^{\text{ML,UUB}}$ 与仿真结果非常类似。

极化码的SC译码²可以简明地表达为完整二叉树上的运动, 了解一些二叉树的规律常常是有益的。我们约定二叉树根节点的层数最高, 其余节点所在的层数由根节点层数不断减1得

¹这是由 \mathcal{X} 的线性决定的。

²如果读者还不知道什么是SC译码器, 那么可以先忽略牵扯到SC译码器的叙述, 等阅读了后续章节之后再回来看。

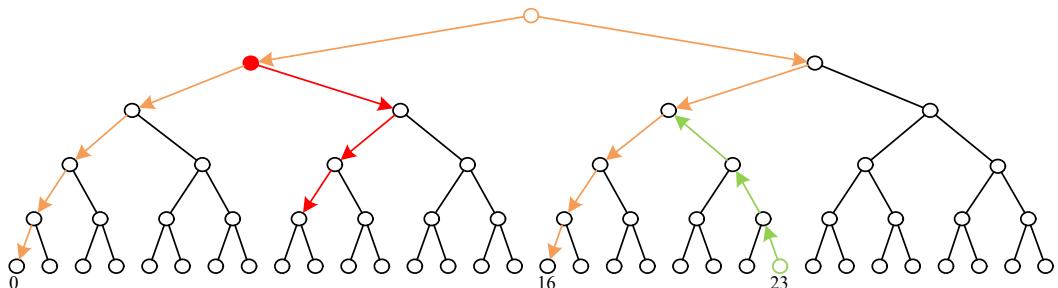


图 1.2 攀升和坠落的例子

到。下面是我自己给出的两个概念，它们在极化码译码时非常有用。下面的概念和SC译码器的关系我也写出了。

定义 1.7 二叉树上的攀升

设 i 是一棵二叉树上任意一个节点，从 i 出发，沿二叉树上的连接线，朝 i 的西北方向走，直到它的西北方向再也没有连接线为止，这一过程称为 i 在二叉树上的攀升。 i 在攀升过程中走过的连接线的段数，称为 i 的攀升次数。一个攀升的例子如图1.2所示，编号为23的叶节点的攀升次数是3次。二叉树的左轮廓上的节点都不能攀升。

从叶节点开始的攀升，对应的是标准SC译码的比特值反馈计算过程。

从非叶节点出发的攀升，对应的是快速SC译码的比特值反馈的计算过程。

定义 1.8 二叉树上的坠落

设 i 是一棵二叉树上除根节点外任意一个节点，从 i 出发，沿二叉树上的连接线，朝 i 的东南方向走一次，到达 i 的右孩子，再从 i 的右孩子出发，一路向西南方向走，这一过程称为 i 在二叉树上的坠落。 i 坠落到 i 的右孩子这一步是必须有的，从 i 的右孩子出发朝西南方向坠落的过程可有可无，可以随时停止。 i 在坠落过程中走过的连接线的段数，称为 i 的坠落次数。二叉树上只有叶节点不能坠落，每个节点坠落的最终点都是叶节点。一个坠落的例子如图1.2所示，其中红色节点坠落次数是3次。

特别地，根节点也可以坠落，且根节点的第一次坠落不限方向：根节点的第一次坠落既可以到达它的左孩子，也可以到达右孩子。但是当根节点坠落到它的某个孩子之后，坠落的方向必须是西南方向。根节点的两种坠落到叶节点的过程如图1.2中的黄线所示。注意根节点不是必须坠落到叶节点。

节点 i 坠落到 i 右孩子，对应的是SC译码过程中 g 函数的计算；节点 i 坠落到 i 右孩子之后向西南方向坠落的过程，对应的是SC译码过程中 f 函数的计算。

节点*i*坠落到叶节点这个过程对应着标准SC译码器的概率计算过程。

节点*i*尚未坠落到叶节点就停止坠落，对应的是快速SC译码的概率计算过程。

只要你稍微看过一些算法的教材，你就知道二叉树上，第*i*层编号为*j*的节点的左孩子在第*i*-1层上的编号为 $2j$ （二进制下， $2j$ 就是在*j*后添一个零），第*i*层编号为*j*的节点的右孩子在第*i*-1层上的编号为 $2j+1$ （二进制下， $2j+1$ 就是在*j*后添一个一）。下面的结论是上述规律的简单推论。

定理 1.7 把二叉树的叶节点从左到右，从0开始，按十进制自然数编号。

编号为*i*叶节点的攀升次数等于*i*的二进制展开中低位连续1的个数。如果某个节点*j*不是叶节点，则*j*必可以由唯一一个叶节点*i*在攀升过程中遇到，那么*j*的攀升次数等于*i*的攀升次数减去*i*走到*j*经过的连接线数。

设某个节点*j*不是叶节点也不是根节点，强行让*j*坠落到叶节点*i*，则*i*是唯一确定的，*j*的坠落次数是*i*的二进制展开中低位连续零的个数加一。其中“*i*的二进制展开中低位连续零的个数”等于*j*向西南的坠楼次数，“加一”指的是*j*坠落到*j*的右孩子的那一次坠落。

证明 略。初学者如果不能独立地写出极化码SC译码器，往往是因为没有清楚地认识到上面的规律。

第2章 长度为2的幂的极化码

本章为偏向于极化码应用研究的同学提供简单易懂的极化码基本概念的说明，本章内容基本都来自文献[3]。

§ 2.1 长度为2的极化码

如图2.1所示，长度为2的极化码是极化码的基本组成模块，其中 \oplus 表示模2加法，即GF(2)上的加法。图2.1中有两个等价的概率分布： (U_1, U_2, Y_1, Y_2) 和 (X_1, X_2, Y_1, Y_2) ，其中 (U_1, U_2) 是信源序列， (Y_1, Y_2) 是接收信号， $(X_1, X_2) = (U_1, U_2)[\begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix}] = (U_1 \oplus U_2, U_2)$ 是 (U_1, U_2) 对应的码字比特， $\mathbf{F} = [\begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix}]$ 是码长 $N = 2$ 的极化码的生成矩阵，两个信道 $W(y_1|x_1)$ 和 $W(y_2|x_2)$ 表示信道 W 的两次复用，一次用来传输码字比特 x_1 ，另一次用来传输码字比特 x_2 。因为 \mathbf{F} 是可逆的，即在GF(2)上 $\mathbf{F}^2 = \mathbf{I}_2$ ，所以 (U_1, U_2) 和 (X_1, X_2) 是一一对应的，从而概率分布 (U_1, U_2, Y_1, Y_2) 和 (X_1, X_2, Y_1, Y_2) 等价： $\Pr(U_1 = u_1, U_2 = u_2, Y_1 = y_1, Y_2 = y_2) = \Pr(X_1 = u_1 \oplus u_2, X_2 = u_2, Y_1 = y_1, Y_2 = y_2)$ 。

下面的定理使得我们的叙述可以继续。

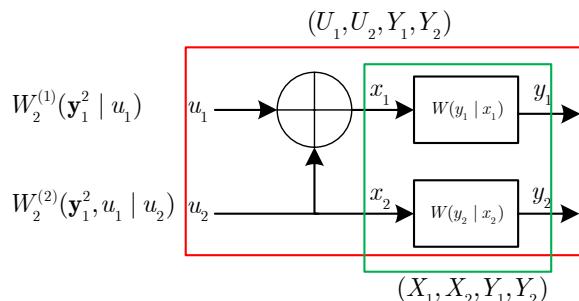


图 2.1 极化码 $N = 2$

定理 2.1 设 N 维随机变量 (U_1, U_2, \dots, U_N) 中的分量是独立同分布的，且 $U_1 \sim \text{Bernoulli}(1/2)$ 。矩阵 \mathbf{G} 是 N 维可逆矩阵。则 N 维随机变量 $(X_1, X_2, \dots, X_N) = (U_1, U_2, \dots, U_N)\mathbf{G}$ 中的分量是独立同分布的，且 $X_1 \sim \text{Bernoulli}(1/2)$ 。

证明 取集合 $\{1, 2, 3, \dots, N\}$ 中任意两个不相交的子集 $\{i_1, i_2, \dots, i_s\}$ 和 $\{j_1, j_2, \dots, j_t\}$ ，其中 $s, t \geq 1$ 。

$$\begin{aligned} \Pr(X_{i_1} = x_{i_1}, \dots, X_{i_s} = x_{i_s}) &= \sum_{\{1, \dots, N\} \setminus \{i_1, \dots, i_s\}} \Pr(X_1 = x_{i_1}, \dots, X_N = x_{i_s}) \\ &= \sum_{\{1, \dots, N\} \setminus \{i_1, \dots, i_s\}} \Pr((U_1, \dots, U_N) = (u_1, \dots, u_N) = (x_1, \dots, x_N)\mathbf{G}^{-1}) \\ &\stackrel{(a)}{=} \sum_{\{1, \dots, N\} \setminus \{i_1, \dots, i_s\}} \frac{1}{2^N} = \frac{2^{N-s}}{2^N} = \frac{1}{2^s}. \end{aligned} \quad (2.1)$$

等号(a)是因为对于任意 (U_1, \dots, U_N) 有 $\Pr(U_1, \dots, U_N) = 2^{-N}$ 。同理可得 $\Pr(X_{j_1} = x_{j_1}, \dots, X_{j_t} = x_{j_t}) = \frac{1}{2^t}$ 和 $\Pr(X_{i_1} = x_{i_1}, \dots, X_{i_s} = x_{i_s}, X_{j_1} = x_{j_1}, \dots, X_{j_t} = x_{j_t}) = \frac{1}{2^{s+t}}$ ，即 $\Pr(X_{i_1} = x_{i_1}, \dots, X_{i_s} = x_{i_s})\Pr(X_{j_1} = x_{j_1}, \dots, X_{j_t} = x_{j_t}) = \Pr(X_{i_1} = x_{i_1}, \dots, X_{i_s} = x_{i_s}, X_{j_1} = x_{j_1}, \dots, X_{j_t} = x_{j_t})$ ，证毕。

推论 2.1 因为 $(X_1, X_2) = (U_1, U_2)\mathbf{F}$ 且 \mathbf{F} 可逆，所以 X_1 与 X_2 独立。

因为 W 是无记忆的且 X_1 与 X_2 独立，所以二维随机变量 (X_1, Y_1) 和 (X_2, Y_2) 独立，即 $\Pr(X_1 = u_1 \oplus u_2, X_2 = u_2, Y_1 = y_1, Y_2 = y_2) = \Pr(X_1 = u_1 \oplus u_2, Y_1) \Pr(X_2 = u_2, Y_2 = y_2)$ 。

从四维随机变量 (U_1, U_2, Y_1, Y_2) 的联合分布 $\Pr(U_1 = u_1, U_2 = u_2, Y_1 = y_1, Y_2 = y_2)$ 出发，我们可以获得一些边缘分布和条件分布。下面的两个条件分布被定义为传输信源比特 u_1 和 u_2 的“极化信道”，其中 $W_2^{(1)}$ 和 $W_2^{(2)}$ 分别表示第一个和第二个极化信道的概率集函数，使得符号容易辨认，避免全都写成 \Pr 。

定义 2.1 长度为2的极化码的两个极化信道

传输信源序列 u_1 的极化信道 $W_2^{(1)}(y_1, y_2 | u_1)$ 的转移概率为：

$$\begin{aligned}
W_2^{(1)}(y_1, y_2 | u_1) &= \Pr(y_1, y_2, u_1) / P(u_1) \stackrel{(a)}{=} 2 \sum_{u_2} \Pr(y_1, y_2, u_1, u_2) \\
&\stackrel{(b)}{=} \frac{1}{2} \sum_{u_2} \Pr(y_1, y_2 | u_1, u_2) \stackrel{(c)}{=} \frac{1}{2} \sum_{u_2} \Pr(y_1, y_2 | x_1, x_2) \\
&\stackrel{(d)}{=} \frac{1}{2} \sum_{u_2} W(y_1 | u_1 \oplus u_2) W(y_1 | u_2).
\end{aligned} \tag{2.2}$$

传输信源序列 u_2 的极化信道 $W_2^{(2)}(y_1, y_2, u_1 | u_2)$ 的转移概率为:

$$\begin{aligned}
W_2^{(2)}(y_1, y_2, u_1 | u_2) &= \Pr(y_1, y_2, u_1, u_2) / P(u_2) \stackrel{(a)}{=} 2 \Pr(y_1, y_2, u_1, u_2) \\
&\stackrel{(b)}{=} \frac{1}{2} \Pr(y_1, y_2 | u_1, u_2) \stackrel{(c)}{=} \frac{1}{2} \Pr(y_1, y_2 | x_1, x_2) \\
&\stackrel{(d)}{=} \frac{1}{2} W(y_1 | u_1 \oplus u_2) W(y_1 | u_2).
\end{aligned} \tag{2.3}$$

定义2.2中等号(a)和(b)是因为假设信源 U_1 和 U_2 是独立同分布的Bernoulli(1/2)随机变量, 等号(c)是因为随机变量 (U_1, U_2, Y_1, Y_2) 和 (X_1, X_2, Y_1, Y_2) 在可逆矩阵 \mathbf{F} 的作用下等价, 等号(d)是因为 W 是无记忆信道。直观上, 式(2.2)和(2.3)的区别在于(2.3)没有对 u_2 的求和号。

信道 $W_2^{(1)}(y_1, y_2 | u_1)$ 的输入是 u_1 , 输出是 y_1, y_2 , 其互信息是 $I(Y_1 Y_2; U_1)$ 。

信道 $W_2^{(2)}(y_1, y_2, u_1 | u_2)$ 的输入是 u_2 , 输出是 y_1, y_2, u_1 , 其互信息是 $I(Y_1 Y_2 U_1; U_2)$ 。

下面分析 $I(Y_1 Y_2; U_1)$ 和 $I(Y_1 Y_2 U_1; U_2)$ 的关系, 默认信源比特 U_1 和 U_2 是独立同分布的Bernoulli(1/2)随机变量。

定理 2.2 $I(Y_1 Y_2; U_1) + I(Y_1 Y_2 U_1; U_2) = 2I(X_1; Y_1) = 2I(W)$, 其中 $I(W) = I(X_1; Y_1) = I(X_2; Y_2)$ 表示信道 W 的互信息。此定理的意义是信道 W 的两次复用所能传递的信息等于极化信道 $W_2^{(1)}$ 和 $W_2^{(2)}$ 所能传递的信息的和, 极化信道不会损失信息传输的能力。

证明 $2I(X_1; Y_1) \stackrel{(a)}{=} I(X_1, X_2; Y_1, Y_2) \stackrel{(b)}{=} I(U_1, U_2; Y_1, Y_2) \stackrel{(c)}{=} I(U_1; Y_1, Y_2) + I(U_2; Y_1, Y_2 | U_1) \stackrel{(d)}{=} I(U_1; Y_1, Y_2) + I(U_2; Y_1, Y_2, U_1)$, 其中等号(a)是因为 (X_1, Y_1) 和 (X_2, Y_2) 独立, 等号(b)是因为 \mathbf{F} 可逆, 等号(c)是互信息的链式法则, 等号(d)是因为 U_1 和 U_2 独立, 证毕。

定理 2.3 $I(Y_1 Y_2; U_1) \leq I(Y_1 Y_2 U_1; U_2)$, 即 $W_2^{(2)}$ 传递信息的能力比 $W_2^{(1)}$ 大。

证明 $I(Y_1 Y_2 U_1; U_2) \stackrel{(a)}{=} I(U_2; Y_2) + I(U_2; Y_1 U_1 | Y_2) \stackrel{(b)}{=} I(X_2; Y_2) + I(U_2; Y_1 U_1 | Y_2)$, 其中等号(a)是互信息链式法则, 等号(b)是因为 $X_2 = U_2$ 。由于 $I(U_2; Y_1 U_1 | Y_2) \geq 0$, 所以 $I(Y_1 Y_2 U_1; U_2) \geq I(W)$, 又因为 $I(Y_1 Y_2; U_1) + I(Y_1 Y_2 U_1; U_2) = 2I(W)$, 所以有 $I(Y_1 Y_2 U_1; U_2) \geq I(W) \geq I(Y_1 Y_2; U_1)$, 证毕。

上面的定理就是极化现象的基础: $W_2^{(2)}$ 比 $W_2^{(1)}$ 具有更大的信道容量。当码长趋于无穷时, 极化信道的容量非零即一, 这一点将在下一节说明。

下面的定理叙述了 $W_2^{(1)}$ 和 $W_2^{(2)}$ 巴特查理亚系数的关系: $Z(W_2^{(2)})$ 的值小, $Z(W_2^{(1)})$ 的值大。

定理 2.4

$$\begin{aligned} Z(W_2^{(1)}) &= \sum_{y_1, y_2} \sqrt{W_2^{(1)}(y_1, y_2 | 0) W_2^{(1)}(y_1, y_2 | 1)} \leq 2Z(W) - [Z(W)]^2, \\ Z(W_2^{(2)}) &= \sum_{y_1, y_2, u_1} \sqrt{W_2^{(2)}(y_1, y_2, u_1 | 0) W_2^{(2)}(y_1, y_2, u_1 | 1)} = [Z(W)]^2, \\ Z(W_2^{(2)}) &\leq Z(W) \leq Z(W_2^{(1)}) \end{aligned} \quad (2.4)$$

第一个式子中的等号成立当且仅当 W 是二进制擦除信道 (Binary Erasure Channel, BEC)。第三个式子中两个等号成立当且仅当 $Z(W) = 1$ 或0。

证明 第二个式子的证明用 $W_2^{(2)}$ 的定义易得, 但第一个和第三个式子证明较长, 请参考文献[3]中的命题5。文献[3]里第三个式子证明过程中使用了闵科夫斯基不等式, 闵科夫斯基不等式刻画了一种推广的欧式空间中的距离的特点, 请注意参数 a_k 和 b_k 的灵活应用。闵科夫斯基不等式有参数 $p > 0$, $p > 1$ 和 $p < 1$ 时不等式的方向不一样, 请注意。

定理 2.5 $W_2^{(1)} \preceq W_2^{(2)}$.

证明 利用 W 做衔接。首先注意到 $W_2^{(2)}(y_1, y_2, u_1 | u_2)$ 的输出符号集合为 (Y_1, Y_2, U_1) , 利用例1.4中的操作, 将 $(Y_1, Y_2 = y_2, U_1)$ 对应的所有列合并 (在 $W_2^{(2)}$ 的转移矩阵中共计 $|\mathcal{Y}| \times 2$ 列相加), 再注意到 $X_2 = U_2$, 即得信道 $W(y_2 | x_2) = W$, 因此 $W \preceq W_2^{(2)}$ 。

下面说明 $W_2^{(1)} \preceq W$ 。设 W 是BI-DMSC且对应的置换是 π , 则存在信道 $P(y_1, y_2 | y) = \frac{1}{2}[W(y_2 | 0)\delta(y_1 = y) + W(y_2 | 1)\delta(y_2 = \pi(y))]$ 使得 $W_2^{(1)}(y_1, y_2 | u_1) = \sum_{y \in \mathcal{Y}} W(y | u_1)P(y_1, y_2 | y)$, 其中 $\delta(x)$ 表示如果 x 取逻辑值1, 则 $\delta(x) = 1$, 如果 $\delta(x)$ 表示如果 x 取逻辑值0, 则 $\delta(x) = 0$ 。

因此 $W_2^{(1)} \preceq W \preceq W_2^{(2)}$, 证毕。该证明取自文献[4]。

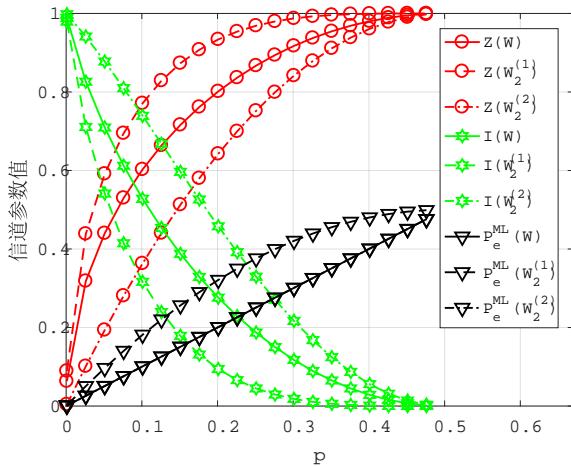


图 2.2 当 W 为 BSC 时, W 、 $W_2^{(1)}$ 和 $W_2^{(2)}$ 对应的 P_e^{ML} 、 I 和 Z

定理 2.6 如果 $W \preceq Q$, 则 $W_2^{(1)} \preceq Q_2^{(1)}$, $W_2^{(2)} \preceq Q_2^{(2)}$, 即本来就好的传输信道在极化之后仍然好。

证明 利用定义1.6和定义2.1易得。

例 2.1 当 W 是二进制对称信道 (Binary Symmetric Channel, BSC) 时, $W_2^{(1)}$ 和 $W_2^{(2)}$ 的例子。设 W 的交叉概率是 p , 则 W , $W_2^{(1)}$ 和 $W_2^{(2)}$ 的转移矩阵如下 (记 $\bar{p} = 1 - p$)。

$$W = \begin{bmatrix} \bar{p} & p \\ p & \bar{p} \end{bmatrix}, \quad (2.5)$$

$$W_2^{(1)} = \frac{1}{2} \begin{bmatrix} p^2 + \bar{p}^2 & 2p\bar{p} & 2p\bar{p} & p^2 + \bar{p}^2 \\ 2p\bar{p} & p^2 + \bar{p}^2 & p^2 + \bar{p}^2 & 2p\bar{p} \end{bmatrix}, \quad (2.6)$$

$$W_2^{(2)} = \frac{1}{2} \begin{bmatrix} \bar{p}^2 & p^2 & p\bar{p} & p\bar{p} & p\bar{p} & p\bar{p} & p^2 & p^2 \\ p^2 & \bar{p}^2 & p\bar{p} & p\bar{p} & p\bar{p} & p\bar{p} & p^2 & \bar{p}^2 \end{bmatrix}. \quad (2.7)$$

注意到 $P_e^{\text{ML}}(W) = P_e^{\text{ML}}(W_2^{(2)}) = p$ 。 W 、 $W_2^{(1)}$ 和 $W_2^{(2)}$ 对应的 P_e^{ML} 、 I 和 Z 随 p 的变化如图2.2所示。

例 2.2 当 W 是 BEC 时, $W_2^{(1)}$ 和 $W_2^{(2)}$ 的例子。设 W 的删除概率是 ϵ , 则 W , $W_2^{(1)}$ 和 $W_2^{(2)}$ 的

转移矩阵如下（记 $\bar{\epsilon} = 1 - \epsilon$ ）。

$$W = \begin{bmatrix} \bar{\epsilon} & \epsilon & 0 \\ 0 & \epsilon & \bar{\epsilon} \end{bmatrix}, \quad (2.8)$$

$$W_2^{(1)} = \frac{1}{2} \begin{bmatrix} \bar{\epsilon}^2 & \bar{\epsilon}^2 & \epsilon\bar{\epsilon} & \epsilon\bar{\epsilon} & 2\epsilon^2 & \epsilon\bar{\epsilon} & \epsilon\bar{\epsilon} & 0 & 0 \\ 0 & 0 & \epsilon\bar{\epsilon} & \epsilon\bar{\epsilon} & 2\epsilon^2 & \epsilon\bar{\epsilon} & \epsilon\bar{\epsilon} & \bar{\epsilon}^2 & \bar{\epsilon}^2 \end{bmatrix} \stackrel{(a)}{=} \begin{bmatrix} \bar{\epsilon}^2 & 2\epsilon - \epsilon^2 & 0 \\ 0 & 2\epsilon - \epsilon^2 & \bar{\epsilon}^2 \end{bmatrix}. \quad (2.9)$$

上式中等号(a)是指合并“不会错误判决的符号”和“无法区分的符号”，这样的操作不会对 $W_2^{(1)}$ 产生任何影响。“不会错误判决的符号” y 是指 $W_2^{(1)}(y|0)$ 和 $W_2^{(1)}(y|1)$ 中有一个为0，而另一个非0；“无法区分的符号” y 是指 $W_2^{(1)}(y|0) = W_2^{(1)}(y|1)$ ，其中 y 是极化信道 $W_2^{(1)}$ 的某个输出符号。可见 $W_2^{(1)}$ 仍是BEC信道。同理可得 $W_2^{(2)}$ 的转移概率矩阵如下。

$$W_2^{(2)} = \begin{bmatrix} 1 - \epsilon^2 & \epsilon^2 & 0 \\ 0 & \epsilon^2 & 1 - \epsilon^2 \end{bmatrix}. \quad (2.10)$$

本例说明了BEC信道在极化过程中产生的极化信道仍然是BEC信道，所以BEC信道的极化规律具有闭合表达式，也就是式(2.4)中取等号的情况。

§ 2.2 长度为 2^n 的极化码

长度为 $N = 2^n$ 的极化码是长度为2的极化码的扩展，即长度为2的极化码产生的极化信道 $W_2^{(1)}$ 和 $W_2^{(2)}$ 被当作另一个长度为2的极化码的 W 。一个长度为4的极化码的极化过程如图2.3所示，其中 (u_1, u_2, u_3, u_4) 是信源比特， (x_1, x_2, x_3, x_4) 是码字比特。依照图中走线，编码过程从左向右看，极化过程从右向左看。

聪明的你马上就能发现规律：当 $N = 2$ 时，极化过程可以表示为 $(W, W) \rightarrow (W_2^{(1)}, W_2^{(2)})$ ；当 $N = 4$ 时， $(W_2^{(1)}, W_2^{(1)}) \rightarrow (W_4^{(1)}, W_4^{(2)})$ ， $(W_2^{(2)}, W_2^{(2)}) \rightarrow (W_4^{(3)}, W_4^{(4)})$ 。

一般来讲，这个规律是： $(W_N^{(i)}, W_N^{(i)}) \rightarrow (W_{2N}^{(2i-1)}, W_{2N}^{(2i)})$ ，其中 $W_N^{(i)}$ 是长度为 N 的极化码的第 i 个极化信道，而 $W_{2N}^{(2i-1)} W_{2N}^{(2i)}$ 是长度为 $2N$ 的极化码的第 $2i-1$ 和 $2i$ 个极化信道。

依照这个规律，一个长度为8的极化码的极化过程如图2.4所示，其中 (u_1, u_2, \dots, u_8) 是信源比特， (x_1, x_2, \dots, x_8) 是码字比特。

当图2.3和2.4已经摆在我面前时，从右向左地发现 $(W_N^{(i)}, W_N^{(i)}) \rightarrow (W_{2N}^{(2i-1)}, W_{2N}^{(2i)})$ 是容易的。接下来就是找到图2.3和图2.4的绘制规律，也就是极化码编码过程或极化过程的通用描

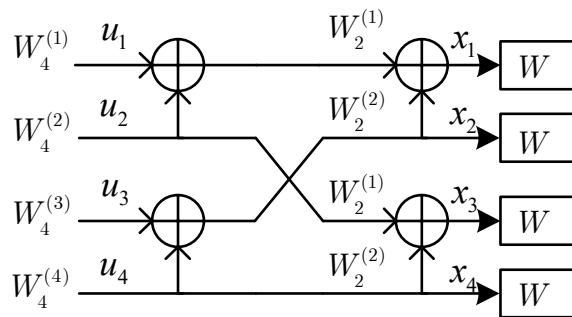


图 2.3 长度为4的极化码

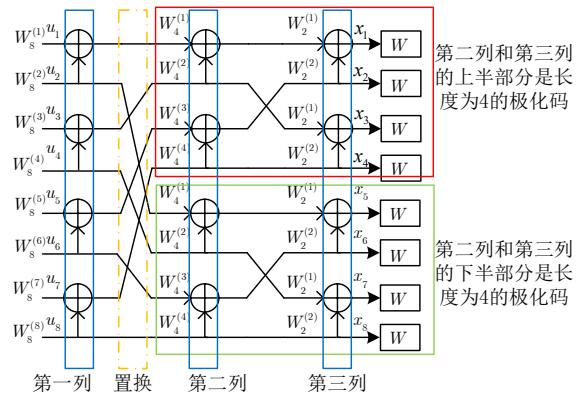


图 2.4 长度为8的极化码

述。此后我们把如图2.3和2.4中所示，由“ \oplus ”和“走线”构成的图称为长度为 N 的极化码的编码图，表示这张图的矩阵被称为生成矩阵 \mathbf{G}_N 。例如当 $N=2$ 是生成矩阵为 $\mathbf{G}_2 = \mathbf{F} = [\begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix}]$

本段话可以参考图2.4阅读。长度为 N 的极化码编码图的最左列是竖着排列的 $N/2$ 个长度为2的极化码的编码图，所有这 $N/2$ 个长度为2的极化码的第一个码字比特 $(u_1 \oplus u_2, u_3 \oplus u_4, \dots, u_{N-1} \oplus u_N)$ 被置换到上一半，所有这 $N/2$ 个长度为2的极化码的第二个码字比特 (u_2, u_4, \dots, u_N) 被置换到下一半。从左侧数第2至第 n 列的上半部分是一个长度为 $N/2$ 的极化码的编码图，从左侧数第2至第 n 列的下半部分和上半部分完全一致，也是一个长度为 $N/2$ 的极化码的编码图。这就是极化码编码图的递归规律，写成矩阵乘法如下。

$$\mathbf{G}_N = (\mathbf{I}_{N/2} \otimes \mathbf{F}) \mathbf{R}_N (\mathbf{I}_2 \otimes \mathbf{G}_{N/2}). \quad (2.11)$$

上式中“ \otimes ”表示Kronecker乘法。第一项 $\mathbf{I}_{N/2} \otimes \mathbf{F}$ 图2.4中的第一列。第二项 \mathbf{R}_N 是一个置换矩阵，对应图2.4中的置换，即 $(a_1, a_2, \dots, a_N) \mathbf{R}_N = (a_1, a_3, a_5, \dots, a_{N-1}, a_2, a_4, a_6, \dots, a_N)$ 。第

三项 $\mathbf{I}_2 \otimes \mathbf{G}_{N/2}$ 对应图中的红色和绿色框。 \mathbf{G}_N 的可逆性显而易见，因为第一项 $\mathbf{I}_{N/2} \otimes \mathbf{F}$ 和第三项 $\mathbf{I}_2 \otimes \mathbf{G}_{N/2}$ 是分块对角阵，对角线上的子矩阵都可逆，第二项 \mathbf{R}_N 是置换矩阵显然可逆，因此 \mathbf{G}_N 的可逆。

定义 2.2 洗牌置换

说一个 $m_1 m_2 \times m_1 m_2$ 矩阵 \mathbf{S}_{m_1, m_2} 是一个洗牌置换矩阵，乃是指 \mathbf{S}_{m_1, m_2} 具有如下的特性。

$$\begin{aligned}\mathbf{S}_{m_1, m_2}(a_1, a_2, a_3, \dots, a_{m_1 m_2})^T &= (a_1, a_{m_1+1}, a_{2m_1+1}, \dots, a_{(m_2-1)m_1+1}, \\ &\quad a_2, a_{m_1+2}, a_{2m_1+2}, \dots, a_{(m_2-1)m_1+2}, \dots, a_{m_1}, a_{2m_1}, a_{3m_1}, \dots, a_{m_2 m_1+1})^T.\end{aligned}\tag{2.12}$$

如果想对行向量进行置换，只需对上式等号两侧都取转置。 \mathbf{S}_{m_1, m_2}^T 具有如下性质。

推论 2.2

$$\mathbf{S}_{m_1, m_2}^T = \mathbf{S}_{m_1, m_2}^{-1} = \mathbf{S}_{m_2, m_1}.\tag{2.13}$$

证明 第一个等号是置换矩阵的基本性质，第二个等号通过计算 $\mathbf{S}_{m_1, m_2} \mathbf{S}_{m_2, m_1} = \mathbf{I}$ 即可发现。

引理 2.1 对于矩阵 \mathbf{A}_{m_1, n_1} 和 \mathbf{B}_{m_2, n_2} ，有下面的恒等式。

$$\mathbf{S}_{m_1, m_2}(\mathbf{B}_{m_2, n_2} \otimes \mathbf{A}_{m_1, n_1}) = (\mathbf{A}_{m_1, n_1} \otimes \mathbf{B}_{m_2, n_2})\mathbf{S}_{n_1, n_2}\tag{2.14}$$

证明 利用Kronecker乘法的定义和 \mathbf{S}_{m_1, m_2} 的定义，对乘积矩阵中的元素逐一计算即可发现规律。

引理 2.2 如果矩阵 \mathbf{A} 和 \mathbf{C} 能做普通乘法，矩阵 \mathbf{B} 和 \mathbf{D} 能做普通乘法，则有 $(\mathbf{A} \otimes \mathbf{B})(\mathbf{C} \otimes \mathbf{D}) = (\mathbf{AC}) \otimes (\mathbf{BD})$ 。

证明 这是Kronecker乘法招牌性质，对等号两侧所有元素逐一验证即可。

定理 2.7 $\mathbf{G}_N = \mathbf{B}_N \mathbf{F}^{\otimes n}$ ，其中 $n = \log_2 N$ ，指数位置上的 \otimes 是Kronecker幂， \mathbf{B}_N 是比特倒序置换矩阵，即 \mathbf{B}_N 把索引 $< i_1 i_2 \dots i_n >_2$ 上的元素置换到索引 $< i_n i_{n-1} \dots i_1 >_2$ 上， $< i_1 i_2 \dots i_n >_2$ 和 $< i_n i_{n-1} \dots i_1 >_2$ 分别是 $i = \sum_{k=1}^n i_k 2^{n-k} k$ 和 $j = \sum_{k=1}^n i_k 2^{k-1}$ 的二进制形式。

证明 利用(2.11)中的递归关系和引理2.1、2.2可得。注意灵活使用单位矩阵进行配凑。

定义 2.3 极化码编码

长度为 N 的极化码的编码过程可以写成GF(2)上的矩阵乘法：

$$\mathbf{x}_1^N = \mathbf{u}_1^N \mathbf{G}_N. \quad (2.15)$$

其中 $\mathbf{x}_1^N = (x_1, x_2, \dots, x_N)$ 是码字比特， $\mathbf{u}_1^N = (u_1, u_2, \dots, u_N)$ 是信源序列。以后我们都用这样的记号表示向量，即 $\mathbf{a}_i^j = (a_i, a_{i+1}, \dots, a_j)$ 。如果 $j < i$ ，则 \mathbf{a}_i^j 等于什么也没有。

从联合分布 $\Pr(\mathbf{U}_1^N, \mathbf{Y}_1^N)$ 出发，其中 \mathbf{U}_1^N 是 N 维信源随机变量， \mathbf{U}_1^N 中所有元素独立同分布， $U_1 \sim \text{Bernoulli}(1/2)$ ， \mathbf{U}_1^N 经过式(2.15)编码得到 \mathbf{X}_1^N ，发送 \mathbf{X}_1^N ，得到接收信号 \mathbf{Y}_1^N ，我们可以得到下面的边缘的、条件的分布，称为极化信道。

定义 2.4 极化信道

第*i*个信源比特 u_i 所经历的第*i*个极化信道具有如下的条件分布，这个条件分布只不过是 $\Pr(\mathbf{U}_1^N, \mathbf{Y}_1^N)$ 的边缘的、条件分布。

$$\begin{aligned} W_N^{(i)}(\mathbf{y}_1^N, \mathbf{u}_1^{i-1} | u_i) &= \Pr(\mathbf{y}_1^N, \mathbf{u}_1^i) / \Pr(u_i) \\ &= 2 \sum_{u_{i+1}^N} \Pr(\mathbf{y}_1^N, \mathbf{u}_1^N) = \frac{2}{\Pr(\mathbf{u}_1^N)} \sum_{u_{i+1}^N} \Pr(\mathbf{y}_1^N | \mathbf{u}_1^N) \\ &= \frac{1}{2^{N-1}} \sum_{u_{i+1}^N} \Pr(\mathbf{y}_1^N | \mathbf{x}_1^N) \stackrel{(a)}{=} \frac{1}{2^{N-1}} \sum_{u_{i+1}^N} \prod_{i=1}^N W(y_i | x_i). \end{aligned} \quad (2.16)$$

其中 $\mathbf{x}_1^N = \mathbf{u}_1^N \mathbf{G}_N$ ，等号(a)是因为信道 W 是无记忆的。

上式中 $W_N^{(i)}$ 表示概率集函数，相当于 \Pr ，也用来代指第*i*个极化信道。如同转移概率 $W(y|x)$ 的定义一样，信道 $W_N^{(i)}(\mathbf{y}_1^N, \mathbf{u}_1^{i-1} | u_i)$ 的输入是 u_i ，输出是 \mathbf{y}_1^N 和 \mathbf{u}_1^{i-1} ，即极化信道 $W_N^{(i)}$ 不仅能观测到物理信道 W 的输出 \mathbf{y}_1^N ，还能观测到比特值 $(u_1, u_2, \dots, u_{i-1})$ 。这是因为极化码使用串行抵消译码，当从 u_1 开始逐一估计信源比特，直到 u_N 。因此，在译码 u_i 时， $(u_1, u_2, \dots, u_{i-1})$ 的值都已经获得，被当作译码 u_i 所需要的反馈。

定义 2.5 极化信道的递归关系

在长度为 N 的极化码中，极化信道有如下的递归关系。下面的两个式子看起来复杂，其

实它们本质上与长度为2的极化码的两个极化信道 $W_2^{(1)}$ 和 $W_2^{(2)}$ 所表达的含义完全一样。

$$W_N^{(2i-1)}(\mathbf{y}_1^N, \mathbf{u}_1^{2i-2} | u_{2i-1}) = \frac{1}{2} \sum_{u_{2i}} W_{N/2}^{(i)}(\mathbf{y}_1^{N/2}, \mathbf{u}_{1,o}^{2i-2} \oplus \mathbf{u}_{1,e}^{2i-2} | u_{2i_1} \oplus u_{2i}) W_{N/2}^{(i)}(\mathbf{y}_{N/2+1}^N, \mathbf{u}_{1,e}^{2i-2} | u_{2i}), \quad (2.17)$$

$$W_N^{(2i)}(\mathbf{y}_1^N, \mathbf{u}_1^{2i-2} | u_{2i}) = \frac{1}{2} W_{N/2}^{(i)}(\mathbf{y}_1^{N/2}, \mathbf{u}_{1,o}^{2i-2} \oplus \mathbf{u}_{1,e}^{2i-2} | u_{2i_1} \oplus u_{2i}) W_{N/2}^{(i)}(\mathbf{y}_{N/2+1}^N, \mathbf{u}_{1,e}^{2i-2} | u_{2i}). \quad (2.18)$$

其中 $\mathbf{u}_{1,o}^{2i-2}$ 表示 \mathbf{u}_1^{2i-2} 中索引为奇数的元素， $\mathbf{u}_{1,e}^{2i-2}$ 表示 \mathbf{u}_1^{2i-2} 中索引为偶数的元素， $\mathbf{u}_{1,o}^{2i-2} \oplus \mathbf{u}_{1,e}^{2i-2}$ 表示向量逐位模2加法。上面两个式子的物理含义和定义2.16中的解释相同。

对于大多数初学者而言，这两个式子具有致命的“复杂度”，文献[3]从联合分布 $\Pr(\mathbf{U}_1^N, \mathbf{Y}_1^N)$ 出发推出了上面的式子，但是对于初学者而言其物理含义仍然不容易看清。下面让我们通过一张图看看这两个式子在说什么。

定义 2.6 极化模块

一个长度为2的极化码称为一个 2×2 极化模块，即图2.1就是 2×2 极化模块。

以下的四个自然段请对应图2.5阅读。我们首先考虑上角标 $2i-1$ 、 $2i$ 和 i 的含义。如果一个比特是 u_{2i-1} ，那么他必然和 u_{2i} 位于同一个 2×2 极化模块的输入端。在极化码编码图的第一列中， u_{2i-1} 和 u_{2i} 对应的 2×2 极化模块就是第 i 个极化模块，换句话说 u_{2i-1} 和 u_{2i} 对应的 2×2 极化模块之前还有 $i-1$ 个 2×2 极化模块，这些前面的极化模块对应的是 $(u_1, u_2), \dots, (u_{2i-3}, u_{2i-2})$ 。

u_{2i-1} 和 u_{2i} 对应的 2×2 极化模块右侧的两根走线分别连接到两个长度为 $N/2$ 的极化码的极化信道上，被连接的极化信道恰都是长度为 $N/2$ 的极化码中的第 i 个极化信道，这是因为长度 $N/2$ 的极化码中的第1至第 $i-1$ 个极化信道都被 u_{2i-1} 和 u_{2i} 对应的 2×2 极化模块之前的 $i-1$ 个 2×2 极化模块连接过了。 $W_{N/2}^{(i)}(\mathbf{y}_1^{N/2}, \mathbf{u}_{1,o}^{2i-2} \oplus \mathbf{u}_{1,e}^{2i-2} | u_{2i_1} \oplus u_{2i})$ 能观测 $\mathbf{y}_1^{N/2}$ ，这在图中是显然的，因为位于上侧的长度 $N/2$ 的极化码的观测值就是 $\mathbf{y}_1^{N/2}$ ； $W_{N/2}^{(i)}(\mathbf{y}_1^{N/2}, \mathbf{u}_{1,o}^{2i-2} \oplus \mathbf{u}_{1,e}^{2i-2} | u_{2i_1} \oplus u_{2i})$ 还能观测到 $\mathbf{u}_{1,o}^{2i-2} \oplus \mathbf{u}_{1,e}^{2i-2}$ ，这也是显然的，因为 $u_1 \oplus u_2, u_3 \oplus u_4, \dots, u_{2i-3} \oplus u_{2i-2}$ 由 u_{2i-1} 和 u_{2i} 对应的 2×2 极化模块之前的 $i-1$ 个 2×2 极化模块所输送，尽管还没有讲译码，但看到这里的读者都知道，极化码的串行抵消译码是序贯译码，在译码 u_{2i-1} 时， u_1 至 u_{2i-2} 的值都已经获得，而 $\mathbf{u}_{1,o}^{2i-2} \oplus \mathbf{u}_{1,e}^{2i-2}$ 只不过是利用 u_1 至 u_{2i-2} 的值算出的结果而已； $W_{N/2}^{(i)}(\mathbf{y}_1^{N/2}, \mathbf{u}_{1,o}^{2i-2} \oplus \mathbf{u}_{1,e}^{2i-2} | u_{2i_1} \oplus u_{2i})$ 的输入为 $u_{2i_1} \oplus u_{2i}$ ，这是因为 u_{2i-1} 和 u_{2i} 对应的 2×2 极化模块将计算结果之一，也就是 $u_{2i_1} \oplus u_{2i}$ 送入了该信道作为输入。

$W_{N/2}^{(i)}(\mathbf{y}_{N/2+1}^N, \mathbf{u}_{1,e}^{2i-2} | u_{2i})$ 的含义按上一段类推。

从而，把 $W_{N/2}^{(i)}(\mathbf{y}_1^{N/2}, \mathbf{u}_{1,o}^{2i-2} \oplus \mathbf{u}_{1,e}^{2i-2} | u_{2i_1} \oplus u_{2i})$ 和 $W_{N/2}^{(i)}(\mathbf{y}_{N/2+1}^N, \mathbf{u}_{1,e}^{2i-2} | u_{2i})$ 视为 W ，把 u_{2i} 视为 u_2 ，即可由式(2.2)写出式(2.17)，由式(2.3)写出式(2.18)。

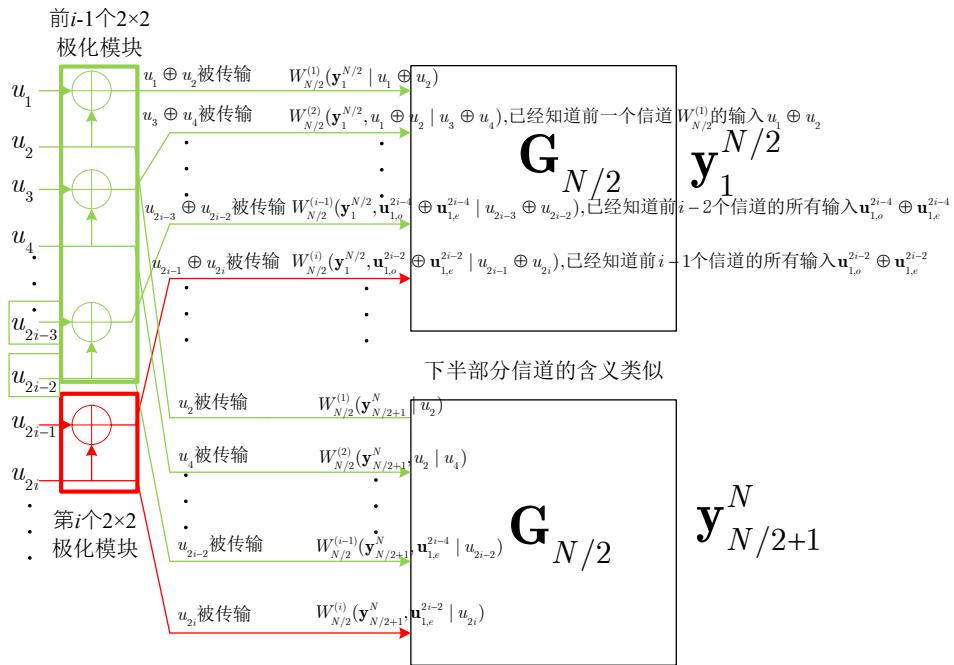


图 2.5 极化信道递归的含义

§ 2.3 长度为 2^n 的极化码可达信道容量

这一节简单地介绍极化码的信道容量可达的性质，即当码字长度 $N = 2^n$ 趋于无穷时，如果极化码的码率低于信道容量，则极化码能够以任意小的错误率传输信息。

推论 2.3 这是定理2.2的推论：

$$\sum_{i=1}^N I(W_N^{(i)}) = \sum_{i=1}^N I(\mathbf{Y}_1^N, \mathbf{U}_1^{i-1}; U_i) = NI(W) = NI(X; Y). \quad (2.19)$$

证明 在极化码编码图上，从右向左，由定理2.2，每个 2×2 极化模块都不损失信息。

定义 2.7 模型等价

设*n*是非负整数， $N = 2^n$ ， $\{0, 1, \dots, N - 1\}$ 是由连续十进制自然数构成的集合，把该集合中的十进制数展开成*n*位二进制数得到集合 $\mathcal{B} = \{(000\dots000), (000\dots001), (000\dots010)\dots, (111\dots111)\}$ ，

则下面的两个系统模型等价。

系统模型1：取出 \mathcal{B} 的一个子集 \mathcal{B}_s ，计算 \mathcal{B}_s 中的元素占 \mathcal{B} 中元素数量的比例。这个结果显然是 $|\mathcal{B}_s|/N$ ，其中 $|\mathcal{B}|$ 表示 \mathcal{B}_s 中元素的数量。

系统模型2：一个随机系统每次会输出 \mathcal{B} 中的一个元素。经过统计发现 \mathcal{B} 中每个元素出现的概率相等，都是 $1/N$ 。取出 \mathcal{B} 的一个子集 \mathcal{B}_s ，计算下一次该随机系统输出的元素属于 \mathcal{B}_s 的概率，这个概率显然是 $|\mathcal{B}_s|/N$ ，其中 $|\mathcal{B}|$ 表示 \mathcal{B}_s 中元素的数量。

因此，当所有单点事件发生的概率相等时，样本点子集 \mathcal{B}_s 中元素的数量占总样本点数量的比例 \Leftrightarrow 样本点子集 \mathcal{B}_s 对应事件发生的概率。

在下面的叙述中，我们的目标是计算“比例”，而所用的工具是一些简单的“概率”。

定义 2.8 I_n 的定义

I_n 是随机变量， I_n 的所有实现值是长度为 $N = 2^n$ 的极化码中 N 个极化信道容量的值，且 I_n 所有实现值出现的概率相等，都是 $1/N$ 。

例 2.3 I_0, I_1, I_n 的分布率如下表。

I_0	$I(W)$
Pr	1

I_1	$I(W_2^{(1)})$	$I(W_2^{(2)})$
Pr	1/2	1/2

I_n	$I(W_N^{(1)})$	$I(W_N^{(2)})$...	$I(W_N^{(N)})$
Pr	1/ N	1/ N	...	1/ N

定义 2.9 Z_n 的定义

Z_n 是随机变量， Z_n 的所有实现值是长度为 $N = 2^n$ 的极化码中 N 个极化信道巴特查理亚系数的值，且 Z_n 所有实现值出现的概率相等，都是 $1/N$ 。

例 2.4 Z_0, Z_1, Z_n 的分布律如下表。

Z_0	$Z(W)$
Pr	1

Z_1	$Z(W_2^{(1)})$	$Z(W_2^{(2)})$
Pr	1/2	1/2

Z_n	$Z(W_N^{(1)})$	$Z(W_N^{(2)})$...	$Z(W_N^{(N)})$
Pr	1/N	1/N	...	1/N

由上面两个例子可见, I_n 和 Z_n 都满足树形随机过程, 如图2.6所示。一旦 I_n 的实现值确定, 那么 $I_{n-1}, I_{n-2}, \dots, I_1, I_0$ 的实现值就确定了, 即只需沿着树的边走到根节点。例如, 从如果 $I_5 = I(W_{32}^{(27)})$ 给定, 则 $I_4 = I(W_{16}^{(14)})$, $I_3 = I(W_8^{(7)})$, $I_2 = I(W_4^{(4)})$, $I_1 = I(W_2^{(2)})$ 和 $I_0 = I(W)$ 就完全确定:

$$\Pr(I_n, I_{n-1}, \dots, I_1, I_0) = \begin{cases} \frac{1}{2^n}, & \text{沿 } I_{n-1}, \dots, I_1, I_0 \text{ 可达根节点,} \\ 0, & \text{非法的样本函数, 不存在于树形随机过程中.} \end{cases} \quad (2.20)$$

此后的叙述需要如下的定义。

定义 2.10 条件期望

设二维离散随机变量 (X, Y) 的联合分布 $\Pr(X = x, Y = y)$ 已知, 则有:

$$E\{X|Y = y\} = \sum_x xP(X = x|Y = y) = \sum_x x \frac{P(X = x, Y = y)}{\Pr(Y = y)}. \quad (2.21)$$

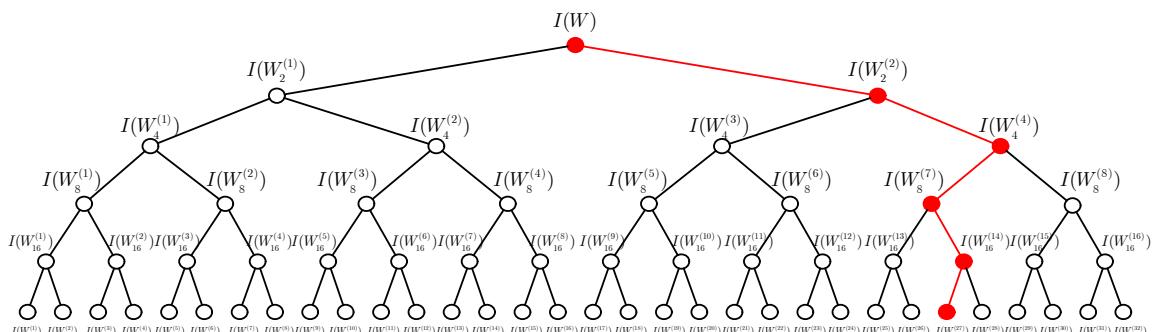


图 2.6 I_n 的树形随机过程, 共计32个样本函数, 每个样本函数的概率是1/32

考虑树形随机过程中的截断 $(I_{n+1}, I_n, \dots, I_1, I_0)$, 利用联合概率分布 $\Pr(I_{n+1}, I_n, \dots, I_1, I_0)$ 可以写出下面的式子。

$$\begin{aligned} E\{I_{n+1}|I_n, \dots, I_0\} &= I_n^+ \Pr(I_{n+1} = I_n^+|I_n, \dots, I_0) + I_n^- \Pr(I_{n+1} = I_n^-|I_n, \dots, I_0) \\ &= I_n^+ \frac{\Pr(I_{n+1} = I_n^+, I_n, \dots, I_0)}{\Pr(I_n, \dots, I_0)} + I_n^- \frac{\Pr(I_{n+1} = I_n^-, I_n, \dots, I_0)}{\Pr(I_n, \dots, I_0)} \\ &\stackrel{(a)}{=} \frac{2^n}{2^{n+1}} I_n^+ + \frac{2^n}{2^{n+1}} I_n^- \stackrel{(b)}{=} I_n. \end{aligned} \quad (2.22)$$

其中 I_n^- 表示从 I_n 的实现值 $I(W_{2^n}^k)$ 出发, 经过一个 2×2 极化模块后所得的 $I(W_{2^{n+1}}^{2k-1})$; I_n^+ 表示从 I_n 的实现值 $I(W_{2^n}^k)$ 出发, 经过一个 2×2 极化模块后所得的 $I(W_{2^{n+1}}^{2k})$, I_{n+1} 只能取 I_n^- 或 I_n^+ , 没有第三个值, 且由定理2.3可知 $I_n^- \leq I_n^+$ 。等号(a)由式(2.20)可得, 等号(b)由定理2.2可得。

同理, 当考虑巴氏系数的随机过程时, 截断 $(Z_{n+1}, Z_n, \dots, Z_1, Z_0)$ 具有下面的性质:

$$\begin{aligned} E\{Z_{n+1}|Z_n, \dots, Z_0\} &= Z_n^+ \Pr(Z_{n+1} = Z_n^+|Z_n, \dots, Z_0) + Z_n^- \Pr(Z_{n+1} = Z_n^-|Z_n, \dots, Z_0) \\ &= Z_n^+ \frac{\Pr(Z_{n+1} = Z_n^+, Z_n, \dots, Z_0)}{\Pr(Z_n, \dots, Z_0)} + Z_n^- \frac{\Pr(Z_{n+1} = Z_n^-, Z_n, \dots, Z_0)}{\Pr(Z_n, \dots, Z_0)} \\ &= \frac{1}{2}(Z_n^+ + Z_n^-) \stackrel{(a)}{\leq} Z_n. \end{aligned} \quad (2.23)$$

其中不等号(a)是因为式(2.4)。

式(2.22)和(2.23)表明了随机过程 I_n 和 Z_n 分别是鞅和上鞅。

定义 2.11 鞅和上鞅

随机过程 $\{X_n, n \geq 0\}$ 是鞅(上鞅), 乃是指它具有如下的性质:

- (i). $\forall n \geq 0, E\{|X_n|\} < +\infty.$ $(0 \leq I_n, Z_n \leq 1)$
- (ii). $\forall n \geq 0, E\{X_{n+1}|X_n, X_{n-1}, \dots, X_0\} = (\leq) X_n.$ (式(2.22)和(2.23))

利用鞅论中的结论(教材[12]第6章第5节), 鞅 $\{I_n, n \geq 0\}$ 满足下式¹:

$$E\{I_\infty\} = E\{I_0\} \stackrel{(a)}{=} I_0. \quad (2.24)$$

¹这虽是随机过程中的简单结论, 但是它的证明对于工学生而言并不简单, 动辄需要实分析和测度论。举个简单例子, 我们既然在式(2.24)中写出了 I_∞ , 那么我们必须证明 I_∞ 是存在的: $\{I_n, n \geq 0\}$ 收敛于 I_∞ , 这已经超过了对工学硕士的要求, 博士除外。

其中等号(a)是由极化码决定的: I_0 是常数。

当 n 趋于无穷时, 直接套用鞅论中的结果, 上鞅 $\{Z_n, n \geq 0\}$ 满足下式:

$$E\{|Z_{n+1} - Z_n|\} \rightarrow 0. \quad (2.25)$$

注意到 Z_{n+1} 以概率 $\frac{1}{2}$ 等于 Z_n^2 , 那么我们写出下式:

$$\begin{aligned} E\{|Z_{n+1} - Z_n|\} &= \sum_{z_n} [\sum_{z_{n+1}} |z_{n+1} - z_n| \Pr\{z_{n+1}|z_n\}] \Pr\{z_n\} \\ &\stackrel{(a)}{=} \sum_{z_n} [\frac{1}{2}|z_n^2 - z_n| + \frac{1}{2}|\xi - z_n|] \Pr\{z_n\} \\ &\stackrel{(b)}{\geq} \frac{1}{2} \sum_{z_n} z_n(1 - z_n) \Pr\{z_n\} = \frac{1}{2} E\{Z_n(1 - Z_n)\}. \end{aligned} \quad (2.26)$$

其中等号(a)中的 ξ 是 Z_{n+1} 除了 z_n^2 外的另一个取值, $\Pr\{z_{n+1}|z_n\}$ 恒等于 $\frac{1}{2}$ 。不等号(b)是因为舍弃带 ξ 的正项, 且 $z_n^2 - z_n < 0$ 。显然 $E\{Z_n(1 - Z_n)\} \geq 0$ 。由式(2.26): $E\{|Z_{n+1} - Z_n|\} \rightarrow 0$, 所以当 n 趋于无穷时, 有 $E\{Z_n(1 - Z_n)\} = 0$, 这表明 $Z_n = 0$ 或 1 (几乎处处)。

由第一章中的式(1.3): $\log_2 \frac{2}{1+Z(W)} \leq I(W) \leq \sqrt{1 - Z(W)^2}$, 当 $Z_n = 0$, I_n 只能是1; 当 $Z_n = 1$, I_n 只能是0。所以 I_∞ 是伯努利随机变量。由式(2.24), 伯努利随机变量 I_∞ 的均值为 I_0 , 所以 I_∞ 的分布律如下:

I_∞	0	1
Pr	$1 - I_0$	I_0

可以看到 $I_\infty = 1$ 的概率是 I_0 , 由定义2.7, 这等价于“当码长为无穷时, 容量为1的极化信道占所有极化信道的比例是 I_0 , 容量为0的极化信道所占比例为 $1 - I_0$ ”。通过把信息比特放置在容量为1的极化信道上, 把取值固定的比特 (称为冻结比特) 仿真容量为0的极化信道上, 我们就有希望在码率低于 I_0 时, 实现任意小错误率的信息传输。下面就说明这么做确实可以让错误率任意小。

定义 2.12 串行抵消 (Successive Cancellation, SC) 译码下的错误事件

考虑联合分布 $(\mathbf{U}_1^N, \mathbf{Y}_1^N)$, 其中 \mathbf{U}_1^N 是信息比特, \mathbf{Y}_1^N 是码字 $\mathbf{X}_1^N = \mathbf{U}_1^N \mathbf{G}_N$ 的接收信号。SC译码逐个估计信息比特: 从 u_1 开始, 到 u_N 为止, 用 $(\mathbf{u}_1^{i-1}, \mathbf{y}_1^N)$ 估计 u_i 的值。更具体讲, SC译码器通过计算 $\Pr\{\mathbf{y}_1^N, \mathbf{u}_1^{i-1} | u_i = 0\}$ 和 $\Pr\{\mathbf{y}_1^N, \mathbf{u}_1^{i-1} | u_i = 1\}$ 的值, 以及判断 u_i 是否为冻结比特来译码。前面两个条件概率就是 $W_N^{(i)}\{\mathbf{y}_1^N, \mathbf{u}_1^{i-1} | u_i\}$ 。我们在后面专门讲译码器的时候再具体说明

计算细节，这里不需要细节也能看懂。

SC译码估计 u_i 的规则是：（其中 u_i 表示比特的发送值， \hat{u}_i 是对 u_i 的估计值）

$$\hat{u}_i = \text{SC}(\hat{\mathbf{u}}_1^{i-1}, \mathbf{y}_1^N) = \begin{cases} u_i, u_i \text{是冻结比特, 其值早已告知SC译码器,} \\ 0, \Pr\{\mathbf{y}_1^N, \hat{\mathbf{u}}_1^{i-1} | u_i = 0\} \geq \Pr\{\mathbf{y}_1^N, \hat{\mathbf{u}}_1^{i-1} | u_i = 1\}, \\ 1, \Pr\{\mathbf{y}_1^N, \hat{\mathbf{u}}_1^{i-1} | u_i = 1\} > \Pr\{\mathbf{y}_1^N, \hat{\mathbf{u}}_1^{i-1} | u_i = 0\}. \end{cases} \quad (2.27)$$

上式暗含“冻结比特不会出错”。我们通过寻找首个错误SC译码比特的方式，把错误事件进行划分。事件“首个错误SC译码比特是 u_i ”指的是下面这个集合：

$$\begin{aligned} \mathcal{B}_i &= \{(\mathbf{u}_1^N, \mathbf{y}_1^N) | \hat{\mathbf{u}}_i^{i-1} = \mathbf{u}_i^{i-1}, \hat{u}_i \neq u_i\} = \{(\mathbf{u}_1^N, \mathbf{y}_1^N) | \hat{\mathbf{u}}_i^{i-1} = \mathbf{u}_i^{i-1}, u_i \neq \text{SC}(\mathbf{u}_1^{i-1}, \mathbf{y}_1^N)\} \\ &\stackrel{(a)}{\subset} \{(\mathbf{u}_1^N, \mathbf{y}_1^N) | u_i \neq \text{SC}(\mathbf{u}_1^{i-1}, \mathbf{y}_1^N)\} \\ &\stackrel{(b)}{\subset} \{(\mathbf{u}_1^N, \mathbf{y}_1^N) | \Pr\{\mathbf{y}_1^N, \mathbf{u}_1^{i-1} | u_i\} \leq \Pr\{\mathbf{y}_1^N, \mathbf{u}_1^{i-1} | u_i \oplus 1\}\} = \mathcal{E}_i \end{aligned} \quad (2.28)$$

(a)是因为我们去掉了限制条件，集合变大。

(b)就比较隐蔽了，也许你会认为这应该是等号，但事实上如果 $u_i = 0$ ，则有可能出现 $\Pr\{\mathbf{y}_1^N, \mathbf{u}_1^{i-1} | 0\} = \Pr\{\mathbf{y}_1^N, \mathbf{u}_1^{i-1} | 1\}$ 的情况。这时，根据式(2.27)的第二行，我们的判决结果是0，这时判断是正确的。所以集合 \mathcal{E}_i 比集合 $\{(\mathbf{u}_1^N, \mathbf{y}_1^N) | u_i \neq \text{SC}(\mathbf{u}_1^{i-1}, \mathbf{y}_1^N)\}$ 大，因为 \mathcal{E}_i 甚至包括一点点 u_i 译码正确的情况。

我们清楚地认识到对于任意 $1 \leq i, j \leq N, i \neq j$ ，有 $\mathcal{B}_i \cap \mathcal{B}_j = \emptyset$ ，因为首个SC译码错误比特是唯一的。但是，设 $\mathcal{D}_i = \{(\mathbf{u}_1^N, \mathbf{y}_1^N) | u_i \neq \text{SC}(\mathbf{u}_1^{i-1}, \mathbf{y}_1^N)\}$ ，其中的含义是SC译码器得到了天使的施舍，SC译码器已经正确地知道 \mathbf{u}_1^{i-1} 的值，即便如此 u_i 的译码仍然是错误的。那么对于 $1 \leq i, j \leq N, i < j$ ，我们不一定有 $\mathcal{D}_i \cap \mathcal{D}_j = \emptyset$ 。这是因为对于某一个实现值 $(\mathbf{u}_1^N, \mathbf{y}_1^N)$ ，存在这样的情况：把 \mathbf{u}_1^{i-1} 的值告诉SC译码器，结果 u_i 估计错误，把 $\mathbf{u}_1^{j-1}, j > i$ 的值告诉SC译码器，结果 u_j 的估计照样错误，从而 $(\mathbf{u}_1^N, \mathbf{y}_1^N) \in \mathcal{D}_i \cap \mathcal{D}_j$ ，这也就是SC比特翻转译码器中错误阶数大于1的情况。我说这个是想请初学者区分各种情形，清楚地看清表达式的含义。

设所有可能的错误事件为 $\mathcal{E} = \bigcup_{i=1}^N \mathcal{B}_i = \bigcup_{i \in \mathcal{A}} \mathcal{B}_i$ ，其中 \mathcal{A} 表示信息比特的位置，即冻结比特因为不会出错所以不考虑。通过上面的分析我们知道对于每个 i ， $\mathcal{B}_i \subset \mathcal{E}_i$ ，从而 $\mathcal{E} = \bigcup_{i \in \mathcal{A}} \mathcal{B}_i \subset \bigcup_{i \in \mathcal{A}} \mathcal{E}_i$ 。那么我们有：

$$\Pr\{\mathcal{E}\} \leq \Pr\{\bigcup_{i \in \mathcal{A}} \mathcal{E}_i\} \leq \sum_{i \in \mathcal{A}} \Pr\{\mathcal{E}_i\} \quad (2.29)$$

$$\begin{aligned}
\Pr\{\mathcal{E}_i\} &\stackrel{(a)}{=} \sum_{(\mathbf{u}_1^N, \mathbf{y}_1^N)} \Pr\{\mathbf{u}_1^N, \mathbf{y}_1^N\} \mathbb{I}\{(\mathbf{u}_1^N, \mathbf{y}_1^N) \in \mathcal{E}_i\} \\
&\stackrel{(b)}{\leq} \sum_{(\mathbf{u}_1^N, \mathbf{y}_1^N)} \Pr\{\mathbf{u}_1^N, \mathbf{y}_1^N\} \sqrt{\frac{\Pr\{\mathbf{y}_1^N, \mathbf{u}_1^{i-1} | u_i \oplus 1\}}{\Pr\{\mathbf{y}_1^N, \mathbf{u}_1^{i-1} | u_i\}}} \\
&\stackrel{(c)}{=} \sum_{(\mathbf{u}_1^i, \mathbf{y}_1^N)} \Pr\{\mathbf{u}_1^i, \mathbf{y}_1^N\} \sqrt{\frac{\Pr\{\mathbf{y}_1^N, \mathbf{u}_1^{i-1} | u_i \oplus 1\}}{\Pr\{\mathbf{y}_1^N, \mathbf{u}_1^{i-1} | u_i\}}} \\
&= \frac{1}{2} \sum_{(\mathbf{u}_1^i, \mathbf{y}_1^N)} \Pr\{\mathbf{u}_1^{i-1}, \mathbf{y}_1^N | u_i\} \sqrt{\frac{\Pr\{\mathbf{y}_1^N, \mathbf{u}_1^{i-1} | u_i \oplus 1\}}{\Pr\{\mathbf{y}_1^N, \mathbf{u}_1^{i-1} | u_i\}}} \stackrel{(d)}{\leq} Z(W_N^{(i)}).
\end{aligned} \tag{2.30}$$

等号(a)中 \mathbb{I} 表示指示函数，如果 \mathbb{I} 后面的括号里的条件满足，那么 $\mathbb{I} = 1$ ，否则 $\mathbb{I} = 0$ 。 \mathbb{I} 的意思就是挑出来满足 \mathbb{I} 后面括号中的条件的事件，把对应概率求和。

不等号(b)是因为，根据 \mathcal{E}_i 的定义， \mathcal{E}_i 中的每一个元素 $(\mathbf{u}_1^N, \mathbf{y}_1^N)$ 都满足 $\frac{\Pr\{\mathbf{y}_1^N, \hat{\mathbf{u}}_1^{i-1} | u_i \oplus 1\}}{\Pr\{\mathbf{y}_1^N, \hat{\mathbf{u}}_1^{i-1} | u_i\}} \geq 1 \geq \mathbb{I}$ 。

等号(c)是把无关的维数 \mathbf{u}_{i+1}^N 求和，让它们消失。

不等号(d)是把 $\frac{1}{2}$ 去掉，并使用巴氏系数的定义。

从而，我们有：

$$\Pr\{\mathcal{E}\} \leq \sum_{i \in \mathcal{A}} \Pr\{\mathcal{E}_i\} \leq \sum_{i \in \mathcal{A}} Z(W_N^{(i)}). \tag{2.31}$$

下面我们不加证明地引入文献[13]中的定理，因为它证明涉及实分析，比较复杂，但结论易懂。

定理 2.8 设 W 是二进制离散无记忆信道，对称容量为 $I(W)$ 。对任意 $R < I(W), \beta < 0.5$ ，随 $N = 2^n$ 的增大，存在集合序列 $\mathcal{A}_N \subset \{1, 2, \dots, N\}, |\mathcal{A}_N| > NR$ ，使 $\sum_{i \in \mathcal{A}_N} Z(W_N^{(i)}) = o(2^{-N^\beta})$ 。

定义 2.13 极化信道选取规则

当我们在 N 个极化信道中选取 K 个极化信道来传输信息比特时，我们总是选取 K 个具有最小巴氏系数的极化信道。

既然定理2.8告诉我们，对于充分大的 N 和信息比特数 K ，当 $K/N < I(W), \beta < 0.5$ 时，存在 $\mathcal{A}' \subset \{1, 2, \dots, N\}, |\mathcal{A}'| > K$ ，使得 $\sum_{i \in \mathcal{A}'} Z(W_N^{(i)}) = o(2^{-N^\beta})$ 。那么根据极化信道选取规则得到的 K 极化信道（其索引集合记为 \mathcal{A} ）必使得 $\sum_{i \in \mathcal{A}} Z(W_N^{(i)}) = o(2^{-N^\beta})$ ，因为这已经是 $\sum_{i \in \mathcal{A}} Z(W_N^{(i)})$ 取值最小的情况了。

所以，在满足定理2.8条件的情况下，对于根据极化信道选取规则获得的集合 \mathcal{A} ，有：

$$\Pr\{\mathcal{E}\} \leq \sum_{i \in \mathcal{A}} Z(W_N^{(i)}) = o(2^{-N^\beta}). \quad (2.32)$$

由此我们说明了当码长趋于无穷时，极化码能够以不超过 $I(W)$ 的速率进行错误率任意小的传输。

第3章 极化码编码

§ 3.1 极化码构造

码长为 $N = 2^n$ 且信息比特数为 K 的极化码构造，指的是在 N 个极化信道 $W_N^{(i)}$, $1 \leq i \leq N$ 中选取 K 个“最可靠极化信道”，用以传输信息比特。极化信道的可靠性一般来讲由三种指标来衡量，分别是对称容量 $I(W_N^{(i)})$ 、巴氏系数 $Z(W_N^{(i)})$ 和最大似然判决错误率 $P_e^{\text{ML}}(W_N^{(i)})$ ：

$$\begin{aligned} I(W_N^{(i)}) &= \frac{1}{2} \sum_{u_i \in \{0,1\}} \sum_{(\mathbf{y}_1^N, \mathbf{u}_1^{i-1}) \in \mathcal{Y}^N \times \{0,1\}^{i-1}} W_N^{(i)}(\mathbf{y}_1^N, \mathbf{u}_1^{i-1}|u_i) \log_2 \frac{W_N^{(i)}(\mathbf{y}_1^N, \mathbf{u}_1^{i-1}|u_i)}{\sum_{u \in \{0,1\}} W_N^{(i)}(\mathbf{y}_1^N, \mathbf{u}_1^{i-1}|u)}, \\ Z(W_N^{(i)}) &= \sum_{(\mathbf{y}_1^N, \mathbf{u}_1^{i-1}) \in \mathcal{Y}^N \times \{0,1\}^{i-1}} \sqrt{W_N^{(i)}(\mathbf{y}_1^N, \mathbf{u}_1^{i-1}|0) W_N^{(i)}(\mathbf{y}_1^N, \mathbf{u}_1^{i-1}|1)}, \\ P_e^{\text{ML}}(W_N^{(i)}) &= \frac{1}{2} \sum_{(\mathbf{y}_1^N, \mathbf{u}_1^{i-1}) \in \mathcal{Y}^N \times \{0,1\}^{i-1}} \min\{W_N^{(i)}(\mathbf{y}_1^N, \mathbf{u}_1^{i-1}|0), W_N^{(i)}(\mathbf{y}_1^N, \mathbf{u}_1^{i-1}|1)\}. \end{aligned} \tag{3.1}$$

上面三个参数中， $I(W_N^{(i)})$ 越大表示 $W_N^{(i)}$ 越可靠， $Z(W_N^{(i)})$ 和 $P_e^{\text{ML}}(W_N^{(i)})$ 越小表示 $W_N^{(i)}$ 越可靠。从目前存在的算法来看，我们一般不计算 $I(W_N^{(i)})$ 的值，因为它非常难算：不光式子复杂 ($\log_2 x$ 的数值计算是非常缓慢的)，而且在极化过程 $(W_N^{(i)}, W_N^{(i)}) \rightarrow (W_{2N}^{(2i-1)}, W_{2N}^{(2i)})$ 中， $I(W_{2N}^{(2i-1)})$ 和 $I(W_{2N}^{(2i)})$ 的值也没有用 $I(W_N^{(i)})$ 表示的解析的界。利用参数 $Z(W_N^{(i)})$ 和 $P_e^{\text{ML}}(W_N^{(i)})$ 的码构造算法都存在，但除了BEC信道外，我们只能计算 $Z(W_N^{(i)})$ 和 $P_e^{\text{ML}}(W_N^{(i)})$ 的近似值。有时候这种近似在整个计算过程中始终保持为 $Z(W_N^{(i)})$ 和 $P_e^{\text{ML}}(W_N^{(i)})$ 的上界（或下界），有时我们也不知道我们的计算结果比 $Z(W_N^{(i)})$ 和 $P_e^{\text{ML}}(W_N^{(i)})$ 的真实值大还是小，我们只是近似而已。

定义 3.1 $Z(W_N^{(i)})$ 上界算法[3]

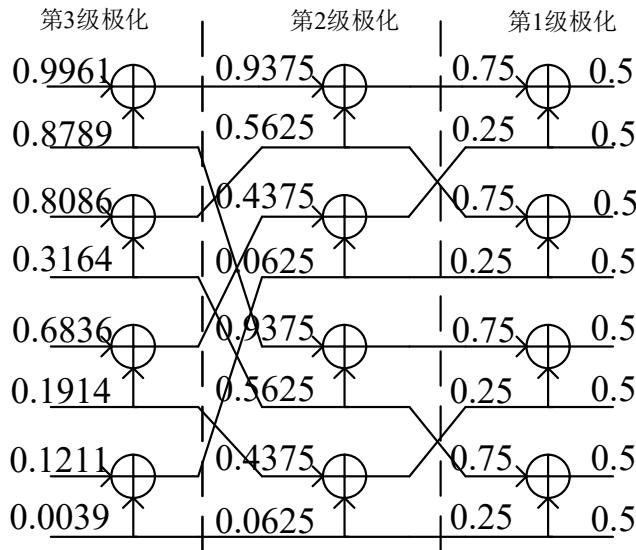


图 3.1 算法3.1的计算过程, 从右向左看

这是恐怕是最简单的构造算法。在上一章的分析中我们知道在极化过程 $(W_N^{(i)}, W_N^{(i)}) \rightarrow (W_{2N}^{(2i-1)}, W_{2N}^{(2i)})$ 中下, 面的式子成立:

$$\begin{aligned} Z(W_{2N}^{(2i-1)}) &\leq 2Z(W_N^{(i)}) - [Z(W_N^{(i)})]^2, \\ Z(W_{2N}^{(2i-1)}) &= [Z(W_N^{(i)})]^2. \end{aligned} \quad (3.2)$$

其中第一个式子里的等号成立当且仅当 $W_N^{(i)}$ 是BEC。但是我们在实用场景下, 不关心BEC信道中的极化码的传输, 所以这个等号一般取不到。但是我们可以指鹿为马, 硬说这个等号总是成立, 那么在 $(W_N^{(i)}, W_N^{(i)}) \rightarrow (W_{2N}^{(2i-1)}, W_{2N}^{(2i)})$ 的过程中的所有计算结果都是闭合的, 所需运算只有乘法和加法, 可以说是非常简单了。下面, 我们假设传输码字比特的信道 $W(y|x)$ 具有 $Z(W) = 0.5$, 那么在本算法下, 当 $N = 8$ 时, 极化码的构造过程如下面的例子所示。

例 3.1 当传输码字比特的信道 $W(y|x)$ 具有 $Z(W) = 0.5$, 码长 $N = 8$ 时, 上述算法的所有计算结果如图3.1所示。如果选择4个极化信道, 那么极化信道索引集合 $\mathcal{A} = \{3, 6, 7, 8\}$, 对应4个具有最小巴氏系数的极化信道。

下面给出一个复杂度是 $O(N)$ 的算法3.1的MATLAB程序。

```

1 function ZWi = get_BEC_ZWi(N, ZW)
2 ZWi = zeros(N, 1);
3 ZWi(1) = ZW;
4 m = 1;
5 while (m <= N/2)
6     for k = 1 : m
7         Z_tmp = ZWi(k);
8         ZWi(k) = 2 * Z_tmp - Z_tmp^2;%use upper bound
9         ZWi(k + m) = Z_tmp^2;
10    end
11    m = m * 2;
12 end
13 ZWi = bitrevorder(ZWi);%perform bit-reversal order.

```

定义 3.2 高斯近似（Gaussian Approximation, GA），来自文献[5]

GA最早是用来在AWGN信道中构造LDPC码的，但是GA对于极化码非常适用，目前许多文献的仿真中都是用GA作为码构造算法。下面我们总是假设信道 $W(y|x)$ 是AWGN信道，极化码长度为 $N = 2^n$ 。

在GA中，假设信源比特是全零比特，那么码字自然也是全零码字。经过BPSK调制 $s_i = 1 - 2x_i, 1 \leq i \leq N$ ，其中 s_i 是第*i*个BPSK符号， x_i 是第*i*个码字比特，那么BPSK发送序列就是全1序列。每个接收BPSK符号是 $y_i = 1 + n_i, 1 \leq i \leq N$ ，其中 n_i 是i.i.d.的加性高斯白噪声，均值为0，方差为 σ^2 。所以 y_i 也是i.i.d.随机变量，且服从高斯分布 $\mathcal{N}(1, \sigma^2)$ 。

在AWGN信道中使用BPSK调制，第*i*个LLR的计算公式为 $L_i = \ln \frac{\frac{1}{\sqrt{2\pi}} e^{-(y_i-1)^2/(2\sigma^2)}}{\frac{1}{\sqrt{2\pi}} e^{-(y_i+1)^2/(2\sigma^2)}} = \frac{2}{\sigma^2} y_i$ 。因为 y_i 服从 $\mathcal{N}(1, \sigma^2)$ ，所以 $\frac{2}{\sigma^2} y_i$ 服从 $\mathcal{N}(\frac{2}{\sigma^2}, \frac{4}{\sigma^4})$ 。我们注意到 $\frac{2}{\sigma^2} y_i$ 的方差是均值的两倍。

GA的思想是：假设在极化过程 $(W_N^{(i)}, W_N^{(i)}) \rightarrow (W_{2N}^{(2i-1)}, W_{2N}^{(2i)})$ 中，三个极化信道 $W_N^{(i)}$ 、 $W_{2N}^{(2i-1)}$ 以及 $W_{2N}^{(2i)}$ 的对数似然比都是“方差为均值两倍的高斯随机变量”。在这种假设下，问题可以描述为“如何用 $W_N^{(i)}$ 中LLR的均值算出 $W_{2N}^{(2i-1)}$ 和 $W_{2N}^{(2i)}$ 中LLR的均值”。

为了解决这个问题，我们首先要算出 $W_{2N}^{(2i-1)}$ 和 $W_{2N}^{(2i)}$ 中LLR的表达式：

$$\begin{aligned}
 L_{2N}^{(2i-1)}(\mathbf{y}_1^{2N}, \mathbf{u}_1^{2i-2}) &= \ln \frac{W_{2N}^{(2i-1)}(\mathbf{y}_1^{2N}, \mathbf{u}_1^{2i-2} | u_{2i-1} = 0)}{W_{2N}^{(2i-1)}(\mathbf{y}_1^{2N}, \mathbf{u}_1^{2i-2} | u_{2i-1} = 1)} \\
 &\stackrel{(a)}{=} \ln \frac{\sum_{u_{2i}} W_N^{(i)}(\mathbf{y}_1^N, \mathbf{u}_{1,o}^{2i-2} \oplus \mathbf{u}_{1,e}^{2i-2} | u_{2i}) W_N^{(i)}(\mathbf{y}_{N+1}^{2N}, \mathbf{u}_{1,e}^{2i-2} | u_{2i})}{\sum_{u_{2i}} W_N^{(i)}(\mathbf{y}_1^N, \mathbf{u}_{1,o}^{2i-2} \oplus \mathbf{u}_{1,e}^{2i-2} | u_{2i} \oplus 1) W_N^{(i)}(\mathbf{y}_{N+1}^{2N}, \mathbf{u}_{1,e}^{2i-2} | u_{2i})} \\
 &\stackrel{(b)}{=} \ln \frac{e^{L_N^{(i)}(\mathbf{y}_1^N, \mathbf{u}_{1,o}^{2i-2} \oplus \mathbf{u}_{1,e}^{2i-2}) + L_N^{(i)}(\mathbf{y}_{N+1}^{2N}, \mathbf{u}_{1,e}^{2i-2})} + 1}{e^{L_N^{(i)}(\mathbf{y}_1^N, \mathbf{u}_{1,o}^{2i-2} \oplus \mathbf{u}_{1,e}^{2i-2})} + e^{L_N^{(i)}(\mathbf{y}_{N+1}^{2N}, \mathbf{u}_{1,e}^{2i-2})}} \\
 &= 2 \tanh^{-1} \left(\tanh \frac{L_N^{(i)}(\mathbf{y}_1^N, \mathbf{u}_{1,o}^{2i-2} \oplus \mathbf{u}_{1,e}^{2i-2})}{2} \tanh \frac{L_N^{(i)}(\mathbf{y}_{N+1}^{2N}, \mathbf{u}_{1,e}^{2i-2})}{2} \right).
 \end{aligned} \tag{3.3}$$

符号 $L_N^{(i)}(\mathbf{y}_1^N, \mathbf{u}_1^{i-1})$ 表示长度为 N 的极化码中第 i 个极化信道的LLR，它的含义再上式中是通用的。等号(a)是极化信道条件概率的递归计算，上一章已经讲过；等号(b)处分子分母同时除以 $W_N^{(i)}(\mathbf{y}_1^N, \mathbf{u}_{1,o}^{2i-2} \oplus \mathbf{u}_{1,e}^{2i-2} | 1) W_N^{(i)}(\mathbf{y}_{N+1}^{2N}, \mathbf{u}_{1,e}^{2i-2} | 1)$ ，并利用LLR的定义可得；等号(c)利用了恒等式 $\ln \frac{e^{x+y} + 1}{e^x + e^y} = 2 \tanh^{-1}(\tanh(x/2) \tanh(y/2))$ 。

由于符号太过复杂，我们下面用 x 表示 $L_N^{(i)}(\mathbf{y}_1^N, \mathbf{u}_{1,o}^{2i-2} \oplus \mathbf{u}_{1,e}^{2i-2})$ ，用 y 表示 $L_N^{(i)}(\mathbf{y}_{N+1}^{2N}, \mathbf{u}_{1,e}^{2i-2})$ ，用 z 表示 $L_{2N}^{(2i-1)}(\mathbf{y}_1^{2N}, \mathbf{u}_1^{2i-2})$ ，那么上式经过简单变形可得：

$$\tanh(z/2) = \tanh(x/2) \tanh(y/2). \tag{3.4}$$

类似地，我们有：

$$\begin{aligned}
 L_{2N}^{(2i)}(\mathbf{y}_1^{2N}, \mathbf{u}_1^{2i-1}) &= \ln \frac{W_{2N}^{(2i)}(\mathbf{y}_1^{2N}, \mathbf{u}_1^{2i-1} | u_{2i} = 0)}{W_{2N}^{(2i)}(\mathbf{y}_1^{2N}, \mathbf{u}_1^{2i-1} | u_{2i} = 1)} \\
 &= (1 - 2u_{2i-1}) L_N^{(i)}(\mathbf{y}_1^N, \mathbf{u}_{1,o}^{2i-2} \oplus \mathbf{u}_{1,e}^{2i-2}) + L_N^{(i)}(\mathbf{y}_{N+1}^{2N}, \mathbf{u}_{1,e}^{2i-2}).
 \end{aligned} \tag{3.5}$$

因为我们假设信源序列全是0，所以 u_{2i} 是0。令 $s = L_{2N}^{(2i)}(\mathbf{y}_1^{2N}, \mathbf{u}_1^{2i-1})$ ，得：

$$s = x + y. \tag{3.6}$$

我们这时注意到 x, y, z, s 都是随机变量，且 x, y 是i.i.d.的，这是由于已经假定 $\mathbf{U}_1^{2N} = \mathbf{0}$ ，且 \mathbf{Y}_1^N 和 \mathbf{Y}_{N+1}^{2N} 独立。对式(3.4)和(3.6)两边取均值，因为 x, y 是i.i.d.的，所以我们统一用 x 的参

数：

$$E\{\tanh(z/2)\} = E\{\tanh(x/2)\}^2. \quad (3.7)$$

$$E\{s\} = 2E\{x\}. \quad (3.8)$$

式(3.7)稍微复杂些，下面把它用期望的定义展开：

$$\int_{-\infty}^{+\infty} \frac{1}{\sqrt{4\pi E\{z\}}} \tanh(z/2) e^{-\frac{(z-E\{z\})^2}{4E\{z\}}} dz = [\int_{-\infty}^{+\infty} \frac{1}{\sqrt{4\pi E\{x\}}} \tanh(x/2) e^{-\frac{(x-E\{x\})^2}{4E\{x\}}} dx]^2. \quad (3.9)$$

等号两边使用了高斯近似假设：所有出现的随机变量都是方差为均值两倍的高斯随机变量。令 $\phi(t) = 1 - \int_{-\infty}^{+\infty} \frac{1}{\sqrt{4\pi t}} \tanh(\alpha/2) e^{-\frac{(\alpha-t)^2}{4t}} d\alpha$ ($\phi(x)$ 事实上是区间 $(0, +\infty)$ 上的单调递减函数， $\phi(0) = 1, \phi(+\infty) = 0$ ，但这个性质的证明太复杂，就略了)，有：

$$E\{z\} = \phi^{-1}\{1 - [1 - \phi(E\{x\})]^2\}. \quad (3.10)$$

式(3.8)和(3.10)就是我们想要的结果，但是(3.10)中函数 ϕ 的计算太复杂，我们用下面的式子近似它：

$$\phi(x) = \begin{cases} e^{-0.4527x^{0.86} + 0.0218}, & 0 < x < 10 \\ \sqrt{\frac{\pi}{x}} e^{-\frac{x}{4}} (1 - \frac{10}{7x}), & x \geq 10. \end{cases} \quad (3.11)$$

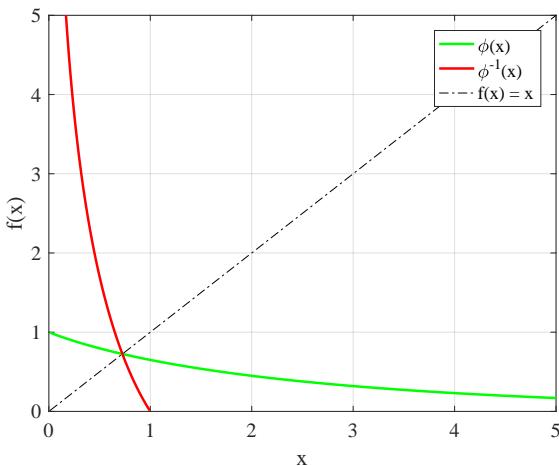
对于上面第一个分段，我们有 $\phi^{-1}(x) = (\frac{0.0218 - \ln x}{0.4527})^{\frac{1}{0.86}}$, $0.3848 < x \leq 1$ 。

对于上面第二个分段，我们可以验证 $\phi(x) = \sqrt{\frac{\pi}{x}} e^{-\frac{x}{4}} (1 - \frac{10}{7x})$, $x \geq 10$ 是单调减函数。欲求 $\phi^{-1}(x) = ?$, 两边同时取 ϕ , 有 $\phi(\phi^{-1}(x)) = x$ 。由此，可用求解 $\phi(\phi^{-1}(x)) - x = 0$ 零点的方式解出 $\phi^{-1}(x)$ 的值。

$\phi(x)$ 和 $\phi^{-1}(x)$ 的图像如图3.2所示。

在极化过程 $(W_M^{(i)}, W_M^{(i)}) \rightarrow (W_{2M}^{(2i-1)}, W_{2M}^{(2i)})$ 中，不断使用式(3.8)和(3.10)，直到计算至预定码字长度 N 。在 $W_N^{(i)}$ 中，选取具有最大的LLR均值的极化信道作为传输信息比特的信道，完成极化码构造。

下面是GA码构造算法的MATLAB代码。

图 3.2 $\phi(x)$ 和 $\phi^{-1}(x)$ 的图像

```

1 function u = GA(sigma, N)
2 u = zeros(1, N);
3 u(1) = 2/sigma^2;
4 for i = 1 : log2(N)
5     j = 2^(i - 1);
6     for k = 1 : j
7         tmp = u(k);
8         u(k) = phi_inverse(1 - (1 - phi(tmp))^2);
9         % You can realize phi(x) and phi_inverse(x) in your own way.
10        sum_iter = sum_iter + num_iter;
11        u(k + j) = 2 * tmp;
12    end
13 end
14 u = bitrevorder(u);
15 end

```

一个GA算法的例子如图3.3所示。在图3.3中，我们想在 $E_b/N_0 = 2.5\text{dB}$ 处，构造极化码 $P(8,4)$ 。我们首先需要计算AWGN信道的噪声方差 $\sigma = \frac{1}{\sqrt{2R}}10^{-\frac{E_b/N_0}{20}} = 0.7499$ ，其中 $R = K/N$ 。然后式(3.8)和(3.10)逐步计算，得到图3.3中的结果。选中的4个极化信道索引为{4, 6, 7, 8}。

文献[6]提出了一对儿极化码构造算法，这两种算法不局限于AWGN信道，对于一般的对称

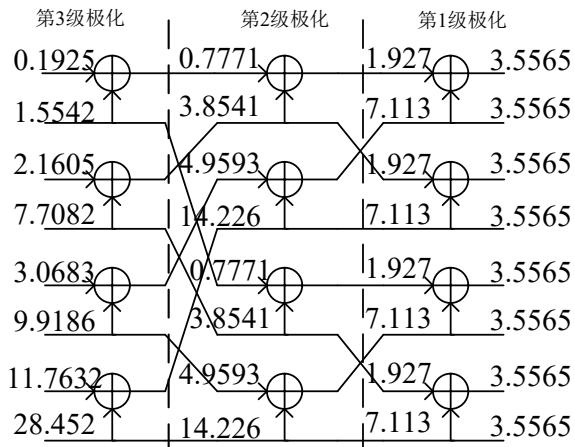


图 3.3 GA 码构造的所有计算数值

信道都适用，离散和连续的对称信道都可以。其中一个是基于信道退化，另一个基于信道提升。这两种算法思想上相同，实现细节上大同小异，性能几乎没有区别。因为基于信道退化的算法实现起来更简单一些，所以这里只讲基于信道退化的极化码构造。下面我会复读文献[6]中的一些结论，这些结论的证明都在文献[6]中，我就不搬运它们的证明了，想看的读者可以直接读文献[6]。

极化码中第*i*个极化信道 $W_N^{(i)}(\mathbf{y}_1^N, \mathbf{u}_1^{i-1} | u_i)$ 的输出符号有 $|\mathcal{Y}|^N \times 2^{i-1}$ 个，其中 $|\mathcal{Y}|$ 是物理信道 $W(y|x)$ 的输出符号数量。极化码构造的主要问题在于 $|\mathcal{Y}|^N \times 2^{i-1}$ 的数值太大，存储不下。文献[6]中信道退化的主要思想是缩减 $W_N^{(i)}(\mathbf{y}_1^N, \mathbf{u}_1^{i-1} | u_i)$ 的输出符号数量，每一次输出符号缩减得到的信道是原信道的退化，即信道容量会减少；同时使用一定的策略，使得每一次信道退化得不那么厉害，尽量保持原信道的容量。

定义 3.3 信道退化极化码构造[6]

设传输码字比特的信道为 $W(y|x)$, $y \in \mathcal{Y}, x \in \{0, 1\}$, W 是对称信道。由 W 进行极化操作，则所得极化信道 $W_N^{(i)}$, $1 \leq i \leq N$ 都是对称信道[3]。

如果 W 是离散的，那么 $W_N^{(i)}$, $1 \leq i \leq N$ 自然是离散的，且每个 $W_N^{(i)}$ 的转移概率都可以抽象地写成下面的形式：

$$W_N^{(i)} = \left[\begin{array}{cccc|ccccc} q_1 & q_2 & \cdots & q_{|\mathcal{Y}|^N \times 2^{i-2}} & p_{|\mathcal{Y}|^N \times 2^{i-2}} & \cdots & p_2 & p_1 \\ p_1 & p_2 & \cdots & p_{|\mathcal{Y}|^N \times 2^{i-2}} & q_{|\mathcal{Y}|^N \times 2^{i-2}} & \cdots & q_2 & q_1 \end{array} \right], \quad (3.12)$$

其中元素的似然比是有序的： $1 < \frac{q_i}{p_i} < \frac{q_{i+1}}{p_{i+1}}$, $1 \leq i \leq |\mathcal{Y}|^N \times 2^{i-2} - 1$ ，设上式中的列 $[\frac{q_i}{p_i}]$ 对应的输出符号叫 y_i ，列 $[\frac{p_i}{q_i}]$ 对应的输出符号叫 \bar{y}_i 。合并输出符号 (y_i, \bar{y}_i) ，指的

是：(a). 在 $W_N^{(i)}$ 中，删除列 $[q_i]$ 和列 $[q_j]$ ，再插入列 $[q_i + q_j]$ ，插入的位置要满足 $\frac{q_k}{p_k} < \frac{q_i + q_j}{p_i + p_j} < \frac{q_{k+1}}{p_{k+1}}$ ，即插入在原第 k 列和原第 $k+1$ 列之间；(b). 在 $W_N^{(i)}$ 中，删除列 $[q_i]$ 和列 $[q_j]$ ，再插入列 $[p_i + p_j]$ ，插入的位置要满足 $\frac{p_s}{q_s} < \frac{p_i + p_j}{q_i + q_j} < \frac{p_{s+1}}{q_{s+1}}$ ，即插入在原第 s 列和原第 $s+1$ 列之间。经过上述操作后，输出符号数量比原信道少了 2 个。

注意到有时 $W_N^{(1)}$ 的输出符号数量不一定是偶数，例如当 W 为 BEC 信道时， $W_N^{(1)}$ 的输出符号数量为 3^N 。这可以通过拆分 W 的删除符号来避免： $W = [\begin{smallmatrix} 1-\epsilon & \epsilon & 0 \\ 0 & \epsilon & 1-\epsilon \end{smallmatrix}]$, $Q = [\begin{smallmatrix} 1-\epsilon & \epsilon/2 & \epsilon/2 & 0 \\ 0 & \epsilon/2 & \epsilon/2 & 1-\epsilon \end{smallmatrix}]$, 显然 $W \preceq Q, Q \preceq W$ ，因此 W 和 Q 没有任何区别。

下面我们叙述文献[6]中的几个结论。

引理 3.1 正数 a, b, c, d, e, f, g, h 满足 $\frac{a}{b} < \frac{c}{d} < \frac{e}{f} < \frac{g}{h}$ ，则 $\frac{a}{b} < \frac{c+e}{d+f} < \frac{g}{h}$ 。

证明 去掉分号易得。

引理 3.2 在式(3.12)中，合并 (y_i, y_j) 会使信道容量减少，且减少的值记为 $\Delta I_{y_i, y_j}$ ：

$$\begin{aligned} \Delta I_{y_i, y_j} = & -(q_i + p_i) \log_2 \frac{q_i + p_i}{2} - (q_j + p_j) \log_2 \frac{q_j + p_j}{2} \\ & + q_i \log_2 q_i + p_i \log_2 p_i + q_j \log_2 q_j + p_j \log_2 p_j \\ & + (q_i + p_i + q_j + p_j) \log_2 \frac{q_i + p_i + q_j + p_j}{2} \\ & - (q_i + q_j) \log_2 (q_i + q_j) - (p_i + p_j) \log_2 (p_i + p_j). \end{aligned} \quad (3.13)$$

证明 由 $I(X; Y) = H(Y) - H(Y|X) = -\sum_i p(y_i) \log_2(p(y_i)) + \sum_i p(y_i|x) \log_2 p(y_i|x)$ 可得。其中，已经假定信道是对称的，输入是均匀分布的。

引理 3.3 在式(3.12)中，考虑输出符号 $y_i, y_j, y_k, i < j < k$ ，那么在对它们进行合并时，互信息的变化量 ΔI 有下面的几个关系成立：

$$\Delta I_{y_i, y_j} = \Delta I_{\bar{y}_i, \bar{y}_j} \leq \Delta I_{\bar{y}_i, y_j} = \Delta I_{y_i, \bar{y}_j}. \quad (3.14)$$

$$\begin{aligned} \Delta I_{y_i, y_j} &\leq \Delta I_{y_i, y_k}, \\ \Delta I_{y_j, y_k} &\leq \Delta I_{y_i, y_k}. \end{aligned} \quad (3.15)$$

式(3.14)说明合并输出符号时不要跨过式(3.12)中间的分块线，不然互信息会损失变大；式(3.15)说明对式(3.12)中的符号合并时，要对下标连续的符号进行合并（通过递推可知），否则互信息损失变大。

证明 这个定理含义清晰，但证明太长，证明过程的有无不影响我们后面的叙述。有兴趣的读者可以阅读文献[6]的附录。

有了上面的几个引理，我们进行如下的“贪心”合并：合并的目的在于减少输出符号的数量，每一步合并减少2个输出符号；贪心的意思是每一步合并都让 ΔI 最小，但不一定是整体最优的合并方式。

信道 $W_N^{(i)}$ 的输出符号有 $|\mathcal{Y}|^N \times 2^{i-1}$ 个，想把这个数减少到一个合理的值，比如 μ 个， μ 的典型值是64左右。那么我们就对式(3.12)进行计算，算出每个 $\Delta I(y_i, y_{i+1})$, $1 \leq i \leq |\mathcal{Y}|^N \times 2^{i-1} - 1$ ，得到 $k = \arg \min_{1 \leq i \leq |\mathcal{Y}|^N \times 2^{i-1} - 1} \{\Delta I(y_i, y_{i+1})\}$ ，合并 (y_k, y_{k+1}) 使输出符号减少2个。由引理3.1可知，新的列 $[\frac{q_k + q_{k+1}}{p_k + p_{k+1}}]$ 的位置还是在原来 k 的位置上，新的列 $[\frac{p_k + p_{k+1}}{q_k + q_{k+1}}]$ 的位置同理可知。重复上面的过程，直到输出符号变成 μ 个，该信道设为 $\underline{W}_N^{(i)}$ 。显然 $\underline{W}_N^{(i)} \preceq W_N^{(i)}$ ，因为每一步合并都是信道退化。那么我们现在可以完整的描述基于信道退化的极化码构造算法了。

- (i). 选定阈值 μ ，初始化 $N = 1$, $W_1^{(1)} = W$ 。
- (ii). 如果 N 等于所需码长，转步骤(iv)；否则，计算 $(W_N^{(i)}, W_N^{(i)}) \rightarrow (W_{2N}^{(2i-1)}, W_{2N}^{(2i)})$ ，令 $N = 2N$ 。
- (iii). 逐一判断 $W_N^{(i)}$, $1 \leq i \leq N$ 的输出符号数量，如果 $W_N^{(i)}$ 没有超过 μ ，不对其进行操作；如果 $W_N^{(i)}$ 超过了 μ ，则对其进行贪心合并，得到 $\underline{W}_N^{(i)}$ 。令 $W_N^{(i)} = \underline{W}_N^{(i)}$ ，转步骤(ii)。
- (iv). 按式(3.1)计算 $P_e^{\text{ML}}(W_N^{(i)})$ ，选取具有最小 $P_e^{\text{ML}}(W_N^{(i)})$ 的信道作为传输信息比特的信道。

注意上面基于信道退化的极化码构造算法仅对 W 是离散对称信道的情况适用。然而很多时候， W 是连续对称信道，例如BPSK-AWGN信道。当 W 是连续对称信道时，首先需要对 W 进行量化，得到一个离散信道。下面以量化BPSK-AWGN信道为例进行说明，该AWGN信道方差为 σ^2 ，BPSK信号取值于 $\{-1, 1\}$ 。

命题 3.1 在BPSK-AWGN信道的接收信号概率密度图上（两个高斯分布： $\mathcal{N}(1, \sigma^2)$ 和 $\mathcal{N}(-1, \sigma^2)$ ），分割正半数轴 $(0, +\infty)$ 为若干 ν 个互不相交的区间 $(0, x_1), (x_1, x_2), \dots, (x_{\nu-1}, +\infty)$ 。

令 $q_i = \frac{1}{\sigma\sqrt{2\pi}} \int_{x_{i-1}}^{x_i} e^{-\frac{(x-1)^2}{2\sigma^2}} dx$, $p_i = \frac{1}{\sigma\sqrt{2\pi}} \int_{x_{i-1}}^{x_i} e^{-\frac{(x+1)^2}{2\sigma^2}} dx$ ，则该BPSK-AWGN信道可以量化为离散信道 W ：

$$W = \left[\begin{array}{cccc|cccc} q_1 & q_2 & \dots & q_\nu & p_\nu & \dots & p_2 & p_1 \\ p_1 & p_2 & \dots & p_\nu & q_\nu & \dots & q_2 & q_1 \end{array} \right], \quad (3.16)$$

且对任意正半轴的分割方法，有 $\frac{q_i}{p_i} < \frac{q_{i+1}}{p_{i+1}}$, $1 \leq i \leq \nu - 1$ 。

证明 首先明确两点：(i). 当 $x > 0$ 时，有 $e^{-\frac{(x-1)^2}{2\sigma^2}} / e^{-\frac{(x+1)^2}{2\sigma^2}} = e^{\frac{4x}{2\sigma^2}}$ 是大于1的增函数。(ii).

对于两个任意的正数列 (a_1, a_2, \dots, a_n) 和 (b_1, b_2, \dots, b_n) , 有 $\min_{1 \leq i \leq N} \{ \frac{a_i}{b_i} \} \leq \frac{\sum_{i=1}^N a_i}{\sum_{i=1}^N b_i} \leq \max_{1 \leq i \leq N} \{ \frac{a_i}{b_i} \}$ 。然后有下式成立:

$$\frac{q_i}{p_i} = \frac{\int_{x_{i-1}}^{x_i} e^{-\frac{(x-1)^2}{2\sigma^2}} dx}{\int_{x_{i-1}}^{x_i} e^{-\frac{(x+1)^2}{2\sigma^2}} dx} \stackrel{(a)}{=} \frac{\sum_k e^{-\frac{(\xi_{k-1})^2}{2\sigma^2}}}{\sum_k e^{-\frac{(\xi_k+1)^2}{2\sigma^2}}} \stackrel{(b)}{\leq} \max_k \left\{ \frac{e^{-\frac{(\xi_{k-1})^2}{2\sigma^2}}}{e^{-\frac{(\xi_k+1)^2}{2\sigma^2}}} \right\} \stackrel{(c)}{=} e^{\frac{4x_i}{2\sigma^2}} \quad (3.17)$$

等号(a)是因为分子分母的积分分割 Δx 取得一样长。每个 Δx 内, 取该区间的右端点 ξ 计算被积函数值。不等号(b)是因为事实(ii), 等号(c)是因为事实(i)。同理有下式成立 (每个 Δx 内取该区间的左端点 ζ 计算被积函数值) :

$$\frac{q_{i+1}}{p_{i+1}} = \frac{\int_{x_i}^{x_{i+1}} e^{-\frac{(x-1)^2}{2\sigma^2}} dx}{\int_{x_i}^{x_{i+1}} e^{-\frac{(x+1)^2}{2\sigma^2}} dx} = \frac{\sum_k e^{-\frac{(\zeta_{k-1})^2}{2\sigma^2}}}{\sum_k e^{-\frac{(\zeta_k+1)^2}{2\sigma^2}}} \geq \min_k \left\{ \frac{e^{-\frac{(\zeta_{k-1})^2}{2\sigma^2}}}{e^{-\frac{(\zeta_k+1)^2}{2\sigma^2}}} \right\} = e^{\frac{4x_i}{2\sigma^2}} \quad (3.18)$$

证毕。

上面的定理说明不管怎么量化BPSK-AWGN信道, 只要按照 q_i 和 p_i 的定义计算, 其比值就是升序的, 如何分割正半轴不是很重要。一种最差的分割方式是干脆不分割, 这时你将得到一个交叉概率为 $\frac{1}{\sigma\sqrt{2\pi}} \int_0^{+\infty} e^{-\frac{(x+1)^2}{2\sigma^2}} dx$ 的BSC信道。文献[6]中给出了一种分割方式, 保证了量化得到的离散信道不会比原来的连续信道“坏太多”, 有兴趣的读者可以去阅读。

现在华为挺厉害了, 讲一个华为的码构造算法。

定义 3.4 极化重量 (Polarization Weight, PW) 算法, 来自文献[7]

我们想构造一个长度是 $N = 2^n$ 的极化码, 任取十进制整数 $0 \leq i \leq N - 1$, 对*i*做二进制展开得到 $< i_n, i_{n-1}, \dots, i_1 >_2$, 其中 i_n 是最高位。则极化信道 $W_N^{(i)}$ 的极化重量PW为:

$$PW_i = \sum_{k=1}^n i_k \beta^{k-1}. \quad (3.19)$$

其中基数 $\beta = 2^{0.25}$ 。获得所有 N 个极化信道 $W_N^{(i)}$ 的PW后, 选取具有最大PW的极化信道传输信息比特。

这个算法看似不太讲理: 它和信道完全无关, 极化码理论里也没有 $\beta = 2^{0.25}$ 这个数。事实上, 基数 $\beta = 2^{0.25}$ 是通过一系列的多项式方程确定的, 同时由基数 β 通过式(3.19)构造的极化码不能和经典构造算法差别过大。另外, 通过PW算法获得长度为 N 的极化码的各个极化信道的极化重量后, 长度 $N/2, N/4, \dots, 2$ 的极化码的极化码重量也一并获得了, 这是因为长度 $N/2^i, i \geq 1$ 的极化码的各个极化信道的PW就是长度为 N 的极化码前 $N/2^i$ 个极化信道的PW。对该算法有兴趣的读者可以阅读参考文献[7]。

除了上面4种码构造算法, 还有一种算法叫蒙特卡罗码构造算法, 我们放到译码器里讲。

§ 3.2 极化码生成矩阵的性质

长度为 N 、信息为长度为 K 的极化码构造，就是从 $W_N^{(i)}, 1 \leq i \leq N$ 中选取 K 个最可靠的极化信道的过程，这些信道的索引构成集合 \mathcal{A} ， $N \times N$ 矩阵 $\mathbf{G}_N = \mathbf{B}_N \mathbf{F}^{\otimes \log_2 N}$ 中挑出集合 \mathcal{A} 对应 K 行，形成 (N, K) 极化码的生成矩阵。本节我们仅分析 $\mathbf{G}_N = \mathbf{B}_N \mathbf{F}^{\otimes \log_2 N}$ 的性质。 $\mathbf{G}_N = \mathbf{B}_N \mathbf{F}^{\otimes \log_2 N}$ 是一个非常古怪的矩阵，它的性质太多了，本章我介绍几个常用的性质。这些性质全都是已经存在的，不是我的贡献。这些性质相对分散，没有明显先后关系，所以本章更像一个公式手册。

本节的结论大多已经非常古老，也说不上证明权归属于谁（反正不属于我），应该就是人类共有的。近些年来的结论我都给了出处。

定理 3.1 $\mathbf{F}^{\otimes n} \mathbf{F}^{\otimes n} = \mathbf{I}_{2^n}$

证明 用归纳法证明。

源头：容易验证 $\mathbf{FF} = \mathbf{I}_2$ 。

归纳假设： $\mathbf{F}^{\otimes k} \mathbf{F}^{\otimes k} = \mathbf{I}_{2^k}$ 。

下一步：

$$\mathbf{F}^{\otimes(k+1)} = \begin{bmatrix} \mathbf{F}^{\otimes k} & \mathbf{O} \\ \mathbf{F}^{\otimes k} & \mathbf{F}^{\otimes k} \end{bmatrix} \quad (3.20)$$

$$\mathbf{F}^{\otimes(k+1)} \mathbf{F}^{\otimes(k+1)} = \begin{bmatrix} \mathbf{F}^{\otimes k} & \mathbf{O} \\ \mathbf{F}^{\otimes k} & \mathbf{F}^{\otimes k} \end{bmatrix} \begin{bmatrix} \mathbf{F}^{\otimes k} & \mathbf{O} \\ \mathbf{F}^{\otimes k} & \mathbf{F}^{\otimes k} \end{bmatrix} = \begin{bmatrix} \mathbf{F}^{\otimes k} \mathbf{F}^{\otimes k} & \mathbf{O} \\ \mathbf{F}^{\otimes k} \mathbf{F}^{\otimes k} + \mathbf{F}^{\otimes k} \mathbf{F}^{\otimes k} & \mathbf{F}^{\otimes k} \mathbf{F}^{\otimes k} \end{bmatrix} \stackrel{(a)}{=} \mathbf{I}_{2^{k+1}}. \quad (3.21)$$

等号(a)使用了归纳假设。

下面我们看一看 $\mathbf{F}^{\otimes n}$ 的子矩阵的零空间的特点。

定理 3.2 设集合 $\mathcal{A} \subset \{1, 2, 3, \dots, 2^n\}$ ， $\mathcal{A}^c = \{1, 2, 3, \dots, 2^n\} \setminus \mathcal{A}$ 。取出 $\mathbf{F}^{\otimes n}$ 中行索引属于 \mathcal{A} 行，形成一个子矩阵，记作 $\mathbf{F}_{\mathcal{A}}$ ；又取出 $\mathbf{F}^{\otimes n}$ 中列索引位于 \mathcal{A}^c 中的列，形成一个子矩阵，记作 $\mathbf{F}_{:, \mathcal{A}^c}$ ，则 $\mathbf{F}_{\mathcal{A}} \mathbf{F}_{:, \mathcal{A}^c} = \mathbf{O}$ ，含义是 $\mathbf{F}_{\mathcal{A}}$ 的零空间是 $\mathbf{F}_{:, \mathcal{A}^c}$ 。

证明 $\mathbf{F}^{\otimes n} \mathbf{F}^{\otimes n} = \mathbf{I}_{2^n}$ 的含义是 $\mathbf{F}^{\otimes n}$ 中的第 i 行只有乘 $\mathbf{F}^{\otimes n}$ 中的第 i 列才不为零。那么显然， $\mathbf{F}_{\mathcal{A}}$ 对应的列一个都不在 $\mathbf{F}_{:, \mathcal{A}^c}$ 之中，证毕。

诚然，用 \mathbf{F} 的克罗尼克幂可以轻便地表达 $\mathbf{F}^{\otimes n}$ ，但是有时候我们也用另一种方式描述它，这就和里德-穆勒（Reed-Muller, RM）码有关系了。之所以谈论RM码，是因为RM码和极化码在一定程度上相似。

定义 3.5 RM 码的 $n + 1$ 个母向量 $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$ 。

设 $N = 2^n$ ，则向量 $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$ 的长度是 N ，他们的分量有如下定义：

$\mathbf{v}_i, 0 \leq i \leq n$ 的第1个元素至第 2^{n-i} 个元素都是1，第 $2^{n-i} + 1$ 个元素至第 2^{n-i+1} 个元素都是0。从第 $2^{n-i+1} + 1$ 个元素开始，不断复制第1个至第 2^{n-i+1} 个元素，直到填满长度 N 。

例 3.2 $n = 4$ 时有：

$$\begin{aligned}\mathbf{v}_0 &= (1111111111111111), \\ \mathbf{v}_1 &= (1111111100000000), \\ \mathbf{v}_2 &= (1111000011110000), \\ \mathbf{v}_3 &= (1100110011001100), \\ \mathbf{v}_4 &= (1010101010101010).\end{aligned}\tag{3.22}$$

定义 3.6 向量逐位乘法

设 α 和 β 是长度位 N 的行向量，则 $\alpha\beta = (\alpha_1\beta_1, \alpha_2\beta_2, \dots, \alpha_N\beta_N)$ 。

定义 3.7 由 $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$ 构成的矩阵

对于整数 $0 \leq k \leq n$ ，那下面所有的行向量摞起来，得到矩阵 \mathbf{R}_k^n 。

$$\begin{aligned}&\mathbf{v}_0, \\ &\mathbf{v}_i, \quad i \in \{1, 2, \dots, n\}, \\ &\mathbf{v}_{i_1}\mathbf{v}_{i_2}, \quad i_1 \neq i_2, i_1, i_2 \in \{1, 2, \dots, n\}, \\ &\mathbf{v}_{i_1}\mathbf{v}_{i_2}\mathbf{v}_{i_3}, \quad i_1, i_2, i_3 \text{互不相等}, \quad i_1, i_2, i_3 \in \{1, 2, \dots, n\}, \\ &\dots \\ &\mathbf{v}_{i_1}\mathbf{v}_{i_2}\dots\mathbf{v}_{i_k}, \quad i_1, i_2, \dots, i_k \text{互不相等}, \quad i_1, i_2, \dots, i_k \in \{1, 2, \dots, n\}.\end{aligned}\tag{3.23}$$

可见, \mathbf{R}_k^n 共有 $K = C_n^0 + C_n^1 + C_n^2 + \dots + C_n^k$ 行。

\mathbf{R}_k^n 就是长度为 2^n , 信息比特比特数为 K 的 RM 码的生成矩阵。

定理 3.3 不记行的排列顺序, 有 $\mathbf{R}_n^n = \mathbf{F}^{\otimes n}$ 。

证明 归纳法。

源头: 当 $n = 1$ 时, 不记行的顺序, 容易验证 $\mathbf{R}_1^1 = \mathbf{F}$ 。

归纳假设: 不记行的顺序, $\mathbf{R}_k^k = \mathbf{F}^{\otimes k}$ 。

下一步: 当 $n = k + 1$ 时, 关键在于用 $n = k$ 时的 $k + 1$ 个母向量 $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_k$ 表示 $n = k + 1$ 时的 $k + 2$ 个母向量 $\nu_0, \nu_1, \nu_2, \dots, \nu_{k+1}$:

$$\nu_0 = (\mathbf{v}_0, \mathbf{v}_0),$$

$$\nu_1 = (\mathbf{v}_0, \mathbf{0}_1^{2^k}),$$

$$\nu_2 = (\mathbf{v}_1, \mathbf{v}_1),$$

(3.24)

$$\nu_3 = (\mathbf{v}_2, \mathbf{v}_2),$$

...

$$\nu_{k+1} = (\mathbf{v}_k, \mathbf{v}_k).$$

其中 $\mathbf{0}_1^{2^k}$ 表示长度为 2^k 的全零向量。用定义 3.5 可以验证上式的正确性。上面所有的向量中, 只有 ν_1 比较特殊: 其一, 只有 ν_1 不能用“并排两个一样的 \mathbf{v} ”的形式表达; 其二, 任何向量和 ν_1 逐位相乘, 计算结果的后一半都会变成零。这启发我们写出 $\mathbf{F}^{\otimes(k+1)}$ 的表达式:

$$\mathbf{F}^{\otimes(k+1)} = \begin{bmatrix} \mathbf{F}^{\otimes k} & \mathbf{O} \\ \mathbf{F}^{\otimes k} & \mathbf{F}^{\otimes k} \end{bmatrix}. \quad (3.25)$$

依据定义 3.7, \mathbf{R}_{k+1}^{k+1} 的行由 $\nu_i, 0 \leq i \leq k + 1$ 所有可能的乘积构成。我们把这些乘积分为两类: 包含 ν_1 , 以及不包含 ν_1 。

第一类：包含 ν_1 的乘积结果。由式(3.24)可见，这些结果明显具有以下形式：

$$(\mathbf{v}_0, \mathbf{0}_1^{2^k})$$

$$(\mathbf{v}_j, \mathbf{0}_1^{2^k}) \quad j \in \{1, 2, \dots, k\},$$

$$(\mathbf{v}_{j_1} \mathbf{v}_{j_2}, \mathbf{0}_1^{2^k}) \quad j_1 \neq j_2, j_1, j_2 \in \{1, 2, \dots, k\},$$

$$(\mathbf{v}_{j_1} \mathbf{v}_{j_2} \mathbf{v}_{j_3}, \mathbf{0}_1^{2^k}) \quad j_1, j_2, j_3 \text{互不相等}, \quad j_1, j_2, j_3 \in \{1, 2, \dots, k\},$$

...

$$(\mathbf{v}_{j_1} \mathbf{v}_{j_2} \dots \mathbf{v}_{j_k}, \mathbf{0}_1^{2^k}) \quad j_1, j_2, \dots, j_k \text{互不相等}, \quad j_1, j_2, \dots, j_k \in \{1, 2, \dots, k\}.$$

注意！由定义3.7，上式的最后一行本应是 $k+1$ 个 ν 连乘，也就是 $k+1$ 个 \mathbf{v} 连乘，但是由于 ν_1 的前一半是 \mathbf{v}_0 （全是1），乘啥也不变，和没乘一样，所以是 k 项连乘。上面的所有式子的个数是 $C_k^0 + C_k^1 + \dots + C_k^k = 2^k$ 。根据归纳假设，上式对应的是 $\mathbf{F}^{\otimes(k+1)}$ 的上半部分，即上式中的左半边的乘积对应的是 $\mathbf{F}^{\otimes k}$ ，右半边的全零分量对应的是零矩阵。

第一类：不包含 ν_1 的乘积结果。由式(3.24)可见，这些结果明显具有以下形式。注意下面 \mathbf{v} 的角标具有一般性。

$$(\mathbf{v}_0, \mathbf{v}_0)$$

$$(\mathbf{v}_j, \mathbf{v}_j) \quad j \in \{1, 2, \dots, k\},$$

$$(\mathbf{v}_{j_1} \mathbf{v}_{j_2}, \mathbf{v}_{j_1} \mathbf{v}_{j_2}) \quad j_1 \neq j_2, j_1, j_2 \in \{1, 2, \dots, k\},$$

$$(\mathbf{v}_{j_1} \mathbf{v}_{j_2} \mathbf{v}_{j_3}, \mathbf{v}_{j_1} \mathbf{v}_{j_2} \mathbf{v}_{j_3}) \quad j_1, j_2, j_3 \text{互不相等}, \quad j_1, j_2, j_3 \in \{1, 2, \dots, k\},$$

...

$$(\mathbf{v}_{j_1} \mathbf{v}_{j_2} \dots \mathbf{v}_{j_k}, \mathbf{v}_{j_1} \mathbf{v}_{j_2} \dots \mathbf{v}_{j_k}) \quad j_1, j_2, \dots, j_k \text{互不相等}, \quad j_1, j_2, \dots, j_k \in \{1, 2, \dots, k\}.$$

注意！由定义3.7，上式的最后一行本应是 $k+1$ 个 ν 连乘，但是现在 ν_1 已经不参与计算且 $n = k+1$ ，所以最多选出 k 项连乘。上面的所有式子的个数是 $C_k^0 + C_k^1 + \dots + C_k^k = 2^k$ 。根据归纳假设，上式对应的是 $\mathbf{F}^{\otimes(k+1)}$ 的下半部分，即上式中的左、右半边的分量完全相同，所有左半边分量摞起来构成 $\mathbf{F}^{\otimes(k+1)}$ 下半部分的一个 $\mathbf{F}^{\otimes k}$ ，所有右半边分量摞起来构成 $\mathbf{F}^{\otimes(k+1)}$ 下半部分的另一个 $\mathbf{F}^{\otimes k}$ 。证毕。

推论 3.1 $\text{rank}(\mathbf{R}_k^n) = C_n^0 + C_n^1 + C_n^2 + \dots + C_n^k$ 。

证明 由定理3.3, 不记行的顺序, 有 $\mathbf{R}_n^n = \mathbf{F}^{\otimes n}$ 。因为 $\mathbf{F}^{\otimes n}$ 可逆, 所以 \mathbf{R}_n^n 可逆, 而 \mathbf{R}_k^n 只不过是由 \mathbf{R}_n^n 中某些行构成的矩阵, 所以 \mathbf{R}_k^n 行满秩, 秩就是行数。

利用定义3.3中 ν 和 \mathbf{v} 间的关系, 我们可以得到下面看似简单但又不是很容易说清楚的引理。

引理3.4 设 $n \geq 2$, 对于任意 $i_1, i_2, i_3, \dots, i_k \in \{1, 2, 3, \dots, n\}$ 且 $i_1, i_2, i_3, \dots, i_k$ 互不相等, 则 $\mathbf{v}_{i_1}\mathbf{v}_{i_2}\dots\mathbf{v}_{i_k}$ 中1的个数是 $\mathbf{v}_{i_1}\mathbf{v}_{i_2}\dots\mathbf{v}_{i_{k-1}}$ 中1的个数的一半。注意这里一直不考虑 \mathbf{v}_0 , 因为 \mathbf{v}_0 的分量全是1, 乘 \mathbf{v}_0 等于什么也发生。

证明 对 n 用归纳法。

源头: 当 $n = 2$, $\mathbf{v}_1 = (1100)$, $\mathbf{v}_2 = (1010)$, $\mathbf{v}_1\mathbf{v}_2 = (1000)$, 定理成立。

归纳假设: 当 $n = k$ 时, $i_1, i_2, i_3, \dots, i_j \in \{1, 2, 3, \dots, k\}$, 且 $i_1, i_2, i_3, \dots, i_j$ 互不相等, 则 $\mathbf{v}_{i_1}\mathbf{v}_{i_2}\dots\mathbf{v}_{i_j}$ 中1的个数是 $\mathbf{v}_{i_1}\mathbf{v}_{i_2}\dots\mathbf{v}_{i_{j-1}}$ 中1的个数的一半。

下一步: 当 $n = k + 1$ 时, 不考虑 \mathbf{v}_0 , 再次用 $n = k$ 时的母向量 $\{\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_k\}$ 表示 $n = k + 1$ 时的母向量 $\{\nu_1, \nu_2, \dots, \nu_{k+1}\}$:

$$\begin{aligned} \nu_1 &= (\mathbf{v}_0, \mathbf{0}_1^{2^k}), \\ \nu_2 &= (\mathbf{v}_1, \mathbf{v}_1), \\ \nu_3 &= (\mathbf{v}_2, \mathbf{v}_2), \\ &\dots, \\ \nu_{k+1} &= (\mathbf{v}_k, \mathbf{v}_k). \end{aligned} \tag{3.28}$$

这样, 对于任意 $i_1, i_2, i_3, \dots, i_{j-1} \in \{1, 2, 3, \dots, k+1\}$ 且 $i_1, i_2, i_3, \dots, i_{j-1}$ 互不相等, 有乘积 $\nu_{i_1}\nu_{i_2}\dots\nu_{i_{j-1}}$ 。下面分两种情况讨论。

情况1: 乘积 $\nu_{i_1}\nu_{i_2}\dots\nu_{i_{j-1}}$ 包括 ν_1 。因为我们定义的向量乘法有交换律, 所以不妨假设 $\nu_{i_1} = \nu_1$ 。由式(3.28), 我们容易写出 $\nu_1\nu_{i_2}\dots\nu_{i_{j-1}}$ 的形式:

$$\nu_1\nu_{i_2}\dots\nu_{i_{j-1}} = (\mathbf{v}_{i_2-1}\dots\mathbf{v}_{i_{j-1}-1}, \mathbf{0}_1^{2^k}). \tag{3.29}$$

上式中 \mathbf{v} 的角标比对应的 ν 的角标少1, 是由于式(3.28)中的角标关系。给 $\nu_1\nu_{i_2}\dots\nu_{i_{j-1}}$ 再乘一个 ν_{i_j} , $i_j \neq 0, 1$, $i_j \neq 0$ 是因为我们从来不考虑 \mathbf{v}_0 , $i_j \neq 1$ 是因为在目前的情况下乘积 $\nu_1\nu_{i_2}\dots\nu_{i_{j-1}}$ 已经包括 ν_1 。我们得到下面的结果:

$$\nu_1\nu_{i_2}\dots\nu_{i_{j-1}}\nu_{i_j} = (\mathbf{v}_{i_2-1}\dots\mathbf{v}_{i_{j-1}-1}\mathbf{v}_{i_j-1}, \mathbf{0}_1^{2^k}). \tag{3.30}$$

对比式(3.29)和(3.30)发现两个式子中的右半边一直是零，不影响1的数量；左半边满足归纳假设。所以情况1证毕。

情况2：乘积 $\nu_{i_1}\nu_{i_2} \dots \nu_{i_{j-1}}$ 不包括 ν_1 。由式(3.28)，我们容易写出 $\nu_{i_1}\nu_{i_2} \dots \nu_{i_{j-1}}$ 的形式：

$$\nu_{i_1}\nu_{i_2} \dots \nu_{i_{j-1}} = (\mathbf{v}_{i_1-1} \dots \mathbf{v}_{i_{j-1}-1}, \mathbf{v}_{i_1-1} \dots \mathbf{v}_{i_{j-1}-1}). \quad (3.31)$$

那么再给上面的式子乘一个 ν_{i_j} , $i_j \neq 0$ 。如果 $\nu_{i_j} = \nu_1$, 则式(3.31)变成了：

$$\nu_{i_1}\nu_{i_2} \dots \nu_{i_{j-1}}\nu_{i_j} = (\mathbf{v}_{i_1-1} \dots \mathbf{v}_{i_{j-1}-1} \mathbf{v}_0, \mathbf{v}_{i_1-1} \dots \mathbf{v}_{i_{j-1}-1} \mathbf{0}_1^{2^k}) = (\mathbf{v}_{i_1-1} \dots \mathbf{v}_{i_{j-1}-1}, \mathbf{0}_1^{2^k}). \quad (3.32)$$

显然，式(3.32)中1的数量是式(3.31)中1的数量的一半。

如果 $\nu_{i_j} \neq \nu_1$, 则式(3.31)变成了：

$$\nu_{i_1}\nu_{i_2} \dots \nu_{i_{j-1}}\nu_{i_j} = (\mathbf{v}_{i_1-1} \dots \mathbf{v}_{i_{j-1}-1} \mathbf{v}_{i_j-1}, \mathbf{v}_{i_1-1} \dots \mathbf{v}_{i_{j-1}-1} \mathbf{v}_{i_j-1}). \quad (3.33)$$

由归纳假设，显然式(3.33)前一半中1的数量是式(3.31)中前一半中1的数量的一半，式(3.33)后一半中1的数量是式(3.31)后一半中1的数量的一半。证毕。

引理 3.5 \mathbf{v}_0 乘任意一个 \mathbf{v}_i , $i \in \{1, 2, \dots, n\}$, 所得结果中的1的数量是 2^{n-1} 。

证明 由 \mathbf{v} 的定义可得。

定理 3.4 $\mathbf{R}_k^n(\mathbf{R}_{n-k-1}^n)^T = \mathbf{O}$, 即以 \mathbf{R}_k^n 为生成矩阵的RM码和以 \mathbf{R}_{n-k-1}^n 为生成矩阵的RM码是对偶(Dual)码。

证明 等价于证明 \mathbf{R}_k^n 的任意一行与 \mathbf{R}_{n-k-1}^n 中的任意的一行的内积是0。

任意取 \mathbf{R}_k^n 中的一行 $\mathbf{r}_1 = \mathbf{v}_{i_1} \mathbf{v}_{i_2} \dots \mathbf{v}_{i_s}$, $1 \leq s \leq k$ 。

任意取 \mathbf{R}_{n-k-1}^n 中的一行 $\mathbf{r}_2 = \mathbf{v}_{j_1} \mathbf{v}_{j_2} \dots \mathbf{v}_{j_t}$, $1 \leq t \leq n-k-1$ 。

记 $\gamma = \mathbf{r}_1 \mathbf{r}_2 = \mathbf{v}_{i_1} \mathbf{v}_{i_2} \dots \mathbf{v}_{i_s} \mathbf{v}_{j_1} \mathbf{v}_{j_2} \dots \mathbf{v}_{j_t}$, 但是这所有 $n-1$ 项乘积中可能有重复的项, 因为乘重复的项完全没有任何影响, 所以我们忽略重复的项, 得到 $\gamma = \mathbf{v}_{p_1} \mathbf{v}_{p_2} \dots \mathbf{v}_{p_q}$, $1 \leq q \leq n-1$, $q=1$ 的意思是至少有一项, 这是肯定的; $q=n-1$ 当且仅当 $s=k, t=n-k-1$ 且集合 $\{i_1, i_2, \dots, i_s\}$ 和 $\{j_1, j_2, \dots, j_t\}$ 中没有重复的元素。

不管 $\gamma = \mathbf{v}_{p_1} \mathbf{v}_{p_2} \dots \mathbf{v}_{p_q}$ 中包不包括 \mathbf{v}_0 , 我们都给他凑上一个 \mathbf{v}_0 , 得到 $\gamma = \mathbf{v}_0 \mathbf{v}_{p_1} \mathbf{v}_{p_2} \dots \mathbf{v}_{p_q}$ 。由引理3.4和3.5可知, 从 \mathbf{v}_0 后, 每乘一个 \mathbf{v} 都导致1的数量变成一半, 那么 γ 中的1的个数是 $2^n/2^q = 2^{n-q} \geq 2$, 也就是说 γ 中1的数量是2的幂, 自然也是偶数: $\sum_{i=1}^{2^n} \gamma_i = 0$, 这就是 \mathbf{r}_1 和 \mathbf{r}_2 的内积。证毕。

定理 3.5 如果 $k < n$, 则:

$$\mathbf{R}_k^n = \begin{bmatrix} \mathbf{R}_{k-1}^{n-1} & \mathbf{O} \\ \mathbf{R}_k^{n-1} & \mathbf{R}_k^{n-1} \end{bmatrix}. \quad (3.34)$$

证明 还是一样的套路, 利用参数对 (n, k) 时的母向量 $(\nu_0, \nu_1, \dots, \nu_n)$, 把 \mathbf{R}_k^n 的每一行都写出来:

$$\nu_0,$$

$$\nu_i, \quad i \in \{1, 2, \dots, n\},$$

$$\nu_{i_1} \nu_{i_2}, \quad i_1 \neq i_2, i_1, i_2 \in \{1, 2, \dots, n\}, \quad (3.35)$$

$$\dots,$$

$$\nu_{i_1} \nu_{i_2} \dots \nu_{i_k}, \quad i_1, i_2, \dots, i_k \text{互不相等}, \quad i_1, i_2, \dots, i_k \in \{1, 2, \dots, n\}.$$

利用式(3.24), 可以把上式用参数对 $(n - 1, k)$ 时的母向量 $(\mathbf{v}_0, \mathbf{v}_1, \dots, \mathbf{v}_{n-1})$ 表示。注意下面式子中下角标具有一般性。

$$(\mathbf{v}_0, \mathbf{v}_0)$$

$$(\mathbf{v}_0, \mathbf{0}_1^{2^{n-1}}), (\mathbf{v}_j, \mathbf{v}_j) \quad j \in \{1, 2, \dots, n - 1\},$$

$$(\mathbf{v}_{j_1} \mathbf{v}_{j_2}, \mathbf{v}_{j_1} \mathbf{v}_{j_2}) \quad j_1 \neq j_2, j_1, j_2 \in \{1, 2, \dots, n - 1\}, \quad (3.36)$$

...

$$(\mathbf{v}_{j_1} \mathbf{v}_{j_2} \dots \mathbf{v}_{j_k}, \mathbf{v}_{j_1} \mathbf{v}_{j_2} \dots \mathbf{v}_{j_k}) \quad j_1, j_2, \dots, j_k \text{互不相等}, \quad j_1, j_2, \dots, j_k \in \{1, 2, \dots, n - 1\}.$$

其中最特殊的还是 $\nu_1 = (\mathbf{v}_0, \mathbf{0}_1^{2^{n-1}})$ 。在式(3.36)中挑出乘积中带有 ν_1 的项：

$$(\mathbf{v}_0, \mathbf{0}_1^{2^{n-1}})$$

$$(\mathbf{v}_j, \mathbf{0}_1^{2^{n-1}}) \quad j \in \{1, 2, \dots, n-1\},$$

$$(\mathbf{v}_{j_1} \mathbf{v}_{j_2}, \mathbf{0}_1^{2^{n-1}}) \quad j_1 \neq j_2, j_1, j_2 \in \{1, 2, \dots, n-1\}, \quad (3.37)$$

...

$$(\mathbf{v}_{j_1} \mathbf{v}_{j_2} \dots \mathbf{v}_{j_{k-1}}, \mathbf{0}_1^{2^{n-1}}) \quad j_1, j_2, \dots, j_{k-1} \text{互不相等}, \quad j_1, j_2, \dots, j_k \in \{1, 2, \dots, n-1\}.$$

注意！上式中的最后一行之所以少了一项乘积（式(3.36)的最后一行有 k 项连乘），是因为事实上已经乘了 ν_1 的前一半，而 ν_1 的前一半全是1，和没乘一样。由定义3.7，上式所有式子的前半部分构成了 \mathbf{R}_{k-1}^{n-1} ，后半部分构成零矩阵。现在我们已经算出了式(3.34)的上半部分。

然后，在式(3.36)中挑出乘积中没有 ν_1 的项：

$$(\mathbf{v}_0, \mathbf{v}_0)$$

$$(\mathbf{v}_j, \mathbf{v}_j) \quad j \in \{1, 2, \dots, n-1\},$$

$$(\mathbf{v}_{j_1} \mathbf{v}_{j_2}, \mathbf{v}_{j_1} \mathbf{v}_{j_2}) \quad j_1 \neq j_2, j_1, j_2 \in \{1, 2, \dots, n-1\}, \quad (3.38)$$

...

$$(\mathbf{v}_{j_1} \mathbf{v}_{j_2} \dots \mathbf{v}_{j_k}, \mathbf{v}_{j_1} \mathbf{v}_{j_2} \dots \mathbf{v}_{j_k}) \quad j_1, j_2, \dots, j_k \text{互不相等}, \quad j_1, j_2, \dots, j_k \in \{1, 2, \dots, n-1\}.$$

注意！尽管 ν_1 已经不参与计算，但是我们仍能选出 k 项 ν 作连乘，这是因为 $k < n$ 。上式所有式子的前半部分和后半部分完全一样。由定义3.7，上式所有式子的前半部分和后半部分构成了 \mathbf{R}_k^{n-1} 。现在我们已经算出了式(3.34)的下半部分。证毕。

定理3.34还有更明确的物理意义。我们考虑三个RM码，它们三个分别以 $\mathbf{R}_k^n, \mathbf{R}_k^{n-1}, \mathbf{R}_{k-1}^{n-1}$ 为生成矩阵。那么根据定理3.34，我们有下面的式子：

$$\mathbf{x}_1^{2^n} = (\alpha_1^{\sum_{i=0}^{k-1} C_{n-1}^i}, \beta_1^{\sum_{i=0}^k C_{n-1}^i}) = \begin{bmatrix} \mathbf{R}_{k-1}^{n-1} & \mathbf{O} \\ \mathbf{R}_k^{n-1} & \mathbf{R}_k^{n-1} \end{bmatrix} \quad (3.39)$$

$$= (\alpha_1^{\sum_{i=0}^{k-1} C_{n-1}^i} \mathbf{R}_{k-1}^{n-1} + \beta_1^{\sum_{i=0}^k C_{n-1}^i} \mathbf{R}_k^{n-1}, \beta_1^{\sum_{i=0}^k C_{n-1}^i} \mathbf{R}_k^{n-1}).$$

其中 $\alpha_1^{\sum_{i=0}^{k-1} C_{n-1}^i} \in \{0, 1\}^{\sum_{i=0}^{k-1} C_{n-1}^i}$ 和 $\beta_1^{\sum_{i=0}^k C_{n-1}^i} \in \{0, 1\}^{\sum_{i=0}^k C_{n-1}^i}$ 是自由的信源变量。可

以看到，码字比特 $\mathbf{x}_1^{2^n}$ 由两部分构成，分别是 $\alpha_1^{\sum_{i=0}^{k-1} C_{n-1}^i} \mathbf{R}_{k-1}^{n-1} + \beta_1^{\sum_{i=0}^k C_{n-1}^i} \mathbf{R}_k^{n-1} = \mathbf{a} + \mathbf{b}$ 和 $\beta_1^{\sum_{i=0}^k C_{n-1}^i} \mathbf{R}_k^{n-1} = \mathbf{b}$ 。设以 \mathbf{R}_k^n 为生成矩阵的RM为 \mathcal{C}_1 ，以 \mathbf{R}_k^{n-1} 为生成矩阵的RM为 \mathcal{C}_2 ，以 \mathbf{R}_{k-1}^{n-1} 为生成矩阵的RM为 \mathcal{C}_3 ，则存在这样的关系： $\mathbf{c} \in \mathcal{C}_1, \mathbf{b} \in \mathcal{C}_2, \mathbf{a} \in \mathcal{C}_3, \mathbf{c} = (\mathbf{a} + \mathbf{b}|\mathbf{b})$ ，其中竖线“—”是就是个分块符号。由此引入下面的定义。

定义 3.8 Bar构造

设 \mathcal{C}_1 和 \mathcal{C}_2 是两个长度相等的线性分组码。取 $\mathbf{a} \in \mathcal{C}_1, \mathbf{b} \in \mathcal{C}_2$ ，记 $\mathbf{c} = (\mathbf{a} + \mathbf{b}|\mathbf{b})$ 。显然，所有 \mathbf{c} 构成了一个线性分组码，记为码字 \mathcal{C}_3 。称 \mathcal{C}_3 是由 \mathcal{C}_1 和 \mathcal{C}_2 通过“Bar构造”得到的，有时这种构造也记为 $\mathcal{C}_3 = (\mathcal{C}_1 + \mathcal{C}_2|\mathcal{C}_2)$ 。

引理 3.6 $\mathcal{C}_3 = (\mathcal{C}_1 + \mathcal{C}_2|\mathcal{C}_2)$ ，码字 \mathcal{C}_1 的生成矩阵的秩是 r_1 ，该生成矩阵的一组基记为 $\alpha_1, \dots, \alpha_{r_1}$ ，码字 \mathcal{C}_2 的生成矩阵的秩是 r_2 ，该生成矩阵的一组基记为 $\beta_1, \dots, \beta_{r_2}$ 那么码字 \mathcal{C}_3 的生成矩阵的秩是 r_1+r_2 ，该生成矩阵的一组基可以表示为 $(\alpha_1|\mathbf{0}), \dots, (\alpha_{r_1}|\mathbf{0}), (\beta_1|\beta_1), \dots, (\beta_{r_2}|\beta_{r_2})$ 。

证明 直接验证即可。首先，你要验证 $(\alpha_1|\mathbf{0}), \dots, (\alpha_{r_1}|\mathbf{0}), (\beta_1|\beta_1), \dots, (\beta_{r_2}|\beta_{r_2})$ 之间线性无关；其次，你要验证 \mathcal{C}_3 中任意一个元素都能用 $(\alpha_1|\mathbf{0}), \dots, (\alpha_{r_1}|\mathbf{0}), (\beta_1|\beta_1), \dots, (\beta_{r_2}|\beta_{r_2})$ 表示。

引理 3.7 $\mathcal{C}_3 = (\mathcal{C}_1 + \mathcal{C}_2|\mathcal{C}_2)$ ，则 $d_{\min}(\mathcal{C}_3) = \min\{d_{\min}(\mathcal{C}_1), 2d_{\min}(\mathcal{C}_2)\}$ ，其中 $d_{\min}(*)$ 表示一个码字的最小汉明重量。这个引理说明了 $d_{\min}(\mathcal{C}_3)$ 不会同时小于 $d_{\min}(\mathcal{C}_1)$ 和 $2d_{\min}(\mathcal{C}_2)$ 。

证明 用夹逼方法。下面我们用符号 $wt(*)$ 表示一个二进制向量的汉明重量。

1. 对于 $\alpha \in \mathcal{C}_1$ 且 $wt(\alpha) = d_{\min}(\mathcal{C}_1)$ ， \mathcal{C}_3 中存在码字 $\gamma_1 = (\alpha|\mathbf{0})$ ，所以 $d_{\min}(\mathcal{C}_3) \leq wt(\gamma_1) = d_{\min}(\mathcal{C}_1)$ 。对于 $\beta \in \mathcal{C}_2$ 且 $wt(\beta) = d_{\min}(\mathcal{C}_2)$ ， \mathcal{C}_3 中存在码字 $\gamma_2 = (\beta|\beta)$ ，所以 $d_{\min}(\mathcal{C}_3) \leq wt(\gamma_2) = 2d_{\min}(\mathcal{C}_2)$ 。综上， $d_{\min}(\mathcal{C}_3) \leq \min\{d_{\min}(\mathcal{C}_1), 2d_{\min}(\mathcal{C}_2)\}$ 。

2. 任取 $(\alpha + \beta|\beta) \in \mathcal{C}_3$ ，其中 $\alpha \in \mathcal{C}_1, \beta \in \mathcal{C}_2$ 。如果 $\alpha \neq \mathbf{0}$ ，则有：

$$wt(\alpha + \beta|\beta) \stackrel{(a)}{=} wt(\alpha + \beta) + wt(\beta) \stackrel{(b)}{\geq} wt(\alpha + \beta + \beta) = wt(\alpha) \stackrel{(c)}{\geq} d_{\min}(\mathcal{C}_1). \quad (3.40)$$

其中等号(a)是只不过是把向量分成两截分段计算，(b)取等号当且仅当 $\alpha + \beta$ 和 β 的支集不相交，(c)是因为 $\alpha \neq \mathbf{0}$ 。

如果 $\alpha = \mathbf{0}$ （注意这时候 β 不能再为 $\mathbf{0}$ 了，不然成全 $\mathbf{0}$ 码字了）：

$$wt(\alpha + \beta|\beta) = wt(\beta|\beta) = 2wt(\beta) \geq 2d_{\min}(\mathcal{C}_2). \quad (3.41)$$

由上面两个式子可得 $d_{\min}(\mathcal{C}_3) \geq d_{\min}(\mathcal{C}_1)$ 和 $2d_{\min}(\mathcal{C}_2)$ ，从而 $d_{\min}(\mathcal{C}_3) \geq \min\{d_{\min}(\mathcal{C}_1), 2d_{\min}(\mathcal{C}_2)\}$ 。综上， $d_{\min}(\mathcal{C}_3) = \min\{d_{\min}(\mathcal{C}_1), 2d_{\min}(\mathcal{C}_2)\}$ 。

下面是一个经典定理。

定理 3.6 以 \mathbf{R}_k^n 为生成矩阵的 RM 码的最小汉明距离是 2^{n-k} 。

证明 对 n 用归纳法。

源头: $n = 1, k = 0$ 时, \mathbf{R}_0^1 的所有码字为 $(11), (00)$, 此时最小汉明距离是 $2 = 2^{1-0}$, 定理成立; $n = 1, k = 1$ 时, \mathbf{R}_1^1 的所有码字为 $(10), (11), (00), (01)$, 此时最小汉明距离是 $1 = 2^{1-1}$, 定理成立。

归纳假设($n-1$): 以 \mathbf{R}_k^{n-1} 为生成矩阵的 RM 码的最小汉明距离是 2^{n-1-k} 。

下一步(n): 以 \mathbf{R}_0^n 为生成矩阵的 RM 码只有两个码字, 全1码字和全0码字, 所以其最小汉明距离是 $2^n = 2^{n-0}$, 定理结论成立。当 $1 \leq p \leq n-1$, 记 \mathbf{R}_{p-1}^{n-1} 为码字 \mathcal{C}_1 , \mathbf{R}_p^{n-1} 为码字 \mathcal{C}_2 , \mathbf{R}_p^n 为码字 \mathcal{C}_3 。由式(3.34) 和定义 3.8, 有 $\mathcal{C}_3 = (\mathcal{C}_1 + \mathcal{C}_2 | \mathcal{C}_2)$ 。再由引理 3.7, 有 $d_{\min}(\mathcal{C}_3) = \min\{d_{\min}(\mathcal{C}_1), 2d_{\min}(\mathcal{C}_2)\} \stackrel{(a)}{=} \min\{2^{n-1-p+1}, 2 \times 2^{n-1-p}\} = 2^{n-p}$, 定理结论成立, 其中等号(a)是归纳假设。当 $p = n$, \mathbf{R}_n^n 是可逆矩阵, 通过高斯消去可得单位矩阵, 所以最小汉明距离为 $1 = 2^{n-n}$, 定理结论成立。证毕。

通过上面冗长的叙述, 我们终于要得到极化码的最小汉明重量的特点了。

定理 3.7 设集合 $\mathcal{A} \subset \{1, 2, 3, \dots, N = 2^n\}$, 取出 $\mathbf{F}^{\otimes n}$ 中行号在 \mathcal{A} 中的行, 得到 $\mathbf{F}^{\otimes n}$ 的子矩阵 $\mathbf{F}_{\mathcal{A}}^{\otimes n}$ 。逐一计算 $\mathbf{F}_{\mathcal{A}}^{\otimes n}$ 中每一行的汉明重量, 得到 $\mathbf{F}_{\mathcal{A}}^{\otimes n}$ 的最小行重是 2^k , 以 $\mathbf{F}_{\mathcal{A}}^{\otimes n}$ 为生成矩阵的线性分组码记为 \mathcal{C} , 则 $d_{\min}(\mathcal{C}) = 2^k$ 。

证明 显然现在我们有 $d_{\min}(\mathcal{C}) \leq 2^k$, 下面我们说明 $\mathbf{F}_{\mathcal{A}}^{\otimes n}$ 是 \mathbf{R}_{n-k}^n 的子矩阵。

依据引理 3.4 和 3.5, 在向量 \mathbf{v} 的连乘中每多乘 1 个不是 \mathbf{v}_0 的 \mathbf{v} 都会使得 1 的数量减半。既然 $\mathbf{F}_{\mathcal{A}}^{\otimes n}$ 的最小行重是 2^k , 那么 $\mathbf{F}_{\mathcal{A}}^{\otimes n}$ 中的行最多是 $n-k$ 个 \mathbf{v} 的连乘: $\mathbf{v}_{i_1} \mathbf{v}_{i_2} \dots \mathbf{v}_{i_{n-k}}$, 否则将出现最小重量小于 2^k 的行, 与题设矛盾。根据定义 3.7, \mathbf{R}_{n-k}^n 的行包含了所有 $\mathbf{v}_{i_1} \mathbf{v}_{i_2} \dots \mathbf{v}_{i_{n-k}}$, 以及乘积项数更少的 \mathbf{v} 的连乘, 所以 $\mathbf{F}_{\mathcal{A}}^{\otimes n}$ 是 \mathbf{R}_{n-k}^n 的子矩阵, $\mathbf{F}_{\mathcal{A}}^{\otimes n}$ 在最小行重为 2^k 时最多和 \mathbf{R}_{n-k}^n 完全一样。所以 $d_{\min}(\mathcal{C}) \stackrel{(a)}{\geq} d_{\min}(\mathcal{C}_1) \stackrel{(b)}{=} 2^k$, 其中 \mathcal{C}_1 是以 \mathbf{R}_{n-k}^n 为生成矩阵的 RM 码, 等号(a)是因为子矩阵形成的码字的最小汉明重量大于等于原码字, 等号(c)是由于定理 3.6。

综上, $d_{\min}(\mathcal{C}) = 2^k$ 。

用任意极化码构造算法获得的信息比特集合都满足 $\mathcal{A} \subset \{1, 2, 3, \dots, N = 2^n\}$, 所以上面的定理适用于极化码。

下面我们讨论比特置换矩阵和 $\mathbf{F}^{\otimes n}$ 可交换性。我们熟知的结论: $\mathbf{B}_N \mathbf{F}^{\otimes \log_2 N} = \mathbf{F}^{\otimes \log_2 N} \mathbf{B}_N$ 只是更广泛情况中的一个特例。

定义 3.9 比特置换及其对应的矩阵

设 $N = 2^n$, 对长度为 N 的向量 $\mathbf{a}_0^{N-1} = (a_0, a_1, \dots, a_{N-1})$ 进行满足如下条件的置换 π , 这种置换 π 就称为比特置换。

$$\begin{aligned} (a_0, a_1, \dots, a_{N-1}) &\stackrel{(a)}{=} (a_{<0_1, \dots, 0_n>_2}, a_{<1_1, \dots, 1_n>_2}, \dots, a_{<N-1_1, \dots, N-1_n>_2}) \\ &\xrightarrow{\pi} (a_{<0_{\pi(1)}, \dots, 0_{\pi(n)}>_2}, a_{<1_{\pi(1)}, \dots, 1_{\pi(n)}>_2}, \dots, a_{<N-1_{\pi(1)}, \dots, N-1_{\pi(n)}>_2}). \end{aligned} \quad (3.42)$$

其中等号(a)表示把 $a_i, 0 \leq i \leq N-1$ 的下标 i 作 n 位二进制展开, $< i_1, i_2, \dots, i_n >_2$ 表示 i 的二进制展开, $i_k, 1 \leq k \leq n$ 表示 i 的二进制展开中的第 k 位, 其中 i_1 是最高位, i_n 是最低位。置换 π 对向量 $(1, 2, \dots, n)$ 作置换得到 $(\pi(1), \pi(2), \dots, \pi(n))$, 也就是把 i 的二进制展开的比特位作了置换。

长度为 N 的向量 \mathbf{a}_0^{N-1} 总计有 $N!$ 种置换方式, 比特置换 π 是所有 $N!$ 个置换构成的集合的子集。 π 共计有 $n!$ 个, 远小于 $N!$ 。

定义 3.10 矩阵的两种等价索引写法

设 $N = 2^n$, 记 $A_{i,j}$ 是矩阵 $N \times N$ 维 \mathbf{A} 中的第 i 行第 j 列的元素, $< i_1, i_2, \dots, i_n >_2$ 表示 i 的二进制展开, $< j_1, j_2, \dots, j_n >_2$ 表示 j 的二进制展开, 那么下面两种记号:

$$A_{i,j} \text{ 和 } A_{<i_1, i_2, \dots, i_n>_2, <j_1, j_2, \dots, j_n>_2},$$

说的是同一个元素。

引理 3.8 设矩阵 \mathbf{P} 是任意一个 $N \times N$ 维置换矩阵, \mathbf{P} 对应的置换记为 ρ 。 \mathbf{A} 是任意一个 $N \times N$ 维矩阵, 按 \mathbf{A} 的每一行进行分块, 分块结果记作 $[\mathbf{a}_1; \mathbf{a}_2; \dots; \mathbf{a}_N]$, 其中 \mathbf{a}_i 是 \mathbf{A} 的第 i 行。按 \mathbf{A} 的每一列进行分块, 分块结果记作 $[\mathbf{A}_1 \ \mathbf{A}_2 \ \dots \ \mathbf{A}_N]$, 其中 \mathbf{A}_i 是 \mathbf{A} 的第 i 列。那么下式成立:

$$\mathbf{PA} = [\mathbf{a}_{\rho(1)}; \mathbf{a}_{\rho(2)}; \dots; \mathbf{a}_{\rho(N)}], \quad (3.43)$$

$$\mathbf{AP}^T = [\mathbf{A}_{\rho(1)} \ \mathbf{A}_{\rho(2)} \ \dots \ \mathbf{A}_{\rho(N)}]. \quad (3.44)$$

证明 记 \mathbf{P} 的第 i 行中的 1 位于第 $\gamma(i)$ 列, 相应地, \mathbf{P}^T 的第 i 列中的 1 位于第 $\gamma(i)$ 行, 一一计算 \mathbf{PA} 的每一行和 \mathbf{AP}^T 的每一列即可。

上面的引理的含义是: 左乘置换矩阵 \mathbf{P} 相当于 \mathbf{A} 的行置换, 右乘 \mathbf{P}^T 相当于对 \mathbf{A} 的列作规律相同的列置换。

下面是一个和置换无关的引理，它描述了 $\mathbf{F}^{\otimes n}$ 中元素的构成规律。

引理 3.9 记 $F_{i,j}^{\otimes n}$ 是 $\mathbf{F}^{\otimes n}$ 中第*i*行、第*j*列的元素，*i*的*n*位二进制展开记为 $< i_1, i_2, \dots, i_n >_2$ ，*j*的*n*位二进制展开记为 $< j_1, j_2, \dots, j_n >_2$ ，则 $F_{i,j}^{\otimes n} = \prod_{\beta=1}^n F_{i_\beta, j_\beta}$ 。

其中 F_{i_β, j_β} 的四种可能取值为 $F_{0,0} = 1, F_{0,1} = 0, F_{1,0} = 1, F_{1,1} = 1$ 。 F_{i_β, j_β} 事实上就是 \mathbf{F} 的四个元素，下标表示行列号：

$$\mathbf{F} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} F_{0,0} & F_{0,1} \\ F_{1,0} & F_{1,1} \end{bmatrix} \quad (3.45)$$

证明 使用归纳法。

源头：当*n* = 1时，由式(3.45)可见， $F_{i,j}^{\otimes n} = \prod_{\beta=1}^n F_{i_\beta, j_\beta}$ 平凡成立。

归纳假设：当*n* = *k*时， $F_{i,j}^{\otimes k} = \prod_{\beta=1}^k F_{i_\beta, j_\beta}$ 成立。

下一次：当*n* = *k* + 1时，可以写出下面的式子：

$$\mathbf{F}^{\otimes(k+1)} = \mathbf{F} \otimes \mathbf{F}^{\otimes k} = \begin{bmatrix} F_{0,0}\mathbf{F}^{\otimes k} & F_{0,1}\mathbf{F}^{\otimes k} \\ F_{1,0}\mathbf{F}^{\otimes k} & F_{1,1}\mathbf{F}^{\otimes k} \end{bmatrix}. \quad (3.46)$$

对 $\mathbf{F}^{\otimes(k+1)}$ 中第*i*行、第*j*列的元素 $\mathbf{F}_{i,j}^{\otimes(k+1)}$ 分情况讨论，注意这时 $0 \leq i, j \leq 2^{k+1} - 1$ 。

情况1： $0 \leq i, j \leq 2^k - 1$ ，即 $i_1 = j_1 = 0$ 。

一方面：

$$\prod_{\beta=1}^{k+1} F_{i_\beta, j_\beta} = F_{0,0} \prod_{\beta=2}^{k+1} F_{i_\beta, j_\beta} = \prod_{\beta=2}^{k+1} F_{i_\beta, j_\beta} \stackrel{(a)}{=} F_{i^*, j^*}^{\otimes k}. \quad (3.47)$$

其中等号(a)是归纳假设。上式表示 $\prod_{\beta=1}^{k+1} F_{i_\beta, j_\beta}$ 等于 $\mathbf{F}^{\otimes k}$ 中第*i**行、第*j**列的元素，其中*i**的*k*位二进制展开为 $< i_2, i_3, \dots, i_{k+1} >_2$ ，*j**的*k*位二进制展开为 $< j_2, j_3, \dots, j_{k+1} >_2$ 。

另一方面：由于 $0 \leq i, j \leq 2^k - 1$ ，所以 $\mathbf{F}_{i,j}^{\otimes(k+1)}$ 位于式(3.46)的左上角的分块中，这个分块就恰是 $\mathbf{F}^{\otimes k}$ ，所以 $\mathbf{F}_{i,j}^{\otimes(k+1)} = \mathbf{F}_{i^*, j^*}^{\otimes k}$ 。

综上， $\mathbf{F}_{i,j}^{\otimes(k+1)} = \mathbf{F}_{i^*, j^*}^{\otimes k} = \prod_{\beta=1}^{k+1} F_{i_\beta, j_\beta}$ 。

其余情况，包括 $0 \leq i \leq 2^k - 1, 2^k \leq j \leq 2^{k+1} - 1, 2^k \leq i \leq 2^{k+1} - 1, 0 \leq j \leq 2^k - 1$ 以及 $2^k \leq i \leq 2^{k+1} - 1, 2^k \leq j \leq 2^{k+1} - 1$ 类似可证，它们的分析只不过是移动到了其余三个分块之中。

定义 3.11 $j \preceq i$

设 $0 \leq i, j \leq N - 1 = 2^n - 1$ 的位进制展开分别是 $< i_1, \dots, i_n >_2, < j_1, \dots, j_n >_2$ ， $j \preceq i$ 是

指 $\forall \beta \in \{1, 2, \dots, n\}, j_\beta \leq i_\beta$ 。

引理 3.10 设 $0 \leq i, j \leq N - 1 = 2^n - 1$ 的位进制展开分别是 $< i_1, \dots, i_n >_2, < j_1, \dots, j_n >_2$ ，则 $\mathbf{G}_{i,j} = 1 \iff j \preceq i$ 。

证明 由引理3.9可知， $\mathbf{G}_{i,j} = \prod_{\beta=1}^n F_{i_\beta, j_\beta}$ 。

$$\mathbf{G}_{i,j} = \prod_{\beta=1}^n F_{i_\beta, j_\beta} = 1 \iff \forall \beta \in \{1, 2, \dots, n\}, F_{i_\beta, j_\beta} = 1.$$

因为 $F_{0,0} = 1, F_{0,1} = 0, F_{1,0} = 1, F_{1,1} = 1$ ，所以 $F_{i_\beta, j_\beta} = 1 \iff j_\beta \leq i_\beta$ 。

从而 $\mathbf{G}_{i,j} = 1 \iff \forall \beta \in \{1, 2, \dots, n\}, j_\beta \leq i_\beta \iff j \preceq i$ 。

定理 3.8 设 π 是比特置换， Π 是 π 对应的置换矩阵，则 $\Pi \mathbf{F}^{\otimes n} \Pi^T = \mathbf{F}^{\otimes n}$ 。这个定理来自[11]

证明 设 $\mathbf{F}^{\otimes n}$ 中第*i*行、第*j*列的元素是 $\mathbf{F}_{i,j}^{\otimes n}$ ，则由引理3.8，经过计算 $\Pi \mathbf{F}^{\otimes n} \Pi^T$ 后， $\mathbf{F}_{i,j}^{\otimes n}$ 跑到了二进制索引($< i_{\pi(1)}, i_{\pi(2)}, \dots, i_{\pi(n)} >_2, < j_{\pi(1)}, j_{\pi(2)}, \dots, j_{\pi(n)} >_2$)的位置上，那么我们测试 $\mathbf{F}_{i,j}^{\otimes n}$ 是否等于 $\mathbf{F}_{< i_{\pi(1)}, i_{\pi(2)}, \dots, i_{\pi(n)} >_2, < j_{\pi(1)}, j_{\pi(2)}, \dots, j_{\pi(n)} >_2}^{\otimes n}$

$$\mathbf{F}_{< i_{\pi(1)}, i_{\pi(2)}, \dots, i_{\pi(n)} >_2, < j_{\pi(1)}, j_{\pi(2)}, \dots, j_{\pi(n)} >_2}^{\otimes n} \stackrel{(a)}{=} \prod_{\beta=1}^n F_{i_{\pi(\beta)}, j_{\pi(\beta)}} \stackrel{(b)}{=} \prod_{\beta=1}^n F_{i_\beta, j_\beta} = \mathbf{F}_{i,j}^{\otimes n}. \quad (3.48)$$

其中等号(a)使用了引理3.9，等号(b)是因为域上乘法具有交换律。证毕。

推论 3.2 $\Pi \mathbf{F}^{\otimes n} = \mathbf{F}^{\otimes n} \Pi$

证明 对于任意置换阵 \mathbf{P} 有 $\mathbf{P}^{-1} = \mathbf{P}^T$ 。 $\Pi \mathbf{F}^{\otimes n} \Pi^T = \mathbf{F}^{\otimes n}$ 两侧乘 \mathbf{P} 即可。

推论 3.3 $\mathbf{B}_N \mathbf{F}^{\otimes \log_2 N} = \mathbf{F}^{\otimes \log_2 N} \mathbf{B}_N$

证明 因为 \mathbf{B}_N 是比特置换。

由推论3.3可知，极化码的编码过程可写为 $\mathbf{x}_1^N = \mathbf{u}_1^N \mathbf{G}_N = (\mathbf{u}_1^N \mathbf{B}_N) \mathbf{F}^{\otimes \log_2 N} = (\mathbf{u}_1^N \mathbf{F}^{\otimes \log_2 N}) \mathbf{B}_N$ ，所以 \mathbf{B}_N 的作用不过是置换 \mathbf{u}_1^N 或 \mathbf{x}_1^N 而已，不会对极化码的BLER性能产生任何影响。因此，在下面的叙述中，我们有时丢掉 \mathbf{B}_N ，只考虑 $\mathbf{F}^{\otimes n}$ 。

定理 3.9 $\mathbf{F}^{\otimes n}$ 第 i , $0 \leq i < 2^n - 1$ 行中 1 的个数是 $2^{wt_n(i)}$, 其中 $wt_n(i)$ 表示 i 的 n 位二进制展开中 1 的个数, 此处的“二进制展开”是指: 如果二进制的位数能够完全表示十进制数 i , 则无事发生; 如果二进制的位数不能完全表示十进制数 i , 则被扔掉的是二进制的高位。这里, $\mathbf{F}^{\otimes n}$ 某一行中 1 的个数也称为该行的行重。这个定理来自文献[3]。

证明 使用归纳法。

源头: 容易验证 \mathbf{F} 满足定理。

归纳假设: $\mathbf{F}^{\otimes k}$ 第 i , $0 \leq i < 2^k - 1$ 行中 1 的个数是 $2^{wt_k(i)}$ 。

下一步:

$$\mathbf{F}^{\otimes(k+1)} = \begin{bmatrix} \mathbf{F}^{\otimes k} & \mathbf{O} \\ \mathbf{F}^{\otimes k} & \mathbf{F}^{\otimes k} \end{bmatrix} \quad (3.49)$$

在 $\mathbf{F}^{\otimes(k+1)}$ 中, 行索引 i 的范围扩大了一倍: $0 \leq i < 2^{k+1} - 1$ 。

情况1: 当行索引 i 满足 $0 \leq i < 2^k - 1$, 即研究 $\mathbf{F}^{\otimes(k+1)}$ 的上半部分时, 我们发现 $\mathbf{F}^{\otimes(k+1)}$ 上半部分的右半边是零矩阵, 因此 $\mathbf{F}^{\otimes(k+1)}$ 上半部分的行重与 $\mathbf{F}^{\otimes k}$ 的行重完全相同。又注意到 $0 \leq i < 2^k - 1$, 即这时 i 的 $k + 1$ 位二进制展开的最高位是 0, 所以有 $wt_{k+1}(i) = wt_k(i)$ 。依据归纳假设, 定理3.9 对 $\mathbf{F}^{\otimes(k+1)}$ 的上半部分成立: 对于行索引 $0 \leq i < 2^k - 1$, 行重是 $2^{wt_k(i)}$, 也是 $2^{wt_{k+1}(i)}$ 。

情况2: 当行索引 i 满足 $2^k \leq i < 2^{k+1} - 1$, 即研究 $\mathbf{F}^{\otimes(k+1)}$ 的下半部分时, 我们发现 $\mathbf{F}^{\otimes(k+1)}$ 下半部分的左半边和右半边都是 $\mathbf{F}^{\otimes k}$, 所以此时 $\mathbf{F}^{\otimes(k+1)}$ 的行重是 $\mathbf{F}^{\otimes k}$ 对应行重的两倍。又注意到此时行索引 i 的 $k + 1$ 位二进制展开的最高位是 1, 所以 $wt_{k+1}(i) = wt_k(i) + 1$ 。根据归纳假设, 定理3.9 对 $\mathbf{F}^{\otimes(k+1)}$ 的下半部分成立: 对于行索引 $2^k \leq i < 2^{k+1} - 1$, 行重是 $2 \times 2^{wt_k(i)}$, 这个数等于 $2^{wt_k(i)+1} = 2^{wt_{k+1}(i)}$ 。

推论 3.4 设 π 是 n 位比特置换, Π 是 π 对应的置换矩阵。沿用定理3.9 中的说法, 则矩阵 $\Pi\mathbf{F}^{\otimes n}$ 中第 i 行的行重也是 $2^{wt_n(i)}$ 。

证明 矩阵 $\Pi\mathbf{F}^{\otimes n}$ 和 $\mathbf{F}^{\otimes n}$ 的差别在于前者是后者的行置换, 置换规则是 Π 。设 $\mathbf{F}^{\otimes n}$ 的第 i 行为 \mathbf{f}_i , 由定理3.9, \mathbf{f}_i 的行重是 $2^{wt_n(i)}$ 。现在依据比特置换 π , 把 \mathbf{f}_i 换到第 j 行, 其中 i 的二进制展开是 $< i_1, i_2, \dots, i_n >_2$, j 的二进制展开是 $< i_{\pi(1)}, i_{\pi(2)}, \dots, i_{\pi(n)} >_2$ 。由于 \mathbf{f}_i 的内容没有任何变化, 所以 $\Pi\mathbf{F}^{\otimes n}$ 第 j 行的行重就是 \mathbf{f}_i 的重量: $2^{wt_n(i)}$ 。注意到比特置换不改变一个数的二进制展开中 1 的数量, 所以 $2^{wt_n(i)} = 2^{wt_n(j)}$, 从而 $\Pi\mathbf{F}^{\otimes n}$ 第 j 行的行重是 $2^{wt_n(j)}$ 。证毕。

推论 3.5 矩阵 $\mathbf{B}_{2^n} \mathbf{F}^{\otimes n}$ 中第 i 行的行重是 $2^{wt_n(i)}$ 。

证明 矩阵 \mathbf{B}_{2^n} 是比特置换矩阵。

定理 3.10 设 $N = 2^n$, 行向量 $\mathbf{u}_1^N = (00\dots01 * * * \dots*)$ 中的首个 1 位于第 i 个位置上, 即 $\mathbf{u}_1^{i-1} = \mathbf{0}, u_i = 1$, \mathbf{u}_{i+1}^N 的取值是任意的, 则 $wt(\mathbf{u}_1^N \mathbf{F}^{\otimes n}) \geq wt(\mathbf{F}_i^{\otimes n})$ 。其中 $wt(\mathbf{a})$ 表示 \mathbf{a} 中 1 的数量, $\mathbf{F}_i^{\otimes n}$ 是 $\mathbf{F}^{\otimes n}$ 中的第 i 行。

证明 这个定理来自文献[2]。对 n 使用归纳法。

源头: 当 $n = 1$ 时, 对于(10), 有 $(10)\mathbf{F} = (10)$, 定理成立; 对于(11), 有 $(11)\mathbf{F} = (01)$, 定理成立; 对于(01), 有 $(01)\mathbf{F} = (11)$, 定理成立。

归纳假设: 当 $n = k$ 时, $\mathbf{u}_1^{i-1} = \mathbf{0}, u_i = 1$, $\mathbf{u}_{i+1}^{2^k}$ 的取值是任意的, 则 $wt(\mathbf{u}_1^{2^k} \mathbf{F}^{\otimes k}) \geq wt(\mathbf{F}_i^{\otimes k})$ 。

下一步: 当 $n = k + 1$ 时, $\mathbf{u}_1^{i-1} = \mathbf{0}, u_i = 1$, $\mathbf{u}_{i+1}^{2^k}$ 的取值是任意的。下面分两种情况讨论。

情况1: $1 \leq i \leq 2^k$ 。此时有:

$$\begin{aligned} \mathbf{u}_1^{2^{k+1}} \mathbf{F}^{\otimes k+1} &= (00\dots01 * \dots * | * * * \dots * *) \begin{bmatrix} \mathbf{F}_i^{\otimes k} & \mathbf{O} \\ \mathbf{F}_i^{\otimes k} & \mathbf{F}_i^{\otimes k} \end{bmatrix} \\ &= ((00\dots01 * \dots *) \mathbf{F}^{\otimes k} + (* * * \dots * *) \mathbf{F}^{\otimes k}, (* * * \dots * *) \mathbf{F}^{\otimes k}) \\ &= (\mathbf{x} + \mathbf{y}, \mathbf{y}) \xrightarrow{\text{取1的个数}} wt(\mathbf{x} + \mathbf{y}) + wt(\mathbf{y}) = wt(\mathbf{x}) + 2wt(\mathbf{y}) - 2wt(\mathbf{xy}) \\ &\stackrel{(a)}{\geq} wt(\mathbf{x}) \stackrel{(b)}{\geq} wt(\mathbf{F}_i^{\otimes k}) \stackrel{(c)}{=} wt(\mathbf{F}_i^{\otimes k+1}), \end{aligned} \quad (3.50)$$

其中不等号(a)是因为 $wt(\mathbf{y}) \geq wt(\mathbf{xy})$, 不等号(b)是归纳假设, 等号(c)是因为 $\mathbf{F}^{\otimes k+1}$ 的右上分块是零矩阵。

情况2: $2^k + 1 \leq i \leq 2^{k+1}$ 。此时有:

$$\begin{aligned} \mathbf{u}_1^{2^{k+1}} \mathbf{F}^{\otimes k+1} &= (\mathbf{0} | 00\dots01 * \dots *) \begin{bmatrix} \mathbf{F}_i^{\otimes k} & \mathbf{O} \\ \mathbf{F}_i^{\otimes k} & \mathbf{F}_i^{\otimes k} \end{bmatrix} \\ &= ((00\dots01 * \dots *) \mathbf{F}^{\otimes k}, (00\dots01 * \dots *) \mathbf{F}^{\otimes k}) \\ &= (\mathbf{y}, \mathbf{y}) \xrightarrow{\text{取1的个数}} 2wt(\mathbf{y}) \stackrel{(a)}{\geq} 2wt(\mathbf{F}_{i-2^k}^{\otimes k}) \stackrel{(b)}{=} wt(\mathbf{F}_i^{\otimes k+1}), \end{aligned} \quad (3.51)$$

其中不等号(a)是归纳假设，等号(b)是因为 $\mathbf{F}^{\otimes k+1}$ 下半部分左右两块完全相同，都是 $\mathbf{F}^{\otimes k}$ 。证毕。

§ 3.3 非系统极化码编码过程

非系统码的意思是码字序列中不直接包含信源比特。事实上无论非系统极化码还是系统极化码，编码的渐进复杂度都是 $O(N \log_2 N)$ 。本节中我们彻底地抛弃置换 \mathbf{B}_N ，研究 $\mathbf{F}^{\otimes n}$ 。设 \mathcal{A} 是通过任意一个码构造算法得到的可靠极化信道集合，此时码长为 N ，信息比特数为 $K = |\mathcal{A}|$ 。在长度为 N 的信源序列 $\mathbf{u}_1^N = (u_1, u_2, \dots, u_N)$ 中，把 K 个信息比特放在 $i \in \mathcal{A}$ 的元素上，其余 $N - K$ 个元素叫做冻结比特，在本教程中总是取固定比特值0。 $\mathbf{x}_1^N = \mathbf{u}_1^N \mathbf{F}^{\otimes n}$ 是 \mathbf{u}_1^N 对应的码字，计算 \mathbf{x}_1^N 的过程就是极化码的非系统编码过程。我们先定义 $\mathbf{F}^{\otimes n}$ 的标准形式。

定义 3.12 $\mathbf{F}^{\otimes n}$ 的标准形式

如果 $\mathbf{F}^{\otimes n}$ 中的 n 列从左至右依次可表达为 $\mathbf{F}^{\otimes n} = [\mathbf{I}_{N/2} \otimes \mathbf{F} \otimes \mathbf{I}_1][\mathbf{I}_{N/4} \otimes \mathbf{F} \otimes \mathbf{I}_2] \dots [\mathbf{I}_1 \otimes \mathbf{F} \otimes \mathbf{I}_{N/2}] = \prod_{\beta=1}^{\log_2 N} [\mathbf{I}_{\frac{N}{2^\beta}} \otimes \mathbf{F} \otimes \mathbf{I}_{2^{\beta-1}}]$ ，则称这种形式是 $\mathbf{F}^{\otimes n}$ 的标准形式。

例 3.3 当 $n = 3$ 时， $\mathbf{F}^{\otimes 3}$ 的标准形式如图3.4所示。

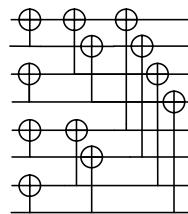


图 3.4 $\mathbf{F}^{\otimes 3}$ 的标准形式

图3.4中的编码过程是从左向右的，最左侧的线头接收信源序列 \mathbf{u} ，最右侧的线头输出编码结果 \mathbf{x} ，连接线表示“直接通过”， \oplus 表示其左侧线头和下方线头上比特值的异或。设 $\mathbf{u}_1^8 = (00010111)$ ，则其对应码字 $\mathbf{x}_1^8 = \mathbf{u}_1^8 \mathbf{F}^{\otimes 3}$ 的计算过程如图3.5所示，其中黑色比特值表示已经存在的计算结果，红色比特值表示本步骤中刚计算出来的比特值。每个 2×2 模块每次只允许输出右上角或右下角的计算结果，不允许同时输出，这是为了模仿下一章要讲的SC译码器，实际中没有这样的限制。按阅读文字的顺序看。

极化码编码大体有两种写法：其一、使用递归；其二，只用**for**循环。显然我们推荐只用**for**循环，因为递归并不能降低任何计算量，且增加程序执行时间。虽然如此，我们还是把两

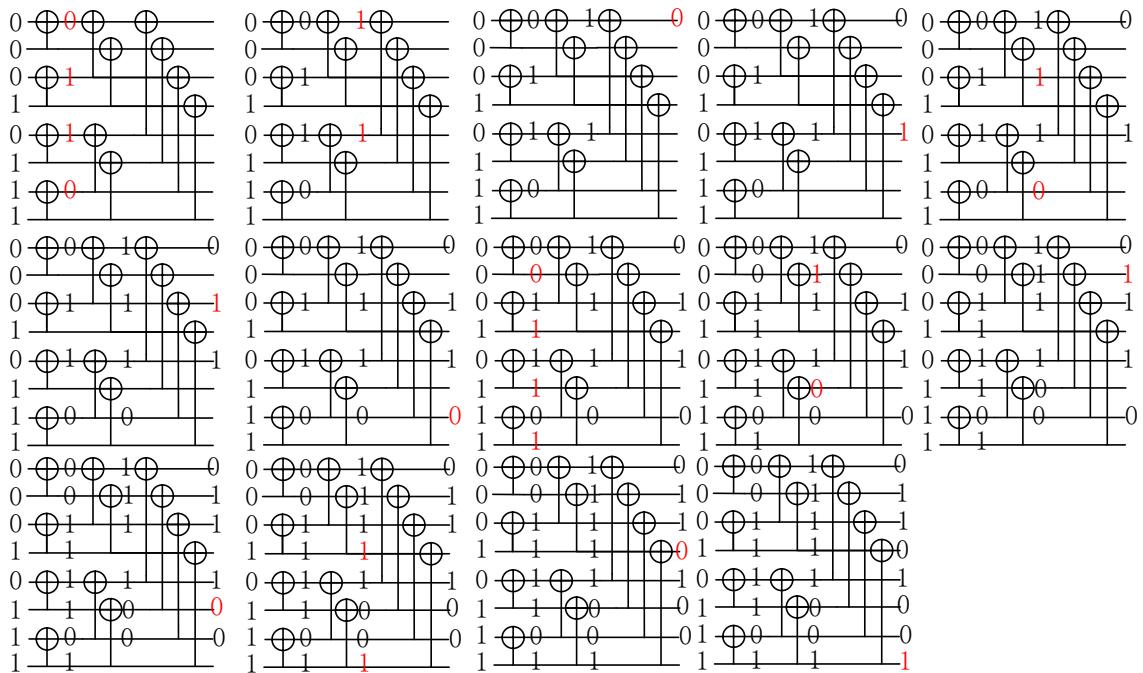


图 3.5 $\mathbf{F}^{\otimes 3}$ 的标准形式的编码过程, 共计 $2N - 2 = 14$ 张图

种形式的MATLAB代码给出了。需要注意的是, 下面两段代码的中间计算结果和图3.5中间计算结果不同, 但最终计算结果相同, 这种现象我们稍后解释。

首先是使用递归的代码, 它虽然短, 但是它的开销大。

```

1 function x = polar_encode(u)
2 N = length(u);
3 if N == 1
4     x = u;
5 else
6     u1u2 = mod(u(1 : N/2) + u(N/2 + 1 : N) , 2);
7     u2 = u(N/2 + 1 : N);
8     x = [polar_encode(u1u2) polar_encode(u2)];
9 end
10 end

```

下面是仅使用**for**循环的代码, 它需要依赖输入参数layer_vec和lambda_offset, 这两个参数

的含义如下，我稍后会给出一个例子，演示它们的使用。

其中layer_vec是一个表示“实际执行层数”的向量，它是一个 $N \times 1$ 维向量，里面的元素是取值于 $[0, n - 1]$ 内的整数。layer_vec中的第*i*, $1 \leq i \leq N$ 个元素layer_vec(i)表示：*i* − 1的二进制展开中“低位连续零的个数”，也就是*i* − 1的质数展开中2的次数，也就是*i* − 1能连续整除2的次数。例如，当*n* = 3时，layer_vec(5) = 2。由于0是特例，0的二进制展开全是0，所以我们不计算它，即layer_vec(1)总是初始值0，这个值也不参与后续计算。

lambda_offset是一个“分段”用的向量，它的长度是*n*+1，元素的值依次是 $2^0, 2^1, \dots, 2^{\log_2 N} = n$ 。lambda_offset中的值给x.internal_value这个存储中间计算结果的向量分了段。

```

1 lambda_offset = 2.^ (0 : log2(N));
2
3 function layer_vec = get_layer(N)
4 layer_vec = zeros(N, 1);
5 for phi = 1 : N - 1
6     psi = phi;
7     layer = 0;
8     while(mod(psi, 2) == 0)
9         psi = floor(psi/2);
10        layer = layer + 1;
11    end
12    layer_vec(phi + 1) = layer;
13 end
14 end
15
16 function x = polar_encode(u, lambda_offset, layer_vec)
17 N = length(u);
18 m = log2(N);
19 x_internal_value = zeros(2 * N - 1, 1);%internal calculation results
20 x_internal_value(end - N + 1 : end) = u;%obtain source vector u
21 x = zeros(N, 1);%final output
22 %calculate x1
23 for i_layer = m - 1 : -1 : 0
24     index_1 = lambda_offset(i_layer + 1);
25     index_2 = lambda_offset(i_layer + 2);
26     for beta = index_1 : index_2 - 1

```

```
27     x_internal_value(beta) = ...
28     x_internal_value(index_2 + beta - index_1) ...
29     + x_internal_value(index_2 + beta);
30 end
31
32 x(1) = x_internal_value(1);
33 %calculate x2,...,xN
34 for phi = 1 : N - 1
35     layer = layer_vec(phi + 1);
36     %directly get bits
37     index_1 = lambda_offset(layer + 1);
38     index_2 = lambda_offset(layer + 2);
39     for beta = index_1 : index_2 - 1
40         x_internal_value(beta) = x_internal_value(index_2 + beta);
41     end
42     %plus
43     for i_layer = layer - 1 : -1 : 0
44         index_1 = lambda_offset(i_layer + 1);
45         index_2 = lambda_offset(i_layer + 2);
46         for beta = index_1 : index_2 - 1
47             x_internal_value(beta) = ...
48             x_internal_value(index_2 + beta - index_1) ...
49             + x_internal_value(index_2 + beta);
50         end
51     end
52     x(phi + 1) = x_internal_value(1);
53 end
54 %mod 2
55 x = mod(x, 2);
56 end
```

上面的代码第一次看的话可能不知所云，也可能你会觉得这个代码太长了，我下面画出它的执行过程，如图3.6所示。每种颜色表示一个分段，每个分段的第一个索引分别是(1, 2, 4, 8)，恰好是lambda_offset中的元素。黄色方块的内容是信源序列u且一直不变，白色方框的内容是码字比特的计算结果，其余颜色方框的内容是中间计算结果。

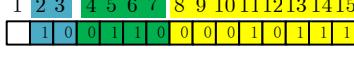
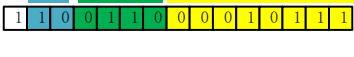
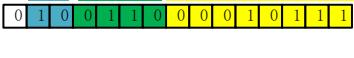
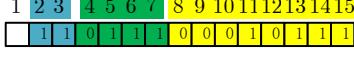
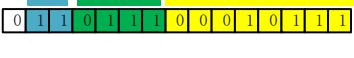
第1步		for beta = 4 : 7 计算绿色比特	$z_4 = z_8 + z_{12}, z_5 = z_9 + z_{13}$ $z_6 = z_{10} + z_{14}, z_7 = z_{11} + z_{15}$
第2步		for beta = 2 : 3 计算蓝色比特	$z_2 = z_4 + z_6, z_3 = z_5 + z_7$
第3步		for beta = 1 : 1 计算白色比特	$z_1 = z_2 + z_3$ 得到x1=0
第4步		for beta = 1 : 1 计算白色比特	$z_1 = z_3$ 得到x2=1
第5步		for beta = 2 : 3 计算蓝色比特	$z_2 = z_6, z_3 = z_7$
第6步		for beta = 1 : 1 计算白色比特	$z_1 = z_2 + z_3$ 得到x3=1
第7步		for beta = 1 : 1 计算白色比特	$z_1 = z_3$ 得到x4=0
第8步		for beta = 4 : 7 计算绿色比特	$z_4 = z_{12}, z_5 = z_{13}$ $z_6 = z_{14}, z_7 = z_{15}$
第9步		for beta = 2 : 3 计算蓝色比特	$z_2 = z_4 + z_6, z_3 = z_5 + z_7$
第10步		for beta = 1 : 1 计算白色比特	$z_1 = z_2 + z_3$ 得到x5=1
第11步		for beta = 1 : 1 计算白色比特	$z_1 = z_3$ 得到x6=0
第12步		for beta = 2 : 3 计算蓝色比特	$z_2 = z_6, z_3 = z_7$
第13步		for beta = 1 : 1 计算白色比特	$z_1 = z_2 + z_3$ 得到x7=0
第14步		for beta = 1 : 1 计算白色比特	$z_1 = z_3$ 得到x8=1

图 3.6 只用**for**的代码的计算过程，每步中15个方框的内容分别为 z_1, \dots, z_{15}

下面是关于 $\mathbf{F}^{\otimes n}$ 的 $n!$ 个“基于置换的端到端等价形式”的内容。

先看一个例子。当 $N = 8$ 时, $\mathbf{F}^{\otimes n}$ 的标准形式位于图3.7的中间, 而图3.7两边的形式是通过置换 $\mathbf{F}^{\otimes n}$ 的标准形式的列得到的: 左边的图置换了第1和2列, 右边的图置换了第2和3列。不难验证对于任意 $\mathbf{u}_1^8 \in \{0, 1\}^8$, 输入到图3.7中任意一个编码图的左侧, 右侧得到计算结果都相同, 这样的现象叫做 $\mathbf{F}^{\otimes n}$ “基于置换的端到端等价形式”, 其中“置换”指的置换标准编码图中的列, “端到端等价”的意思是对于任意 $\mathbf{u}_1^8 \in \{0, 1\}^8$, 经过置换的标准编码图最右侧输出的计算结果和标准编码图最右侧输出的结果一样, 但是中间计算结果不一定一样。

显然, $\mathbf{F}^{\otimes n}$ 的标准编码图中共有 n 列, 对应的列置换了 $n!$ 个, 下面的定理说明这 $n!$ 个置换编码图和标准编码图端到端等价。

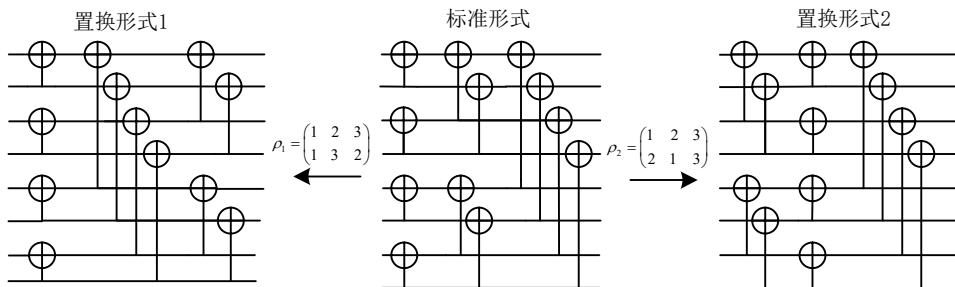


图 3.7 $\mathbf{F}^{\otimes 3}$ 基于置换的端到端等价形式

定理 3.11 设 $N = 2^n$, 记图3.7中间的形式为标准形式的 $\mathbf{F}^{\otimes n}$, $\mathbf{F}^{\otimes n}$ 共有 n 列, 从左向右依次标号为 $\{1, 2, 3, \dots, n\}$ 。设置置换 ρ 是定义集合 $\{1, 2, 3, \dots, n\}$ 上的置换, $\mathbf{F}_{\rho}^{\otimes n}$ 表示 $\mathbf{F}^{\otimes n}$ 的列按 $(\rho(1), \rho(2), \dots, \rho(n))$ 的顺序排列, 也就是用 ρ 置换 $\mathbf{F}^{\otimes n}$ 的列, 则对于任意 $\mathbf{u}_1^N \in \{0, 1\}^N$, 有 $\mathbf{u}_1^N \mathbf{F}^{\otimes n} = \mathbf{u}_1^N \mathbf{F}_{\rho}^{\otimes n}$ 。

证明 对 n 用归纳法。

源头: $n = 1$ 过于平凡, 从 $n = 2$ 开始, 如图3.8所示, 此时只存在 $2!$ 个置换, 定理成立。

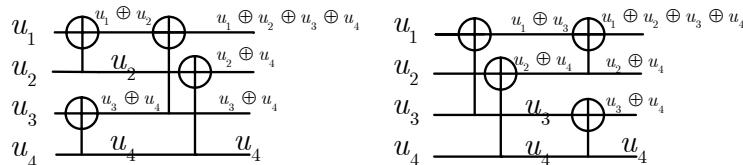


图 3.8 当 $N = 2^2$ 时, 对于任意 $\mathbf{u}_1^4 \in \{0, 1\}^4$, 定理结论成立

归纳假设：当 $n = k$ 时，对于定义在 $\{1, 2, 3, \dots, k\}$ 上的任意置换 ρ ，有 $\mathbf{u}_1^{2^k} \mathbf{F}^{\otimes k} = \mathbf{u}_1^{2^k} \mathbf{F}_{\rho}^{\otimes k}$ 。

下一步：当 $n = k + 1$ 时，与 $n = k$ 时主要有两点不同：(i). 定义在 $\{1, 2, 3, \dots, k+1\}$ 上的置换比定义在 $\{1, 2, 3, \dots, k\}$ 上的置换多了 $k+1$ 倍；(ii). $\mathbf{F}^{\otimes k+1}$ 由上下两个 $\mathbf{F}^{\otimes k}$ 拼接而成，拼接方式如图3.9所示，即使用 $\mathbf{F} \otimes \mathbf{I}_{N/2}$ 把两个 $\mathbf{F}^{\otimes k}$ 连起来。图3.9中包含了 $k!$ 种基于列置换的端到端等价形式：在固定第 $k+1$ 列不动的情况下，由归纳假设，置换前 k 列不改变任何计算结果（这时置换前 k 列等价于上下两个 $\mathbf{F}^{\otimes k}$ 同时执行相同的置换）。

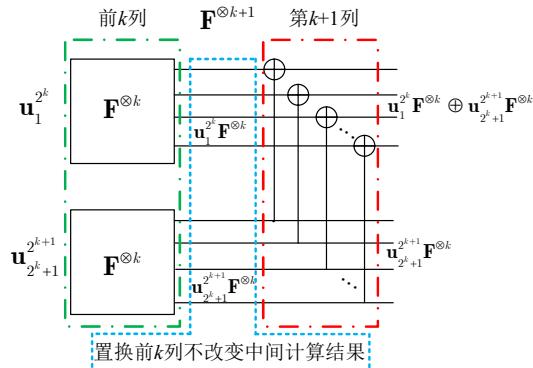


图 3.9 $\mathbf{F}^{\otimes k+1}$ 和 $\mathbf{F}^{\otimes k}$ 的关系， $\mathbf{F} \otimes \mathbf{I}_{N/2}$ 对应的列放在最后面

类似地，这次我们把 $\mathbf{F} \otimes \mathbf{I}_{N/2}$ 对应的列放在最前面，如图3.10所示。可见图3.9和3.10计算结果相同。现在我们已经证明了 $2(k!)$ 种情况，还剩 $(k-1)(k!)$ 种情况。

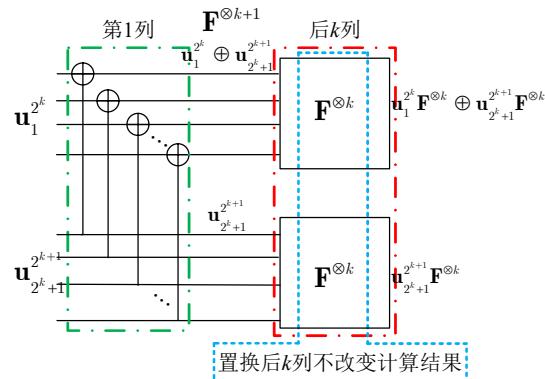


图 3.10 $\mathbf{F} \otimes \mathbf{I}_{N/2}$ 对应的列放在最前面

剩下的情况无非就是把 $\mathbf{F} \otimes \mathbf{I}_{N/2}$ 对应的列置于中间，如图3.12所示。其中 $\mathbf{F}_{\rho,1:k_1}^{\otimes k} = \prod_{\beta=1}^{k_1} [\mathbf{I}_{\frac{N}{2\rho(\beta)}} \otimes \mathbf{F} \otimes \mathbf{I}_{2\rho(\beta)-1}]$ 表示 $\mathbf{F}_{\rho}^{\otimes k}$ 的前 k_1 , $1 \leq k_1 \leq k-1$ 列， $\mathbf{F}_{\rho,k_1+1:n}^{\otimes k} = \prod_{\beta=k_1+1}^{\log_2 N} [\mathbf{I}_{\frac{N}{2\rho(\beta)}} \otimes \mathbf{F} \otimes \mathbf{I}_{2\rho(\beta)-1}]$ 表示 $\mathbf{F}_{\rho}^{\otimes k}$ 的后 $k-k_1$ 列。

$\mathbf{I}_{2\rho(\beta)-1}$]表示 $\mathbf{F}_\rho^{\otimes k}$ 的后 $n - k_1$ 列，同时注意到 $\mathbf{F}^{\otimes k} = \mathbf{F}_\rho^{\otimes k} = \mathbf{F}_{\rho,1:k_1}^{\otimes k} \mathbf{F}_{\rho,k_1+1:n}^{\otimes k}$ ，第一个等号使用了归纳假设，第二个等号基于如图3.11中的事实。可见图3.9、3.10和3.12计算结果相同。图3.12中间的列共有 $k - 1$ 个插入位置，证毕。

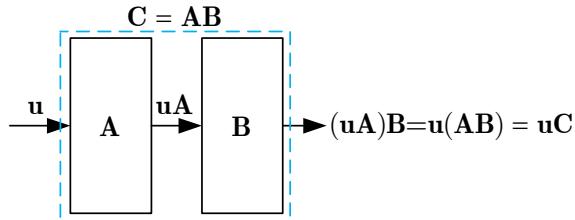


图 3.11 第一个框是矩阵**A**对应编码过程，第二个框是矩阵**B**对应编码过程，记**C** = **AB**，则本图中的计算结果为(**uA**)**B** = **uC**

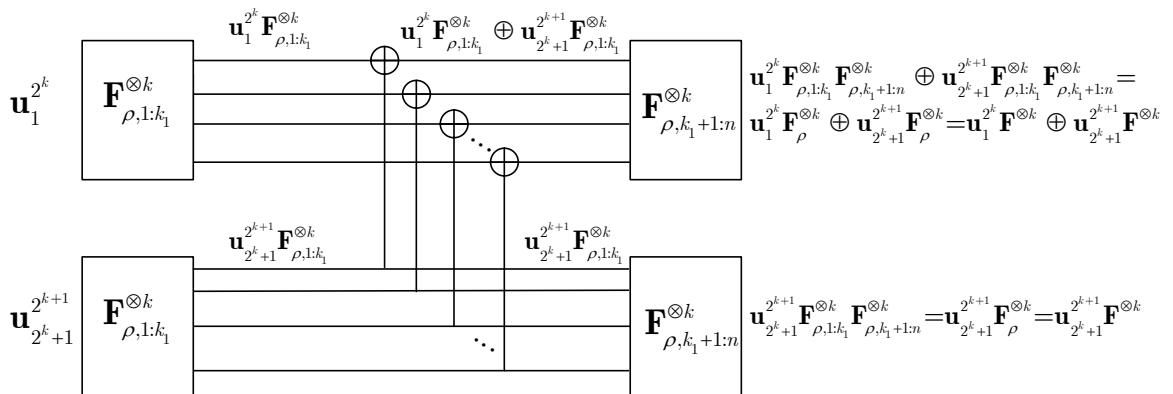


图 3.12 $\mathbf{F} \otimes \mathbf{I}_{N/2}$ 对应的列放在中间

§ 3.4 系统极化码编码过程

系统码就是能够直接从码字序列中提取信息比特的码字，一般信息比特位于整个码字序列的前端。但是极化码的系统码稍微复杂一些，因为信息比特散落在系统极化码的码字序列中[8]，并不集中位于码字前端或后端。

设我们的目标系统极化码的码长为 $N = 2^n$ ，信息比特数为 K 。 $\mathcal{A}, |\mathcal{A}| = K$ 是通过某种极化码构造算法得到的信息比特的索引集合， $\mathcal{A}^c = \{1, 2, \dots, N\} \setminus \mathcal{A}, |\mathcal{A}^c| = N - K$ 是冻结比特的索引集合，冻结比特的值全为0。为了记号的简便，本节中记**G** = $\mathbf{F}^{\otimes n}$ 。记**G** 中行号位于 \mathcal{A} 中的行

构成子矩阵 \mathbf{G}_A , \mathbf{G}_{A^c} 的含义以此类推。记信源序列 $\mathbf{u} = (u_1, u_2, \dots, u_N)$ 中位于索引集合 A 中的元素形成的子向量为 \mathbf{u}_A , 位于索引集合 A^c 中的元素形成的子向量为 \mathbf{u}_{A^c} , \mathbf{x}_A 和 \mathbf{x}_{A^c} 的含义以此类推。那么, 极化码的编码过程可以表达如下:

$$\mathbf{x} = (\mathbf{u}_A | \mathbf{u}_{A^c} = \mathbf{0}) \left[\begin{array}{c} \mathbf{G}_A \\ \mathbf{G}_{A^c} \end{array} \right] = \mathbf{u}_A \mathbf{G}_A. \quad (3.52)$$

下一步强行把 \mathbf{x} 分为 $\mathbf{x} = (\mathbf{x}_A | \mathbf{x}_{A^c})$ 两部分。其中 \mathbf{x}_A 就是想要传输的 K 个信息比特, 所以如果有了 \mathbf{x} , 我们就能直接提取 \mathbf{x}_A , 这就是系统极化码的意义所在; \mathbf{x}_{A^c} 是 $N - K$ 个校验位, \mathbf{x}_{A^c} 是 \mathbf{x}_A 的函数。有了 $\mathbf{x} = (\mathbf{x}_A | \mathbf{x}_{A^c})$, 我们能做进一步的分割:

$$\mathbf{x} = (\mathbf{x}_A | \mathbf{x}_{A^c}) = (\mathbf{u}_A \mathbf{G}_{AA} | \mathbf{u}_A \mathbf{G}_{AA^c}). \quad (3.53)$$

其中 \mathbf{G}_{AA} 表示 \mathbf{G}_A 中列号位于 A 中的列组成的子矩阵, \mathbf{G}_{AA^c} 的含义以此类推。上式写成两个式子, 就是:

$$\mathbf{x}_A = \mathbf{u}_A \mathbf{G}_{AA}, \quad (3.54)$$

$$\mathbf{x}_{A^c} = \mathbf{u}_A \mathbf{G}_{AA^c}. \quad (3.55)$$

由上面两个式子我们可以轻易地获得 \mathbf{x}_{A^c} 的表达式:

$$\mathbf{x}_{A^c} = \mathbf{x}_A \mathbf{G}_{AA}^{-1} \mathbf{G}_{AA^c}. \quad (3.56)$$

上式的意义是: 对于系统极化码, 我首先接收到 K 个信息比特, 我把他们放到 \mathbf{x}_A 的位置上, 剩下的 \mathbf{x}_{A^c} 由上式计算得到, 这样我们就得到了系统极化码的一个码字。

然而, 上式虽然简单, 但却是基于向量-矩阵乘法, 复杂度为 $O(N^2)$, 显然只能用于教材中做个引子, 实际中是不可能使用的, 因为 $\mathbf{G}_{AA}^{-1} \mathbf{G}_{AA^c}$ 不是低密度的。下面, 我们基于上式, 介绍几个复杂度为 $O(N \log_2 N)$ 的系统编码方法。

方法1: 递归法, 来自文献[8]。

设 \mathbf{x} 的长度为 $N = 2^n$, \mathbf{x}_1 表示 \mathbf{x} 中前一半元素, \mathbf{x}_2 表示 \mathbf{x} 中后一半元素, \mathbf{u}_1 和 \mathbf{u}_2 的含义以此类推。根据 $\mathbf{F}^{\otimes n}$ 和 $\mathbf{F}^{\otimes n-1}$ 的关系, 我们得到下式:

$$(\mathbf{x}_1 | \mathbf{x}_2) = (\mathbf{u}_1 | \mathbf{u}_2) \left(\begin{array}{cc} \mathbf{F}^{\otimes n-1} & \mathbf{O} \\ \mathbf{F}^{\otimes n-1} & \mathbf{F}^{\otimes n-1} \end{array} \right) \quad (3.57)$$

上式等价于：

$$\mathbf{x}_1 \oplus \mathbf{x}_2 = \mathbf{u}_1 \mathbf{F}^{\otimes n-1}, \quad (3.58)$$

$$\mathbf{x}_2 = \mathbf{u}_2 \mathbf{F}^{\otimes n-1}. \quad (3.59)$$

上面的两个式子就把长度 N 的编码问题分治为两个长度是 $N/2$ 的编码问题，所以复杂度为 $O(N \log_2 N)$ ，这是算法理论中的简单结论。上面两个式子的处理方式为：先解决式(3.59)，获得式(3.59)的结果后，把 \mathbf{x}_2 的结果带入式(3.58)。那么式(3.59)怎么解决？当然是重复分治算法，直到 \mathbf{x}_2 的值已知。下面举例说明。在下图中，请按红色数字的顺序阅读。每当一个分块中的等号左边的值完全已知，则停止分治。

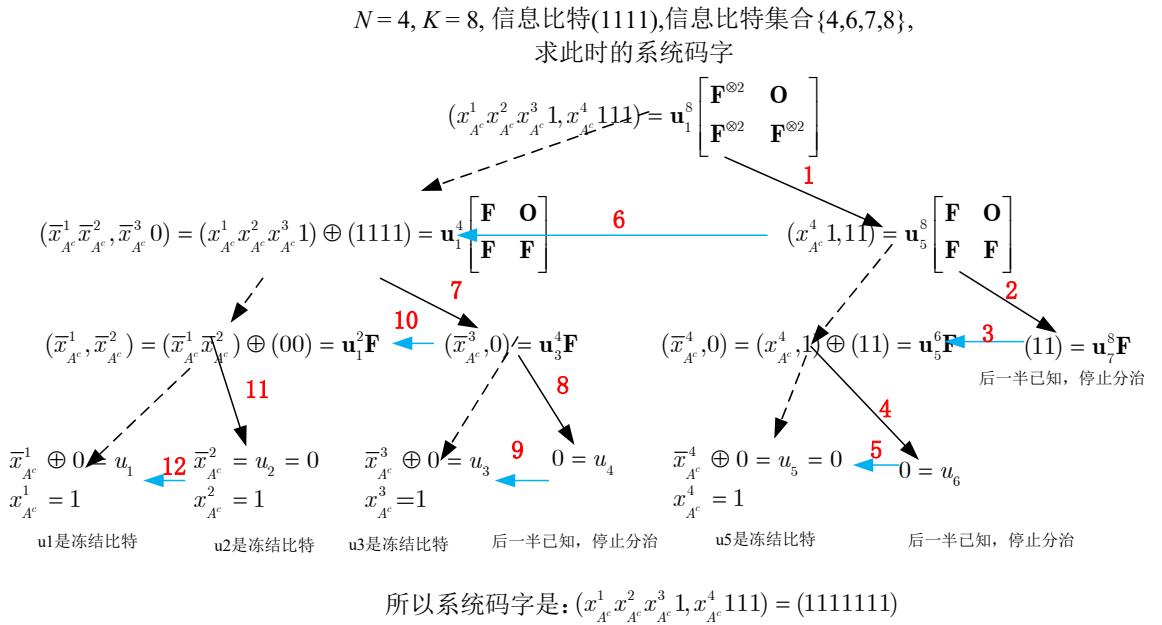


图 3.13 系统极化码递归编码举例

我们很容易就能看出上面算法的缺点：我们也不知道它什么时候会停止，只有走到了那一步才知道。所以这种“不确定”的结构也不会在实际中使用。实际中使用的是一种“固定”的编码方式。

方法2：两步编码法，来自文献[4][9][10]，这几篇文献本质上没有区别，因为研究早期，大家难免会重复一些普遍的结论。

这个方法基于极化信道的偏序关系，所以先讲这个偏序关系。我们注意到这些偏序关系的结论很简单，但是其证明并没有很简单，所以我在此就不抄袭它们的证明了。有兴趣的读者可以阅读我列出的文献，没兴趣的读者只掌握结论也不错。

定义 3.13 偏序关系1[1]

设 $N = 2^n$ ，极化信道顺序标号为 $W_N^{(i)}, 0 \leq i \leq N - 1$ 。十进制数 $0 \leq p, q \leq N - 1$ 的 n 位二进制展开有这样的特点： p 的 n 位二进制展是 $\langle p_n, \dots, p_s = 0, \dots, p_t = 1, \dots, p_1 \rangle_2$ ，其中左边是高位。交换 p_s 和 p_t 上的0和1，得到 $\langle p_n, \dots, p_s = 1, \dots, p_t = 0, \dots, p_1 \rangle_2$ 恰是 q 的二进制展开，则 $I(W_N^{(q)}) \geq I(W_N^{(p)})$ 。

简单地说，交换高位的0和低位的1会得到一个容量更大的极化信道。注意我没有说 $Z(W_N^{(q)})$ 和 $Z(W_N^{(p)})$ 之间存在任何关系。

定义 3.14 偏序关系2[4]

设 $N = 2^n$ ，极化信道顺序标号为 $W_N^{(i)}, 0 \leq i \leq N - 1$ 。十进制数 $0 \leq p, q \leq N - 1$ 的 n 位二进制展开有这样的特点： p 的 n 位二进制展是 $\langle p_n, \dots, p_s = 0, \dots, p_1 \rangle_2$ ，其中左边是高位。把 p_s 上的0变成1，得到 $\langle p_n, \dots, p_s = 1, \dots, p_1 \rangle_2$ 恰是 q 的二进制展开，则 $W_N^{(p)} \preceq W_N^{(q)}$ ，从而 $I(W_N^{(q)}) \geq I(W_N^{(p)})$ ， $Z(W_N^{(q)}) \leq Z(W_N^{(p)})$ ， $P_e^{\text{ML}}(W_N^{(q)}) \leq P_e^{\text{ML}}(W_N^{(p)})$ 。

简单地说，把某一位的0和变成1会得到一个更好的极化信道。这个结论比上一个结论强。

上面两个偏序关系说明这样的问题：如果某个 $W_N^{(i)}$ 入选集合 \mathcal{A} ，那么所有比 $W_N^{(i)}$ “更好”的极化信道必入选集合 \mathcal{A} ，否则你的构造就有问题；如果某个 $W_N^{(i)}$ 入未选集合 \mathcal{A} ，那么所有比 $W_N^{(i)}$ “更差”的极化信道必不能入选集合 \mathcal{A} ，否则你的构造就有问题。上面两个偏序关系把所有 N 个极化信道“部分”地联系了起来，所谓部分是指任取两个极化信道 $W_N^{(s)}$ 和 $W_N^{(t)}$ ，对于某些 s, t ，我们可以依据上面两个关系断定 $W_N^{(s)}$ 和 $W_N^{(t)}$ 哪个更好；而对于某些 s, t ，我们不能依据上面的两个关系断定哪个更好。

这两个偏序关系可以画成图形，随着码字的增长（每次增长从 N 变成 $2N$ ），表示偏序关系的图形呈现出递归的结构。这种表示偏序的图形你自己思考，可能非常复杂；但它事实上是非常简单的，有需要的读者可以阅读文献[7]，我在此就不复读了。

为了结论我的完整我把两种偏序关系写出来了，下面我们其实只用关系2。下面的结论主要来自文献[9][10]。

引理 3.11 设极化码构造满足偏序关系2： p 的 n 位二进制展是 $\langle p_n, \dots, p_s = 0, \dots, p_1 \rangle_2$ ， q 的 n 位二进制展是 $\langle p_n, \dots, p_s = 1, \dots, p_1 \rangle_2$ ，如果 p 入选信息集合 \mathcal{A} ，则 q 必入选。那么 $\mathbf{G}_{\mathcal{A}^c \setminus \mathcal{A}} = \mathbf{0}$ 。

证明 任取 $s \in \mathcal{A}^c, t \in \mathcal{A}$, 设 s, t 的位进制展开分别是 $< s_n, \dots, s_1 >_2, < t_n, \dots, t_1 >_2$ 。必存在 $1 \leq \alpha \leq n$ 使得 $s_\alpha = 0 < t_\alpha = 1$, 否则所有 $s_\alpha \geq t_\alpha$, 那么 s 应属于 \mathcal{A} , 与题设中的偏序关系 2 矛盾。然后, 由引理 3.9 可知, $\mathbf{G}_{s,t} = \prod_{\beta=1}^n F_{s_\beta, t_\beta}$, 对于其中 $\beta = \alpha$ 这一项, 因为 $F_{0,0} = 1, F_{0,1} = 0, F_{1,0} = 1, F_{1,1} = 1$ 且 $s_\alpha = 0 < t_\alpha = 1$, $F_{s_\alpha=0, t_\alpha=1} = 0$, 所以导致 $\mathbf{G}_{s,t} = 0$ 。证毕。

引理 3.12 设 \mathbf{y} 是任意 $1 \times N$ 维向量, A 是 $N \times N$ 维矩阵, 如果 $\mathbf{y}\mathbf{A} = \mathbf{y}$ 恒成立, 则 \mathbf{A} 是单位矩阵。

证明 用基 $\{\mathbf{e}_1, \dots, \mathbf{e}_N\}$ 验证。

定理 3.12 设极化码构造满足偏序关系 2: p 的 n 位二进制展是 $< p_n, \dots, p_s = 0, \dots, p_1 >_2$, q 的 n 位二进制展是 $< p_n, \dots, p_s = 1, \dots, p_1 >_2$, 如果 p 入选信息集合 \mathcal{A} , 则 q 必入选。那么 $\mathbf{G}_{\mathcal{A}\mathcal{A}} = \mathbf{G}_{\mathcal{A}\mathcal{A}}^{-1}$ 。

证明 因为 $\mathbf{GG} = \mathbf{I}_N$ 且 $\mathbf{x} = \mathbf{u}\mathbf{G}$, 所以 $\mathbf{u} = \mathbf{x}\mathbf{G}$, 对这个等号右边的项分块相乘, 有:

$$\mathbf{u} = (\mathbf{x}_{\mathcal{A}}, \mathbf{x}_{\mathcal{A}^c}) \left[\begin{array}{c|c} \mathbf{G}_{\mathcal{A}} & \\ \hline & \mathbf{G}_{\mathcal{A}^c} \end{array} \right] = \mathbf{x}_{\mathcal{A}}\mathbf{G}_{\mathcal{A}} + \mathbf{x}_{\mathcal{A}^c}\mathbf{G}_{\mathcal{A}^c}. \quad (3.60)$$

再取出 \mathbf{u} 位于 \mathcal{A} 中的元素, 有:

$$\mathbf{u}_{\mathcal{A}} = \mathbf{x}_{\mathcal{A}}\mathbf{G}_{\mathcal{A}\mathcal{A}} + \mathbf{x}_{\mathcal{A}^c}\mathbf{G}_{\mathcal{A}^c\mathcal{A}}. \quad (3.61)$$

由引理 3.11, $\mathbf{G}_{\mathcal{A}^c\mathcal{A}} = \mathbf{O}$, 所以 $\mathbf{u}_{\mathcal{A}} = \mathbf{x}_{\mathcal{A}}\mathbf{G}_{\mathcal{A}\mathcal{A}}$, 由式(3.54)有: $\mathbf{x}_{\mathcal{A}} = \mathbf{u}_{\mathcal{A}}\mathbf{G}_{\mathcal{A}\mathcal{A}}$, 即 $\mathbf{u}_{\mathcal{A}} = \mathbf{u}_{\mathcal{A}}\mathbf{G}_{\mathcal{A}\mathcal{A}}\mathbf{G}_{\mathcal{A}\mathcal{A}}$ 。因为 $\mathbf{u}_{\mathcal{A}}$ 的取值是任意的, 从而根据引理 3.12 有: $\mathbf{G}_{\mathcal{A}\mathcal{A}}\mathbf{G}_{\mathcal{A}\mathcal{A}} = \mathbf{I}_{|\mathcal{A}|}$ 。

由式(3.56)和上面的定理可知:

$$\mathbf{x}_{\mathcal{A}^c} = \mathbf{u}_{\mathcal{A}}\mathbf{G}_{\mathcal{A}\mathcal{A}}\mathbf{G}_{\mathcal{A}\mathcal{A}^c}. \quad (3.62)$$

上式说明了系统极化码可以通过两步非系统编码获得。第一步: 把信息比特放在 $\mathbf{u}_{\mathcal{A}}$ 上, 冻结比特全取 0, 计算 $\mathbf{v} = \mathbf{u}\mathbf{G}$ 。第二步, 令 $\mathbf{v}_{\mathcal{A}^c}$ 为零, $\mathbf{v}_{\mathcal{A}}$ 中的元素维持原样, 得到 \mathbf{v}' , 计算 $\mathbf{w} = \mathbf{v}'\mathbf{G}$, 取出 $\mathbf{w}_{\mathcal{A}^c}$, 则 $\mathbf{w}_{\mathcal{A}^c} = \mathbf{x}_{\mathcal{A}^c}$, 完成校验位 $\mathbf{x}_{\mathcal{A}^c}$ 的计算。与图 3.13 对应的一个算例如图 3.14 所示, 其中码长 $N = 8$, 信息比特数 $K = 4$, $\mathcal{A} = \{4, 6, 7, 8\}$, $\mathbf{u}_{\mathcal{A}} = (1111)$, 冻结比特全取 0。

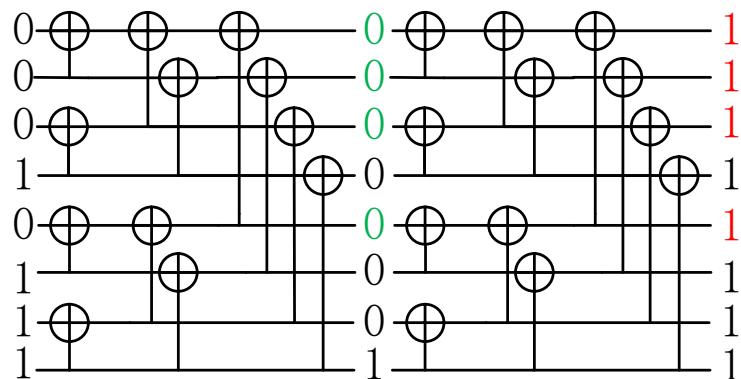


图 3.14 两步系统编码，绿色的零表示令 v_{A^c} 为零，红色元素表示 w_{A^c}

第4章 串行抵消译码

串行抵消（Successive Cancellation, SC）译码[3]是极化码的经典译码算法之一。对于长度为 2^n 的极化码，SC译码器收到接收信号，首先利用接收信号译码 u_1 ，然后利用利用接收信号和 u_1 的估计译码 u_2 ，利用接收信号和 u_1, \dots, u_{i_1} 的估计译码 u_i ，直到 u_N 为止。

本章介绍极化码的SC译码过程，先讲长度为2的极化码的SC译码，然后推广到长度为 2^n 的极化码，最后讲一些变体SC译码算法。本章仅叙述基于LLR的SC译码过程，因为二元输入信道中LLR是接收信号的充分统计量，数值稳定，且实际系统中也使用LLR译码。

本章给出了各种SC译码器的MATLAB代码，这些代码几乎可以不加修改地变成C语言代码。

§ 4.1 长度为2的极化码的SC译码过程

考虑联合分布 $\Pr(U_1, U_2, Y_1, Y_2)$ ，这些随机变量的关系为： U_1, U_2 是独立同分布的Bernoulli(0.5)随机变量， $(X_1, X_2) = (U_1, U_2)\mathbf{F} = (U_1 \oplus U_2, U_2)$ ， X_1, X_2 经过二元输入无记忆信道 W 的传输，分别形成接收信号 $Y_1 Y_2$ 。现在想用观测值 Y_1, Y_2 估计发送值 U_1, U_2 。

用条件概率 $\Pr(Y_1 Y_2 | U_1)$ 作为判决 U_1 的依据：如果 $\Pr(Y_1 Y_2 | U_1 = 0) \geq \Pr(Y_1 Y_2 | U_1 = 1)$ ，则将 U_1 判决为0；如果 $\Pr(Y_1 Y_2 | U_1 = 0) < \Pr(Y_1 Y_2 | U_1 = 1)$ ，则将 U_1 判决为1。 $\Pr(Y_1 Y_2 | U_1)$ 只不过

是 $\Pr(U_1, U_2, Y_1, Y_2)$ 的简单变体，它的计算过程如下：

$$\begin{aligned}\Pr(Y_1 Y_2 | U_1) &= \frac{\Pr(Y_1 Y_2 U_1)}{\Pr(U_1)} = \frac{\sum_{u_2 \in \{0,1\}} \Pr(Y_1 Y_2 U_1 U_2)}{\Pr(U_1)} \\ &= \frac{\sum_{u_2 \in \{0,1\}} \Pr(Y_1 Y_2 | U_1 U_2) \Pr(U_1 U_2)}{\Pr(U_1)} \stackrel{(a)}{=} \frac{1}{2} \sum_{u_2 \in \{0,1\}} \Pr(Y_1 Y_2 | X_1 X_2) \\ &\stackrel{(b)}{=} \frac{1}{2} \sum_{u_2 \in \{0,1\}} \Pr(Y_1 | X_1) \Pr(Y_2 | X_2) = \frac{1}{2} \sum_{u_2 \in \{0,1\}} \Pr(Y_1 | U_1 \oplus U_2) \Pr(Y_2 | U_2).\end{aligned}\quad (4.1)$$

其中等号(a)是因为对于任意 U_1, U_2 , $\Pr(U_1) = 0.5, \Pr(U_1 U_2) = 0.25$, 且 U_1, U_2 和 X_1, X_2 一一对应。等号(b)是因为 W 是无记忆信道。由此, 很容易计算对数似然比 $\ln \frac{\Pr(Y_1 Y_2 | U_1 = 0)}{\Pr(Y_1 Y_2 | U_1 = 1)}$:

$$\begin{aligned}\ln \frac{\Pr(Y_1 Y_2 | U_1 = 0)}{\Pr(Y_1 Y_2 | U_1 = 1)} &= \ln \frac{\sum_{u_2 \in \{0,1\}} \Pr(Y_1 | U_2) \Pr(Y_2 | U_2)}{\sum_{u_2 \in \{0,1\}} \Pr(Y_1 | 1 \oplus U_2) \Pr(Y_2 | U_2)} \\ &= \ln \frac{\Pr(Y_1 | 0) \Pr(Y_2 | 0) + \Pr(Y_1 | 1) \Pr(Y_2 | 1)}{\Pr(Y_1 | 1) \Pr(Y_2 | 0) + \Pr(Y_1 | 0) \Pr(Y_2 | 1)} \\ &= \ln \frac{\frac{\Pr(Y_1 | 0) \Pr(Y_2 | 0)}{\Pr(Y_1 | 1) \Pr(Y_2 | 1)} + 1}{\frac{\Pr(Y_2 | 0)}{\Pr(Y_2 | 1)} + \frac{\Pr(Y_1 | 0)}{\Pr(Y_1 | 1)}} \stackrel{(a)}{=} \ln \frac{1 + e^{L_1 + L_2}}{e^{L_1} + e^{L_2}}.\end{aligned}\quad (4.2)$$

等号(a)中 $L_1 = \ln \frac{\Pr(Y_1 | 0)}{\Pr(Y_1 | 1)}, L_2 = \ln \frac{\Pr(Y_2 | 0)}{\Pr(Y_2 | 1)}$ 是接收信号的LLR。典型信道的LLR有典型的形式。例如BSC(ϵ)中只有两种LLR, 分别是 $\ln \frac{1-\epsilon}{\epsilon}$ 和 $\ln \frac{\epsilon}{1-\epsilon}$; BEC(ϵ)的LLR有三个值, 分别是 $+\infty, 0, -\infty$, 虽然这三个值和 ϵ 无关, 但是 ϵ 越小, 取值为无穷的LLR就会越多, 译码越正确; 二元输入AWGN信道是连续输出信道, 如果使用BPSK调制 $s = 1 - 2x$, 其中 $x \in \{0, 1\}$ 是比特值, $s \in \{1, -1\}$ 是发送BPSK信号, $y = s + n$ 是接收信号, 其中 n 是均值为0、方差为 σ^2 的高斯加性白噪声, 则接收信号 y 的LLR的形式为 $\ln \frac{\frac{1}{\sigma\sqrt{2\pi}} e^{-(y-1)^2/(2\sigma^2)}}{\frac{1}{\sigma\sqrt{2\pi}} e^{-(y+1)^2/(2\sigma^2)}} = \frac{2}{\sigma^2}y$ 。有的人喜欢把BPSK写成 $s = 2x - 1$, 那接收LLR加个符号就行了: $-\frac{2}{\sigma^2}y$ 。

式(4.2)常在文献中被称为 f 运算。由于式(4.2)的计算非常复杂, 同时包括指数和对数运算, 不利于硬件实现, 我们常常对它进行近似, 一种常用的形式为:

$$\ln \frac{1 + e^{L_1 + L_2}}{e^{L_1} + e^{L_2}} \approx \text{sign}(L_1) \text{sign}(L_2) \min\{|L_1|, |L_2|\}. \quad (4.3)$$

其中 sign 表示取符号位。

由此我们就完成了 U_1 的译码: 我们一旦接收到 Y_1 和 Y_2 , 就计算 Y_1 和 Y_2 的对数似然比 L_1 和 L_2 , 把 L_1 和 L_2 带入式(4.2), 就得到了 $\ln \frac{\Pr(Y_1 Y_2 | U_1 = 0)}{\Pr(Y_1 Y_2 | U_1 = 1)}$ 。如果 U_1 是冻结比特, 我们直接把它判决为预

设的值；如果 $\ln \frac{\Pr(Y_1 Y_2 | U_1 = 0)}{\Pr(Y_1 Y_2 | U_1 = 1)} \geq 0$ ，说明 $\frac{\Pr(Y_1 Y_2 | U_1 = 0)}{\Pr(Y_1 Y_2 | U_1 = 1)} \geq 1$ ，那么 U_1 判决为0；反之， U_1 判决为1。

下面该判决 U_2 了，我们利用 U_1 的值判决 U_2 ，所考虑的条件概率是 $\Pr(Y_1 Y_2 U_1 | U_2)$ ，它只不过是 $\Pr(U_1, U_2, Y_1, Y_2)$ 的简单变体而已，下面的计算过程类比式(4.1)可得：

$$\begin{aligned} \Pr(Y_1 Y_2 U_1 | U_2) &= \frac{\Pr(Y_1 Y_2 U_1 U_2)}{\Pr(U_2)} = \frac{\Pr(Y_1 Y_2 | U_1 U_2) \Pr(U_1 U_2)}{\Pr(U_1)} \\ &= \frac{1}{2} \Pr(Y_1 Y_2 | U_1 U_2) = \frac{1}{2} \Pr(Y_1 Y_2 | X_1 X_2) \\ &= \frac{1}{2} \Pr(Y_1 | X_1) \Pr(Y_2 | X_2) = \frac{1}{2} \Pr(Y_1 | U_1 \oplus U_2) \Pr(Y_2 | U_2). \end{aligned} \quad (4.4)$$

上式与式(4.1)乍一看上去的区别是：式(4.1)比式(4.4)多了个求和号；式(4.4)需要 U_1 的值才能计算，幸运的是我们已经通过上面的过程已经估计出了 U_1 的值。

类比式(4.2)，可以得到对数似然比 $\ln \frac{\Pr(Y_1 Y_2 U_1 | U_2 = 0)}{\Pr(Y_1 Y_2 U_1 | U_2 = 1)}$ ：

$$\begin{aligned} \ln \frac{\Pr(Y_1 Y_2 U_1 | U_2 = 0)}{\Pr(Y_1 Y_2 U_1 | U_2 = 1)} &= \ln \frac{\Pr(Y_1 | U_1) \Pr(Y_2 | 0)}{\Pr(Y_1 | U_1 \oplus 1) \Pr(Y_2 | 1)} \\ &= \ln \frac{\Pr(Y_1 | U_1)}{\Pr(Y_1 | U_1 \oplus 1)} + \ln \Pr(Y_2 | 0) \Pr(Y_2 | 1) = (1 - 2U_1)L_1 + L_2. \end{aligned} \quad (4.5)$$

上式的最后一个等号是一种变通的表达形式，上式在文献中常被称为 g 运算。由此，我们完成了 U_2 的译码：在获得 L_1, L_2, U_1 后，带入上式，得到 $\ln \frac{\Pr(Y_1 Y_2 U_1 | U_2 = 0)}{\Pr(Y_1 Y_2 U_1 | U_2 = 1)}$ 的值，如果 U_2 是冻结比特，我们直接把它判决为预设的值；如果 $\ln \frac{\Pr(Y_1 Y_2 U_1 | U_2 = 0)}{\Pr(Y_1 Y_2 U_1 | U_2 = 1)} \geq 0$ ，说明 $\frac{\Pr(Y_1 Y_2 U_1 | U_2 = 0)}{\Pr(Y_1 Y_2 U_1 | U_2 = 1)} \geq 1$ ， U_2 判决为0；反之， U_2 判决为1。

下面我给出一套完成的编译码流程，它虽然简单，但却具有代表性，不然我也就不写了。我们考虑在AWGN信道中进行长度为2的极化码的传输，信息比特的集合为 $\mathcal{A} = \{2\}$ ，信息比为1，冻结比特总是为0。二元输入AWGN信道的信噪比为 $E_b/N_0 = 4\text{dB}$ ，其中 $N_0 = 2\sigma^2$ 是加性高斯白噪声的单边功率谱密度。

第一步：给出信息比特，本例中信息比特 $U_2 = 1$ 。

第二步：计算码字，本例中 $(U_1, U_2) = (01)$ ，则 $(X_1, X_2) = (11)$ ，其实是个重复码。

第三步：BPSK调制， $(X_1, X_2) = (11)$ 经过BPSK调制后变成 $(s_1, s_2) = (-1, -1)$ 。

第四步：加噪声，噪声方差的计算公式为 $\frac{E_s}{RN_0} \stackrel{(a)}{=} \frac{1}{2R\sigma^2} = \frac{E_b}{N_0}$ ，从而 $\sigma = \frac{1}{\sqrt{2R}} \frac{1}{\sqrt{E_b/N_0}}$ 。这个式子中的值都是线性值，如果 E_b/N_0 是分贝值，则需转化为线性值： $\sigma = \frac{1}{\sqrt{2R}} \frac{1}{\sqrt{E_b/N_0}} = \frac{1}{\sqrt{2R}} \frac{1}{\sqrt{10^{(E_b/N_0)/10}}} = \frac{1}{\sqrt{2R}} 10^{-(E_b/N_0)/20}$ 。其中等号(a)是因为 E_s 已经归一化为1， R 是码率，本例

中为 $R = 0.5$ 。由此可得 $\sigma = 10^{-4/20}$ 。用程序生成两个独立的零均值、方差为 σ^2 的高斯随机变量当作噪声，生成的两个噪声值为 $(n_1, n_2) = (0.3392, 1.1571)$ 。

第五步：获得接收信号 $(y_1, y_2) = (-1 + n_1, -1 + n_2) = (-0.6608, 0.1571)$ ，从而 $(L_1, L_2) = (\frac{2}{\sigma^2}y_1, \frac{2}{\sigma^2}y_2) = (-3.3195, 0.7893)$ 。

第六步：SC译码，首先判决 U_1 ，这里使用近似公式(4.3)，则 U_1 对应的对数似然比为 $\text{sign}(L_1)\text{sign}(L_2)\min\{|L_1|, |L_2|\} = -0.7893$ ，但是 U_1 是冻结比特，所以我们直接令 $U_1 = 0$ 。然后判决 U_2 ，由式(4.5)， U_2 对应的LLR为 $(1 - 2U_1)L_1 + L_2 = -2.5302$ ，所以 U_2 判决为1，译码结束，译码正确。

把上述过程画在图上，就有了图4.1，编码过程从左往右看，译码过程从右往左看。这个图可以简单描述为：信道接收LLR放在右边，左上角用 f 函数判决 u_1 ，右下角用 g 函数判决 u_2 。

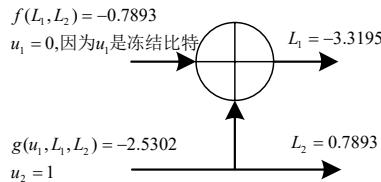


图 4.1 长度为2的极化码的译码举例

§ 4.2 长度为 2^n 的极化码的SC译码过程

通过上一章的叙述我们知道了长度为 2^n 的极化码只不过是用 $n2^{n-1}$ 个 2×2 基本极化码模块拼凑起来而已，可以推测，长度为 2^n 的极化码的SC译码也只不过是用 $n2^{n-1}$ 个 2×2 基本极化码模块拼凑起来而已。上一节的叙述可以不加修改地推广到长度为 2^n 的极化码的SC译码中。我们直接用一个码长 $N = 8$ 的例子看看一般的SC译码过程是怎样的。

例 4.1 配置如下(AWGN信道传输， $E_b/N_0 = 4\text{dB}$)：

码长 $N = 8$ ，信息比特数 $K = 4$ ，码率 $R = K/N = 0.5$ 。

信息比特集合 $\mathcal{A} = \{4, 6, 7, 8\}$ ，冻结比特全部取0。

4个信息比特为(1111)，则 $\mathbf{u}_1^8 = (00010111)$ ，编码后 $\mathbf{x}_1^8 = \mathbf{u}_1^8 \mathbf{F}^{\otimes 3} = (01101001)$ 。

\mathbf{x}_1^8 对应的BPSK序列为 $\mathbf{s}_1^8 = (1, -1, -1, 1, -1, 1, 1, -1)$

用MATLAB随机生成一个噪声序列 $\mathbf{n}_1^8 = (-1.4, 0.5, 0.2, -0.8, -0.3, 0.2, 2.3, 1.7)$

接收信号为 $\mathbf{y}_1^8 = \mathbf{s}_1^8 + \mathbf{n}_1^8 = (-0.4, -0.5, -0.8, 0.2, -1.3, 1.2, 3.3, 0.7)$

接收对数似然比为 $\mathbf{L}_1^8 = \frac{2}{\sigma^2}\mathbf{y}_1^8 = (-2.0, -2.5, -4.0, 1.0, -6.5, 6.0, 16.6, 3.5)$ 。

下面图片给出了所有计算过程（ f 运算使用近似公式(4.3)），该计算过程是长度为2的极化码的SC译码的推广，与上一节的不同之处仅在于“每个 2×2 模块的引脚伸得不一样长”，每个 2×2 模块在计算时所使用的数据是它引脚处的数据。本例中译码结果是正确结果。

例 4.2 在熟练掌握图4.2后，可以轻易画出另一种图形表示，如图4.3。图4.3是二叉树，所使用的一切数据与图4.2一致，每个步骤也是完全对应的。二叉树中每个节点对应两个数组，一个数组是该节点的LLR数据，另一个数组是该节点对应的比特值。

通过这两张大图的描述，我想你已经完全掌握了码长为8时的SC译码的运算规律。不难把这种规律推广到一般情况，如图4.4所示。图4.4中的运算规则对照图4.2和4.3，不证自明：当你向下计算子节点的LLR时，需要跨越母节点长度的一半来选取两个被运算的LLR值；当你向上返回比特值时，“ $u_1 \oplus u_1$ ”与“ u_2 ”的距离是母节点长度的一半。看懂图4.4你就会明白，SC过程中每一步运算的规律完全一致，只不过是运算数据的长度不同而已。

有了一般的运算规律我们就可以编制程序，下面的递归形式的代码简明表达了SC译码过程。但是我们不推荐在仿真程序中使用递归，因为递归的开销太大，程序运行缓慢。在下面的程序中，llr是图4.4中的 α ，x是图4.4中的 β ，alpha_left是图4.4中的 α^l ，alpha_right是图4.4中的 α^r ，beta_left是图4.4中的 β^l ，beta_right是图4.4中的 β^r ，u是信息比特。

```

1 function [u, x] = recursive_SC_decoder(llr, frozen_bits)
2 N = length(llr);
3 if N == 1
4     if frozen_bits == 0 % info bit
5         u = llr < 0;%llr < 0, u = 1; llr >= 0, u = 0;
6     else % frozen bit — assumed to be zero
7         u = 0;
8     end
9     x = u;
10 else
11     alpha_left = f(llr(1 : N/2), llr(N/2 + 1 : N));
12     [u_left_child, x_left_child] = ...
13     recursive_SC_decoder(alpha_left, frozen_bits(1 : N/2));
14     alpha_right = g(x_left_child, llr(1 : N/2), llr(N/2 + 1 : N));
15     [u_right_child, x_right_child] = ...
16     recursive_SC_decoder(alpha_right, frozen_bits(N/2 + 1 : N));
17     u = [u_left_child u_right_child];
18     x = zeros(1, N);

```

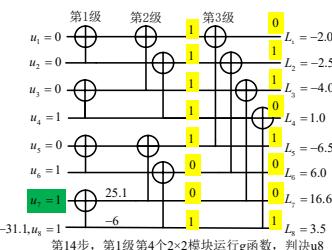
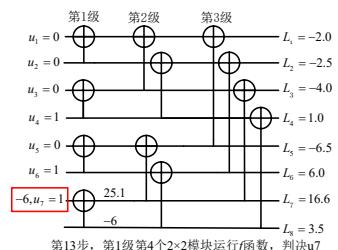
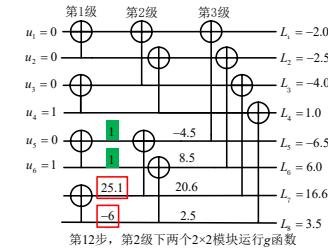
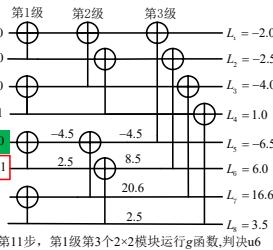
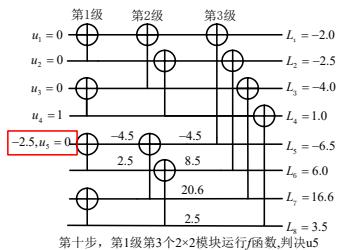
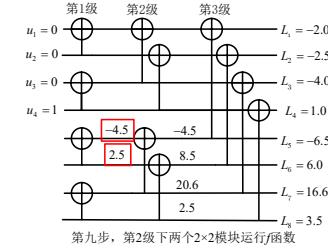
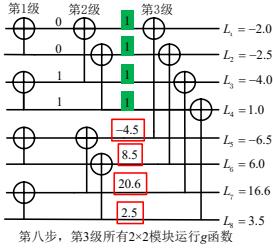
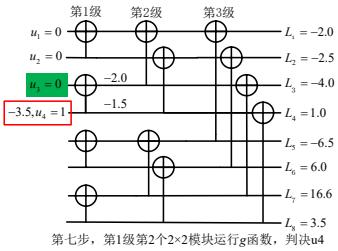
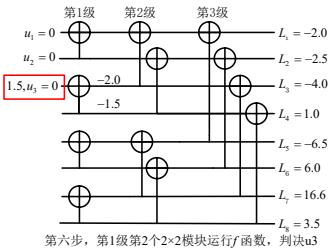
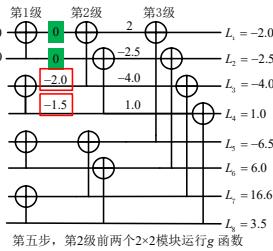
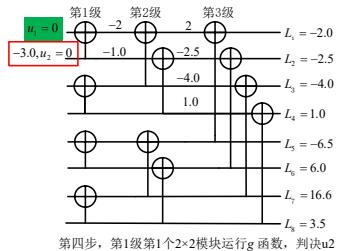
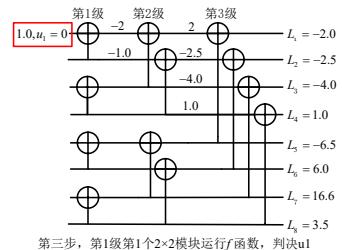
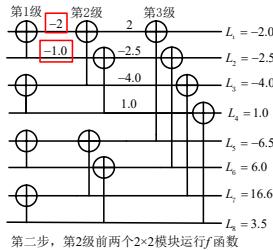
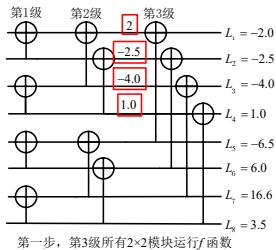


图 4.2 基于编码图表示的SC译码举例, 其中 u_1, u_2, u_3, u_5 是取值为0的冻结比特, 总是判决为0, 红框中的数字表示当前步骤刚算出的数字, 绿框中的比特表示当前步骤中 g 运算所需的比特值, 黄框表示最后一步顺带算出码字

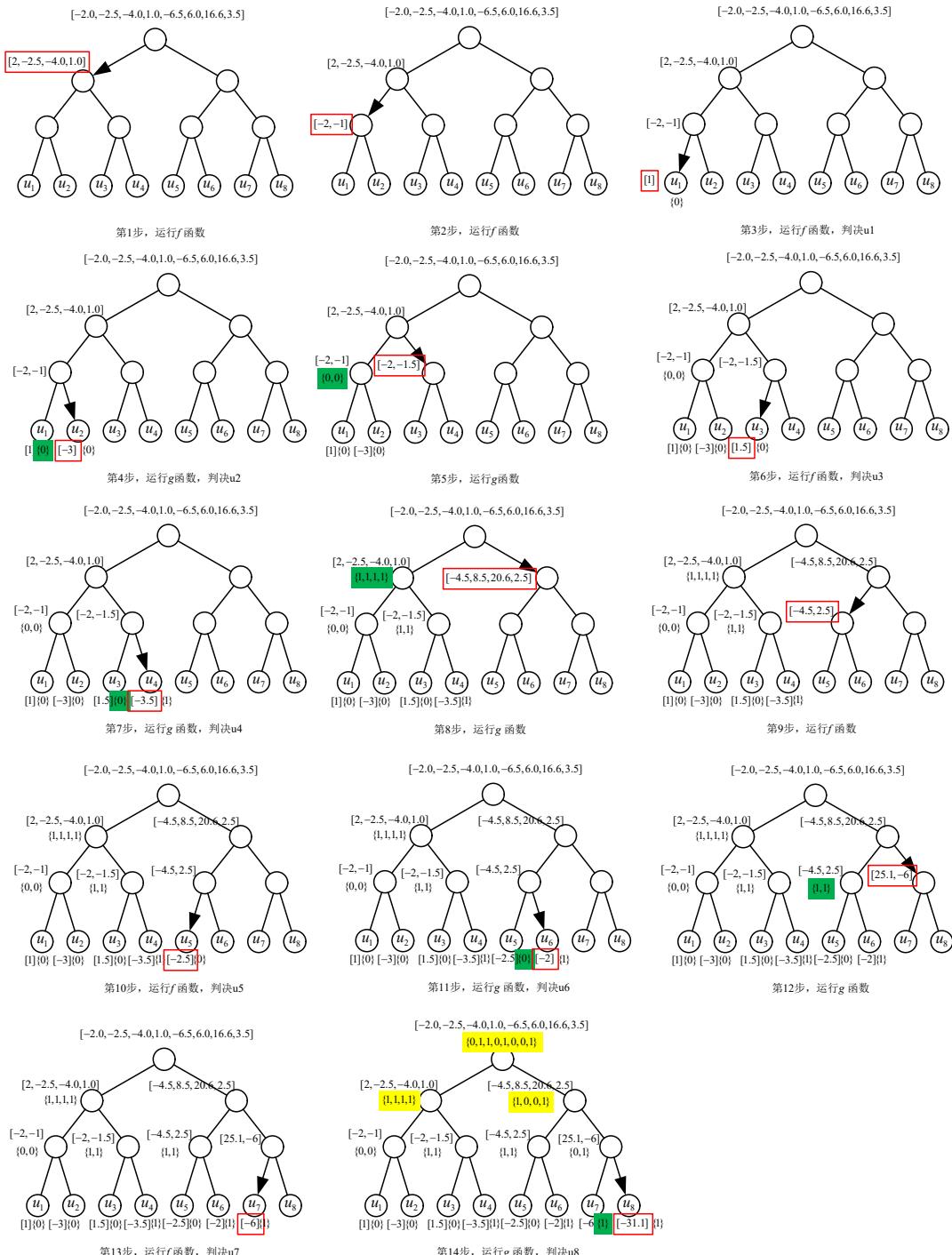


图 4.3 基于二叉树表示的SC译码举例，其中 u_1, u_2, u_3, u_5 是取值为0的冻结比特，总是判决为0，红框中的数字表示当前步骤刚算出的数字，绿框中的比特表示当前步骤中 g 运算所需的比特值，黄框表示最后一步顺带算出码字，中括号内为LLR，大括号内为比特

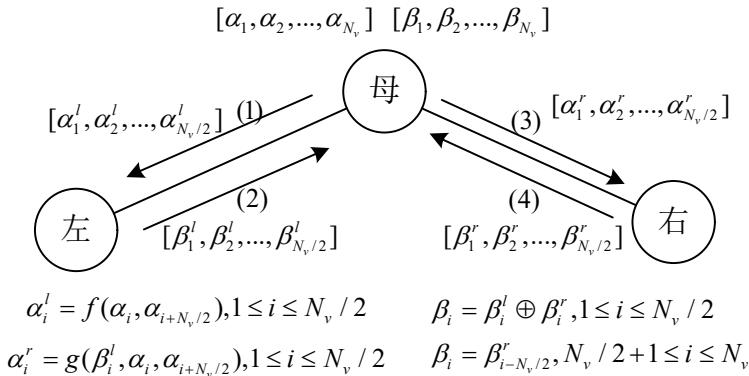


图 4.4 SC译码的二叉树表示中每个节点的一般运算规律。母节点数据的长度是 N_v , 子节点数据长度是 $N_v/2$ 。母节点的LLR和比特值分别用 α 和 β 表示, 左孩子的LLR和比特值分别用 α^l 和 β^l 表示, 右孩子的LLR和比特值分别用 α^r 和 β^r 表示。如果左孩子节点是叶节点(代表信息比特), 则其仅接收到一个LLR: α_1^l , 硬判决 α_1^l 得到 β_1^l , 将 β_1^l 返回母节点; 如果右孩子节点是叶节点(代表信息比特), 则其仅接收到一个LLR: α_1^r , 硬判决 α_1^r 得到 β_1^r , 将 β_1^r 返回母节点。括号里的数字是计算的顺序

```

19     x(1 : N/2) = mod(x_left_child + x_right_child, 2);
20     x(N/2 + 1 : N) = x_right_child;
21 end
22 end
23 function z = f(x,y)%f function
24 z = sign(x) .* sign(y) .* min(abs(x), abs(y));
25 end
26 function z = g(u, x, y)%g function
27 z = (1 - 2 * u) .* x + y;
28 end

```

下面我们给出一个不用递归的SC译码器。它的代码比上一个算法更长, 但是执行起来却更快。下面的代码有几个输入参数需要解释:

其中lambda_offset是一个“分段”用的向量, 它的长度是 $n+1$, 元素的值依次是 $2^0, 2^1, \dots, 2^{\log_2 N} = n$. lambda_offset中的值给 P 和 C 这两个存储中间计算结果的向量分了段。

llr_layer_vec是一个表示“LLR计算实际执行层数”的向量, 它是一个 $N \times 1$ 维向量, 里面的元素是取值于 $[0, n - 1]$ 内的整数。llr_layer_vec中的第 $i, 1 \leq i \leq N$ 个元素llr_layer_vec(i)表示: $i - 1$ 的二进制展开中“低位连续0的个数”, 也就是 $i - 1$ 的质数展开中2的次数, 也就是 $i - 1$ 能连续整除2的次数。例如, 当 $n = 3$ 时, llr_layer_vec(5) = 2。由于0是特例, 0的二进制

展开全是0，所以我们不计算它，即llr_layer_vec(1)总是初始值0，这个值也不参与后续计算。

bit_layer_vec是一个表示“比特值返回时实际执行层数”的向量，它是一个 $N \times 1$ 维向量，里面的元素是取值于 $[0, n - 2]$ 内的整数。bit_layer_vec中的第*i*, $1 \leq i \leq N$ 个元素bit_layer_vec(*i*)表示：*i*-1的二进制展开中“低位连续1的个数减1”。例如，当*n* = 3时，llr_layer_vec(8) = 2。

当*N* = 2^n 给定之后，lambda_offset、llr_layer_vec和bit_layer_vec的值就确定了，只需计算一遍，存起来即可。

程序中开辟的数组P用来存储LLR的中间计算结果，数组C用于存储比特值的中间计算结果。

可能你会觉得这个程序太难读了，这个程序之后，我会把每一步的计算结果全部用图画出来。

```

1 lambda_offset = 2.^0 : log2(N);
2 function u = SC_decoder(llr , frozen_bits , K, ...
3   lambda_offset , llr_layer_vec , bit_layer_vec)
4 N = length(llr);%llr refers to channel LLR.
5 n = log2(N);
6 P = zeros(N - 1 , 1);%channel llr is not include in P.
7 %P stores internal LLRs, P(1) is used for decision.
8 C = zeros(2 * N - 1 , 2);%C stores internal bit values
9 u = zeros(K, 1);%record information bits
10 cnt_u = 1;%a counter used by vector u
11 for phi = 0 : N - 1%decode each u_phi
12   switch phi
13     case 0%for decoding u_1
14       index_1 = lambda_offset(n);
15       for beta = 0 : index_1 - 1%use llr vector
16         P(beta + index_1) =...
17           sign(llr(beta + 1)) * sign(llr(beta + 1 + index_1)) *...
18           min(abs(llr(beta + 1)), abs(llr(beta + 1 + index_1)));
19       end
20       for i_layer = n - 2 : -1 : 0%use P vector
21         index_1 = lambda_offset(i_layer + 1);
22         index_2 = lambda_offset(i_layer + 2);
23         for beta = index_1 : index_2 - 1
24           P(beta) =...

```

```
25         sign(P(beta + index_1)) * sign(P(beta + index_2)) *...
26         min(abs(P(beta + index_1)), abs(P(beta + index_2)));
27     end
28 end
29 case N/2%for decoding u_{N/2 + 1}
30     index_1 = lambda_offset(n);
31     for beta = 0 : index_1 - 1%use llr_vector. g function.
32         P(beta + index_1) =...
33             (1 - 2 * C(beta + index_1, 1)) *...
34             llr(beta + 1) + llr(beta + 1 + index_1);
35     end
36     for i_layer = n - 2 : -1 : 0%use P_vector. f function
37         index_1 = lambda_offset(i_layer + 1);
38         index_2 = lambda_offset(i_layer + 2);
39         for beta = index_1 : index_2 - 1
40             P(beta) =...
41                 sign(P(beta + index_1)) * sign(P(beta + index_2)) *...
42                 min(abs(P(beta + index_1)), abs(P(beta + index_2)));
43         end
44     end
45 otherwise
46     llr_layer = llr_layer_vec(phi + 1);
47     index_1 = lambda_offset(llr_layer + 1);
48     index_2 = lambda_offset(llr_layer + 2);
49     for beta = index_1 : index_2 - 1%g function is first implemented.
50         P(beta) =...
51             (1 - 2 * C(beta, 1)) *...
52             P(beta + index_1) + P(beta + index_2);
53     end
54     for i_layer = llr_layer - 1 : -1 : 0%then f function.
55         index_1 = lambda_offset(i_layer + 1);
56         index_2 = lambda_offset(i_layer + 2);
57         for beta = index_1 : index_2 - 1
58             P(beta) =...
59                 sign(P(beta + index_1)) * sign(P(beta + index_2)) *...
```

```

60           min(abs(P(beta + index_1)), abs(P(beta + index_2)));
61       end
62   end
63 end
64 phi_mod_2 = mod(phi, 2);
65 if frozen_bits(phi + 1) == 1%frozen bitpartialorder
66     C(1, 1 + phi_mod_2) = 0;
67 else%information bit
68     u_phi = P(1) < 0;%decision
69     C(1, 1 + phi_mod_2) = u_phi;%store internal bit values
70     u(cnt_u) = u_phi;%record decoding results
71     cnt_u = cnt_u + 1;
72 end
73 if phi_mod_2 == 1%whether to perform bit recursion ?
74     bit_layer = bit_layer_vec(phi + 1);
75     for i_layer = 0 : bit_layer - 1%give values to the 2nd column of C
76         index_1 = lambda_offset(i_layer + 1);
77         index_2 = lambda_offset(i_layer + 2);
78         for beta = index_1 : index_2 - 1
79             C(beta + index_1, 2) = mod(C(beta, 1) + C(beta, 2), 2);
80             C(beta + index_2, 2) = C(beta, 2);
81         end
82     end
83     index_1 = lambda_offset(bit_layer + 1);
84     index_2 = lambda_offset(bit_layer + 2);
85     for beta = index_1 : index_2 - 1%give values to the 1st column of C
86         C(beta + index_1, 1) = mod(C(beta, 1) + C(beta, 2), 2);
87         C(beta + index_2, 1) = C(beta, 2);
88     end
89 end
90 end
91 end

```

```
1 function layer_vec = get_llr_layer(N)
```

```

2 layer_vec = zeros(N , 1);
3 for phi = 1 : N - 1
4     psi = phi;
5     layer = 0;
6     while(mod(psi , 2) == 0)
7         psi = floor(psi/2);
8         layer = layer + 1;
9     end
10    layer_vec(phi + 1) = layer;
11 end
12 end

```

```

1 function layer_vec = get_bit_layer(N)
2 layer_vec = zeros(N , 1);
3 for phi = 0 : N - 1
4     psi = floor(phi/2);
5     layer = 0;
6     while(mod(psi , 2) == 1)
7         psi = floor(psi/2);
8         layer = layer + 1;
9     end
10    layer_vec(phi + 1) = layer;
11 end
12 end

```

下面的图4.5展示了当 $N = 8$ 时，上面的算法的所有执行过程和结果，所使用的数据和前面的例子完全一致。其中，颜色相同的方块具有同等的意义，红色框表示当前步骤中刚计算出来的结果，橘黄色框表示当前步骤中 g 函数使用的比特值。可以看到，数组C右下方的连续8个黄方块一直没有被使用，它确实浪费了存储。

上面几个图片中的计算步骤的数量无一例外的都是 $2 \times 8 - 2 = 14$ 个。推而广之，当码长为 $N = 2^n$ 时，SC算法的执行步骤（或者说是时钟数，也可以说是译码时延）是 $2+4+\dots+N=2N-2$ ，其中 f 函数和 g 函数各占 $N-1$ 。注意，此时无“硬件资源限制”，意思是至少有 $N/2$ 个运算器，使得一个时钟内可以同时执行 $N/2$ 个 f （或 g ）函数。这对应于第 n 级的 $N/2$ 个 2×2 模块，因为只有第 n 级的 $N/2$ 个 2×2 模块需要一次性计算 $N/2$ 个 f （或 g ）函数，其余的 $n-1$ 级都不需要这么多（从右往左数，每一级所需的LLR计算次数依次为 $N/2, N/4, \dots, 2, 1$ ）。所以，

给码长为 N 的极化码设置 $N/2$ 个运算器事实上是浪费了资源，因为大多数情况下都不需要这么多。我们不考虑中间比特值计算的时延，因为我们认为比特异或很容易实现。

如果只有 P 个运算器， $P < N/2$ 且 P 是2的幂。例如，第 n 级在计算 f 或 g 函数的时候需要算 $N/2$ 次，现在只有 P 个运算器，一个时钟固定计算 P 次，显然需要 $N/(2P)$ 个时钟才能完成 $N/2$ 次计算。随着SC译码逐渐接近二叉树的叶节点，每次所需计算的 f 或 g 函数的次数也在减少。当某层所需计算的 f 或 g 函数的次数小于等于 P 时，那么一个时钟就能完成所需的计算量。依照此规律读者可以推算，SC译码器的时延如下。如果对此有疑问，可以阅读[22]：

$$L_{\text{SC}}(N, P) = 2N + \frac{N}{P} \log_2 \frac{N}{4P} \quad (4.6)$$

特例1： $P = N/2$, $L(N, P) = 2N - 2$, 符合上面的分析结果。

特例2： $P = 1$, $L(N, P) = N \log_2 N$, 这个结果显然正确。

SC算法所需的存储已经清楚地表示在图4.5中了。图4.5中的存储数组C略有冗余，比如C数组右下角的8个黄色方块始终未使用，你也可以略微改动数组的大小使得每个存储都物尽其用。之所以画成这样，是因为这样程序写起来比较规整。

到此，我们讲完了SC的标准（或者说是传统）执行过程。

§ 4.3 快速SC译码算法

如果你不是随便翻翻，你应该已经注意到上一节的算法里存在大量的冗余计算，比如某些比特是冻结比特，它的判决值不依赖LLR，但我们还是算出了它的LLR，这是没有必要的。本节我们试图在SC算法执行到半截的时候就想办法中止它，同时还能得到码字比特的估计，这就是快速SC译码算法的核心思想。这种思想在远古时代早已有之，最近的发掘是文献[14] [15]。

我们先观察码字比特的中间计算值的关系，如图4.6所示，左边是编码图形式，右边是二叉树形式，“同样的框”圈出来的部分等价。其中最重要的规律是：二叉树形式中的每一个子树都是一个极化码，只不过是码长变短了，例如右侧黄色框就是一个长为4个极化码。子二叉树对应的极化码我们称之为“子极化码”。

如果你熟练掌握了上一节中的SC运算，你就会清楚地认识到，长度为 N_v 子极化码的“根节点”会接收到长度 N_v 的LLR序列，子极化码的译码规律和整个极化码的规律完全相同，只不过数据长度变短了。我们现在想做的事是：对某个子极化码，不用SC译码，直接估计这个子极化码的所有码字比特，注意是码字比特。下面我们定义四种特殊的极化码。下面四种码字是最基本的分类，在较为前沿的研究中分类早已不止四种，有兴趣的读者可以参考文献[16]。

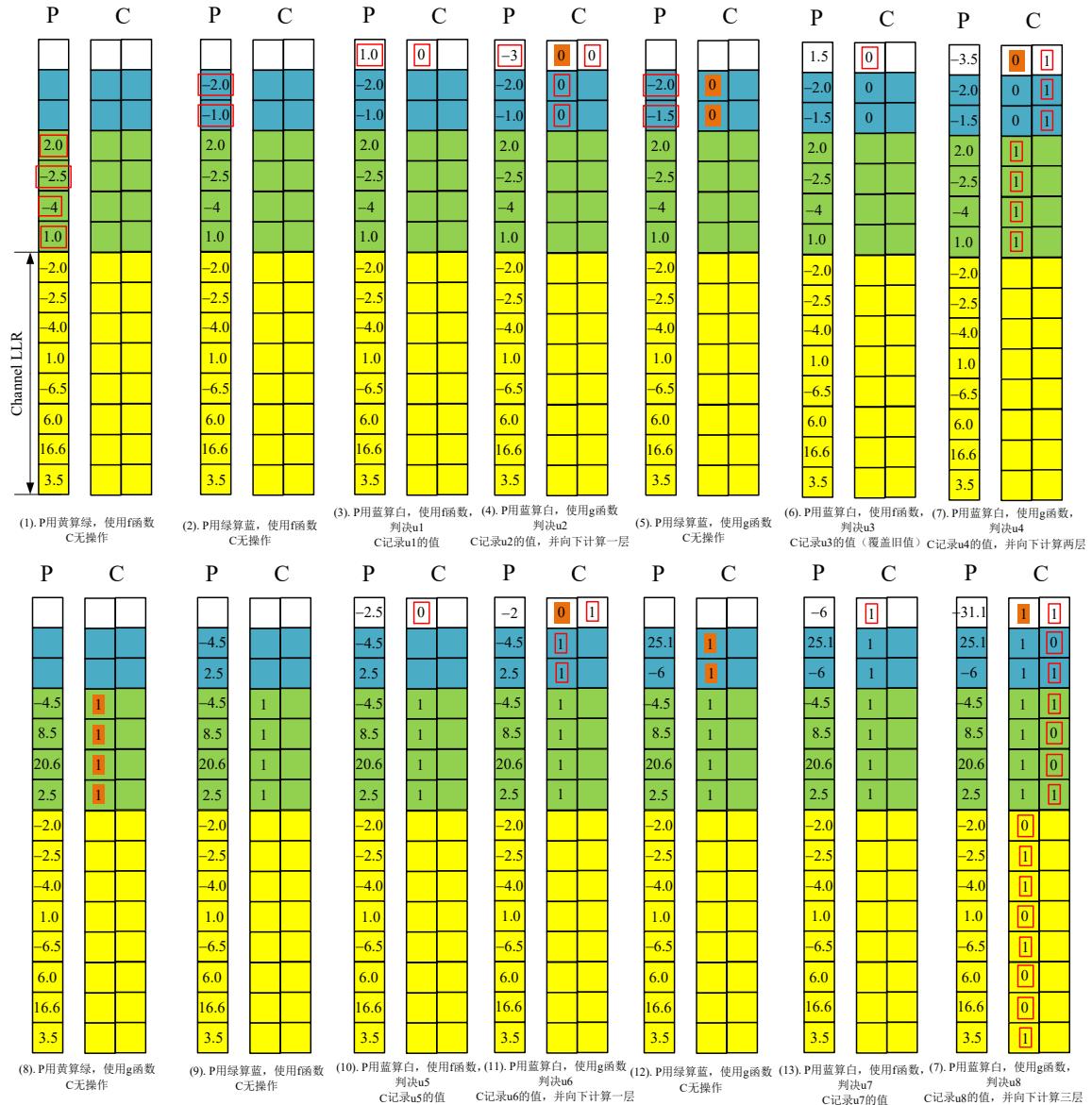
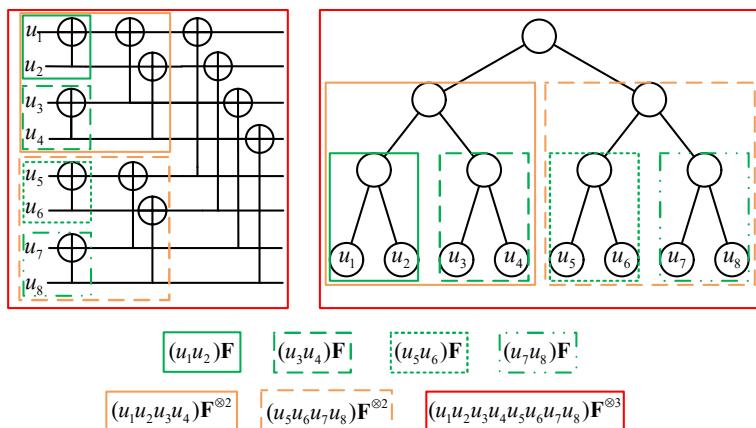


图 4.5 不使用递归的算法的执行过程

图 4.6 码字比特的中间计算结果在编码形式和二叉树形式中的对比，码长 $N = 2^3$

定义 4.1 码率零 (Rate 0, R0)

如果一个长为 $N = 2^n$ 的极化码没有信息比特，只有冻结比特，则称为R0极化码。这样的极化码不携带信息，当然也不会在系统中使用，但是却有可能作为一个子极化码出现，例如一个冻结比特就是一个平凡的R0极化码。一个长度为8的R0极化码的例子如图4.7所示，其中白色节点代表其所有叶节点都是冻结比特。

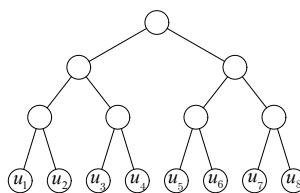


图 4.7 长度为8的R0极化码

定义 4.2 重复码 (Repetition, Rep)

如果一个长为 $N = 2^n$ 的极化码只有 u_N 是信息比特，其余 u_1, u_2, \dots, u_{N-1} 是冻结比特，则该极化码为重复码，即所有码字比特相等。这是显然的，因为此时的生成矩阵为全一行向量。一个长度为8的重复极化码的例子如图4.8所示，其中白色节点代表其所有叶节点都是冻结比特，黑色节点代表其所有叶节点都是信息比特，灰色节点表示其叶节点既包括冻结比特也包括信息比特。

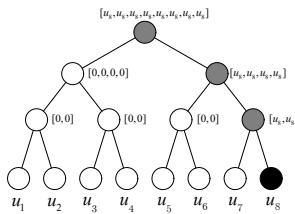


图 4.8 长度为8的重复极化码

定义 4.3 单偶校验码 (Single Parity Check, SPC)

如果一个长为 $N = 2^n$ 的极化码只有 $u_1 = 0$ 是冻结比特，其余 u_2, u_3, \dots, u_N 都是信息比特，则该极化码为SPC，即所有码字比特的和为0。

下面的定理保证了满足上述定义的码字的所有码字比特的和为零。

定理 4.1 如果一个长为 $N = 2^n$ 的极化码只有 $u_1 = 0$ 是冻结比特，其余 u_2, u_3, \dots, u_N 都是信息比特，则对于任意的该极化码的码字，所有码字比特的和为0。

证明 这是一个简单的结论，我们使用对 n 用归纳法。

源头：当 $n = 1$ ，信源为 $(0, u_2)$ ， $(0, u_2)\mathbf{F} = (u_2, u_2)$ ，显然无论 u_2 取值为何，有 $u_2 \oplus u_2 = 0$ 。

归纳假设：当 $n = k$ 时，只有 $u_1 = 0$ 是冻结比特，其余 u_2, u_3, \dots, u_{2^k} 都是信息比特，则对于任意的该极化码的码字，所有码字比特的和为0。

下一步：当 $n = k + 1$ 时，信源序列可以写为 $(0, u_2, u_3, \dots, u_{2^k} | u_{2^k+1}, \dots, u_{2^{k+1}})$ ，其中竖线表示把信源序列分为前一半和后一半， $u_2, u_3, \dots, u_{2^k+1}$ 的取值任意。对于任意一个满足题设条件的码长为 2^{k+1} 的码字 \mathbf{x}

$$\mathbf{x} = (0, u_2, u_3, \dots, u_{2^k} | u_{2^k+1}, \dots, u_{2^{k+1}}) \begin{bmatrix} \mathbf{F}^{\otimes k} & \mathbf{O} \\ \mathbf{F}^{\otimes k} & \mathbf{F}^{\otimes k} \end{bmatrix} \quad (4.7)$$

$$= ((0, u_2, u_3, \dots, u_{2^k})\mathbf{F}^{\otimes k} \oplus (u_{2^k+1}, \dots, u_{2^{k+1}})\mathbf{F}^{\otimes k} | (u_{2^k+1}, \dots, u_{2^{k+1}})\mathbf{F}^{\otimes k}).$$

把上式中所有比特求和，两个相同的 $(u_{2^k+1}, \dots, u_{2^{k+1}})\mathbf{F}^{\otimes k}$ 部分抵消，剩下 $(0, u_2, \dots, u_{2^k})\mathbf{F}^{\otimes k}$ ，此时根据归纳假设，这一部分的和为0，证毕。

一个长度为8的SPC码的例子如图4.9所示，其中白色节点代表其所有叶节点都是冻结比特，黑色节点代表其所有叶节点都是信息比特，灰色节点表示其叶节点既包括冻结比特也包括信息比特。

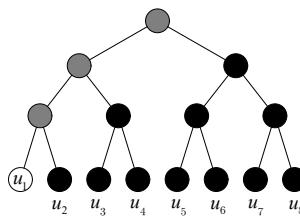


图 4.9 长度为8的SPC码

定义 4.4 码率一 (Rate 1, R1)

如果一个长为 $N = 2^n$ 的极化码所有信源比特都是信息比特，则该极化码为R1码。此时码率为 $R = 1$ ，信源序列和码字序列的数量都是 2^n 个，等价于没有编码。

一个长度为8的R1C码的例子如图4.10所示，其中黑色节点代表其所有叶节点都是信息比特。

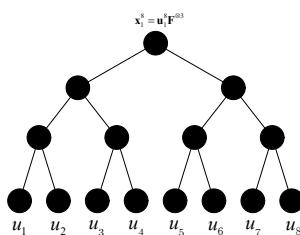


图 4.10 长度为8的R1码

上述四种极化码存在快速最大似然译码方法，使得整个译码能在一个时钟内完成，从而不必再用时延较大的SC译码过程。上面四种码字中最简单的情况是R0码，因为它的值确定，根本无需译码。下面分别给出Rep、SPC和R1码的最大似然译码方法。

定理 4.2 Rep码的最大似然译码。

发送长度为 $N = 2^n$ 的Rep码，经过转移概率为 $\Pr(y|x), x \in \{0, 1\}$ 的无记忆信道传输，接收信号为 (y_1, \dots, y_N) ，则其最大似然译码方法如下：

$$S = \sum_{i=1}^N \ln \frac{\Pr(y_i|0)}{\Pr(y_i|1)} \quad (4.8)$$

S 就是接收对数似然比的和。如果 $S \geq 0$ ，则Rep码译码为全0序列，否则译码为全1序列。

证明

$$S = \sum_{i=1}^N \ln \frac{\Pr(y_i|0)}{\Pr(y_i|1)} = \ln \frac{\prod_{i=1}^N \Pr(y_i|0)}{\prod_{i=1}^N \Pr(y_i|1)} = \ln \frac{\Pr(\mathbf{y}_1^N | \mathbf{0}_1^N)}{\Pr(\mathbf{y}_1^N | \mathbf{1}_1^N)} \quad (4.9)$$

$\mathbf{1}_1^N$ 和 $\mathbf{0}_1^N$ 分别是全1向量和全0向量，最后一个等号是因为信道是无记忆的。由上式可以判决究竟是 $\mathbf{1}_1^N$ 的似然概率大还是 $\mathbf{0}_1^N$ 的似然概率大，证毕。

我们的后续叙述需要下面的引理。

引理 4.1 设 $x = (x_1, \dots, x_N) \in \mathcal{C}$ 是长度为 N 的线性分组码 \mathcal{C} 中的任意一个码字，现在用转移概率为 $\Pr(y|x)$, $x \in \{0, 1\}$ 的无记忆信道传输 \mathbf{x} ，得到的接收信号为 $\mathbf{y} = (y_1, \dots, y_N)$ ，对 \mathbf{y} 做硬判决，得到硬判决序列 $(\beta_1, \dots, \beta_N)$ ，其中 $\beta_i = \frac{1 - \text{sign}(\ln \frac{\Pr(y_i|0)}{\Pr(y_i|1)})}{2}$ 。如果 $(\beta_1, \dots, \beta_N) \in \mathcal{C}$ ，则 $(\beta_1, \dots, \beta_N)$ 是 \mathbf{y} 的最大似然译码结果。

证明 这是一个简单的结论，用反证法。

假设 $(\beta_1, \dots, \beta_N) \in \mathcal{C}$ 不是 \mathbf{y} 的最大似然译码结果，那么就存在 $(\alpha_1, \dots, \alpha_N) \in \mathcal{C}$ 是最大似然译码结果：

$$\frac{\Pr(\mathbf{y}|(\alpha_1, \dots, \alpha_N))}{\Pr(\mathbf{y}|(\beta_1, \dots, \beta_N))} = \frac{\prod_{i=1}^N \Pr(y_i|\alpha_i)}{\prod_{i=1}^N \Pr(y_i|\beta_i)} = \prod_{d \in \mathcal{D}} \frac{\Pr(y_d|\beta_d \oplus 1)}{\Pr(y_d|\beta_d)} > 1. \quad (4.10)$$

其中第一个等号是因为传输信道是无记忆的。 $(\alpha_1, \dots, \alpha_N)$ 和 $(\beta_1, \dots, \beta_N)$ 在有些位置上的比特相同，相同的比特在第一个等号后其实已经约去，我们只关心 $(\alpha_1, \dots, \alpha_N)$ 和 $(\beta_1, \dots, \beta_N)$ 取值不同的比特位：集合 $\mathcal{D} = \{d | \alpha_d \neq \beta_d\}$ 。

对于每个 $d \in \mathcal{D}$ ，存在下面两种情况：

(i). $\beta_d = 0$ 。按照硬判决规则，此时 $\Pr(y_d|0) \geq \Pr(y_d|1)$ ，即 $\Pr(y_d|\beta_d) \geq \Pr(y_d|\beta_d \oplus 1)$ ， $1 \geq \frac{\Pr(y_d|\beta_d \oplus 1)}{\Pr(y_d|\beta_d)}$ ；

(ii). $\beta_d = 1$ 。按照硬判决规则，此时 $\Pr(y_d|1) \geq \Pr(y_d|0)$ ，即 $\Pr(y_d|\beta_d) \geq \Pr(y_d|\beta_d \oplus 1)$ ， $1 \geq \frac{\Pr(y_d|\beta_d \oplus 1)}{\Pr(y_d|\beta_d)}$ ；

这说明式(4.10)最后一个等号中每一个乘积项都小于等于1，式(4.10)大于1肯定不成立。所以假设错误， $(\beta_1, \dots, \beta_N)$ 是最大似然译码结果。

注意到上面的引理的成立条件事实上非常苛刻：当码长 N 比较大时，硬判决结果 $(\beta_1, \dots, \beta_N)$ 是码字的可能性微乎其微。

定理 4.3 SPC极化码的最大似然译码。

发送长度为 $N = 2^n$ 的SPC码，经过转移概率为 $\Pr(y|x)$, $x \in \{0, 1\}$ 的无记忆信道传输，接

收信号为 (y_1, \dots, y_N) , 首先硬判决每一个接收信号, 得到硬判决比特序列 $(\beta_1, \dots, \beta_N)$:

$$\beta_i = \frac{1 - \text{sign}(\ln \frac{\Pr(y_i|0)}{\Pr(y_i|1)})}{2}. \quad (4.11)$$

如果 $\bigoplus_{i=1}^N \beta_i = 0$, 则译码结束, 且 $(\beta_1, \dots, \beta_N)$ 是最大似然译码结果; 如果 $\bigoplus_{i=1}^N \beta_i = 1$, 选取 $p = \arg \min_{1 \leq i \leq N} \{|\ln \frac{\Pr(y_i|0)}{\Pr(y_i|1)}|\}$, 令 $\beta_p = \beta_p \oplus 1$ (翻转具有最小LLR绝对值的接收信号对应的硬判决比特), 则 $(\beta_1, \dots, \beta_N)$ 是最大似然译码结果。

证明 我下面给出一个非常莽的证明, 没有任何巧劲。

如果 $\bigoplus_{i=1}^N \beta_i = 0$, 这说明 $(\beta_1, \dots, \beta_N)$ 是一个SPC码字, 由引理4.1可知 $(\beta_1, \dots, \beta_N)$ 是接收信号 (y_1, \dots, y_N) 的最大似然译码结果。

如果 $\bigoplus_{i=1}^N \beta_i = 0$, 这说明 $(\beta_1, \dots, \beta_N)$ 不是SPC码字, 但是找到满足定理条件的 p 后, $(\beta_1, \dots, \beta_p \oplus 1, \dots, \beta_N)$ 是SPC码字。我们现在想说明 $(\beta_1, \dots, \beta_p \oplus 1, \dots, \beta_N)$ 是最大似然译码结果, 那么任选SPC码中一个码字 $(\alpha_1, \dots, \alpha_N)$, 看看下式是不是总是成立?

$$\frac{\Pr(\mathbf{y}_1^N | (\beta_1, \dots, \beta_p \oplus 1, \dots, \beta_N))}{\Pr(\mathbf{y}_1^N | (\alpha_1, \dots, \alpha_N))} = \frac{\prod_{i=1}^{p-1} \Pr(y_i | \beta_i)}{\prod_{i=1}^{p-1} \Pr(y_i | \alpha_i)} \left\{ \frac{\Pr(y_p | \beta_p \oplus 1)}{\Pr(y_p | \alpha_p)} \right\} \frac{\prod_{i=p+1}^N \Pr(y_i | \beta_i)}{\prod_{i=p+1}^N \Pr(y_i | \alpha_i)} \stackrel{(?)}{\geq} 1. \quad (4.12)$$

在第一个分段 $(\beta_1, \dots, \beta_{p-1}), (\alpha_1, \dots, \alpha_{p-1})$ 中, 有些位置上的比特相同, 这些相同的比特对应的条件概率在式(4.12)其实已经约去。我们找出比特索引集合: $\mathcal{D} = \{d | \alpha_d = \beta_d \oplus 1, 1 \leq d \leq p-1\}$ 。

在第二个分段 $(\beta_{p+1}, \dots, \beta_N), (\alpha_{p+1}, \dots, \alpha_N)$ 中执行相同的操作: $\mathcal{D}' = \{d' | \alpha'_d = \beta'_d \oplus 1, 1 \leq d' \leq p-1\}$ 。

那么式(4.12)就变成了:

$$\frac{\prod_{d \in \mathcal{D}} \Pr(y_d | \beta_d)}{\prod_{d \in \mathcal{D}} \Pr(y_d | \beta_d \oplus 1)} \left\{ \frac{\Pr(y_p | \beta_p \oplus 1)}{\Pr(y_p | \alpha_p)} \right\} \frac{\prod_{d' \in \mathcal{D}'} \Pr(y_{d'} | \beta_{d'})}{\prod_{d' \in \mathcal{D}'} \Pr(y_{d'} | \beta_{d'} \oplus 1)} \stackrel{(?)}{\geq} 1. \quad (4.13)$$

因为 $\beta_d, d \in \mathcal{D}$ 是硬判决结果, 所以 $\Pr(y_d | \beta_d) \geq \Pr(y_d | \beta_d \oplus 1), d \in \mathcal{D}$ 肯定成立。 $\beta_{d'}, d' \in \mathcal{D}'$ 同理 (只不过是另一个分段)。所以 $\frac{\prod_{d \in \mathcal{D}} \Pr(y_d | \beta_d)}{\prod_{d \in \mathcal{D}} \Pr(y_d | \beta_d \oplus 1)} \geq 1$, $\frac{\prod_{d' \in \mathcal{D}'} \Pr(y_{d'} | \beta_{d'})}{\prod_{d' \in \mathcal{D}'} \Pr(y_{d'} | \beta_{d'} \oplus 1)} \geq 1$ 成立。

如果 $\beta_p \oplus 1 = \alpha_p$, 那么中间项 $\frac{\Pr(y_p | \beta_p \oplus 1)}{\Pr(y_p | \alpha_p)}$ 约去, 式(4.13)中的不等号成立。

如果 $\beta_p \oplus 1 \neq \alpha_p$, 即 $\beta_p \oplus 1 = \alpha_p \oplus 1, \beta_p = \alpha_p$, 中间项 $\frac{\Pr(y_p | \beta_p \oplus 1)}{\Pr(y_p | \alpha_p)}$ 变成了 $\frac{\Pr(y_p | \beta_p \oplus 1)}{\Pr(y_p | \beta_p)}$ 。因为 β_p 是硬判决结果, 所以 $\Pr(y_d | \beta_p) \geq \Pr(y_d | \beta_p \oplus 1)$, $\frac{\Pr(y_d | \beta_p \oplus 1)}{\Pr(y_d | \beta_p)} \leq 1$ 肯定成立。这时我们任意

从 $\frac{\prod_{d \in \mathcal{D}} \Pr(y_d | \beta_d)}{\prod_{d \in \mathcal{D}} \Pr(y_d | \beta_d \oplus 1)}$ 或 $\frac{\prod_{d' \in \mathcal{D}'} \Pr(y_{d'} | \beta_{d'})}{\prod_{d' \in \mathcal{D}'} \Pr(y_{d'} | \beta_{d'} \oplus 1)}$ 中选出一个乘积项，记为 $\frac{\Pr(y_s | \beta_s)}{\Pr(y_s | \beta_s \oplus 1)}$ ，注意到 $\frac{\Pr(y_s | \beta_s)}{\Pr(y_s | \beta_s \oplus 1)} \geq 1$ 。那么有以下四种情况。

(i). $\beta_p = 0, \beta_s = 0$ 。考虑 $\ln \frac{\Pr(y_s | \beta_s)}{\Pr(y_s | \beta_s \oplus 1)} + \ln \frac{\Pr(y_p | \beta_p \oplus 1)}{\Pr(y_p | \beta_p)}$ 的符号。因为 $\frac{\Pr(y_d | \beta_p \oplus 1)}{\Pr(y_d | \beta_p)} \leq 1$ ，所以变成了 $|\ln \frac{\Pr(y_s | \beta_s)}{\Pr(y_s | \beta_s \oplus 1)}| - |\ln \frac{\Pr(y_p | \beta_p \oplus 1)}{\Pr(y_p | \beta_p)}| = |\ln \frac{\Pr(y_s | \beta_s)}{\Pr(y_s | \beta_s \oplus 1)}| - |\ln \frac{\Pr(y_p | \beta_p)}{\Pr(y_p | \beta_p \oplus 1)}|$ ，带入 $\beta_p = 0, \beta_s = 0$ ，有 $|\ln \frac{\Pr(y_s | 0)}{\Pr(y_s | 1)}| - |\ln \frac{\Pr(y_p | 0)}{\Pr(y_p | 1)}|$ 。由题设， $p = \arg \min_{1 \leq i \leq N} \{|\ln \frac{\Pr(y_i | 0)}{\Pr(y_i | 1)}|\}$ ，所以 $|\ln \frac{\Pr(y_s | 0)}{\Pr(y_s | 1)}| - |\ln \frac{\Pr(y_p | 0)}{\Pr(y_p | 1)}| \geq 0$ ，即 $\ln \frac{\Pr(y_s | \beta_s)}{\Pr(y_s | \beta_s \oplus 1)} + \ln \frac{\Pr(y_p | \beta_p \oplus 1)}{\Pr(y_p | \beta_p)} \geq 0$ ， $\frac{\Pr(y_s | \beta_s)}{\Pr(y_s | \beta_s \oplus 1)} \frac{\Pr(y_p | \beta_p \oplus 1)}{\Pr(y_p | \beta_p)} \geq 1$ 。式(4.13)中除了 $\frac{\Pr(y_s | \beta_s)}{\Pr(y_s | \beta_s \oplus 1)} \frac{\Pr(y_p | \beta_p \oplus 1)}{\Pr(y_p | \beta_p)}$ 的项都大于等于1，故不等号成立。

(ii). $\beta_p = 0, \beta_s = 1$ 。与上一步类似，我们有下面一套式子成立：

$$\begin{aligned} \ln \frac{\Pr(y_s | \beta_s)}{\Pr(y_s | \beta_s \oplus 1)} + \ln \frac{\Pr(y_p | \beta_p \oplus 1)}{\Pr(y_p | \beta_p)} &= |\ln \frac{\Pr(y_s | \beta_s)}{\Pr(y_s | \beta_s \oplus 1)}| - |\ln \frac{\Pr(y_p | \beta_p \oplus 1)}{\Pr(y_p | \beta_p)}| \\ &= |\ln \frac{\Pr(y_s | \beta_s \oplus 1)}{\Pr(y_s | \beta_s)}| - |\ln \frac{\Pr(y_p | \beta_p)}{\Pr(y_p | \beta_p \oplus 1)}| = |\ln \frac{\Pr(y_s | 0)}{\Pr(y_s | 1)}| - |\ln \frac{\Pr(y_p | 0)}{\Pr(y_p | 1)}| \geq 0. \end{aligned}$$

故 $\ln \frac{\Pr(y_s | \beta_s)}{\Pr(y_s | \beta_s \oplus 1)} + \ln \frac{\Pr(y_p | \beta_p \oplus 1)}{\Pr(y_p | \beta_p)} \geq 0$ ， $\frac{\Pr(y_s | \beta_s)}{\Pr(y_s | \beta_s \oplus 1)} \frac{\Pr(y_p | \beta_p \oplus 1)}{\Pr(y_p | \beta_p)} \geq 1$ 。

式(4.13)中除了 $\frac{\Pr(y_s | \beta_s)}{\Pr(y_s | \beta_s \oplus 1)} \frac{\Pr(y_p | \beta_p \oplus 1)}{\Pr(y_p | \beta_p)}$ 的项都大于等于1，故不等号成立。

(iii). $\beta_p = 1, \beta_s = 0$ 。与上一步类似，我们有下面一套式子成立：

$$\begin{aligned} \ln \frac{\Pr(y_s | \beta_s)}{\Pr(y_s | \beta_s \oplus 1)} + \ln \frac{\Pr(y_p | \beta_p \oplus 1)}{\Pr(y_p | \beta_p)} &= |\ln \frac{\Pr(y_s | \beta_s)}{\Pr(y_s | \beta_s \oplus 1)}| - |\ln \frac{\Pr(y_p | \beta_p \oplus 1)}{\Pr(y_p | \beta_p)}| \\ &= |\ln \frac{\Pr(y_s | 0)}{\Pr(y_s | 1)}| - |\ln \frac{\Pr(y_p | 0)}{\Pr(y_p | 1)}| \geq 0. \end{aligned}$$

故 $\ln \frac{\Pr(y_s | \beta_s)}{\Pr(y_s | \beta_s \oplus 1)} + \ln \frac{\Pr(y_p | \beta_p \oplus 1)}{\Pr(y_p | \beta_p)} \geq 0$ ， $\frac{\Pr(y_s | \beta_s)}{\Pr(y_s | \beta_s \oplus 1)} \frac{\Pr(y_p | \beta_p \oplus 1)}{\Pr(y_p | \beta_p)} \geq 1$ 。

式(4.13)中除了 $\frac{\Pr(y_s | \beta_s)}{\Pr(y_s | \beta_s \oplus 1)} \frac{\Pr(y_p | \beta_p \oplus 1)}{\Pr(y_p | \beta_p)}$ 的项都大于等于1，故不等号成立。

(iv). $\beta_p = 1, \beta_s = 1$ 。与上一步类似，我们有下面一套式子成立：

$$\begin{aligned} \ln \frac{\Pr(y_s | \beta_s)}{\Pr(y_s | \beta_s \oplus 1)} + \ln \frac{\Pr(y_p | \beta_p \oplus 1)}{\Pr(y_p | \beta_p)} &= |\ln \frac{\Pr(y_s | \beta_s)}{\Pr(y_s | \beta_s \oplus 1)}| - |\ln \frac{\Pr(y_p | \beta_p \oplus 1)}{\Pr(y_p | \beta_p)}| \\ &= |\ln \frac{\Pr(y_s | \beta_s \oplus 1)}{\Pr(y_s | \beta_s)}| - |\ln \frac{\Pr(y_p | \beta_p \oplus 1)}{\Pr(y_p | \beta_p)}| = |\ln \frac{\Pr(y_s | 0)}{\Pr(y_s | 1)}| - |\ln \frac{\Pr(y_p | 0)}{\Pr(y_p | 1)}| \geq 0. \end{aligned}$$

故 $\ln \frac{\Pr(y_s | \beta_s)}{\Pr(y_s | \beta_s \oplus 1)} + \ln \frac{\Pr(y_p | \beta_p \oplus 1)}{\Pr(y_p | \beta_p)} \geq 0$ ， $\frac{\Pr(y_s | \beta_s)}{\Pr(y_s | \beta_s \oplus 1)} \frac{\Pr(y_p | \beta_p \oplus 1)}{\Pr(y_p | \beta_p)} \geq 1$ 。

式(4.13)中除了 $\frac{\Pr(y_s | \beta_s)}{\Pr(y_s | \beta_s \oplus 1)} \frac{\Pr(y_p | \beta_p \oplus 1)}{\Pr(y_p | \beta_p)}$ 的项都大于等于1，故不等号成立。

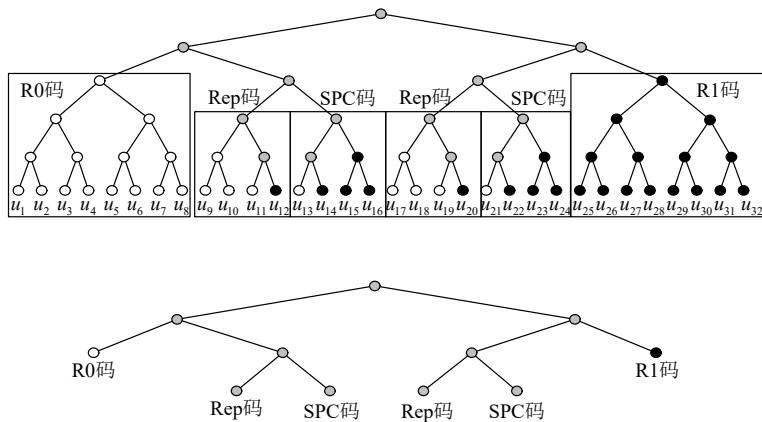


图 4.11 $N = 32, K = 16$ 时, 使用高斯近似在 $E_b/N_0 = 2.5\text{dB}$ 处构造的极化码结构

这个定理的含义很明确: 如果硬判决结果 $(\beta_1, \dots, \beta_N)$ 是一个SPC码字, 则译码成功; 如果 $(\beta_1, \dots, \beta_N)$ 不是一个SPC码字, 那么翻转其中“可靠度”(LLR的绝对值)最低的硬判决比特的值就完成了最大似然译码。

定理 4.4 R1码的最大似然译码。

发送长度为 $N = 2^n$ 的R1码, 经过转移概率为 $\Pr(y|x), x \in \{0, 1\}$ 的无记忆信道传输, 接收信号为 (y_1, \dots, y_N) 。硬判决每一个接收信号¹, 得到硬判决比特序列 $(\beta_1, \dots, \beta_N)$:

$$\beta_i = \frac{1 - \text{sign}(\ln \frac{\Pr(y_i|0)}{\Pr(y_i|1)})}{2}. \quad (4.14)$$

$(\beta_1, \dots, \beta_N)$ 就是 (y_1, \dots, y_N) 的最大似然译码结果。

证明 因为此时码率为1, 任意一个比特序列 $(b_1, \dots, b_N) \in \{0, 1\}^N$ 都是码字, 从而 $(\beta_1, \dots, \beta_N)$ 是码字。由引理4.1, $(\beta_1, \dots, \beta_N)$ 是最大似然译码结果。证毕。

这样我们就说明了4种特殊极化码的最大似然译码方式, 注意到上面的方法无一例外估计码字比特, 而非直接估计信息比特。下面举例说明它们的应用。

例 4.3 $N = 32, K = 16$ 时, 使用高斯近似在 $E_b/N_0 = 2.5\text{dB}$ 处构造的极化码结构如图4.11所示, 其中白色节点代表其所有叶节点都是冻结比特, 黑色节点代表其所有叶节点都是信息比特, 灰色节点表示其叶节点既包括冻结比特也包括信息比特。

我们在上图的基础上说明如何使用R0、Rep、SPC和R1码进行快速SC译码。

例 4.4 如何利用R0、Rep、SPC和R1码进行快速SC译码 仿真配置如下：

信息比特索引集合 $\mathcal{A} = \{12, 14, 15, 16, 20, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32\}$, $R = 16/32$;

信源序列 $\mathbf{u}_1^{32} = (000000000001011100010111111111)$;

码字序列 $\mathbf{x}_1^{32} = \mathbf{u}_1^{32} \mathbf{F}^{\otimes 5} = (0000000101101000011010000000001)$

传输信噪比 $E_b/N_0 = 5$ dB, AWGN信道噪声的标准差 $\sigma = 10^{-5/20}$;

BPSK调制, 接收LLR为 $\text{llr}_1^{32} = (8.2, 12.8, -1.7, 9.4, 7.5, 1.7, 4.8, -5.1, 19.1, 3.5, -11.1, 17.1, -3.7, 6.1, 8.9, 5.6, 5.9, -1.0, -1.3, 11.4, -3.9, 2.0, 8.9, 12.1, 8.1, 10, 8.9, 5.2, 7.4, 3.5, 9.5, -10.4)$

使用R0、Rep、SPC和R1的最大似然译码的快速SC译码过程如图4.12（前七步）和图4.13（后七步）所示，其中红色框内的数据为当前步骤刚算出来的数据，黄色框表示当前步骤 g 函数所需的比特值。可见，仅需14步就完成了长度为 $N = 32$ 的极化码的译码，传统SC译码需要 $2 \times 32 - 2 = 62$ 步。对照仿真配置中 \mathbf{x}_1^{32} 的值，发现译码结果正确。

熟练掌握了图4.12和图4.13中的译码规律后，我们就能编制程序了。编制程序的第一步不是写译码器，而是识别4种特殊的极化子码。也许你有你独到的识别方式，但我这里给出一个较为通用的识别程序。我这个程序虽然是递归的，但是这没关系，因为它仅执行一遍。

函数`get_node_structure(frozen_bits)`是主函数，`frozen_bits`是长度为 N 的向量，`frozen_bits(i)=1`表示 u_i 是冻结比特，`frozen_bits(i)=0`表示 u_i 是信息比特。`code_structure`初始化为 N 维矩阵：`code_structure(i,1)`表示整个极化码种第*i*个子码的起始索引位置，`code_structure(i,2)`表示整个极化码种第*i*个子码的长度，`code_structure(i,3)`指示子码类型：-1指R0，1指R1，2指Rep，3指SPC。

`cnt_structure`只不过是个计数变量，每识别完毕一个节点，它的值加一。

```

1 function node_type_structure = get_node_structure(frozen_bits)
2 global code_structure cnt_structure
3 code_structure = zeros(length(frozen_bits), 3);
4 cnt_structure = 1;
5 node_identifier(frozen_bits, 1 : length(frozen_bits));
6 code_structure = code_structure(code_structure ~= 0);
7 code_structure = reshape(code_structure, length(code_structure)/3, 3);
8 node_type_structure = code_structure;
9 clear global;
10 end
11 % -1 RATE0
12 % 1 RATE 1
13 % 2 REP
14 % 3 SPC

```

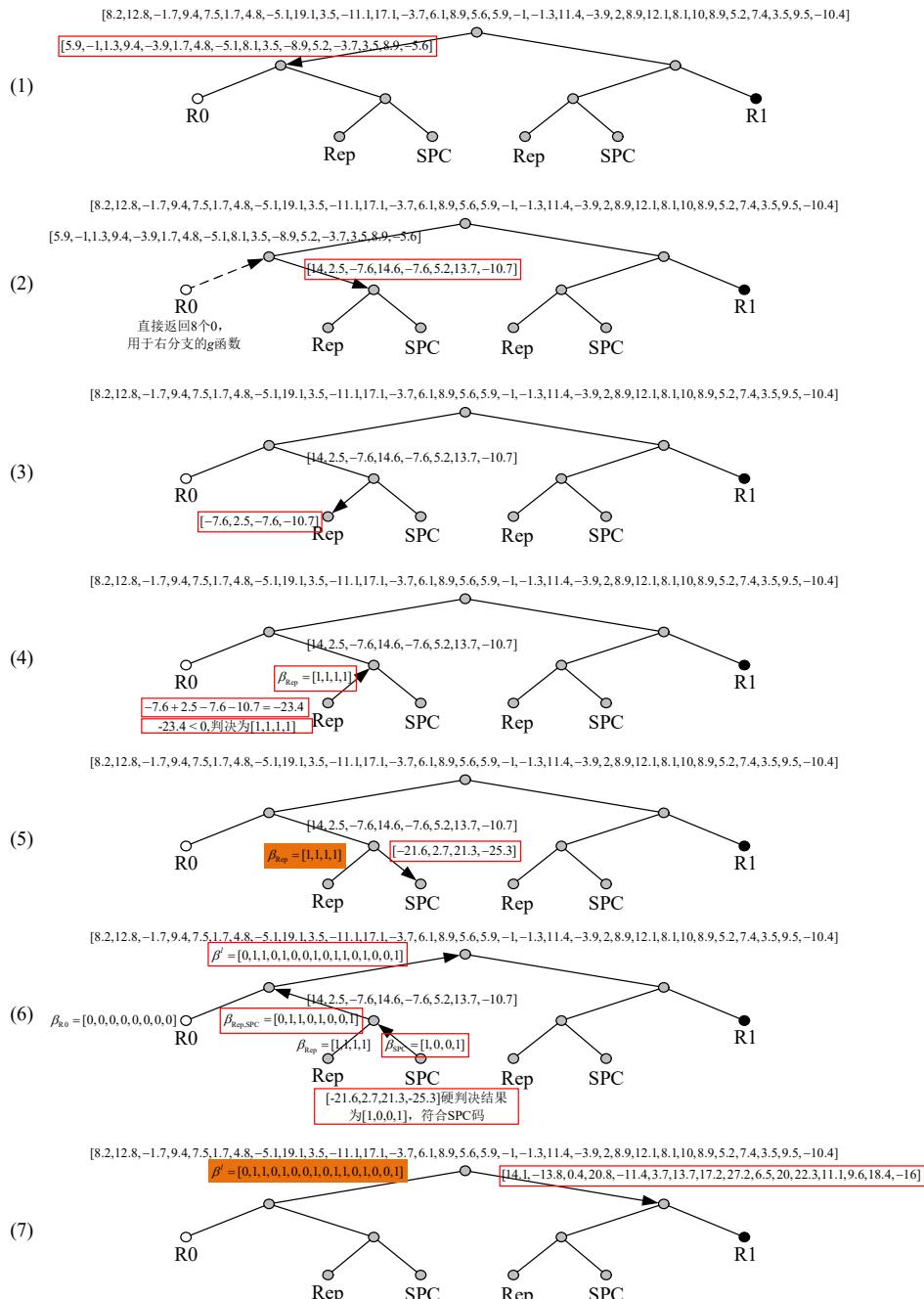


图 4.12 快速SC译码的例子, 前7步

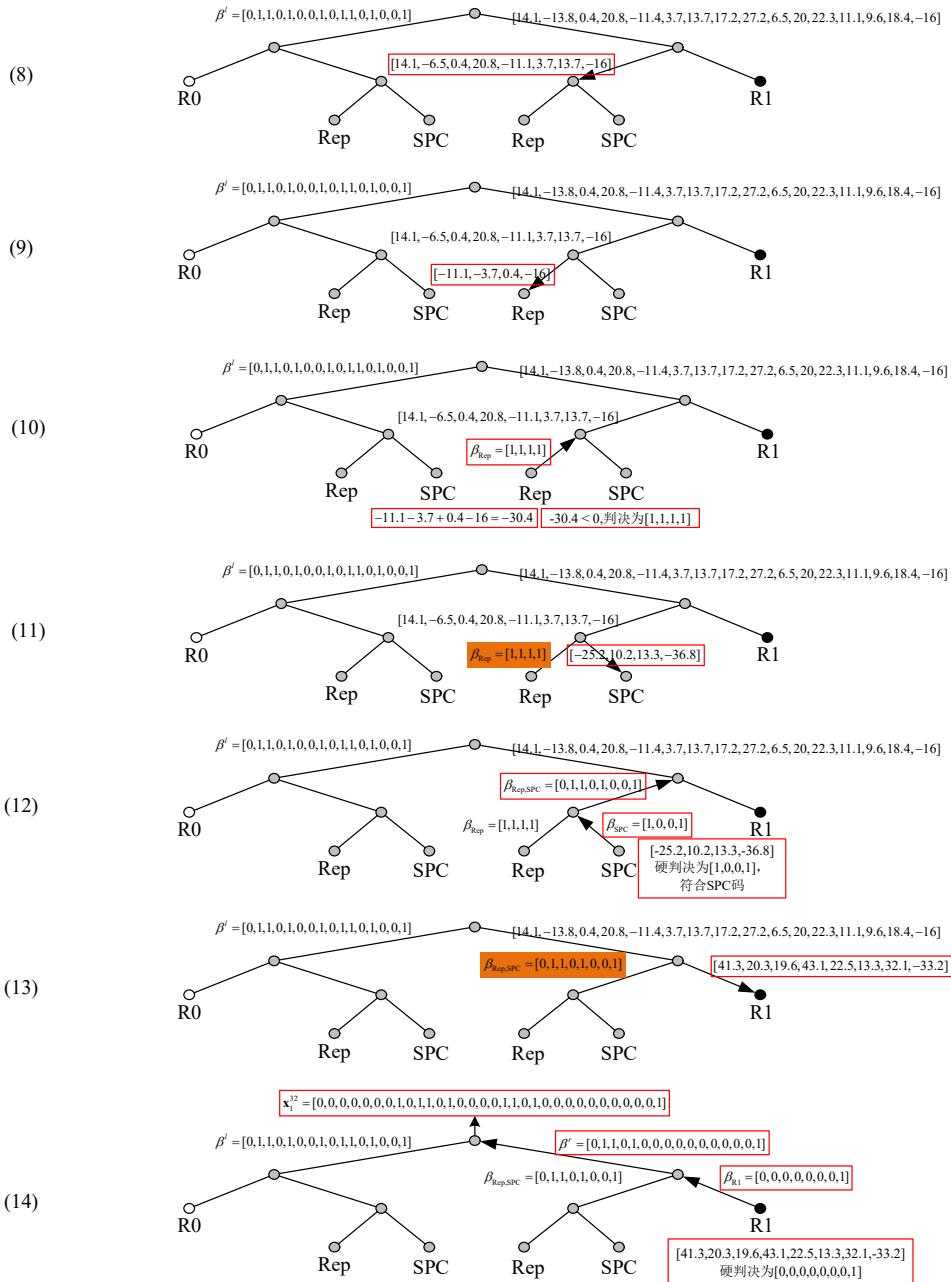


图 4.13 快速SC译码的例子, 后7步

```
1 function node_identifier(f, z)
2 N = length(f);
3 global code_structure cnt_structure
4 if all(f(1 : end - 1) == 1) && (f(end) == 0)%REP
5     code_structure(cnt_structure, 1) = z(1);
6     code_structure(cnt_structure, 2) = N;
7     code_structure(cnt_structure, 3) = 2;
8     cnt_structure = cnt_structure + 1;
9 else
10    if (f(1) == 1) && all(f(2 : end) == 0)%SPC
11        code_structure(cnt_structure, 1) = z(1);
12        code_structure(cnt_structure, 2) = N;
13        code_structure(cnt_structure, 3) = 3;
14        cnt_structure = cnt_structure + 1;
15    else
16        if all(f == 0)%R1
17            code_structure(cnt_structure, 1) = z(1);
18            code_structure(cnt_structure, 2) = N;
19            code_structure(cnt_structure, 3) = 1;
20            cnt_structure = cnt_structure + 1;
21    else
22        if all(f == 1)%R0
23            code_structure(cnt_structure, 1) = z(1);
24            code_structure(cnt_structure, 2) = N;
25            code_structure(cnt_structure, 3) = -1;
26            cnt_structure = cnt_structure + 1;
27        else
28            node_identifier(f(1:N/2), z(1:N/2));
29            node_identifier(f(N/2+1:end), z(N/2+1:end));
30        end
31    end
32 end
33 end
34 end
```

由极化码的基本规律可知，当子码长度 $N_v = 2$ 时，不存在R0, Rep之外的结构。所以上面的程序不存在“识别不了”的子码。

举个例子，图4.11种的frozen_bits= [1111111111010001110100000000000]，程序的执行结果为：

$$\text{node_type_structure} = \begin{bmatrix} 1 & 8 & -1 \\ 9 & 4 & 2 \\ 13 & 4 & 3 \\ 17 & 4 & 2 \\ 21 & 4 & 3 \\ 25 & 8 & 1 \end{bmatrix} \quad (4.15)$$

含义为（对照图4.11）：

从 u_1 开始，是一个长度为8的R0码；

从 u_9 开始，是一个长度为4的Rep码；

从 u_{13} 开始，是一个长度为4的SPC码；

从 u_{17} 开始，是一个长度为4的Rep码；

从 u_{21} 开始，是一个长度为4的SPC码；

从 u_{25} 开始，是一个长度为8的R1码；

下面是快速SC译码的程序，输入参数中的node_type_structure就是形如式(4.15)的矩阵。这个程序是没有任何递归的，它在执行过程中不调用任何子函数，这保证了它的执行效率。

lambda_offset、llr.layer_vec和bit_layer_vec与SC译码器中的含义完全相同。

psi_vec是一个计算中间比特值时需要使用的量，它的大小是：node_type_structure的行数×1。

psi_vec(i)的值的含义是：第*i*个极化子码的长度是 N_v ，去掉 $i - 1$ 的二进制展开最后的 $\log_2 N_v$ 个1，得到一个数 γ ，psi_vec(i)= γ 。

参照图4.12和图4.13，自己逐一地计算所需参数，一步步地手工执行下面的代码，我想一个初学者半天就能看懂。

```

1 function x = FastSCdecoder(llr, node_type_structure, ...
2 lambda_offset, llr.layer_vec, psi_vec, bit_layer_vec)
3 N = length(llr);

```

```
4 n = log2(N);
5 C = zeros(2 * N - 1, 2);%Bit vector
6 P = zeros(2 * N - 1, 1);%LLR vector
7 P(end - N + 1 : end) = llr;%LLR initialization
8 for i_node = 1 : size(node_type_structure, 1)
9     M = node_type_structure(i_node, 2);%size of subcode
10    reduced_layer = log2(M);
11    %reduced_layer denotes where to stop LLR calculation
12    %reduced_layer denotes where to start Internal Bits calculation.
13    llr_layer = llr_layer_vec(node_type_structure(i_node, 1));
14    %llr_layer denotes where to start LLR calculation
15    bit_layer = bit_layer_vec(node_type_structure(i_node, 1) + M - 1);
16    %bit_layer denotes where to stop Internal Bits calculation.
17    psi = psi_vec(i_node);%This psi is used for bits recursion
18    psi_mod_2 = mod(psi, 2);
19    if i_node == 1%first LLR calculation only uses f function.
20        for i_layer = n - 1 : -1 : reduced_layer
21            index_1 = lambda_offset(i_layer + 1);
22            index_2 = lambda_offset(i_layer + 2);
23            for beta = index_1 : index_2 - 1
24                P(beta) = sign(P(beta + index_1)) * ...
25                    sign(P(beta + index_2)) * ...
26                    min(abs(P(beta + index_1)), abs(P(beta + index_2)));
27            end
28        end
29    else
30        index_1 = lambda_offset(llr_layer + 1);
31        index_2 = lambda_offset(llr_layer + 2);
32        for beta = index_1 : index_2 - 1
33            P(beta) = (1 - 2 * C(beta, 1)) * ...
34                P(beta + index_1) + P(beta + index_2);
35        end
36        for i_layer = llr_layer - 1 : -1 : reduced_layer
37            index_1 = lambda_offset(i_layer + 1);
38            index_2 = lambda_offset(i_layer + 2);
```

```

39   for beta = index_1 : index_2 - 1
40     P(beta) = sign(P(beta + index_1)) * ...
41     sign(P(beta + index_2)) * ...
42     min(abs(P(beta + index_1)), abs(P(beta + index_2)));
43   end
44 end
45
46 switch node_type_structure(i_node, 3)
47 case -1%RATE 0
48   for j = M : 2 * M - 1
49     C(j, psi_mod_2 + 1) = 0;
50   end
51 case 1%RATE 1
52   for j = M : 2 * M - 1
53     C(j, psi_mod_2 + 1) = P(j) < 0;
54   end
55 case 2%REP
56   sum_llr = 0;
57   for j = M : 2 * M - 1
58     sum_llr = sum_llr + P(j);
59   end
60   rep_bit = sum_llr < 0;
61   for j = M : 2 * M - 1
62     C(j, psi_mod_2 + 1) = rep_bit;
63   end
64 case 3%spc
65   llr_sub_polar_code = zeros(M, 1);
66   for j = M : 2 * M - 1
67     llr_sub_polar_code(j - M + 1) = P(j);
68   end
69   x = llr_sub_polar_code < 0;
70   if mod(sum(x), 2) ~= 0%if SPC constraint is not satisfied
71     alpha_plus = abs(llr_sub_polar_code);
72     [~, min_index] = min(alpha_plus);
73     x(min_index) = mod(x(min_index) + 1, 2);

```

```

74      end
75      for j = M : 2 * M - 1
76          C(j, psi_mod_2 + 1) = x(j - M + 1);
77      end
78
79      if psi_mod_2 == 1%bit recursion
80          for i_layer = reduced_layer : bit_layer - 1
81              index_1 = lambda_offset(i_layer + 1);
82              index_2 = lambda_offset(i_layer + 2);
83              for beta = index_1 : index_2 - 1
84                  C(beta + index_1, 2) = mod(C(beta, 1) + C(beta, 2), 2);
85                  C(beta + index_2, 2) = C(beta, 2);
86              end
87
88              index_1 = lambda_offset(bit_layer + 1);
89              index_2 = lambda_offset(bit_layer + 2);
90              for beta = index_1 : index_2 - 1
91                  C(beta + index_1, 1) = mod(C(beta, 1) + C(beta, 2), 2);
92                  C(beta + index_2, 1) = C(beta, 2);
93              end
94
95      end
96      x = C(end - N + 1 : end, 1);
97  end

```

```

1 function psi_vec = get_psi(node_type_matrix)
2 psi_vec = zeros(size(node_type_matrix, 1), 1);
3 for i = 1 : length(psi_vec)
4     psi = node_type_matrix(i);
5     M = node_type_matrix(i, 2);
6     reduced_layer = log2(M);
7     for j = 1 : reduced_layer
8         psi = floor(psi/2);
9     end

```

```

10     psi_vec(i) = psi;
11 end
12 end

```

§ 4.4 比特翻转SC译码器

比特翻转SC译码器是一种比较有趣的译码器，来自文献[17]。它的BLER性能优于单纯的SC译码器，但是它的译码复杂度很不确定，所以没法在实际的系统中使用，只能活在论文里。

比特翻转SC译码器使用的不是单纯的极化码，而是CRC码和极化码形成的简单级联码。一个CRC-polar级联码的例子如下图所示。其中 \mathbf{d}_1^K 是用户数据， \mathbf{c}_1^r 是 r 个CRC校验位，把 $[\mathbf{d}_1^K, \mathbf{c}_1^r]$ 放在信息比特索引 \mathcal{A} 上，冻结比特全取0，得到 \mathbf{u}_1^N ，将 \mathbf{u}_1^N 进行极化码编码，就得到了CRC-polar级联码的一个码字 \mathbf{x}_1^N 。可以看到CRC-polar的码字其实还是一个极化码的码字。



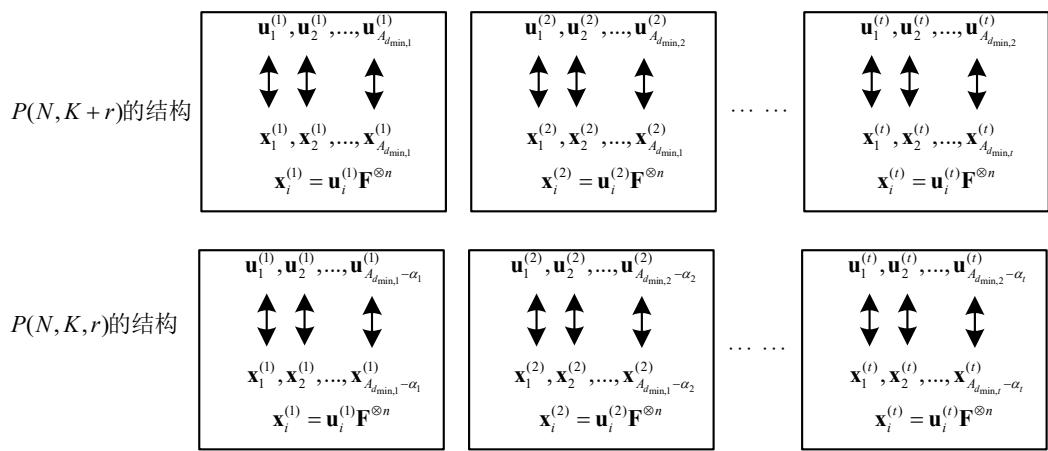
图 4.14 CRC-polar级联码编码过程

单纯极化码用 $P(N, K)$ 表示，其中 N 是码长， K 是信息数量。CRC-polar级联码记为 $P(N, K, r)$ ，其中 r 表示使用了长度为 r 的CRC校验位。必须注意到CRC-polar级联码 $P(N, K, r)$ 的码率是 K/N 而不是 $(K + r)/N$ ，这是因为 r 个CRC校验位不携带信息。单纯极化码 $P(N, K + r)$ 和CRC-polar级联码 $P(N, K, r)$ 的区别如下图所示。

图4.15中参数的含义：

$P(N, K + r)$ 中码字重量的分布为 $\{d_{\min,1}, d_{\min,2}, \dots, d_{\min,t}\}$ ， $\{A_{d_{\min,1}}, A_{d_{\min,2}}, \dots, A_{d_{\min,t}}\}$ ，其中 $d_{\min,1}$ 是最小汉明距离， $d_{\min,2}$ 是次小汉明距离，以此类推。 $A_{d_{\min,i}}$ 表示重量为 $d_{\min,i}$ 的码字的个数。 $\mathbf{x}_k^{(i)}, 1 \leq k \leq A_{d_{\min,i}}$ 表示该码字是第 k 个汉明重量为 $A_{d_{\min,i}}$ 的码字，它对应的信源序列是 $\mathbf{u}_k^{(i)}$ ，并设 $\mathbf{u}_k^{(i)}$ 对应的信息比特为 $\mathbf{d}_k^{(i)}$ 。在用下标 k 对 $\mathbf{u}_k^{(i)}, 1 \leq k \leq A_{d_{\min,i}}$ 进行编号时，如果 $\mathbf{d}_k^{(i)}$ 是CRC码字，我们把 $\mathbf{u}_k^{(i)}$ 放在前 $A_{d_{\min,i}} - \alpha_i$ 个，也就是说，后 α_i 个 $\mathbf{u}_k^{(i)}$ 对应的信息比特 $\mathbf{d}_k^{(i)}$ 不是CRC码字，故不会出现在CRC-polar级联码中。

由图4.15可见， $P(N, K, r)$ 比 $P(N, K + r)$ 的码字数量少，至于“如何减少”，是由所使用的CRC码决定的。我们知道校验位长为 r 的CRC码有多种生成多项式，即便固定 r ，不同的CRC生成多项式也将导致不同的 $\alpha_1, \alpha_2, \dots, \alpha_t$ 。如果你选的CRC比较好，有可能出



现 $\alpha_1 = A_{d_{\min,1}}$ 的情况，意味着 $P(N, K+r)$ 中重量为 $d_{\min,1}$ 的码字消失。这将极大地提升CRC-polar码的BLER性能。

在介绍了CRC-polar级联码后，我们就能介绍比特翻转SC译码器了。

定义 4.5 单一比特翻转SC译码器的工作过程

(1) 对接收信号 \mathbf{y}_1^N 执行传统的SC译码过程，得到极化码信息比特的估计 $[\hat{\mathbf{d}}_1^K, \hat{\mathbf{c}}_1^r]$ ；

(2) 用CRC校验 $[\hat{\mathbf{d}}_1^K, \hat{\mathbf{c}}_1^r]$ 。如果 $[\hat{\mathbf{d}}_1^K, \hat{\mathbf{c}}_1^r]$ 满足CRC校验，或者说 $[\hat{\mathbf{d}}_1^K, \hat{\mathbf{c}}_1^r]$ 是一个CRC码字，则译码结束；如果 $[\hat{\mathbf{d}}_1^K, \hat{\mathbf{c}}_1^r]$ 未通过CRC校验，则需要再次执行SC译码，这时设最大允许翻转次数为 T ，当前比特翻转次数 $t = 1$ ，转入步骤(3)。

(3) 如果 $t > T$ ，翻转次数超过了预设值，宣布译码失败；如果 $t \leq T$ ，选出一个最不可靠信息比特 $u_i, i \in \mathcal{A}$ ，在判决 u_1, \dots, u_{i-1} 的过程中，执行传统SC过程。

在判决 u_i 时，将 u_i 的值按照其对应的LLR的反方向判决：如果 $\ln \frac{W_N^{(i)}(\mathbf{y}_1^N, \mathbf{u}_1^{i-1} | 0)}{W_N^{(i)}(\mathbf{y}_1^N, \mathbf{u}_1^{i-1} | 1)} \geq 0$ ，则 $u_i = 1$ ，反之 $u_i = 0$ 。

$\ln \frac{W_N^{(i)}(\mathbf{y}_1^N, \mathbf{u}_1^{i-1} | 0)}{W_N^{(i)}(\mathbf{y}_1^N, \mathbf{u}_1^{i-1} | 1)}$ 的值就是前面SC译码器代码中 P 数组中的第一个元素 $P(1)$ 。

随后，在判决 u_{i+1}, \dots, u_N 的过程中，执行传统SC过程。

注意到修改一个 u_i 的值会引起前面程序中 C 数组里的元素值联锁改变，因此修改 u_i 的值看似改变了一个比特值，实际上数不清改变了多少中间比特的值。同时我们也必须注意到， u_i 必须是错误SC译码中的首个错误判决的信息比特，否则永远不可能获得正确译码结果。

(4) 如果本次译码结果通过了CRC校验，则译码结束；如果仍未通过CRC校验，则 $t = t + 1$ ，返回第(3)步选另外一个 u_i 进行翻转。

到这里我们可以发现，比特翻转SC译码器的复杂度是不固定的：它有时候只需一次SC译码就成功了，有时需要执行多次带有比特翻转的SC译码才成功，有时在最大允许次数 T 内没有译码成功。

热情的读者会发现，单一比特翻转SC译码的核心问题是如何选择不可靠信息比特 $u_i, i \in \mathcal{A}$ ，来保证该 u_i 是错误SC译码中的首个错误比特，因为必须纠正首个错误比特才有可能获得正确的译码。文献[17]给出一种感性的方法：在首次SC译码过程中，我们可以记录 $\ln \frac{W_N^{(i)}(\mathbf{y}_1^N, \mathbf{u}_1^{i-1}|0)}{W_N^{(i)}(\mathbf{y}_1^N, \mathbf{u}_1^{i-1}|1)}$, $i \in \mathcal{A}$ 的值。如果首次SC译码失败了，我们可以选择：

$$s = \arg \min_{i \in \mathcal{A}} \left\{ \left| \ln \frac{W_N^{(i)}(\mathbf{y}_1^N, \mathbf{u}_1^{i-1}|0)}{W_N^{(i)}(\mathbf{y}_1^N, \mathbf{u}_1^{i-1}|1)} \right| \right\}, \quad (4.16)$$

在随后的比特翻转译码过程中翻转 u_s 的值，因为小的LLR绝对值表示判决不可靠。如果翻转 u_s 的值后仍未成功译码，则继续找出次小LLR绝对值对应的信息比特作为被翻转比特，以此类推，直到译码结果通过CRC校验或比特翻转译码次数超过了允许值 T 。

文献[18]给出了另一种识别不可靠信息比特的方法，称为“关键集合（Critical Set, CS）”法。文献[18]认为位于关键集合内的信息比特不可靠。关键集合由每个R1子码中第一个比特构成，如图4.16所示。图4.16中共有七个R1子码（此处仅考虑R0和R1子码，不考虑SPC和Rep子码），七个R1子码的首个信息比特一起构成了关键集合 $CS = \{u_{12}, u_{14}, u_{15}, u_{20}, u_{22}, u_{23}, u_{25}\}$ 。在选取需要被翻转的信息比特时，就在CS里面挑。

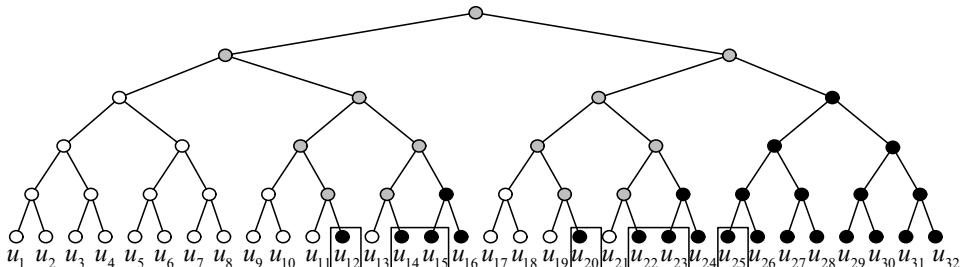
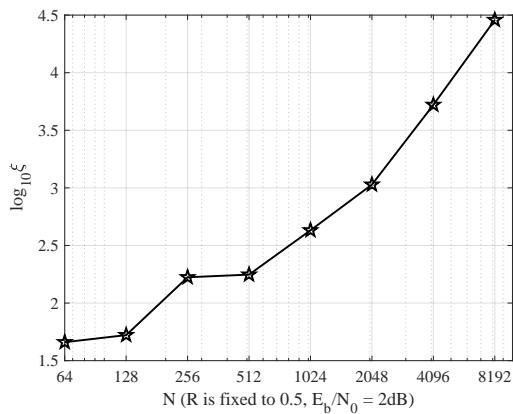
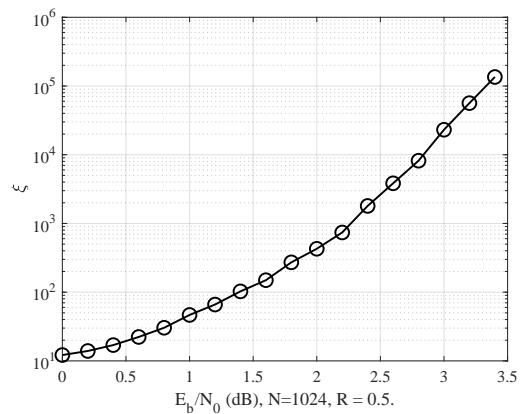


图 4.16 关键集合的例子， $P(32, 16)$

为什么CS中的比特不可靠？文献[18]中自有论述，我不复读他了，我在这里说说文献[18]中没有提及的内容。首先，从极化码结构来看，每个R1子码都是一个极化码，只不过有点短。设一个长度为 $N_v = 2^m$ 的R1子码里的每个信息比特标号为 $u_0, u_1, \dots, u_{N_v-1}$ ，它们对应的极化信道记为 $W_{N_v}^{(0)}, \dots, W_{N_v}^{(N_v-1)}$ 。那么 u_0 入选关键集合。仅在这个长度为 N_v 的R1码里考虑问

图 4.17 ξ 与码长 N 的关系图 4.18 ξ 与信噪比 E_b/N_0 的关系

题：0的二进制展开全是0，其余数 $1, \dots, N_v - 1$ 的二进制展开中至少有一个1。根据我们前面叙述的极化信道偏序的结果， $W_{N_v}^{(0)}$ 是 $W_{N_v}^{(0)}, \dots, W_{N_v}^{(N_v-1)}$ 中最差的信道，所以 u_0 不可靠。

也可以从高斯近似来说明CS中的信息比特不可靠。我们简单地定义一个比值 ξ ：

$$\xi = \frac{\sum_{i \in \text{CS}} P_e(W_N^{(i)})}{\sum_{i \in \mathcal{A} \setminus \text{CS}} P_e(W_N^{(i)}), \quad (4.17)}$$

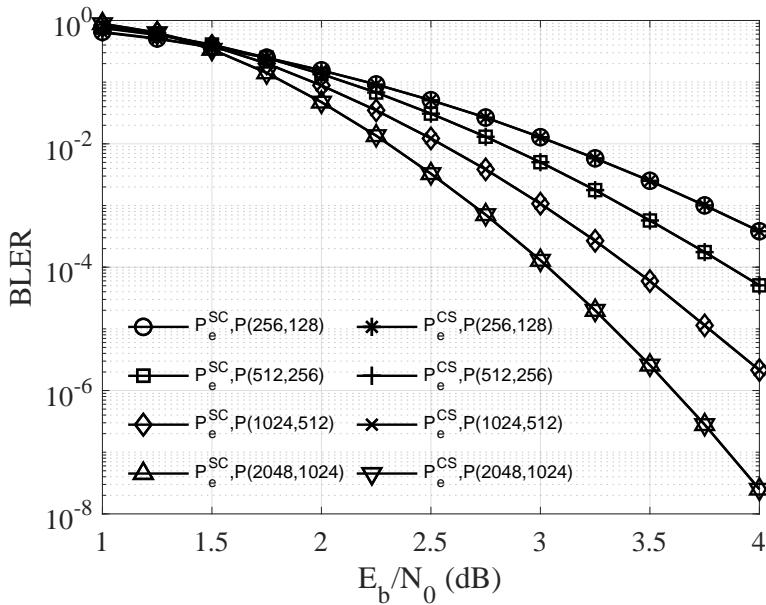
$$P_e(W_N^{(i)}) = Q(\sqrt{E\{L_N^{(i)}\}/2}).$$

其中 $L_N^{(i)} = \ln \frac{W_N^{(i)}(\mathbf{y}_1^N, \mathbf{u}_1^{i-1} | 0)}{W_N^{(i)}(\mathbf{y}_1^N, \mathbf{u}_1^{i-1} | 1)}$ 。

显然 ξ 直观上反映了“CS中的信息比特的错误率是 $\mathcal{A} \setminus \text{CS}$ 中信息比特错误率的多少倍”，当然这只是个不严谨的说法。下面的几张图说明了在各种情况下 ξ 的值都很大，这表明了CS中的信息比特的错误率大。

如果你觉得 ξ 不严谨，我们可以轻松地写出下面的条件概率连乘，它表示SC译码器正确译码的概率：

$$\begin{aligned} \Pr(\hat{u}_1 = u_1, \hat{u}_2 = u_2, \dots, \hat{u}_N = u_N) &= \Pr(\hat{u}_1 = u_1) \Pr(\hat{u}_2 = u_2 | \hat{u}_1 = u_1) \\ &\times \Pr(\hat{u}_3 = u_3 | \hat{u}_1 = u_1, \hat{u}_2 = u_2) \dots \Pr(\hat{u}_N = u_N | \hat{u}_1 = u_1, \dots, \hat{u}_{N-1} = u_{N-1}) \quad (4.18) \\ &\stackrel{(a)}{=} \prod_{i=1}^N (1 - P_e(W_N^{(i)})) \stackrel{(b)}{=} \prod_{i \in \mathcal{A}} (1 - P_e(W_N^{(i)})). \end{aligned}$$

图 4.19 P_e^{CS} 和 P_e^{SC} 的对比

其中(a)使用了高斯近似, (b)是因为冻结比特错误率为0。则SC译码器错误译码概率为:

$$P_e^{\text{SC}} = 1 - \prod_{i \in \mathcal{A}} (1 - P_e(W_N^{(i)})). \quad (4.19)$$

我们考虑位于CS中的信息比特的对于错误概率的“贡献”：

$$P_e^{\text{CS}} = 1 - \prod_{i \in \text{CS}} (1 - P_e(W_N^{(i)})). \quad (4.20)$$

$P_e^{\text{CS}} < P_e^{\text{SC}}$ 显然成立, 那么 P_e^{CS} 到底小了多少? 答案是几乎没变小, 如图4.4所示。这说明CS中的信息比特“贡献”了几乎全部的错误率, 所以CS中的比特不可靠, 在比特翻转过程中是首选的被翻转比特。

上面只讲了单一比特翻转SC译码器, 事实上在一次错误的SC译码中, 仅纠正第一个错误比特有时可能无济于事, 因为它后面还有错误的信息比特: 给定 u_1, \dots, u_{s-1} 的正确值, u_s 的译码出错了, u_s 是首个错误判决, 然后我们幸运地识别出了 u_s , 并且把 u_s 翻转为正确值, 随后 u_s, \dots, u_{t-1} 的译码也是正确的, 这相当于 $u_1, u_2, \dots, u_s, \dots, u_{t-1}$ 全部译码正确, 但是 u_t 的译码又错误了, u_t 是第二个错误判决。在这种情况下, 在一次比特翻转SC译码过程中, 我们必须同时翻转 u_s 和 u_t 的值才有可能获得正确的译码结果, 这时问题就变得困难了: 识别两

个错误比特比识别一个错误比特难得多。问题也会变得更困难：在某次错误的SC译码中存在 $u_{i_1}, u_{i_2}, \dots, u_{i_t}$ 是必须要纠正错误判决， $t \geq 3$ ，这时必须把这 t 个错误比特全部找出来并且执行比特翻转。

现在已经存在多比特翻转SC译码来解决上面的困难，例如文献[18][19]。这些译码器在一次试探性比特翻转SC译码过程中选出多个信息比特 $u_{i_1}, u_{i_2}, \dots, u_{i_t}, i_1 < i_2 < \dots < i_t$ 进行翻转（按照其对应LLR的反方向判决），因此有机会纠正更多的错误，但也引入了更大的复杂度：从 K 个信息比特里选出 t 个不可靠信息比特有 C_K^t 种选法，必须进行大量的试探性比特翻转译码才有可能找出来。有兴趣的读者可以阅读文献[18][19]。

有时候分析比特翻转SC译码器的BLER值的下界（也就是其所能达到的最佳性能）会让我们更清醒一些。这可以通过“精灵的帮助”（Genie-aided）来实现，说白了就是把发送信息 \mathbf{u}_1^N 的值在仿真程序里直接输入给译码器，让译码器自己对照译码结果和正确值。如果发现有一个 u_i 的译码值和 u_i 的正确值不符，则译码器就用 u_i 的正确值覆盖错误的译码结果，同时给比特 u_i 记一个错误。这样一趟SC译码走下来，译码器可能给若干个 $u_{i_1}, u_{i_2}, \dots, u_{i_t}$ 记录了错误。

大量运行重复的SC译码过程，我们会得到以下数据：

- (1). 总运行次数 N_{\max} ；
- (2). 事件“在一次SC译码过程中，存在 i 个需要纠正的判决”出现的次数 T_i 。

纠正 i 个错误的比特翻转SC译码器肯定不能纠正大于 i 个错误。记 P_e^i 为“每次至多能纠正 i 个错误的SC译码器的错误率”，则 $P_e^i \geq \sum_{k=i+1} T_k / N_{\max}$ ，其中等号仅在该译码器有能力纠正所有不超过 i 个错误时才能取到，因此 $\sum_{k=i+1} T_k / N_{\max}$ 是 P_e^i 的下界。

比特翻转SC译码器BLER性能的下界如图4.20所示，其中Genie-aided*i*表示该译码器一定能够纠正不超过*i*个错误判决的信息比特。

§ 4.5 SC译码器在编码中的应用

SC译码器在编码中竟然有用？那是当然的。

4.5.1 用SC译码器进行系统极化码的编码

这种方法来自文献[8]，巨佬在提出这个方法的时候只是一笔带过，后来人们在进行系统极化码编码时也不提及此方法，但是我认为有必要说一下。这并不是因为这个方法很重要，而是因为分析这个方法可以使我们更好地认识极化码。

本小节用 \mathcal{A} 表示信息比特集合， $|\mathcal{A}| = K$ 本小节中的 \mathcal{A} 不是任意的， \mathcal{A} 必须符合上一章讲的极化规律（或者说是极化信道偏序关系）：在长度为 $N = 2^n$ 的极化码中，设 $1 \leq i, j \leq N =$

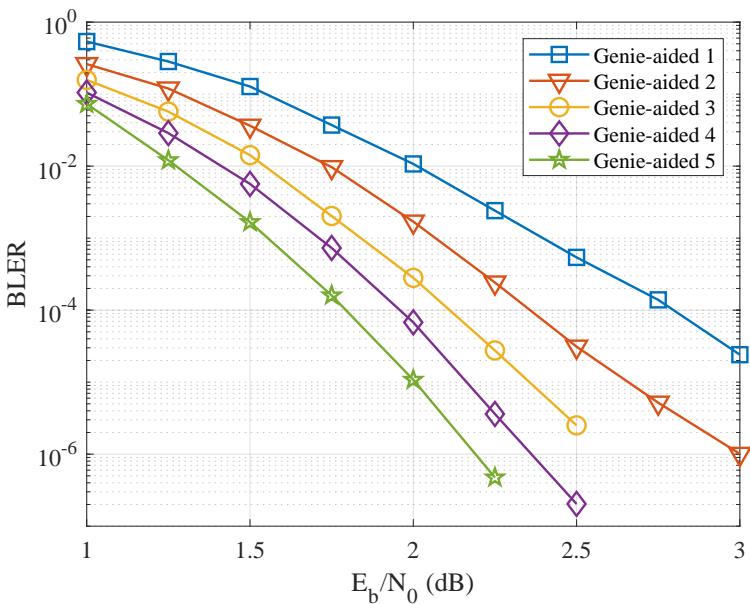


图 4.20 Genie-aided比特翻转SC译码器的BLER性能，仿真码字 $P(1024, 512)$ 由高斯近似在 $E_b/N_0 = 2.5$ dB处构造

2^n , $i-1, j-1$ 的 n 位二进制展开分别是 $(i_n, i_{n-1}, \dots, i_1)$, $(j_n, j_{n-1}, \dots, j_1)$, 其中左边是高位, 且 $i_k \geq j_k$ 对任意 $1 \leq k \leq n$ 成立, 这种现象也记作 $j \preceq i$ 。如果 $j \in \mathcal{A}$, 则 $i \in \mathcal{A}$ 。

上一章里已经讲过, 系统极化码的编码过程的矩阵乘法的写法如下 (默认冻结比特全部为0, $\mathbf{G} = \mathbf{F}^{\otimes n}$) :

$$\begin{aligned}
 \mathbf{x} &= \mathbf{u}_{\mathcal{A}} \mathbf{G}_{\mathcal{A}} \\
 (\mathbf{x}_{\mathcal{A}}, \mathbf{x}_{\mathcal{A}^c}) &= (\mathbf{u}_{\mathcal{A}} \mathbf{G}_{\mathcal{A}\mathcal{A}}, \mathbf{u}_{\mathcal{A}} \mathbf{G}_{\mathcal{A}\mathcal{A}^c}) \\
 \mathbf{x}_{\mathcal{A}} &= \mathbf{u}_{\mathcal{A}} \mathbf{G}_{\mathcal{A}\mathcal{A}} \\
 \mathbf{x}_{\mathcal{A}^c} &= \mathbf{u}_{\mathcal{A}} \mathbf{G}_{\mathcal{A}\mathcal{A}^c} = \mathbf{x}_{\mathcal{A}} \mathbf{G}_{\mathcal{A}\mathcal{A}}^{-1} \mathbf{G}_{\mathcal{A}\mathcal{A}^c}
 \end{aligned} \tag{4.21}$$

其中 $\mathbf{u}_{\mathcal{A}}$ 表示 \mathbf{u} 中位于 \mathcal{A} 内的元素形成的子向量, $\mathbf{x}_{\mathcal{A}}$ 和 $\mathbf{x}_{\mathcal{A}^c}$ 同理。 $\mathbf{G}_{\mathcal{A}}$ 表示 \mathbf{G} 中行号在 \mathcal{A} 中的行形成的子矩阵, $\mathbf{G}_{\mathcal{A}\mathcal{A}}$ 表示 $\mathbf{G}_{\mathcal{A}}$ 中列号在 \mathcal{A} 中的列形成的子矩阵, $\mathbf{G}_{\mathcal{A}\mathcal{A}^c}$ 表示 $\mathbf{G}_{\mathcal{A}}$ 中列号在 \mathcal{A}^c 中的列形成的子矩阵, $\mathbf{G}_{\mathcal{A}\mathcal{A}}$ 可逆是显然的, 因为它是下三角阵且对角线元素全为1。

定义 4.6 用SC译码器进行系统极化码编码

(1).生成待传输数据 \mathbf{d}_1^K ;

(2). 生成序列 L_1^K : $L_1^K = [L_1, \dots, L_K] = [(1-2d_1) \times \infty, \dots, (1-2d_K) \times \infty]$, 即如果 $d_i = 0$, 则 $L_i = +\infty$; 如果 $d_i = 1$, 则 $L_i = -\infty$ 。

(3). 构造长度为 N 的LLR序列 llr : $\text{llr}_{\mathcal{A}} = L_1^K$, $\text{llr}_{\mathcal{A}^c} = \mathbf{0}$ 。其中 $\text{llr}_{\mathcal{A}}$ 表示 llr 中位于索引集合 \mathcal{A} 上的元素, $\text{llr}_{\mathcal{A}^c}$ 表示 llr 中位于索引集合 \mathcal{A}^c 上的元素, $\mathbf{0}$ 是全0向量。

(4). 将 llr 作为接收LLR输入SC译码器, SC译码器(二叉树形式的根节点处)获得码字估计 \mathbf{x} , 则 \mathbf{x} 是 \mathbf{d}_1^K 对应的系统码字。

热情的读者马上就发现了问题: 上述步骤(2),(3),(4)等价于把 \mathbf{x} 放在BEC信道里传输¹, 恰好 $\mathbf{x}_{\mathcal{A}}$ 中的元素都没有被擦除, 而 $\mathbf{x}_{\mathcal{A}^c}$ 中的元素全部被擦除, 然后把接收信号输入SC译码器进行译码。为什么如此操作, 在SC译码器二叉树形式的根节点处能得到系统码字 \mathbf{x} ? 下面将说明这个问题, 这里我们注意到: 信息比特集合 \mathcal{A} 不是任意的, \mathcal{A} 必须符合极化规律: 在长度为 $N = 2^n$ 的极化码中, 设 $1 \leq i, j \leq N = 2^n$, $i-1, j-1$ 的 n 位二进制展开分别是 $(i_n, i_{n-1}, \dots, i_1), (j_n, j_{n-1}, \dots, j_1)$, 其中左边是高位, 且 $i_k \geq j_k$ 对任意 $1 \leq k \leq n$ 成立 ($j \prec i$)。如果 $j \in \mathcal{A}$, 则 $i \in \mathcal{A}$ 。

定理 4.5 设 \mathcal{A} 满足上述极化规律, llr 是通过定义 4.6 中的操作获得的, 将 llr 输入 SC 译码器, SC 译码器二叉树形式的根节点处会获得码字估计 \mathbf{x} , 则 $\mathbf{x}_{\mathcal{A}}$ 是 $\text{llr}_{\mathcal{A}}$ 的硬判决结果。

证明 使用归纳法。

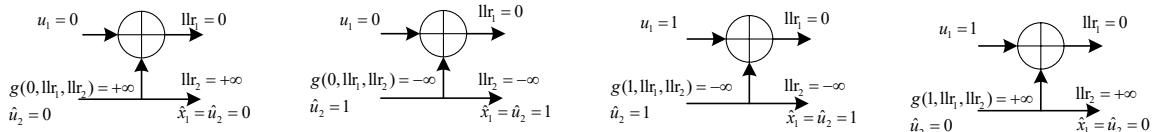
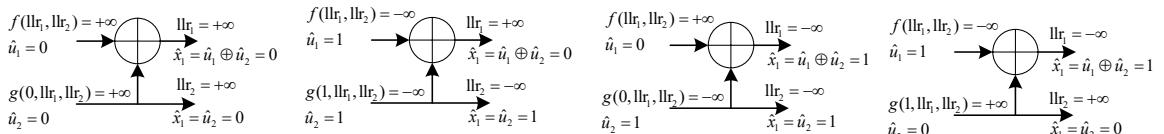
源头: 验证 $N = 2$ 时定理结论成立。当 $N = 2$ 时, \mathcal{A} 有共计存在 4 种情况: $\mathcal{A} = \emptyset$, $\mathcal{A} = \{1\}$, $\mathcal{A} = \{2\}$, $\mathcal{A} = \{1, 2\}$, 其中 $\mathcal{A} = \{1\}$ 不符合极化规律, 不对它进行讨论。因为本定理只关心 $\mathbf{x}_{\mathcal{A}}$ 是否为 $\text{llr}_{\mathcal{A}}$ 的硬判决结果, 而当 $\mathcal{A} = \emptyset$ 时, $\mathbf{x}_{\mathcal{A}}$ 和 $\text{llr}_{\mathcal{A}}$ 都不存在, 所以不考虑 $\mathcal{A} = \emptyset$ 。由此, 只剩下 $\mathcal{A} = \{2\}$ 和 $\mathcal{A} = \{1, 2\}$ 两种情况。

如果 $\mathcal{A} = \{2\}$, 那么存在如图 4.21 所示的 4 种情况。图 4.21 中左边两个 2×2 模块表示冻结比特取 0 的情况, 右边两个 2×2 模块表示冻结比特取 1 的情况。图 4.21 说明, 无论冻结比特取什么值, 在 $\mathcal{A} = \{2\}$ 的情况下定理结论成立: \hat{x}_2 总是等于 llr_2 的硬判决。

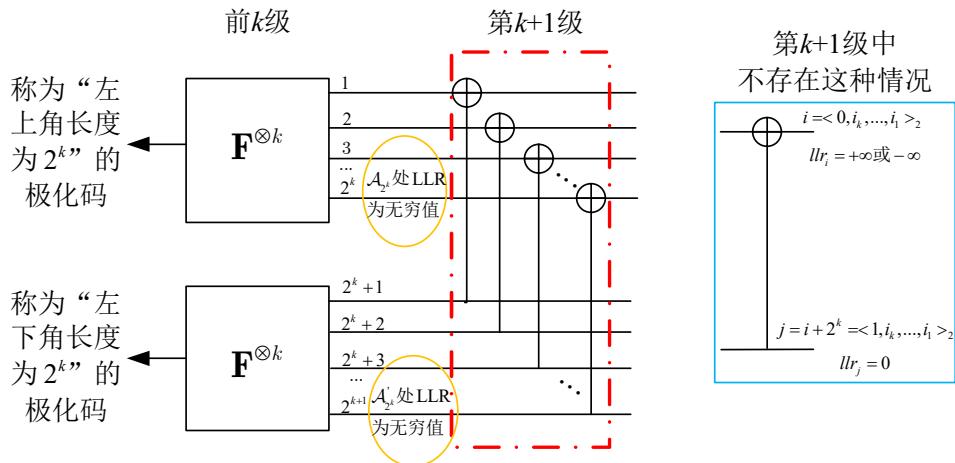
如果 $\mathcal{A} = \{1, 2\}$, 那么长度为 2 的极化码在 BEC 中传输的所有可能结果如图 4.22 所示。由图 4.22 可见, 当 $\mathcal{A} = \{1, 2\}$ 时, SC 译码器获得的 \hat{x}_i 总是等于 llr_i 的硬判决, $i \in \{1, 2\}$ 。

归纳假设: $N = 2^k$ 时, 定理结论成立: \mathcal{A} 满足极化规律, llr 是通过定义 4.6 中的操作获得的, 将 llr 输入 SC 译码器, SC 译码器二叉树形式的根节点处会获得码字估计 \mathbf{x} , 则 $\mathbf{x}_{\mathcal{A}}$ 是 $\text{llr}_{\mathcal{A}}$ 的硬判决结果。

¹ 注意到在BEC中有: $\ln \frac{\Pr(0|0)}{\Pr(0|1)} = +\infty$, $\ln \frac{\Pr(?|0)}{\Pr(?|1)} = 0$, $\ln \frac{\Pr(1|0)}{\Pr(1|1)} = -\infty$ 。在无穷的加法运算上使用柯西主值, 即 $+\infty + (-\infty) = 0$ 。

图 4.21 $\mathcal{A} = \{2\}$ 的情况图 4.22 $\mathcal{A} = \{1, 2\}$ 的情况

下一步： $N = 2^{k+1}$ 。长度为 2^k 的极化码和长度为 2^{k+1} 的极化码的关系如图4.23所示，注意到我们下面将频繁的使用措辞：“左上角长度为 2^k 的极化码”，它指的是图4.23中左上角的 $\mathbf{F}^{\otimes k}$ ；“左下角长度为 2^k 的极化码”，它指的是图4.23中左下角的 $\mathbf{F}^{\otimes k}$ 。我们必须明确下面几个问题。

图 4.23 长度为 2^k 的极化码和长度为 2^{k+1} 的极化码的关系

提示(1).第 $k+1$ 级中的 2×2 模块不存在图4.23右边的 2×2 模块的情况，因为这违反了 \mathcal{A} 的极化规律：第 $k+1$ 级 2×2 模块右边两个引脚的索引值（右上引脚的索引设为 i ，右下引脚的索引为 j ）的关系是 $i, 1 \leq i \leq 2^k, j = i + 2^k, 2^k + 1 \leq j \leq 2^{k+1}$ ，这意味着 $i-1, j-1$ 的二进制展开可以表示为 $(0, i_k, i_{k-1}, \dots, i_1), (1, i_k, i_{k-1}, \dots, i_1)$ ，即 $j-1$ 的最高位是1， $i-1$ 最高位是0，其余位相同。根据极化规律，如果 $i \in \mathcal{A}$ ，则 $j \in \mathcal{A}$ 。因此不存在图4.23右边的 $\times 2$ 模块的情况。

提示(2).图4.23左上角的长度为 2^k 的极化码（或者说是长度为 2^k 的SC译码器的根节点）会收到来自第 $k+1$ 级所有 2×2 模块利用 f 函数算出的LLR，其中有些LLR是0，有些LLR是无穷值。我们给图4.23 左上角长度为 2^k 的极化码右侧 2^k 个引脚依次标号为 $1, 2, \dots, 2^k$ ，并记左上角长度为 2^k 的极化码右侧取值为无穷的LLR所在的索引集合为 \mathcal{A}_{2^k} 。为了能使用归纳假设，我们必须说明 \mathcal{A}_{2^k} 满足极化规律：设 $i \in \{1, 2, \dots, 2^k\}$ 且 $i \in \mathcal{A}_{2^k}$ （即索引*i*处的LLR为无穷），如果存在*l*使得 $i \prec l$ ，则 $l \in \mathcal{A}_{2^k}$ ，也即索引*l*处的LLR取值为无穷。

为了说明上述问题，注意到如果某个 $i \in \{1, 2, \dots, 2^k\}$ 处的LLR为无穷，那么索引*i*在第 $k+1$ 级中对应的 2×2 模块¹右侧两个引脚的LLR必同时取值为无穷，如图4.24中的红色 2×2 模块所示。现有任取 $l \in \{1, 2, \dots, 2^k\}$ 满足 $i \prec l$ ，那么*l*处的LLR是不是无穷值？

如图4.25中绿色 2×2 模块所示，在第 $k+1$ 级右侧有 $i \prec l$ 成立²，由于第 $k+1$ 级右侧 $i \in \mathcal{A}$ （这是因为第 $k+1$ 级右侧的*i*处LLR为无穷），根据极化规律，第 $k+1$ 级右侧的 $l \in \mathcal{A}$ 成立，即第 $k+1$ 级右侧*l*的LLR为无穷；显然我们有 $i + 2^k \prec l + 2^k$ 成立³，由于第 $k+1$ 级右侧 $i + 2^k \in \mathcal{A}$ （这是因为第 $k+1$ 级右侧*i + 2^k*处LLR为无穷），根据极化规律， $l + 2^k \in \mathcal{A}$ 成立，即第 $k+1$ 级右侧*l + 2^k*处LLR为无穷。既然绿色模块右侧两个LLR都为无穷，所以绿色模块左上角的*l*处LLR为无穷⁴，从而绿色模块左上角的*l*处 $\in \mathcal{A}_{2^k}$ 。

到此，我们说明了 \mathcal{A}_{2^k} 满足极化规律：如果左上角长为 2^k 极化码的右侧的 $i \in \mathcal{A}_{2^k}$ ，则任意满足 $i \prec l$ 的*l*属于 \mathcal{A}_{2^k} 。

提示(3).图4.23左下角的长度为 2^k 的极化码（或者说是长度为 2^k 的SC译码器的根节点）会收到来自第 $k+1$ 级所有 2×2 模块利用 g 函数算出的LLR，其中有些LLR是0，有些LLR是无穷值。我们也给图4.23左下角长度为 2^k 的极化码右侧 2^k 个引脚依次标号为 $1, 2, \dots, 2^k$ ，并记左下角长度为 2^k 的极化码右侧取值为无穷的LLR所在的索引集合为 \mathcal{A}'_{2^k} 。为了能使用归纳假设，我们必须说明 \mathcal{A}'_{2^k} 满足极化规律：设 $i \in \{1, 2, \dots, 2^k\}$ 且 $i \in \mathcal{A}'_{2^k}$ （即索引*i*处的LLR为无穷），如果存在*l*使得 $i \prec l$ ，则 $l \in \mathcal{A}'_{2^k}$ ，也即索引*l*处的LLR取值为无穷。

如果某个 $i \in \{1, 2, \dots, 2^k\}$ 处的LLR为无穷，那么索引*i*在第 $k+1$ 级中对应的 2×2 模块⁵右下角的LLR必为无穷，如图4.26中的红色 2×2 模块所示。任取 $l \in \{1, 2, \dots, 2^k\}$ 满足 $i \prec l$ ，则*l*处的LLR是不是无穷值？

如图4.27中绿色 2×2 模块所示，在绿色模块右侧有 $i + 2^k \prec l + 2^k$ 成立⁶。由于第 $k+1$ 级右侧的 $i + 2^k \in \mathcal{A}$ （这是因为第 $k+1$ 级右侧*i + 2^k*处的LLR为无穷），根据极化规律，第 $k+1$ 级右侧*l + 2^k ∈ A*成立，即第 $k+1$ 级右侧*l + 2^k*处LLR为无穷。

¹即*i*是该模块的左上角

²仅仅是*i, l*的二进制展开最高位都变成了0，其余位的关系被继承

³仅仅是*i, l*的二进制展开最高位都变成了1，其余位的关系被继承

⁴*l*处LLR由 f 函数计算

⁵即*i*是该模块的左下角

⁶*l + 2^k*的二进制展开最高位是1，*i + 2^k*二进制展开最高位是0，其余位完全相同，继承了*i < l*的关系

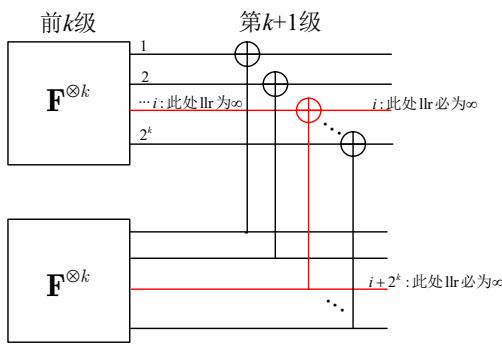


图 4.24 看红色模块，若其左上角*i*处LLR为无穷值，则其右侧两个LLR都是无穷。

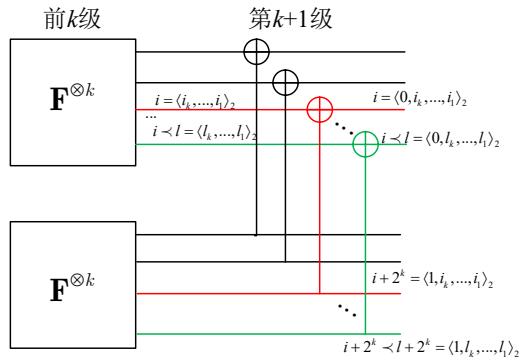


图 4.25 看绿色模块，因为红色模块右侧两个LLR都是无穷，而*i \prec l, i + 2^k \prec l + 2^k*，所以绿色右侧两个LLR都是无穷，所以绿色模块左上角*l*处LLR为无穷。

现在问题分为两种情况：

如果图4.27中绿色模块右上角的LLR是0，那么显然无论绿色模块在进行 g 函数计算时使用的比特是0还是1，绿色模块左下角的LLR必为无穷。

如果图4.27中绿色模块右上角的LLR是无穷，则对图4.27左上角长为 2^k 的极化码使用归纳假设：在绿色模块右侧两个LLR都是无穷的情况下，绿色模块用 f 函数算出其左上角的LLR，该LLR必为无穷，则计算结果必落入图4.22的四种情况之中。归纳假设用在了这里：此时绿色模块左上角的LLR必为无穷，归纳假设保证了左上角 2^k 长的极化码在绿色模块左上角返回的比特值是绿色模块左上角LLR的硬判决结果，所以绿色模块的计算结果必落入图4.22的四种情况之中。对照图4.22的四种情况可见，绿色模块左下角的LLR必为无穷。

综上，无论图4.27中绿色模块右上角的LLR是0还是无穷，绿色模块左下角的LLR都是无穷。

既然绿色模块左下角*l*处的LLR总是为无穷，从而绿色模块左下角的*l* $\in \mathcal{A}'_{2^k}$ 。到此，我们说明了 \mathcal{A}'_{2^k} 满足极化规律：如果左下角的长度为 2^k 的极化码右侧的某个*i* $\in \mathcal{A}'_{2^k}$ ，则任意满足*i \prec l*的*l*属于 \mathcal{A}'_{2^k} 。

我们上面的讨论是为了能够对左上角和左下角的两个长为 2^k 的极化码使用归纳假设。

现在来到证明的最后一步。任取第*k+1*级中的一个2模块，称该模块为Module，则Module右侧两个LLR存在三种情况：

(1).Module右侧两个LLR全为0，即Module右侧两个引脚的索引不属于 \mathcal{A} ，不属于定理讨论的情况。

(2).Module右侧两个LLR仅有一个为0。根据前面的讨论可知，只可能是Module右上

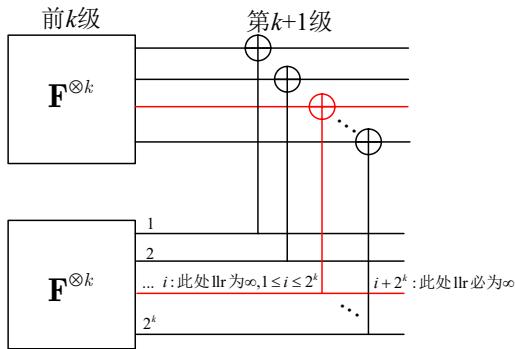


图 4.26 看红色模块，若红色模块左下角*i*处LLR为无穷值，则红色模块右下角的LLR必为无穷。

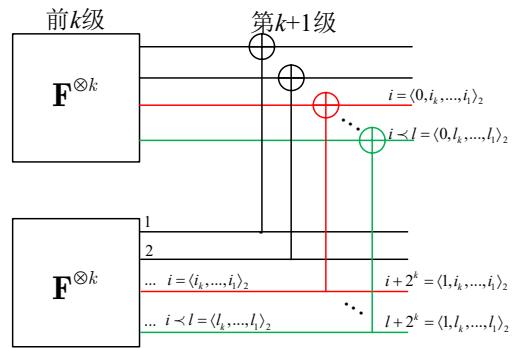


图 4.27 看绿色模块，因为红色模块右下角LLR是无穷，而*i* \prec *l*, *i* + 2^k \prec *j* + 2^k，所以绿色右下角LLR是无穷。根据文中讨论可知，无论绿色模块右上角处LLR是0还是无穷，绿色模块左下角LLR都是无穷。

角LLR为0，右下角LLR为无穷。此时，根据Module上 g 函数的计算规则，该模块左下角的LLR必等于右下角的LLR。这时我们对左下角长度为2^k的极化码使用归纳假设：左下角长度为2^k的极化码进行SC译码¹，左下角长度为2^k的极化码在Module左下角返回的比特值必是Module左下角LLR的硬判决结果。由此，Module所有可能的状态就是图4.21中的4种状态。从而，在Module右上LLR为0，且右下LLR为无穷的情况下，对照图4.21，Module右下脚由SC译码返回的比特值等于Module右下脚LLR的硬判决值。定理结论成立。

(2).Module右侧两个LLR都是无穷。Module在SC译码过程中首先执行 f 函数，得到其左上角LLR，该LLR是无穷值。现使用归纳假设：左上角长度为2^k的极化码进行SC译码²，左上角长度为2^k的极化码在Module左上角返回的比特值等于Module左上角LLR的判决值；随后，Module执行 g 函数。由图4.22可见，此时Module算出左下角LLR必为无穷。现再次使用归纳假设：左下角长度为2^k的极化码进行SC译码³，左下角长度为2^k的极化码在Module左下角返回的比特值等于Module左下角LLR的判决值。经过两次使用归纳假设，我们可以断定，在Module右侧两个LLR都是无穷的情况下，Module的所有可能状态就是图4.22中的四种状态，从而参照图4.22，Module右侧由SC译码返回的两个比特值分别等于右侧两个LLR的硬判决值。定理结论成立。

经过冗长的讨论我们终于得到了证明。但我们的任务还没结束，还差一个唯一性。

定理4.5说明了如果进行定义4.6中的过程，则可以得到一个译码码字 \mathbf{x} ，满足 \mathbf{x}_A 是 llr_A 的硬

¹译码所需LLR来自第k+1级所有2×2模块的 g 函数计算

²译码所需LLR来自第k+1级所有2×2模块的 f 函数计算

³译码所需LLR来自第k+1级所有2×2模块的 g 函数计算

判决结果。因为 $\mathbf{x}_{\mathcal{A}^c} = \mathbf{x}_\mathcal{A} \mathbf{G}_{\mathcal{A}\mathcal{A}}^{-1} \mathbf{G}_{\mathcal{A}^c}$ 是唯一的，所以我们断定我们完成了系统编码。

既然可以通过一次SC译码完成系统极化码编码，那么定义4.6中的编码过程的复杂度就是 $O(N \log_2 N)$ ，属于低复杂度编码。

本小节的最后，对BEC信道中的SC译码器做一点提示。BEC信道中接收LLR取值为 $\{+\infty, 0, -\infty\}$ ，但我们在编制程序的时候不便使用无穷值。结合 f 和 g 函数的运算规则不难发现，我们可以用整数+1代替 $+\infty$ ，用整数-1代替 $-\infty$ ，0还是0。这样的代换使译码的数值稳定。

4.5.2 使用SC译码器进行Monte-Carlo码构造

这个方法在文献[3]中就被提出了，它的原理很简单：通过重复运行大量的SC译码，找出可靠性高的比特信道。

设信息传输过程中某一次的实现值为 $(\mathbf{y}_1^N, \mathbf{u}_1^N)$ 。我们直接把 \mathbf{u}_1^N 的发送值输入给SC译码器。假设当前待译码比特为 u_i ，使用LLR的SC译码器进行传统的译码过程，最后计算得到判决量 $L_N^{(i)} = \ln \frac{\Pr(\mathbf{y}_1^N, \mathbf{u}_1^{i-1} | u_i=0)}{\Pr(\mathbf{y}_1^N, \mathbf{u}_1^{i-1} | u_i=1)}$ 并对 $L_N^{(i)}$ 进行硬判决。这时SC译码器也去看看 u_i 的发送值是什么。如果 u_i 的发送值等于当前的判决值，那么无事发生，SC译码器继续译码 u_{i+1} ；如果当前的判决值不等于 u_i 的发送值，那就给索引*i*记一个错误，同时使用 u_i 的发送值覆盖当前判决值，继续译码 u_{i+1} 。

在重复了大量的上述过程后，每一个索引 $i, i \in \{1, 2, \dots, N\}$ 处记录了数量不等的错误次数。显然我们选择错误次数最少的 K 的信道索引号作为集合 \mathcal{A} 。

有关SC译码的初级知识到这里就全部讲完了。

第5章 串行抵消列表译码

极化码的串行抵消列表译码（Successive Cancellation List, SCL）器[20]是目前极化码应用最广泛的译码器。我在这里根本不打算用一两段白话描述SCL译码器，因为说不清楚。初学者也不要指望看一段说明就懂了SCL译码器，我们要严肃点。

最好看完上一章再看本章，一是因为上一章中有些记号已经出现过，本章可能不再解释；二是因为SC译码不熟练的话，SCL译码就无法理解。本章中我们约定冻结比特总是取0。

§ 5.1 基于LLR的路径度量

我们先引入基于LLR的路径度量（Path Metric, PM），再讨论SCL算法，磨刀不误砍柴工。考虑长度为 $N = 2^n$ 的极化码，自然地，我们关心的联合分布为 $\Pr(\mathbf{U}_1^N, \mathbf{Y}_1^N)$ 。

所谓路径度量就是某个译码结果的后验概率： $\Pr(\mathbf{u}_1^i | \mathbf{y}_1^N)$ 。这个值越大， \mathbf{u}_1^i 正确的概率越大，顺着 \mathbf{u}_1^i 继续用SC译码器译码 u_{i+1}, \dots, u_N ，最终译码正确的概率就越大。

记 $L_N^{(i)} = \ln \frac{\Pr(\mathbf{y}_1^N, \mathbf{u}_1^{i-1} | u_i=0)}{\Pr(\mathbf{y}_1^N, \mathbf{u}_1^{i-1} | u_i=1)}$ 。文献[21]中的定理表明， $\Pr(\mathbf{u}_1^i | \mathbf{y}_1^N)$ 可以用 $L_N^{(1)}, L_N^{(2)}, \dots, L_N^{(i)}$ 表示。虽然文献[21]给了证明，但是他的叙述不连贯，我翻证一下。

定理 5.1 $-\ln \Pr(\mathbf{u}_1^i | \mathbf{y}_1^N) = \sum_{k=1}^i \ln(1 + e^{-(1-2u_k)L_N^{(k)}})$ ， $-\ln \Pr(\mathbf{u}_1^i | \mathbf{y}_1^N)$ 称为路径度量，这里“路径”指的是译码路径，也即译码序列 \mathbf{u}_1^i 。

证明 我们用类似多米诺骨牌的方法完成证明，考虑下面的比值：

$$\begin{aligned} \frac{\Pr(\mathbf{u}_1^{i-1}|\mathbf{y}_1^N)}{\Pr(\mathbf{u}_1^i|\mathbf{y}_1^N)} &= \frac{\Pr(\mathbf{y}_1^N, \mathbf{u}_1^{i-1})}{\Pr(\mathbf{y}_1^N, \mathbf{u}_1^{i-1}, u_i)} \\ &= \frac{\Pr(\mathbf{y}_1^N, \mathbf{u}_1^{i-1}|u_i = 0) \Pr(u_i = 0) + \Pr(\mathbf{y}_1^N, \mathbf{u}_1^{i-1}|u_i = 1) \Pr(u_i = 1)}{\Pr(\mathbf{y}_1^N, \mathbf{u}_1^{i-1}|u_i) \Pr(u_i)} \quad (5.1) \\ &\stackrel{(a)}{=} 1 + \left[\frac{\Pr(\mathbf{y}_1^N, \mathbf{u}_1^{i-1}|u_i = 0)}{\Pr(\mathbf{y}_1^N, \mathbf{u}_1^{i-1}|u_i = 1)} \right]^{-(1-2u_i)} \end{aligned}$$

(a): (1).无论 $u_i = 0$ 还是 1 , $\Pr(u_i) = 0.5$, 分子分母上这一项约去; (2).分别取 $u_i = 0$ 和 1 可以验证(a)成立。

上式其实是递归的关系, 考虑下面的连乘:

$$\begin{aligned} \frac{\Pr(u_1|\mathbf{y}_1^N)}{\Pr(\mathbf{u}_1^i|\mathbf{y}_1^N)} &= \frac{\Pr(u_1|\mathbf{y}_1^N)}{\Pr(\mathbf{u}_1^2|\mathbf{y}_1^N)} \frac{\Pr(\mathbf{u}_1^2|\mathbf{y}_1^N)}{\Pr(\mathbf{u}_1^3|\mathbf{y}_1^N)} \cdots \frac{\Pr(\mathbf{u}_1^{i-1}|\mathbf{y}_1^N)}{\Pr(\mathbf{u}_1^i|\mathbf{y}_1^N)} \\ &= \{1 + \left[\frac{\Pr(\mathbf{y}_1^N, u_1|u_2 = 0)}{\Pr(\mathbf{y}_1^N, u_1|u_2 = 1)} \right]^{-(1-2u_2)}\} \cdots \{1 + \left[\frac{\Pr(\mathbf{y}_1^N, \mathbf{u}_1^{i-1}|u_i = 0)}{\Pr(\mathbf{y}_1^N, \mathbf{u}_1^{i-1}|u_i = 1)} \right]^{-(1-2u_i)}\} \quad (5.2) \end{aligned}$$

我们同时注意到:

$$\begin{aligned} \frac{1}{\Pr(u_1|\mathbf{y}_1^N)} &= \frac{\Pr(\mathbf{y}_1^N)}{\Pr(\mathbf{y}_1^N, u_1)} \\ &= \frac{\Pr(\mathbf{y}_1^N|u_1 = 0) \Pr(u_1 = 0) + \Pr(\mathbf{y}_1^N|u_1 = 1) \Pr(u_1 = 1)}{\Pr(\mathbf{y}_1^N|u_1) \Pr(u_1)} \quad (5.3) \\ &= \{1 + \left[\frac{\Pr(\mathbf{y}_1^N|u_1 = 0)}{\Pr(\mathbf{y}_1^N|u_1 = 1)} \right]^{-(1-2u_1)}\} \end{aligned}$$

把式(5.2)第一个等号左侧的 $\Pr(u_1|\mathbf{y}_1^N)$ 除到右边, 并将式(5.3)带入, 再取对数:

$$-\ln \Pr(\mathbf{u}_1^i|\mathbf{y}_1^N) = \sum_{k=1}^i 1 + \left[\frac{\Pr(\mathbf{y}_1^N, \mathbf{u}_1^{k-1}|u_k = 0)}{\Pr(\mathbf{y}_1^N, \mathbf{u}_1^{k-1}|u_k = 1)} \right]^{-(1-2u_k)} \quad (5.4)$$

显然 $\frac{\Pr(\mathbf{y}_1^N, \mathbf{u}_1^{k-1}|u_k = 0)}{\Pr(\mathbf{y}_1^N, \mathbf{u}_1^{k-1}|u_k = 1)} = e^{L_N^{(k)}}$, 证毕。

定理5.1含义丰富。

一、定理5.1说明了比特序列 \mathbf{u}_1^i 后验概率 $\Pr(\mathbf{u}_1^i|\mathbf{y}_1^N)$ 可以用SC译码器的既往计算结果计

算：SC在译码 $u_k, 1 \leq k \leq i$ 时已经算出了 $L_N^{(k)}$ ，仅利用 u_k 和 $L_N^{(k)}$ 就能表示 $\Pr(\mathbf{u}_1^i | \mathbf{y}_1^N)$ 。

二、我们都希望译码结果正确，所以 $\Pr(\mathbf{u}_1^i | \mathbf{y}_1^N)$ 的值越大越好，从而 $-\ln \Pr(\mathbf{u}_1^i | \mathbf{y}_1^N)$ 的值越小越好。我们一般认为 $-\ln \Pr(\mathbf{u}_1^i | \mathbf{y}_1^N)$ 的初始值为零，即定理5.1中的累加从0开始，加 $\ln(1 + e^{-(1-2u_1)L_N^{(1)}})$ ，再加 $\ln(1 + e^{-(1-2u_2)L_N^{(2)}})$...，直到 $\ln(1 + e^{-(1-2u_N)L_N^{(N)}})$ 为止。

三、定理5.1等号两端的部分显然都恒为正实数。等号右边的项可以进行近似表示¹：如果 $u_k = 0, L_N^{(k)} \geq 0$ ，则 $\ln(1 + e^{-(1-2u_k)L_N^{(k)}}) \approx 0$ ；如果 $u_k = 1, L_N^{(k)} \geq 0$ ，则 $\ln(1 + e^{-(1-2u_k)L_N^{(k)}}) \approx L_N^{(k)}$ ；如果 $u_k = 0, L_N^{(k)} \leq 0$ ，则 $\ln(1 + e^{-(1-2u_k)L_N^{(k)}}) \approx -L_N^{(k)}$ ；如果 $u_k = 1, L_N^{(k)} \leq 0$ ，则 $\ln(1 + e^{-(1-2u_k)L_N^{(k)}}) \approx 0$ 。

从上面四种情况可见，当 u_k 的值符合 $L_N^{(k)}$ 的硬判决时， $\ln(1 + e^{-(1-2u_k)L_N^{(k)}})$ 总是约为零；反之，当 u_k 的值不等于 $L_N^{(k)}$ 的硬判决时， $\ln(1 + e^{-(1-2u_k)L_N^{(k)}})$ 总是约为 $|L_N^{(k)}|$ 。因为 $-\ln \Pr(\mathbf{u}_1^i | \mathbf{y}_1^N)$ 的值越小越好，所以 u_k 的值不等于 $L_N^{(k)}$ 的硬判决将导致 $-\ln \Pr(\mathbf{u}_1^i | \mathbf{y}_1^N)$ 增大，增大的量约为 $|L_N^{(k)}|$ ，这可以形象地理解为一种“惩罚值”。

热情的读者很快就发现，SC译码器在使用接收信号 \mathbf{y}_1^N 完成对某个 $u_i, 1 \leq i \leq N$ 的估计时，译码器里仅存在一个译码序列 \mathbf{u}_1^i 。不管 \mathbf{u}_1^i 对应的 $\Pr(\mathbf{u}_1^i | \mathbf{y}_1^N)$ 的值好或者不好，你都别无选择。能不能让SC译码器不再局限于单一序列的估计，而是获得 L 个译码序列，每个译码序列记为 $\mathbf{u}_{1,l}^i = (u_{1,l}, \dots, u_{i,l}), 1 \leq l \leq L$ ，同时使得这 L 个译码序列都尽可能正确，即使得每个 $\Pr(\mathbf{u}_{1,l}^i | \mathbf{y}_1^N), 1 \leq l \leq L$ 尽可能大？答案是可能的，这就是SCL译码器。

§ 5.2 基于LLR的SCL译码器的执行过程

提醒：初学者必须非常数量地掌握SC译码的所有细节才能继续阅读。我们先用语言描述SCL译码器的运作过程，再用一张图表示SCL译码器的运作过程，最后举一个算例表示SCL译码器的运作过程。

(1). SCL译码器内部并行地放置了 L 个SC译码器，分别记为编号为第1, 2, ..., L 个SC译码器，记为 $\text{SC}_1, \text{SC}_2, \dots, \text{SC}_N$ 。所以，大致地，SCL译码器所需存储几乎是SC译码器的 L 倍²。

(2). SCL译码器获得接收对数似然比序列 (L_1, L_2, \dots, L_N) 后，初始化路径度量为0，且仅激活1号SC译码器，其余未激活的译码器的编号为 $S_{\text{sleep}} = \{2, 3, \dots, L\}$ ，我们认为 $S_{\text{sleep}} = \{2, 3, \dots, L\}$ 是一个堆栈，即如果要从 $S_{\text{sleep}} = \{2, 3, \dots, L\}$ 弹出元素，那么弹出来的依次是 $L, L-1, \dots, 2, 1$ 。

1号SC译码器开始使用标准SC译码流程估计 u_1 的值，即计算 $L_N^{(1)} = \ln \frac{\Pr(\mathbf{y}_1^N | u_1=0)}{\Pr(\mathbf{y}_1^N | u_1=1)}$ ，也就是计算上一章中LLR数组P的第一个元素P(1)。

¹我们近似认为如果 $x > 0$ ，则 $e^x \gg 1$ ；如果 $x < 0$ ，则 $e^x \approx 0$ 。

²但这并不意味着SCL译码器的时延和译码复杂度几乎是SC译码器的 L 倍。

算出 $L_N^{(1)}$ 后，1号SC译码器判断 u_1 是否为冻结比特。如果 u_1 是冻结比特，则不激活其他SC译码器，直接令 $u_1 = 0$ ，并计算此时的路径度量为 $\ln(1 + e^{-(1-2u_1)L_N^{(1)}})$ 。然后1号译码器继续使用标准SC译码流程译码 $u_2, u_3 \dots u_{i-1}$ ，并更新路径度量为 $\sum_{k=1}^{i-1} \ln(1 + e^{-(1-2u_k)L_N^{(k)}})$ ，直到遇到 u_i ，其中 u_i 是极化码中的首个信息比特。

(3). 这时SCL译码器发现尚有 $L - 1$ 个未激活的SC译码器，则从 $S_{\text{sleep}} = \{2, 3, \dots, L\}$ 中抬弹出 L ，激活编号为 L 的SC译码器 SC_L 。 SC_L “继承了” SC_1 的全部数据，即LLR和中间比特值，也就是上一章程序中的P数组和C数组。对于初学者而言，你可以简单地把“继承”理解为直接把 SC_1 的数据全部赋值给 SC_L ¹。

此时，我们称 SC_L 是由 SC_1 克隆（Clone）而来的。在克隆时我们遵从一个原则²：不管 $L_N^{(i)}$ 的取什么值，原始译码路径（ SC_1 ）总是令 $u_i = 0$ ，克隆路径（ SC_L ）总是令 $u_i = 1$ 。这时就出现了两个译码结果 $\mathbf{u}_{1,1}^i = (u_{1,1} = 0, u_{2,1} = 0, \dots, u_{i-1,1} = 0, u_{i,1} = 0)$ 和 $\mathbf{u}_{1,L}^i = (u_{1,L} = 0, u_{2,L} = 0, \dots, u_{i-1,L} = 0, u_{i,L} = 1)$ ， $\mathbf{u}_{1,1}^i$ 和 $\mathbf{u}_{1,L}^i$ 分别对应度量值：

$$\begin{aligned}\Pr(\mathbf{u}_{1,1}^i | \mathbf{y}_1^N) &= \sum_{k=1}^{i-1} \ln(1 + e^{-(1-2u_{k,1})L_N^{(k)}}) + \ln(1 + e^{-L_N^{(i)}}), \\ \Pr(\mathbf{u}_{1,L}^i | \mathbf{y}_1^N) &= \sum_{k=1}^{i-1} \ln(1 + e^{-(1-2u_{k,L})L_N^{(k)}}) + \ln(1 + e^{L_N^{(i)}}).\end{aligned}\tag{5.5}$$

说白了就是SCL译码器同时保留了 $u_i = 0$ 和 1 这两种译码结果。

(4). 在 u_i 之后的译码中，由于 SC_1 和 SC_L 的中间比特值数组已经不同，即上一章程序中的C数组内容不同，这将导致这两个SC译码器即便使用同样的LLR进行 g 函数计算， g 函数的输出结果也不同，这使得 SC_1 和 SC_L 的译码结果从此分道扬镳。在 u_i 之后的译码中， SC_1 和 SC_L 使用各自的数据独立地译码，独立地计算路径度量，直到遇见极化码中的第二个信息比特 u_j 。这时记 SC_1 的路径度量为 $PM_1 = \sum_{k=1}^{j-1} \ln(1 + e^{-(1-2u_{k,1})L_{N,1}^{(k)}})$ ， SC_L 的路径度量为 $PM_L = \sum_{k=1}^{j-1} \ln(1 + e^{-(1-2u_{k,L})L_{N,L}^{(k)}})$ 。此时SCL译码器将执行与步骤(3)中相似的操作。

处理已经激活的 SC_1 ：从 $S_{\text{sleep}} = \{2, 3, \dots, L - 1\}$ 中弹出 $L - 1$ ，并把 SC_1 的数据都复制给 SC_{L-1} 。遵从上面的原则， SC_1 将令 $u_{j,1} = 0$ ，而 SC_{L-1} 将令 $u_{j,L-1} = 1$ 。 SC_1 对应的度量变成 $PM_1 + \ln(1 + e^{-L_{N,1}^{(j)}})$ ， SC_{L-1} 对应的度量变成 $PM_1 + \ln(1 + e^{L_{N,1}^{(j)}})$ 。

处理已经激活的 SC_{L-1} ：从 $S_{\text{sleep}} = \{2, 3, \dots, L - 2\}$ 中弹出 $L - 2$ ，并把 SC_L 的数据都复制给 SC_{L-2} 。遵从上面的原则， SC_L 将令 $u_{j,L} = 0$ ，而 SC_{L-2} 将令 $u_{j,L-2} = 1$ 。 SC_L 对应的度量变成 $PM_L + \ln(1 + e^{-L_{N,L}^{(j)}})$ ， SC_{L-2} 对应的度量变成 $PM_1 + \ln(1 + e^{L_{N,L}^{(j)}})$ 。

¹事实上存在一种赋值操作Lazy Copy[20]，Lazy Copy不直接复制数据，仅指明数据的来源。Lazy Copy对于大多数初学者而言有致命的难度，我将在本节最后的算例中说明这种方法。

²这个原则你可以自己定，别把自己绕晕了就行

说白了就是SCL译码器同时保留了 $(u_i, u_j) = (0,0)、(0,1)、(1,0)$ 以及 $(1,1)$ 这四种译码结果。

(5). 以上过程一直持续，直到遇见第 $\log_2 L$ 个信息比特 u_s ¹。这是一个临界点，因为此时 $S_{\text{sleep}} = \emptyset$ ，所有 L 个SC译码器都被激活了，都获得了各自的数据（LLR和中间比特值值）。

(6). 设 u_t 是 u_s 之后的第一个信息比特。在译码 u_t 时，所有 L 个已经激活的SC译码器各自独立地执行标准SC译码过程，每个译码器分别获得了判决 u_t 用的LLR： $L_{N,l}^{(t)}$, $1 \leq l \leq L$ ，且拥有各自的路径度量 PM_l , $1 \leq l \leq L$ 。这时每一个SC译码仍想要同时保留 $u_s = 0$ 和 1 两种译码结果，但是硬件上不允许了：我们至多只有 L 套译码器，现在要把 L 个译码结果变成 $2L$ 个，注定有 L 个译码结果存储不下。所以，SCL译码器将从 $2L$ 个译码结果中选择 L 个具有最小路径度量值的译码结果，作为“幸存”的结果。我们用下面的 $2 \times L$ 维矩阵记录 $2L$ 个度量值：

$$\mathbf{PM} = \begin{bmatrix} PM_1 + \ln(1 + e^{-L_{N,1}^{(t)}}) & \dots & PM_l + \ln(1 + e^{-L_{N,l}^{(t)}}) & \dots & PM_L + \ln(1 + e^{-L_{N,L}^{(t)}}) \\ PM_1 + \ln(1 + e^{L_{N,1}^{(t)}}) & \dots & PM_l + \ln(1 + e^{L_{N,l}^{(t)}}) & \dots & PM_L + \ln(1 + e^{L_{N,L}^{(t)}}) \end{bmatrix} \quad (5.6)$$

PM中每个元素的意义如下：**PM**第 l 列第一行的元素表示译码器 SC_l 遵循上面的原则，保存了 $u_{t,l} = 0$ 后所导致的路径度量。译码器 SC_l 遵循上面的原则，想要克隆出一个新的且保存了 $u_{t,l} = 1$ 的路径，这个新路径的度量值就是**PM**第 l 列第二行的元素。

因为定理5.1中的度量值越小越好，所以我们从**PM**中的 $2L$ 个元素里选出 L 个最小的值。此时**PM**中的每一列可能出现三种情况：

情况一： **PM**中第 $l, 1 \leq l \leq L$ 列中的两个元素都未入选 L 个最小的值，此时我们宣布 SC_l 译码器死亡，并把 l 的值压入堆栈 S_{sleep} ；

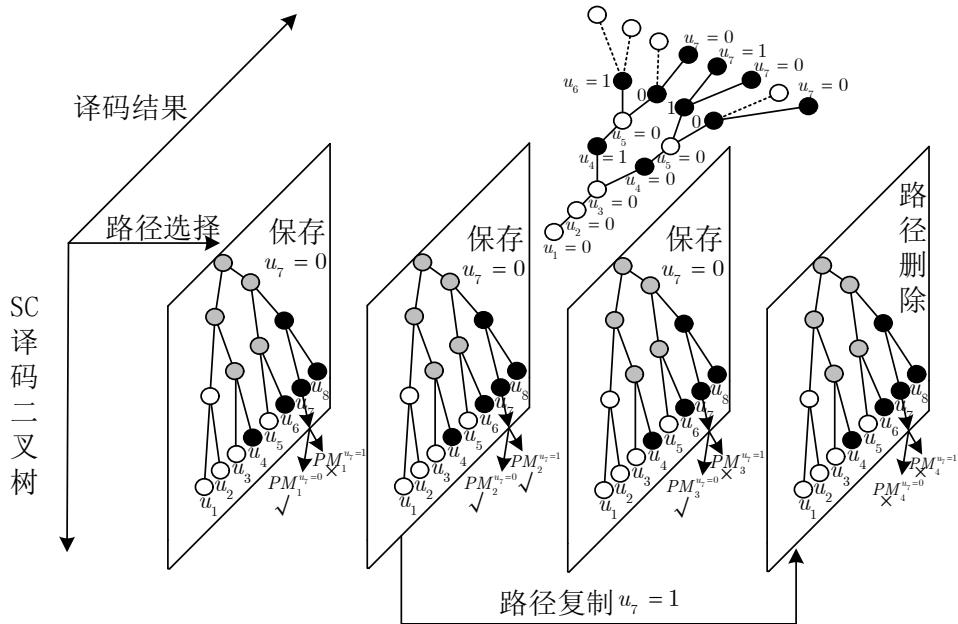
情况二： **PM**中第 $l, 1 \leq l \leq L$ 列中仅有一个元素入选 L 个最小的值，这时译码器 SC_l 仅保存入选的值对应的比特：若第 l 列第1行的值入选，则保存 $u_{t,l} = 0$ ，并更新其路径度量维 $PM_l = PM_l + \ln(1 + e^{-L_{N,l}^{(t)}})$ ；若第 l 列第2行的值入选，则保存 $u_{t,l} = 1$ ，并更新其路径度量维 $PM_l = PM_l + \ln(1 + e^{L_{N,l}^{(t)}})$ 。

情况三： **PM**中第 $l, 1 \leq l \leq L$ 列中两个元素均入选 L 个最小的值。此时我们可以发现，一旦本情况出现，则情况一必出现，否则幸存路径就多于 L 个。这时 SC_l 遵循上面的原则，保留判决值 $u_{t,l} = 0$ ，并更新路径度量 $PM_l = PM_l + \ln(1 + e^{-L_{N,l}^{(t)}})$ 。随后， SC_l 需要进行克隆。这时我们从 S_{sleep} 中弹出一个一个元素²，设这个弹出的元素为 p ，则唤醒 SC_p ：把 SC_l 的所有存储都赋值给 SC_p ，但是 SC_p 保留 $u_{t,p} = 1$ ，同时令 $PM_p = PM_l + \ln(1 + e^{L_{N,l}^{(t)}})$ 。

上面三种情况不一定会同时出现，有时候可能仅有情况二出现。

¹研究极化码的老哥们都喜欢把 L 设为2的幂，即 $L = \{2, 4, 8, 16, 32\}$ 。本讲义也设 L 是2的幂，但事实上 L 等于几都行。

²由于情况一此时必已经出现，所以 S_{sleep} 不为空

图 5.1 SCL译码器举例, $N = 8, K = 4, L = 4$

(7). u_t 之后的信息比特的译码不过是重复度步骤(6)的过程而已。看到这里你会发现我只讨论了信息比特，即第一个信息比特 u_i ，第二个信息比特 u_j ，第 $\log_2 L$ 个信息比特 u_s ，等等。如果遇到冻结比特怎么办？其实冻结比特的情况非常简单，遇到某个冻结比特 u_q 后，因为 u_q 只能等于0，所以不存在路径克隆的情况。这时每个激活的SC译码器仅需保留 $u_q = 0$ ，同时把各自的路径度量加上 $\ln(1 + e^{L_{N,l}^{(q)}})$ ，这样就完成了冻结比特 u_q 的译码。冻结比特的译码和步骤(6)中的情况二类似，只不过此时不论 $L_{N,l}^{(q)}$ 取什么值，每个激活的SC译码器都令 $u_q = 0$ 。

(8). 当完成了 u_N 的译码后，获得了 L 个译码结果，选择对应最小路径度量值的译码结果作为SCL译码器的输出。

到此我已经非常尽力地用语言叙述了SCL译码器，我们下面看一个图。图5.1画出了SCL译码器的执行过程，画这个图我也是费了不少劲。

图5.1中间的四个二叉树表示列表数 $L = 4$ 时放置的4个SC译码器，从左至右依次叫做SC₁, SC₂, SC₃, SC₄。图5.1上方的树形结构表示译码结果。虽然图5.1展示的是译码 u_7 的过程，但是这不妨碍我们开始叙述。

图5.1中的SCL译码器从 u_1 开始估计，直到 u_3 为止，都仅有最左侧的一个SC译码器是激活的，因为没遇到信息比特。这时的译码结果($u_1 = 0, u_2 = 0, u_3 = 0$)的度量记为 PM 。

u_4 是第一个信息比特，这时激活第4个SC译码器，且SC₁保留 $u_{4,1} = 0$, SC₄保留 $u_{4,4} = 1$ ，如图5.1上方的树形结构写有 u_4 那一层。这时SC₁和SC₄的路径度量分别为 PM_1 =

$PM + \ln(1 + e^{-L_{8,1}^{(4)}})$ 和 $PM_4 = PM + \ln(1 + e^{L_{8,1}^{(4)}})$ 。

u_5 是冻结比特，不进行克隆， SC_1 和 SC_4 均保留 $u_5 = 0$ ，并分别更新路径度量为 $PM_1 = PM_1 + \ln(1 + e^{-L_{8,1}^{(5)}})$ 和 $PM_4 = PM_4 + \ln(1 + e^{-L_{8,4}^{(5)}})$ 。

然后译码信息比特 u_6 ， SC_1 唤醒 SC_3 ， SC_4 唤醒 SC_2 ，此时 4 个 SC 译码器都已经激活。

然后译码 u_7 ，这时每个激活的 SC 译码器都想同时保留 $u_7 = 0$ 和 1 两个值，每个 SC 译码器都算出了 $u_7 = 0$ 和 1 时的度量值，如图 5.1 每个二叉树下 u_7 的位置处所示。经过路径度量值的排序发现， SC_1 和 SC_3 落入上面步骤(6)中的情况二，且都保留了 $u_7 = 0$ ； SC_4 落入上面步骤(6)中的情况一，它死亡了； SC_2 落入上面步骤(6)中的情况三， SC_2 保留了 $u_7 = 0$ ，并且唤醒了 SC_4 ， SC_4 获得了 SC_2 的所有数据，但保留 $u_7 = 1$ 。

信息比特 u_8 的译码过程未画出，其实 u_8 的译码只不过重复了 u_7 的过程。

到此我认为读者已经掌握了 SCL 译码的原理和操作过程，下面我将举一个算例，这个算例比上面的叙述更难一点，因为我在算例中考虑了 Lazy Copy，即克隆路径时，并不把原始路径所有数据都赋值给新的路径，只是告诉了新生路径其所需的数据从哪里读取。

例 5.1 SCL 译码器算例，带有 Lazy Copy。

配置如下(AWGN 信道传输， $E_b/N_0 = 4$ dB)：

码长 $N = 8$ ，信息比特数 $K = 4$ ，码率 $R = K/N = 0.5$ ，列表数量 $L = 4$ 。

信息比特集合 $\mathcal{A} = \{4, 6, 7, 8\}$ ，冻结比特全部取 0。

4 个信息比特为 (1111)，则 $\mathbf{u}_1^8 = (00010111)$ ，编码后 $\mathbf{x}_1^8 = \mathbf{u}_1^8 \mathbf{F}^{\otimes 3} = (01101001)$ 。

\mathbf{x}_1^8 对应的 BPSK 序列为 $\mathbf{s}_1^8 = (1, -1, -1, 1, -1, 1, 1, -1)$

用 MATLAB 随机生成一个噪声序列 $\mathbf{n}_1^8 = (-1.4, 0.5, 0.2, -0.8, -0.3, 0.2, 2.3, 1.7)$

接收信号为 $\mathbf{y}_1^8 = \mathbf{s}_1^8 + \mathbf{n}_1^8 = (-0.4, -0.5, -0.8, 0.2, -1.3, 1.2, 3.3, 0.7)$

接收对数似然比为 $\mathbf{L}_1^8 = \frac{2}{\sigma^2} \mathbf{y}_1^8 = (-2.0, -2.5, -4.0, 1.0, -6.5, 6.0, 16.6, 3.5)$ 。

下面图片 5.2-5.9 给出了所有计算过程¹。其中：

$\mathbf{P}^{[i]}$ 表示位于二叉树第 i 层（叶节点是第 1 层）的存储 LLR 的数组， $\mathbf{P}^{[i]}$ 的大小为 $2^{i-1} \times L$ 。 $\mathbf{P}^{[i]}$ 的第 l 列是 SC_l 对应的 LLR 存储。 $\mathbf{P}^{[i]}$ 的每一列用竖线隔开，表示它们隶属于各个 SC 译码器。图 5.2-5.9 中没有画出 $\mathbf{P}^{[4]}$ ，是因为 $\mathbf{P}^{[4]}$ 就是信道接收 LLR，所有 SC 译码器都使用同样的信道接收 LLR。 $P_{i,j}^{[i]}$ 表示 $\mathbf{P}^{[i]}$ 第 i 行第 j 列的元素， $P_{:,j}^{[i]}$ 表示 $\mathbf{P}^{[i]}$ 的第 j 列。

$\mathbf{C}^{[i]}$ 表示位于二叉树第 i 层（叶节点是第 1 层）的存储中间比特值的数组， $\mathbf{C}^{[i]}$ 的大小为 $2^{i-1} \times 2L$ 。 $\mathbf{C}^{[i]}$ 的第 $2l-1$ 和 $2l$ 列是 SC_l 对应的中间比特值存储。 $\mathbf{C}^{[i]}$ 的每两列用竖线隔

¹ f 运算使用近似公式。路径度量也使用近似公式：如果 u_k 是 $L_N^{(k)}$ 的硬判决结果， $\ln(1 + e^{-(1-2u_k)L_N^{(k)}}) \approx 0$ ；如果 u_k 不是 $L_N^{(k)}$ 的硬判决， $\ln(1 + e^{-(1-2u_k)L_N^{(k)}}) \approx |L_N^{(k)}|$ 。

开，表示它们隶属于各个SC译码器。 $C_{i,j}^{[i]}$ 表示 $C^{[i]}$ 第*i*行第*j*列的元素， $C_{:,j}^{[i]}$ 表示 $C^{[i]}$ 的第*j*列。由此， SC_l 拥有存储 $C_{:,2l-1}^{[i]}$ 和 $C_{:,2l}^{[i]}$ 。

带有*号的位置表示尚未被赋值的存储；

图5.2中的 $3 \times 4 = \log_2 N \times L$ 的数组就是Lazy Copy数组。记Lazy Copy 第*i*列第*j*行的元素为 $LC_{i,j}$ ，则 $LC_{i,j}$ 表示：

(i). 我们在熟练掌握SC译码后就能发现，除了 u_1 的译码只用 f 函数外，其余比特的译码均是从 g 函数开始的，而后一直用 f 函数。译码 u_1 时肯定仅有一个激活的SC译码器，不用考虑Lazy Copy。 u_1 之后的信息比特译码时需要考虑Lazy Copy。 SC_l 在计算 $L_{N,l}^{(i)}$ 时，发现 $i-1$ 的*n*位二进制展开的低位连续零有 $m, 0 \leq m \leq n-1$ 个零，这意味着 g 函数的计算将使用 $P_{:,LC_{m+2,l}}^{[m+2]}$ 中的LLR和 $C_{:,2l-1}^{[m+1]}$ 中的比特值，按照译码规则， g 函数的计算为：

$$P_{i,l}^{[m+1]} = (1 - 2C_{i,2l-1}^{[m+1]})P_{i,LC_{m+2,l}}^{[m+2]} + P_{i+2^m,LC_{m+2,l}}^{[m+2]}, 1 \leq i \leq 2^{m+1}. \quad (5.7)$$

$C_{i,2l-1}^{[m+1]}$ 角标没有Lazy Copy是因为 SC_l 对应的存储在译码 u_{i-1} 时获得了属于自己的比特值，正如下面(ii)所解释的。

执行完 g 函数，后续还需执行 f 函数：

$$P_{i,l}^{[s]} = \text{sign}(P_{i,l}^{[s+1]})\text{sign}(P_{i+2^{s-1},l}^{[s+1]}) \min\{|P_{i,l}^{[s+1]}|, |P_{i+2^{s-1},l}^{[s+1]}|\}, 1 \leq s \leq m, 1 \leq i \leq 2^{s-1} \quad (5.8)$$

这里根本没有Lazy Copy，是因为式(5.7)已经给 $P_{:,l}^{[m+1]}$ 赋予了属于 SC_l 自己的LLR 值。在式(5.7)中的 g 函数执行完毕后，剩下的 f 函数只需在 $P_{:,l}^{[m]}, \dots, P_{:,l}^{[1]}$ 中依次计算。

(ii). 我们在数掌握SC译码后就能发现，当C数组的更新没到最后一步时，计算得到的比特总是填入C数组的第二列；当C数组的更新走到最后一步时，计算得到的比特总是填入C数组的第一列。那么， SC_l 在获得 $u_{i,l}$ 的值后，如果*i*是奇数，则将 $u_{i,l}$ 填入C数组这个过程就是C数组更新的最后一步，所以把 $u_{i,l}$ 填入 $C_{1,2l-1}^{[i]}$ 。如果*i*是偶数，那么将 $u_{i,l}$ 填入 $C_{1,2l}^{[i]}$ 这个过程不是C数组更新的最后一步，C数组还需继续计算。设此时 $i-1$ 的*n*位二进制展开低位有连续的*m*个1，后续计算依次为 $C_{:,2l}^{[2]}, C_{:,2l}^{[3]}, \dots, C_{:,2l}^{[m]}, C_{:,2l-1}^{[m+1]}$ 赋值。在为 $C_{:,2l}^{[s]}, 2 \leq s \leq m$ 赋值的计算过程中，根据译码规则：

$$\begin{aligned} C_{i,2l}^{[s]} &= C_{i,2LC_{s-1,l}-1}^{[s-1]} \oplus C_{i,2l}^{[s-1]}, \\ C_{i+2^{s-2},2l}^{[s]} &= C_{i,2l}^{[s-1]}, \\ 2 \leq s \leq m, 1 \leq i \leq 2^{s-2}. \end{aligned} \quad (5.9)$$

可见，计算结果总是赋值给 SC_l 拥有的存储： $C_{:,2l-1}^{[s]}$ 和 $C_{:,2l}^{[s]}$ 。计算时左侧列元素（ \oplus 左侧的

元素) 来自Lazy Copy, 而右侧列元素来自 SC_l 自己。这是因为右侧列元素从 $u_{i,l}$ 填入 $\mathbf{C}_{1,2l}^{[1]}$ 的那一刻起, 就总是新计算出的, 且属于 SC_l , 而左侧列元素是继承来的。

在给 $\mathbf{C}_{:,2l-1}^{[m+1]}, 2 \leq s \leq m$ 赋值的计算过程中, 根据译码规则:

$$\mathbf{C}_{i,2l}^{[m+1]} = \mathbf{C}_{i,2l}^{[m]} \oplus \mathbf{C}_{i,2l}^{[m]},$$

$$\mathbf{C}_{i+2^{m-1},2l}^{[m+1]} = \mathbf{C}_{i,2l}^{[m]}, \quad (5.10)$$

$$1 \leq i \leq 2^{m-1}.$$

对照上面的式子, 逐一地看下面的图。

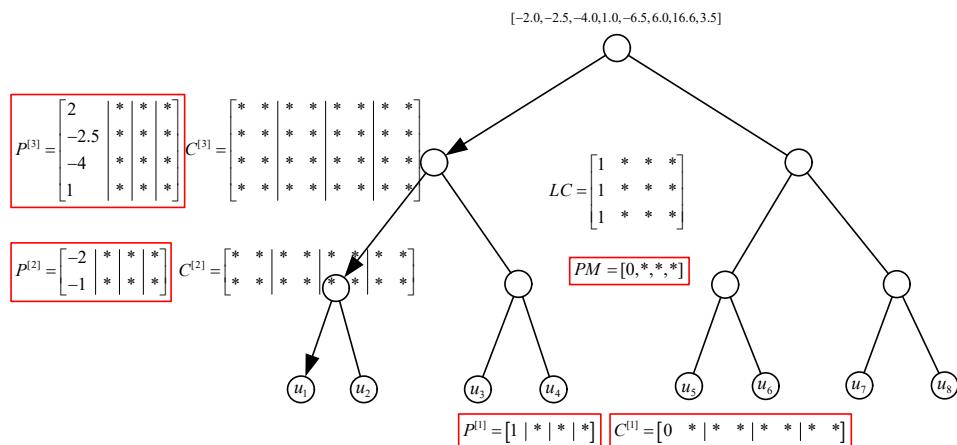


图 5.2 u_1 的译码, 红色框是本步骤的计算结果

和SC译码一样, 目前仅第一个SC译码器被激活, 其余3个SC译码器没存任何数, 用*表示。Lazy Copy第1列初始化为全1, 表示1号SC译码器所需的数据都来自它自己对应的存储。1号SC译码器算出 $L_{8,1}^{(1)} = 1$, 在自己的C数组里保留 $u_1 = 0$ 。因为 $u_1 = 0$ 符合 $L_{8,1}^{(1)} = 1$ 的硬判决, 所以1号路径度量没有加任何惩罚, 维持初始值零。

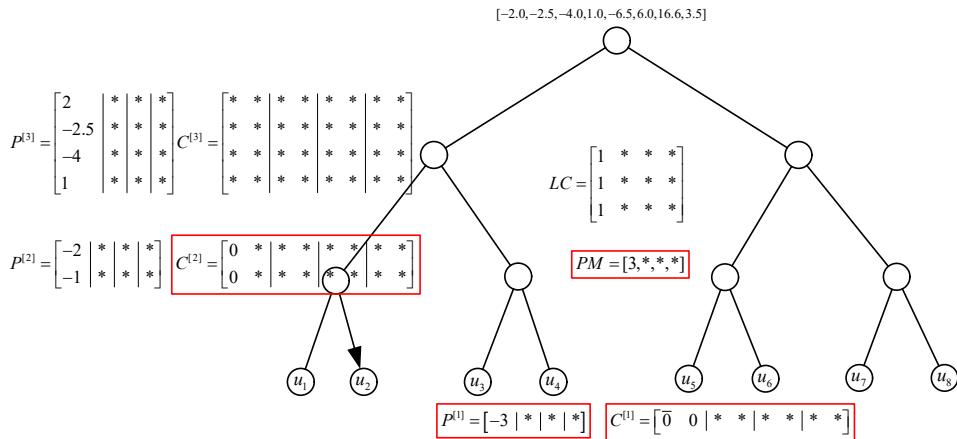


图 5.3 u_2 的译码。C 数组中带有上划线的比特是本步骤 g 函数使用的比特。红色框是本步骤的计算结果

因为 u_2 是冻结比特，所以仍仅有 1 号 SC 译码器是激活的。1 号 SC 译码器算出 $L_{8,1}^{(2)} = -3$ ，在自己的 C 数组里保留 $u_2 = 0$ 。因为 $u_2 = 0$ 不符合 $L_{8,1}^{(2)} = -3$ 的硬判决，所以 1 号路径度量加上惩罚值 $|L_{8,1}^{(2)}| = 3$ 。

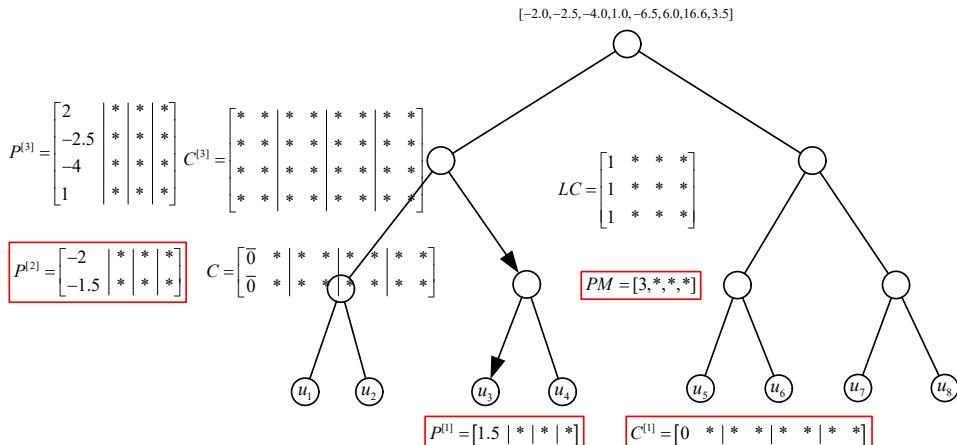


图 5.4 u_3 的译码。C 数组中带有上划线的比特是本步骤 g 函数使用的比特。红色框是本步骤的计算结果

因为 u_3 是冻结比特，所以仍仅有 1 号 SC 译码器是激活的。1 号 SC 译码器算出 $L_{8,1}^{(3)} = 1.5$ ，在自己的 C 数组里保留 $u_3 = 0$ 。因为 $u_3 = 0$ 符合 $L_{8,1}^{(3)} = 1.5$ 的硬判决，所以 1 号路径度量没有加惩罚。

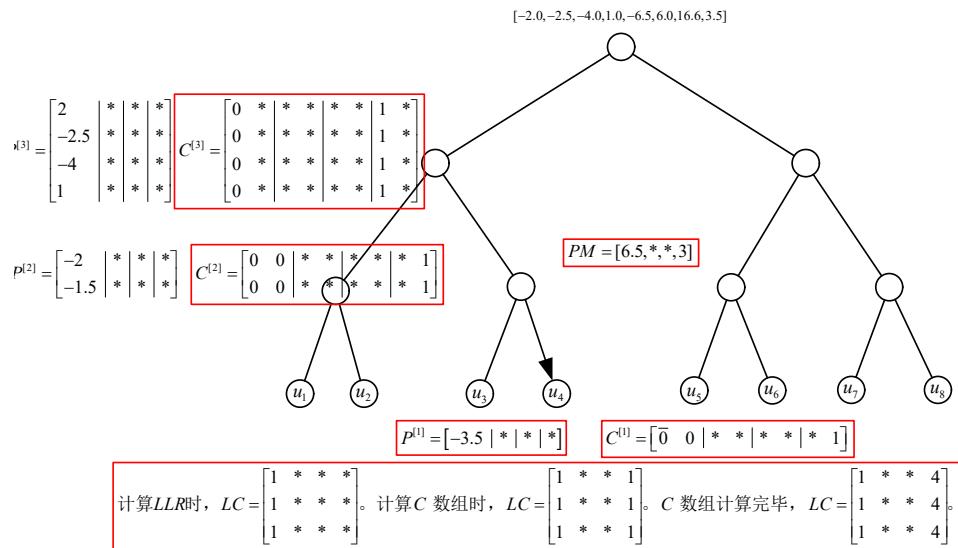


图 5.5 u_4 的译码。C数组中带有上划线的比特是本步骤 g 函数使用的比特。红色框是本步骤的计算结果

在计算 $L_{8,1}^{(4)}$ 时, 仍然只有1号SC译码器激活, 1号SC译码器算出 $L_{8,1}^{(4)} = -3.5$ 。由于 u_4 是第一个信息比特, 需要进行克隆。从 $S_{\text{sleep}} = \{2, 3, 4\}$ 中弹出4, 则4号译码器 SC_4 是 SC_1 的克隆。这时 SC_1 保留 $u_{4,1} = 0$, 并更新 SC_1 自己的C数组, 同时因为 $u_{4,1} = 0$ 和 $L_{8,1}^{(4)} = -3.5$ 的硬判决不符, 所以1号路径度量加上惩罚值 $|L_{8,1}^{(1)}| = 3.5$ 。

克隆 SC_4 时, LC 数组的第1列复制给第4列, 这表明 SC_4 所需计算数据来自 SC_1 。 SC_4 保留 $u_{4,4} = 1$, 并利用 SC_1 对应的 $C_{:,1}^{[1]}$ 和 $C_{:,1}^{[2]}$ 中的元素, 更新 SC_4 自己的C数组 $C_{:,8}^{[2]}, C_{:,7}^{[3]}$ 。同时因为 $u_{4,4} = 1$ 和 $L_{8,1}^{(4)} = -3.5$ 的硬判决相符, 所以4号路径度量没有加惩罚。

由于下一个比特是 u_5 , $5 - 1 = 4 = <100>_2$, 4的二进制开展后有 $m = 2$ 个连续的零。所有激活路径对应的LC中的列, 前 $m + 1 = 3$ 行元素修改为对应激活译码器的编号。

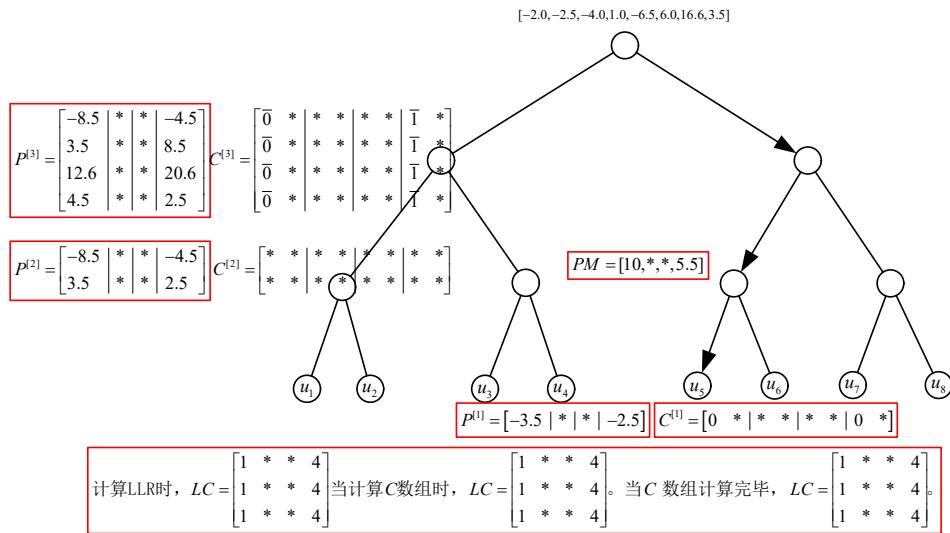


图 5.6 u_5 的译码。C数组中带有上划线的比特是本步骤 g 函数使用的比特。红色框是本步骤的计算结果，带有*的存储位置表示这个位置没有被赋值过，或者这个位置曾经被赋值，但是这个位置的值已经无用

因为 u_5 是冻结比特，所以 SC_1 和 SC_4 用各自的数据计算 $L_{8,1}^{(5)} = -3.5$ 和 $L_{8,4}^{(5)} = -2.5$ ，且都保留 $u_5 = 0$ 。 $u_5 = 0$ 不是 $L_{8,1}^{(5)} = -3.5$ 和 $L_{8,4}^{(5)} = -2.5$ 的硬判决结果，故需要给1号和4号路径分别加惩罚值 $|L_{8,1}^{(5)}| = 3.5$ 和 $|L_{8,4}^{(5)}| = 2.5$ 。

在计算 $L_{8,1}^{(5)}$ 和 $L_{8,4}^{(5)}$ 时，Lazy Copy数组无变化。当 SC_1 和 SC_4 给各自的C数组存上 $C_{1,1}^{[1]} = C_{1,7}^{[1]} = u_5 = 0$ 后，因为5是奇数，所以不再更新C数组。

下一个比特为 u_6 ， $6 - 1 = 5 = <101>_2$ ，5的二进制展开后有 $m = 0$ 个连续的零（就是没有连续的零），所有激活路径对应的LC中的列，前 $m + 1 = 1$ 行元素修改为对应激活译码器的编号（结果没有产生任何变化）。

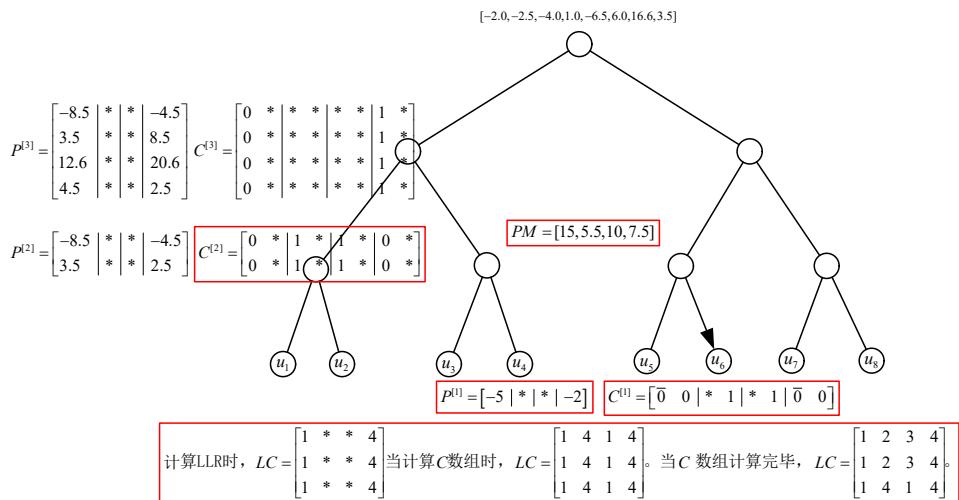


图 5.7 u_6 的译码。C数组中带有上划线的比特是本步骤 g 函数使用的比特。红色框是本步骤的计算结果

1号SC译码器算出 $L_{8,1}^{(6)} = -5$, 4号SC译码器算出 $L_{8,4}^{(6)} = -2$, 此时需要克隆。

从 $S_{\text{sleep}} = \{2, 3\}$ 中弹出3, 则3号译码器 SC_3 是 SC_1 的克隆。这时把 LC 的第一列看复制给第三列, 表示 SC_3 所需数据来自 SC_1 。 SC_1 保留 $C_{1,2}^{[1]} = u_{6,1} = 0$, 并更新 SC_1 自己的C数组: $C_{:,1}^{[2]} = [C_{1,1}^{[1]} \oplus C_{1,2}^{[1]} \ C_{1,2}^{[1]}]$ 。因为 $u_{6,1} = 0$ 和 $L_{8,1}^{(6)} = -5$ 的硬判决不符, 所以1号路径度量加上惩罚值 $|L_{8,1}^{(6)}| = 5$ 。

SC_3 保留 $C_{1,6}^{[1]} = u_{6,3} = 1$, 并更新 SC_3 自己的C数组: $C_{:,5}^{[2]} = [C_{1,1}^{[1]} \oplus C_{1,6}^{[1]} \ C_{1,6}^{[1]}]$ 。因为 $u_{6,3} = 1$ 和 $L_{8,1}^{(6)} = -5$ 的硬判决相符, 所以3号路径度量没有加惩罚值。

从 $S_{\text{sleep}} = \{2\}$ 弹出2, 则2号译码器 SC_2 是 SC_4 的克隆, 这时 $S_{\text{sleep}} = \emptyset$ 。把 LC 的第4列复制给第2列, 表示 SC_2 所需数据来自 SC_4 。 SC_4 保留 $C_{1,8}^{[1]} = u_{6,4} = 0$, 并更新 SC_4 自己的C数组: $C_{:,7}^{[2]} = [C_{1,7}^{[1]} \oplus C_{1,8}^{[1]} \ C_{1,8}^{[1]}]$ 。因为 $u_{6,4} = 0$ 和 $L_{8,4}^{(6)} = -2$ 的硬判决不符, 所以4号路径度量加上惩罚值 $|L_{8,4}^{(6)}| = 2$ 。

SC_2 保留 $C_{1,4}^{[1]} = u_{6,2} = 1$, 并更新 SC_2 自己的C数组: $C_{:,3}^{[2]} = [C_{1,7}^{[1]} \oplus C_{1,4}^{[1]} \ C_{1,4}^{[1]}]$ 。因为 $u_{6,2} = 1$ 和 $L_{8,4}^{(6)} = -2$ 的硬判决相符, 所以2号路径度量没有加惩罚值。

由于下一个比特是 u_7 , $7 - 1 = 6 = < 110 >_2$, 6的二进制开展后有 $m = 1$ 个连续的零。所有激活路径对应的 LC 中的列, 前 $m + 1 = 2$ 行元素修改为对应激活译码器的编号。

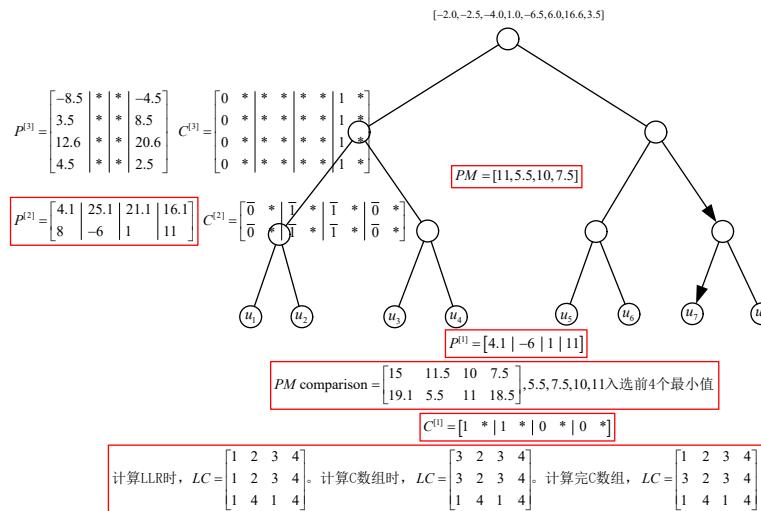


图 5.8 u_7 的译码。C数组中带有上划线的比特是本步骤 g 函数使用的比特。红色框是本步骤的计算结果

1,2,3,4号SC译码器分别算出 $L_{8,1}^{(7)} = 4.1$, $L_{8,2}^{(7)} = -6$, $L_{8,3}^{(7)} = 1$, $L_{8,4}^{(7)} = 11$ 。此时需要继续扩展路径到8条, 再淘汰其中的4条。

1号路径原度量值为15, $u_{7,1} = 0$ 和 1 对应的度量分别为 PMcomparison 第1列第1行和第1列第2行的元素, 所以 PMcomparison 的第一列为 $\begin{bmatrix} 15 \\ 19.1 \end{bmatrix}$, PMcomparison 其余列中的元素同理可得。经过排序发现 5.5, 7.5, 10, 11 入选最小的 $L = 4$ 个元素。

这意味着1号SC译码器死亡, 落入我们上述的情况一; 2, 4号SC译码器落入我们上述的情况二; 3号译码器落入上述的情况三。

对于1号译码器, 发现它死亡了, 收尸, 把编号1压入堆栈 S_{sleep} , 从而 $S_{sleep} = \{1\}$ 。

对于2号SC译码器, 此时维持LC数组的第2列不变, 2号译码器在自己的C数组中保留 $C_{1,3}^{[1]} = u_{7,2} = 1$, 因为7是奇数, 所以不再更新C数组。2号路径度量为5.5。

对于4号SC译码器, 此时维持LC数组的第4列不变, 4号译码器在自己的C数组中保留 $C_{1,7}^{[1]} = u_{7,4} = 0$, 因为7是奇数, 所以不再更新C数组。4号路径度量为7.5。

对于3号译码器, 它需要克隆, 此时从 $S_{sleep} = \{1\}$ 弹出1, 唤醒1号译码器, LC数组的第3列复制给第1列, 说明1号译码器所需数字来自3号译码器。按原则, 3号译码器在自己的C数组中保留 $C_{1,5}^{[1]} = u_{7,3} = 0$, 因为7是奇数, 所以不再更新C数组。3号路径度量为10。1号译码器在自己的C数组中保留 $C_{1,1}^{[1]} = u_{7,1} = 1$, 因为7是奇数, 所以不再更新C数组。1号路径度量为11。

完成上面所有操作后, 下一个比特是 u_8 , $8 - 1 = 7 = < 111 >_2$, 7的二进制开展后有 $m = 0$ 个连续的零。所有激活路径对应的LC中的列, 前 $m + 1 = 1$ 行元素修改为对应激活译码器的编号。

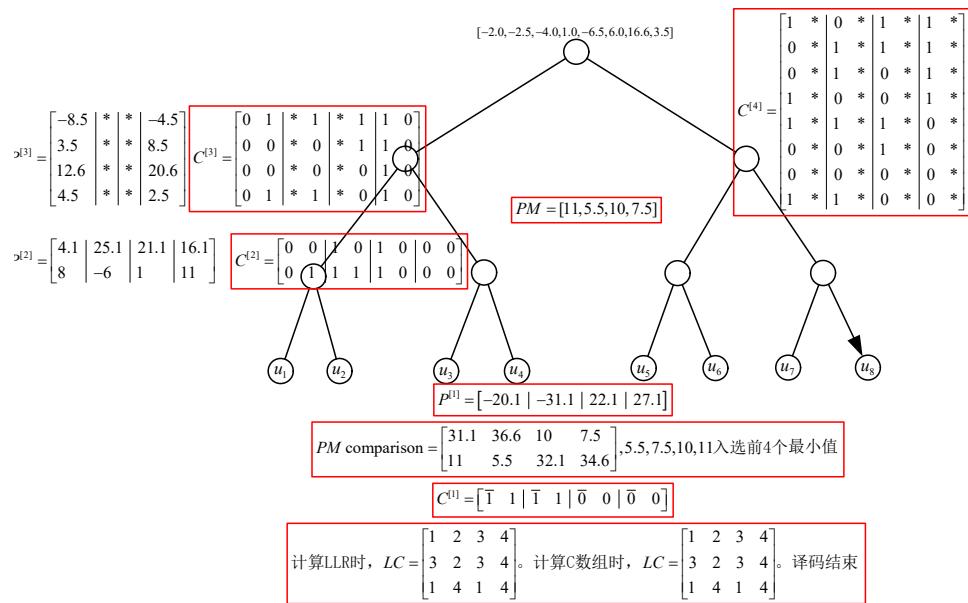


图 5.9 u_8 的译码。C数组中带有上划线的比特是本步骤 g 函数使用的比特。红色框是本步骤的计算结果

1,2,3,4号SC译码器分别算出 $L_{8,1}^{(8)} = -20.1$, $L_{8,2}^{(8)} = -31.1$, $L_{8,3}^{(8)} = 22.1$, $L_{8,4}^{(8)} = 27.1$ 。此时需要继续扩展路径到8条, 再淘汰其中的4条。

1号路径原度量值为11, $u_{8,1} = 0$ 和1对应的度量分别为PMcomparison第1列第1行和第1列第2行的元素, 所以PMcomparison的第一列为 $\begin{bmatrix} 31.1 \\ 11 \end{bmatrix}$, PMcomparison其余列中的元素同理可得。经过排序发现11,5.5,10,7.5 入选最小的 $L = 4$ 个元素。

此时1,2,3,4号译码器均落入上述的情况二, 不存在死亡, 也没有克隆。1,2,3,4号译码器分别保留 $C_{1,2l}^{[1]} = u_{8,1} = 1$, $C_{1,4}^{[1]} = u_{8,2} = 1$, $C_{1,6}^{[1]} = u_{8,3} = 0$, $C_{1,8}^{[1]} = u_{8,4} = 0$, 然后按照LC数组的指示更新各自的C数组:

$$C_{:,2l}^{[2]} = [C_{1,2l-1}^{[1]} \oplus C_{1,2l}^{[1]} \ C_{1,2l}^{[1]}], 1 \leq l \leq 4.$$

$$C_{:,2}^{[3]} = [C_{:,5}^{[2]} \oplus C_{:,2}^{[2]} \ C_{:,2}^{[2]}],$$

$$C_{:,2l}^{[3]} = [C_{:,2l-1}^{[2]} \oplus C_{:,2l}^{[2]} \ C_{:,2l}^{[2]}], l = 2, 3, 4.$$

$$C_{:,1}^{[4]} = [C_{:,1}^{[3]} \oplus C_{:,2}^{[3]} \ C_{:,2}^{[3]}],$$

$$C_{:,3}^{[4]} = [C_{:,7}^{[3]} \oplus C_{:,4}^{[3]} \ C_{:,4}^{[3]}].$$

$$C_{:,5}^{[4]} = [C_{:,1}^{[3]} \oplus C_{:,6}^{[3]} \ C_{:,6}^{[3]}],$$

$$C_{:,7}^{[4]} = [C_{:,7}^{[3]} \oplus C_{:,8}^{[3]} \ C_{:,8}^{[3]}].$$

$C_{:,1}^{[4]}, C_{:,3}^{[4]}, C_{:,5}^{[4]}, C_{:,7}^{[4]}$ 是4种可能的译码码字。由于2号路径的度量值最小，所以 $C_{:,3}^{[4]}$ 胜出， $C_{:,3}^{[4]}$ 是译码结果。对照题设可知，此时译码正确，译码结束。

从上面的算例可以看出，我们虽然复制了Lazy Copy中的列，但我们从未复制任何LLR数据和比特数据。下面是SCL译码器的MATLAB代码，其中输入参数和上一章中的SC译码器大小小异，就是多了列表数量 L 而已。从下面的代码也可以看出，我们只在第148行复制了Lazy Copy中列，在202-211行进行了Lazy Copy中编号的更新，你找不到任何复制LLR数据和中间比特值的操作。至于程序中大小为 $K \times L$ 的数组u，这是一个记录信息比特译码结果的数组，根据个人喜好，它是可有可无的。

```

1 function polar_info_esti = CASCL_decoder(llr , L, K, frozen_bits , ...
2 lambda_offset , llr_layer_vec , bit_layer_vec)
3 %LLR-based SCL decoder , a single function , no other sub-functions .
4 %const
5 N = length(llr);
6 m = log2(N);
7 %memory declared
8 lazy_copy = zeros(m, L);
9 P = zeros(N - 1, L);
10 %Channel llr is public-used , so N - 1 is enough .
11 C = zeros(2 * N - 1, 2 * L);
12 u = zeros(K, L);
13 %unfrozen bits that polar codes carry .
14 PM = zeros(L, 1);%Path metrics
15 activepath = zeros(L, 1);
16 %Indicate if a path is active. '1',active; '0',otherwise .
17 cnt_u = 1;%information bit counter
18 %initialize
19 activepath(1) = 1;
20 lazy_copy(:, 1) = 1;
21 %default: in the case of path clone ,
22 %the original path always corresponds to bit 0,
23 %while the new path bit 1.
24 for phi = 0 : N - 1
25     layer = llr_layer_vec(phi + 1);
26     phi_mod_2 = mod(phi, 2);

```

```
27     for l_index = 1 : L
28         if activepath(l_index) == 0
29             continue;
30     end
31     switch phi
32         %Decoding bits u_0 and u_N/2 needs channel LLR,
33         %so the decoding of them is separated from other bits.
34         case 0
35             index_1 = lambda_offset(m);
36             for beta = 0 : index_1 - 1
37                 P(beta + index_1 , l_index) = ...
38                     sign(1lr(beta + 1)) * ...
39                     sign(1lr(beta + index_1 + 1)) * ...
40                     min(abs(1lr(beta + 1)), ...
41                         abs(1lr(beta + index_1 + 1)));
42             end
43             for i_layer = m - 2 : -1 : 0
44                 index_1 = lambda_offset(i_layer + 1);
45                 index_2 = lambda_offset(i_layer + 2);
46                 for beta = 0 : index_1 - 1
47                     P(beta + index_1 , l_index) = ...
48                         sign(P(beta+index_2 , l_index))* ...
49                         sign(P(beta+ index_1+index_2 ,l_index))* ...
50                         min(abs(P(beta + index_2 ,l_index)), ...
51                             abs(P(beta+index_1+index_2 ,l_index)));
52                 end
53             end
54         case N/2
55             index_1 = lambda_offset(m);
56             for beta = 0 : index_1 - 1
57                 x_tmp = C(beta + index_1 , 2 * l_index - 1);
58                 P(beta + index_1 , l_index) = ...
59                     (1 - 2 * x_tmp) * 1lr(beta + 1) + ...
60                     1lr(beta + 1 + index_1);
61             end
```

```
62     for i_layer = m - 2 : -1 : 0
63         index_1 = lambda_offset(i_layer + 1);
64         index_2 = lambda_offset(i_layer + 2);
65         for beta = 0 : index_1 - 1
66             P(beta + index_1, l_index) = ...
67             sign(P(beta+index_2, l_index))*...
68             sign(P(beta+index_1+index_2, l_index))*...
69             min(abs(P(beta+index_2, l_index)), ...
70                 abs(P(beta+index_1+index_2, l_index)));
71         end
72     end
73     otherwise
74         index_1 = lambda_offset(layer + 1);
75         index_2 = lambda_offset(layer + 2);
76         for beta = 0 : index_1 - 1
77             P(beta + index_1, l_index) =...
78             (1-2*C(beta+index_1, 2*l_index-1))*...
79             P(beta+index_2, lazy_copy(layer+2, l_index))+...
80             P(beta+index_1+index_2, lazy_copy(layer+2, l_index));
81         end
82         for i_layer = layer - 1 : -1 : 0
83             index_1 = lambda_offset(i_layer + 1);
84             index_2 = lambda_offset(i_layer + 2);
85             for beta = 0 : index_1 - 1
86                 P(beta + index_1, l_index) = ...
87                 sign(P(beta+index_2, l_index))*...
88                 sign(P(beta+index_1+index_2, l_index))*...
89                 min(abs(P(beta+index_2, l_index)), ...
90                     abs(P(beta+index_1+index_2, l_index)));
91             end
92         end
93     end
94     if frozen_bits(phi + 1) == 0
95         %if now we decode an unfrozen bit
```

```
97 PM_pair = realmax * ones(2, L);
98 for l_index = 1 : L
99     if activepath(l_index) == 0
100        continue;
101    end
102    if P(1, l_index) >= 0
103        PM_pair(1, l_index) = PM(l_index);
104        PM_pair(2, l_index) = PM(l_index) + P(1, l_index);
105    else
106        PM_pair(1, l_index) = PM(l_index) - P(1, l_index);
107        PM_pair(2, l_index) = PM(l_index);
108    end
109 end
110 middle = min(2 * sum(activepath), L);
111 PM_sort = sort(PM_pair(:));
112 PM_cv = PM_sort(middle);
113 compare = PM_pair <= PM_cv;
114 kill_index = zeros(L, 1);
115 %record the index of the path that is killed
116 kill_cnt = 0;
117 %the total number of killed path
118 %the above two variables consist of a stack
119 for i = 1 : L
120     if (compare(1, i) == 0)&&(compare(2, i) == 0)
121         %indicates that this path should be killed
122         activepath(i) = 0;
123         kill_cnt = kill_cnt + 1;%push stack
124         kill_index(kill_cnt) = i;
125     end
126 end
127 for l_index = 1 : L
128     if activepath(l_index) == 0
129        continue;
130    end
131    path_state=compare(1, l_index)*2+compare(2, l_index);
```

```
132     switch path_state
133     %path_state can equal to 0,
134     %but in this case we do no operation.
135     case 1
136         u(cnt_u , 1_index) = 1;
137         C(1 , 2 * 1_index - 1 + phi_mod_2) = 1;
138         PM(1_index) = PM_pair(2 , 1_index);
139     case 2
140         u(cnt_u , 1_index) = 0;
141         C(1 , 2 * 1_index - 1 + phi_mod_2) = 0;
142         PM(1_index) = PM_pair(1 , 1_index);
143     case 3
144         index = kill_index(kill_cnt);
145         kill_cnt = kill_cnt - 1;%pop stack
146         activepath(index) = 1;
147         %lazy copy
148         lazy_copy(:, index) = lazy_copy(:, 1_index);
149         u(:, index) = u(:, 1_index);
150         u(cnt_u , 1_index) = 0;
151         u(cnt_u , index) = 1;
152         C(1 , 2 * 1_index - 1 + phi_mod_2) = 0;
153         C(1 , 2 * index - 1 + phi_mod_2) = 1;
154         PM(1_index) = PM_pair(1 , 1_index);
155         PM(index) = PM_pair(2 , 1_index);
156     end
157 end
158 cnt_u = cnt_u + 1;
159 else%frozen bit operation
160     for 1_index = 1 : L
161         if activepath(1_index) == 0
162             continue;
163         end
164         if P(1 , 1_index) < 0
165             PM(1_index) = PM(1_index) - P(1 , 1_index);
166         end
```

```
167     if phi_mod_2 == 0
168         C(1, 2 * l_index - 1) = 0;
169     else
170         C(1, 2 * l_index) = 0;
171     end
172 end
173
174 for l_index = 1 : L%partial-sum return
175 if activepath(l_index) == 0
176     continue;
177 end
178 if phi_mod_2 == 1
179     layer = bit_layer_vec(phi + 1);
180     for i_layer = 0 : layer - 1
181         index_1 = lambda_offset(i_layer + 1);
182         index_2 = lambda_offset(i_layer + 2);
183         for beta = index_1 : 2 * index_1 - 1
184             C(beta + index_1, 2 * l_index) = ...
185             mod(C(beta, 2*lazy_copy(i_layer+1, l_index)-1)+...
186                 C(beta, 2 * l_index), 2);
187             C(beta+index_2, 2*l_index)=C(beta, 2*l_index);
188         end
189     end
190     index_1 = lambda_offset(layer + 1);
191     index_2 = lambda_offset(layer + 2);
192     for beta = index_1 : 2 * index_1 - 1
193         C(beta + index_1, 2 * l_index - 1) = ...
194         mod(C(beta, 2*lazy_copy(layer+1, l_index)-1)+...
195             C(beta, 2*l_index), 2);
196         C(beta+index_2, 2*l_index-1)=C(beta, 2*l_index);
197     end
198 end
199
200 end
201 %lazy copy
```

```

202 if phi < N - 1
203     for i_layer = 1 : llr_layer_vec(phi + 2) + 1
204         for l_index = 1 : L
205             if activepath(l_index)
206                 lazy_copy(i_layer, l_index) = l_index;
207             end
208         end
209     end
210 end
211
%path selection.
213 [~, min_index] = min(PM);
214 polar_info_esti = u(:, min_index);
215 end

```

本节的最后分析码长为 N , 列表数为 L 的SCL译码器的存储量、运算量和时延。提示: $1 + 2 + 4 + \dots + 2^n = 2^{n+1} - 1$ 。

首先看看SCL译码器占用的硬件存储。

LLR耗费的存储分为两部分:

第一部分是信道LLR占用的存储, 大小为 NQ_{LLR} , 其中 N 是码长, Q_{LLR} 是LLR的量化位数, 即硬件中用 Q_{LLR} 个比特的截断来表示一个LLR实数。 Q_{LLR} 一般取6就足够了。

第二部分是中间计算结果LLR占用的存储, 大小为 $(N - 1)LQ_{\text{LLR}}$ 。

比特值耗费的存储分为两部分:

第一部分是每个比特 u_i 判决时所需的存储, 显然需要 L 比特的存储。

第二部分是中间比特值占用的存储, 大小为 $2(N - 1)L$, 有2表示C数组是两列的。这里忽略了我图5.9中的 $C^{[4]}$, 因为我们其实不需要码字估计 $\hat{\mathbf{x}}$, 只要顺序记录每个信息比特 $u_i, i \in \mathcal{A}$ 的估计就行了。我在图5.9中画出 $C^{[4]}$ 其实是为了C数组递归计算的完整。

路径度量也占用存储: LQ_{PM} , 其中 Q_{PM} 是每个度量值的量化位数。

综上, 一个标准的SCL译码器需要的存储是:

$$M_{\text{SCL}} = [N + (N - 1)L]Q_{\text{LLR}} + LQ_{\text{PM}} + (2N - 1)L. \quad (5.11)$$

可见, L 和量化位数 Q 确定时, 存储量是码长 N 的线性函数。

然后, 看看SCL译码器的运算复杂度。在这里运算复杂度指的是在一次完整的SCL译码

过程中， f 和 g 函数总计执行的次数 $T_{f,g}$ ¹，因为 f 和 g 函数译码计算的基本单元。假设执行一次 f 或 g 函数需要 q 焦耳的能量，那一次SCL译码所需的能量的下界为 $T_{f,g}q$ 焦耳。

$T_{f,g}$ 由 L, N 和 \mathcal{A} 共同决定。虽然 L, N 和 \mathcal{A} 确定时， $T_{f,g}$ 是定值，但一般而言无法一眼看出 $T_{f,g}$ 的取值，只能在程序里用一个计数变量辛苦地统计。不过 $T_{f,g}$ 有一个简单的上界：

$$T_{f,g} < LN \log_2 N. \quad (5.12)$$

上式之所以成立是因为右边的值是 L 个SC译码器总是在同时且从头到尾地工作所产生的复杂度，而无论 \mathcal{A} 怎样，译码 u_1 时，实际上SCL译码器中总是仅有一个SC译码器在运作，而不是 L 个SC译码器都在运作。下面给出几个常见码长和码率下的 $T_{f,g}$ 的取值。

	$R = 1/3$	$R = 1/2$	$R = 2/3$
$N = 256$	19204	24152	25090
$N = 512$	46476	55112	59462
$N = 1024$	121624	125828	137184
$N = 2048$	268864	284940	304288

表 5.1 $T_{f,g}$ 在常见码长和码率下的取值。所有极化码用高斯近似在 $E_b/N_0 = 2.5\text{dB}$ 处构造。

既然式(5.12)成立，那我们可以断定SCL译码器的计算复杂度是 $O(N \log_2 N)$ 。但是数据的拷贝也是占用时间的，如果不用Lazy Copy，在克隆时总是把母译码器的数据全部拷贝给新译码器，则我们可以简单的计算数据拷贝次数的上界：

$$K \times [(N - 1) + 2(N - 1)] \times \frac{L}{2}. \quad (5.13)$$

上式的意义是：不用Lazy Copy，译码每个信息比特 $u_i, i \in \mathcal{A}$ 时，最坏情况下，所有路径总是处于克隆的状态，因此每个 $u_i, i \in \mathcal{A}$ 引入 $L/2$ 次复制，每次复制 $N - 1$ 个中间LLR值和 $2(N - 1)$ 个中间比特值。当码率为 R ， $K = NR$ ，上式显然是 N^2 的复杂度，这个复杂度在实时通信里是不能接受的。

那么使用了Lazy Copy后，从上述SCL运行过程，我们发现如今不再复制任何LLR和比特数据，而仅复制Lazy Copy中的列。那么译码每个信息比特 $u_i, i \in \mathcal{A}$ 时，最坏情况下，所有路径总是处于克隆的状态，因此每个 $u_i, i \in \mathcal{A}$ 引入 $L/2$ 次复制，每次复制Lazy Copy的某一列，则一次复制的元素个数为 n 。由此，最坏情况下，一次SCL译码数据复制的次数为 $K \times n \times \frac{L}{2}$ ，渐近数量为 $O(LN \log N)$ 。

¹这里我们忽略中间比特值的计算量，因为中间比特值的计算要么是直接复制，要么仅有异或，我们认为这些操作非常容易完成。

最后，看看SCL译码器的时延。SCL译码器的时延显然和硬件实现的架构有关，这里我们使用如下的架构[21]。

(a). 列表数量为 L 的SCL译码器内部放置 L 个SC译码器，每个SC译码器完全一样，每个SC译码器都有 P 个处理器，即每个SC译码器在一个时钟内固定完成 P 次 f 或 g 函数的计算，且 L 个SC译码器能够并行运作；

(b). 信息比特路径度量排序在需要一个时钟。

(c). 中间比特值计算不引入时延。

(a)说明 L 个SC译码器在计算 f 或 g 函数时的操作是同时的，完全一致的，不同之处仅在于运算的数据不同。这样，虽然是 L 个SC译码器在运算，但是时延和一个SC译码器相等。

这就好像一个人吃一个面包用 t 秒吃完，现有一百个人，每个人都吃一个面包，同时吃，也用 t 秒，大家都吃完。一个人吃面包就是SC译码器，一百个人一起吃的就是SCL译码器。

一个人吃面包，发现是豆沙馅的，但是这个人不喜欢豆沙，没办法，他也得吃完，这对应SC译码器只能保存一条译码路径的事实。一百个人一起吃，甲发现自己的面包是豆沙的，但甲喜欢吃肉松的；丁发现自己的面包是肉松的，但丁喜欢豆沙，这样丁和甲交换了面包，大家都高兴了。这就对应SCL译码器中路径的选择。

满足(a)(b)(c)的SCL译码器的时延是：

$$L_{\text{SCL}}(N, P, A) = [2N + \frac{N}{P} \log_2 \frac{N}{4P}] + |\mathcal{A}|. \quad (5.14)$$

$2N + \frac{N}{P} \log_2 \frac{N}{4P}$ 这一项在上一章说过，是一个SC译码器的时延，因为条件(a)，所以SCL中由SC译码器引入的时延也是这个值。

$|\mathcal{A}|$ 这一项表示：冻结比特处没有克隆，也就不需要路径度量的排序，所以仅在信息比特处排序路径度量。信息比特有 $|\mathcal{A}|$ 个，因为条件(b)，所以加上 $|\mathcal{A}|$ 。

虽然上式说明SCL译码器的时延和 L 无关，但是显然 L 越大，SCL译码器功耗越大，所需存储越大，所需运算器越多，路径排序也越复杂。华为在其公开的论文中推荐 L 取8。

§ 5.3 快速SCL译码

本节的主要参考文献是[23][24][25]，其中[23]是一个早期成果，总结了各种节点下路径度量的计算方法；[24]也是一个早期成果，提出了R1节点的快速译码算法；[25]是一个集成的结果，将R0、R1、SPC和Rep节点的快速译码综合在了一起。我认为以上文献的方法是经典的。

讲SC译码器的时候我们讲过快速SC译码器。快速SC译码的原理是在SC过程运行到某个特殊子码时候，不再使用SC过程继续译码，而是直接使用最大似然译码，完成该子码的码字比特

的估计。快速SC译码器处理的特殊子码分别是R0,Rep,SPC和R1子码。同样地，在SCL译码过程中，也能够利用这四种子码进行快速译码，所得译码器称为快速SCL译码器。本节介绍基于上述四种子码的快速SCL译码器。注意到在一些比较新的研究中¹，特殊子码的种类远不止这四种，有兴趣的读者可以参考[26][27]。

在开始下面的长篇大论之前，本段先简述快速SCL译码器的原理。极化码的二叉树译码形式中，考虑SCL译码器中的某个SC译码器。在该SC译码器内，一个长为 N_v 的子码的根节点收到LLR序列 $\text{llr}_1^{N_v}$ ，则该SC译码器试图找到 $\text{llr}_1^{N_v}$ 的最大似然译码结果和一些“次最大似然译码结果”²。每个SC译码都执行上述操作。这样，SCL译码中的每一个SC译码器在该子码处都得到了若干个码字估计，这些估计的数量加起来，可能超过 $2L$ ，但是没关系，我们最终只能选择 L 个最好的结果保留。所以快速SCL译码器有两个核心问题，一是在如何在子码码字的层面上某条路径是好是坏，二是如何找到“次最大似然译码结果”。第一个问题早已被解决，将在下面的小节中阐述；第二个问题永远也不会很好地解决，因为每一种子码都有它的特殊结构，只能说你可以很好地解决你已经找到的子码，但是还有很多你没找到的子码。这些子码长度大，结构复杂，无法用一个统一的、闭合的、自动的方式解决其译码的问题。这就和村官处理日常纠纷一样，天天鸡飞狗跳，没有统一办法。

5.3.1 路径度量的另一种计算方式

我们已经知道SCL译码器路径度量的计算公式：

$$-\ln \Pr(\mathbf{u}_1^i | \mathbf{y}_1^N) = \sum_{k=1}^i \ln(1 + e^{-(1-2u_k)L_N^{(k)}}).$$

这个公式依赖的是判决信息比特用的LLR，也就是 $L_N^{(k)}$ 。下面的引理告诉我们，即使没有各个 $L_N^{(k)}$ 的值，我们仍能计算某个子码引入的路径度量的增量。

引理 5.1 设在GF(2)上有 $x_1 = u_1 \oplus u_2$, $x_2 = u_2$ ，在实数域上有 $\alpha_1 = \ln \frac{1+e^{\beta_1+\beta_2}}{e^{\beta_1}+e^{\beta_2}}$, $\alpha_2 = (1-2u_1)\beta_1 + \beta_2$ ，则对于任意 $(u_1, u_2) \in \{0, 1\}^2$ 和任意实数 β_1, β_2 有下式成立：

$$\ln(1 + e^{-(1-2u_1)\alpha_1}) + \ln(1 + e^{-(1-2u_2)\alpha_2}) = \ln(1 + e^{-(1-2x_1)\beta_1}) + \ln(1 + e^{-(1-2x_2)\beta_2}). \quad (5.15)$$

证明 穷举 (u_1, u_2) 的四种取值，一一验证即可。

¹我写这句话的时间是2019年3月8日10点57分

²注意这些译码结果仍是码字估计，不是信息比特估计

这个引理的意义是，对于长度为2的极化码，它的信道接收对数似然比为 (β_1, β_2) ，SC译码器计算出的判决 u_1, u_2 使用的LLR分别是 α_1, α_2 ，我们既可以用式(5.15)等号左边的形式计算路径度量，这是我们已经熟悉的方式；同时我们也可以用式(5.15)等号右边的形式计算路径度量，即计算码字层面的 x_1, x_2 对应的度量，在这种情况下，对应的信息比特就是 $(u_1, u_2) = (x_1, x_2)\mathbf{F}$ 。注意到如果使用式(5.15)等号右边的形式计算路径度量，首先要选择合法的码字 (x_1, x_2) 。例如，假设 $u_1 = 0$ 是冻结比特， u_2 是信息比特，则 (x_1, x_2) 就不能选 $(1, 0)$ 和 $(0, 1)$ 。

上面的结果可以轻易地进行如下拓展。

定理5.2 设 (l_1, \dots, l_N) 是任意实数序列， $n = \log_2 N$ ，把 (l_1, \dots, l_N) 当作信道接收LLR序列输入给SC译码器，SC译码器计算得到判决各个信息比特的LLR序列 $(L_N^{(1)}, \dots, L_N^{(1)})$ ，并得到信源序列的判决 $(\hat{u}_1, \dots, \hat{u}_N)$ 。根据定理5.1， $(\hat{u}_1, \dots, \hat{u}_N)$ 对应的路径度量是 $\sum_{i=1}^N \ln(1 + e^{-(1-2\hat{u}_i)L_N^{(i)}})$ ，则 $\sum_{i=1}^N \ln(1 + e^{-(1-2\hat{u}_i)L_N^{(i)}}) = \sum_{i=1}^N \ln(1 + e^{-(1-2\hat{x}_i)l_i})$ ，其中 $(\hat{x}_1, \dots, \hat{x}_N) = (\hat{u}_1, \dots, \hat{u}_N)\mathbf{F}^{\otimes n}$ ，这保证了 $(\hat{x}_1, \dots, \hat{x}_N)$ 必是码字。

证明 使用归纳法。

源头：当 $n = 1$ ，由引理5.1可知定理结论成立。

归纳假设：当 $n = k$ 时，有 $\sum_{i=1}^{2^k} \ln(1 + e^{-(1-2\hat{u}_i)L_N^{(i)}}) = \sum_{i=1}^{2^k} \ln(1 + e^{-(1-2\hat{x}_i)l_i})$ ，其中 $(\hat{x}_1, \dots, \hat{x}_{2^k}) = (\hat{u}_1, \dots, \hat{u}_{2^k})\mathbf{F}^{\otimes k}$ 。

下一步：当 $n = k + 1$ 时，如图5.10所示，流程如下所述。

(i). $(l_1, \dots, l_{2^{k+1}})$ 是信道接收LLR，最右侧一列的 2^k 个 2×2 模块用 f 函数算出 $(\alpha_1, \dots, \alpha_{2^k})$ 。

(ii). 处理左上角长度为 2^k 的极化码：把 $(\alpha_1, \dots, \alpha_{2^k})$ 当作接收LLR序列，输入长度为 2^k 的SC译码器，得到左上角长度为 2^k 的极化码的信源估计 $\hat{u}_1^{2^k}$ 。由归纳假设得图5.10上面红框中的式子。

(iii). 处理左下角长度为 2^k 的极化码：把 $(\alpha_{2^k+1}, \dots, \alpha_{2^{k+1}}) = ((1 - 2\hat{x}_1)l_1 + l_{2^k+1}, \dots, (1 - 2\hat{x}_{2^k})l_{2^k} + l_{2^{k+1}})$ 当作接收LLR序列，输入长度为 2^k 的SC译码器，得到左下角长度为 2^k 的极化码的信源估计 $\hat{u}_{2^k+1}^{2^{k+1}}$ 。由归纳假设得图5.10下面红框中的式子。

(iv). 最右侧一列 2^k 个 2×2 模块从上往下编号为 $1, 2, \dots, 2^k$ 。由引理5.1，对于第 i 个 2×2 模块：

$$\ln(1 + e^{-(1-2\hat{x}_i)\alpha_i}) + \ln(1 + e^{-(1-2\hat{x}_{i+2^k})\alpha_{i+2^k}}) = \ln(1 + e^{-(1-2(\hat{x}_i \oplus \hat{x}_{i+2^k}))l_i}) + \ln(1 + e^{-(1-2\hat{x}_{i+2^k})l_{i+2^k}}). \quad (5.16)$$

从而 $\sum_{i=1}^{2^k} \ln(1 + e^{-(1-2\hat{x}_i)\alpha_i}) + \sum_{i=2^k+1}^{2^{k+1}} \ln(1 + e^{-(1-2\hat{x}_i)\alpha_i}) = \sum_{i=1}^{2^{k+1}} \ln(1 + e^{-(1-2\hat{\mu}_i)l_i})$ ，其中 $\hat{\mu}_1^{2^{k+1}} = [\hat{x}_1^{2^k} \oplus \hat{x}_{2^k+1}^{2^{k+1}} \ \hat{x}_{2^k+1}^{2^{k+1}}] = (\hat{u}_1^{2^k}, \hat{u}_{2^k+1}^{2^{k+1}})\mathbf{F}^{\otimes k+1}$ 是长度为 2^{k+1} 的极化码的码字估计。

证毕。

$$\sum_{i=1}^{2^k} \ln(1 + e^{-(1-2\hat{u}_i)L_N^{(i)}}) = \sum_{i=1}^{2^k} \ln(1 + e^{-(1-2\hat{x}_i)\alpha_i})$$

$\mathbf{F}^{\otimes k}$

\hat{x}_1	α_1	\oplus	$\hat{\mu}_1$	l_1
\hat{x}_2	α_2	\oplus	$\hat{\mu}_2$	l_2
\hat{x}_3	α_3	\oplus	$\hat{\mu}_3$	l_3
\vdots	\ddots	\vdots	\vdots	\vdots
\hat{x}_{2^k}	α_{2^k}	\oplus	$\hat{\mu}_{2^k}$	l_{2^k}

$\mathbf{F}^{\otimes k}$

\hat{x}_{2^k+1}	$\alpha_{2^k+1} = (1-2\hat{x}_1)l_1 + l_{2^k+1}$	$\hat{\mu}_{2^k+1}$	l_{2^k+1}
\hat{x}_{2^k+2}	$\alpha_{2^k+2} = (1-2\hat{x}_2)l_2 + l_{2^k+2}$	$\hat{\mu}_{2^k+2}$	l_{2^k+2}
\hat{x}_{2^k+3}	$\alpha_{2^k+3} = (1-2\hat{x}_3)l_3 + l_{2^k+3}$	$\hat{\mu}_{2^k+3}$	l_{2^k+3}
\vdots	\ddots	\ddots	\ddots
$\hat{x}_{2^{k+1}}$	$\alpha_{2^{k+1}} = (1-2\hat{x}_{2^k})l_{2^k} + l_{2^{k+1}}$	$\hat{\mu}_{2^{k+1}}$	$l_{2^{k+1}}$

$$\sum_{i=2^k+1}^{2^{k+1}} \ln(1 + e^{-(1-2\hat{u}_i)L_N^{(i)}}) = \sum_{i=2^k+1}^{2^{k+1}} \ln(1 + e^{-(1-2\hat{x}_i)\alpha_i})$$

图 5.10 定理5.2归纳法的过程说明

5.3.2 路径度量的无损简化计算公式

下面你将看到一个恒等式。

$$\ln(1 + e^a) - \ln(1 + e^{-a}) = a. \quad (5.17)$$

用 $-(1-2x)\alpha$ 替换上面的 a , 得到下面熟悉的样式 (其中 α 的含义是一个对数似然比, $x = (1 - \text{sign}(\alpha))/2$ 相当于按 α 的符号判决的比特值, 并注意到 $-(1-2(x \oplus 1))\alpha = (1-2x)\alpha$)

$$\ln(1 + e^{-(1-2(x \oplus 1))\alpha}) = \ln(1 + e^{-(1-2x)\alpha}) + (1-2x)\alpha. \quad (5.18)$$

式(5.18)物理意义很明确: 等号左侧表示按照对数似然比 α 的反方向判决 (x 已经取反) 所得的路径度量, 等号右侧的第一项表示按照对数似然比 α 判决所得的路径度量, 因为 $x = (1 - \text{sign}(\alpha))/2$, 所以等号右侧的第二项 $(1-2x)\alpha = |\alpha|$ 恒成立, 表示一个惩罚项。式(5.18)告诉我们: 对于一个码字比特 x , 如果不按照其对应的对数似然比 α 判决, 那么路径度量就会加上 $|\alpha|$ 作为惩罚。

下面我们会大量执行“加上对数似然比的绝对值 $|\alpha|$ ”这种操作。我写这一节是为了告诉

读者，这是精确的计算，而不是“硬件友好计算”。尽管在标准SCL译码器中，“硬件友好计算”在惩罚路径时也是加上对数似然比的绝对值，但不要和本章的内容混淆。

5.3.3 R0节点的快速SCL译码方法

R0节点中全是冻结比特（假定取值为0），不存在任何路径克隆。长度为 N_v 的R0节点只会返回长度为 N_v 的全0比特值。

对于列表数量为 L 的SCL译码器，任取一个激活的SC译码器 $SC_l, 1 \leq l \leq L$ ，当 SC_l 内的SC译码过程走到一个R0节点时（至于如何判断走到了一个R0节点，见上一章中的快速SC译码器部分），计算R0节点对应的LLR序列，再把路径度量加上惩罚值，如图5.11所示。其中 PM_l 是 SC_l 在译码R0前已经具有的度量值。

别忘了全零序列要填入C数组的对应位置。一般来讲，R0节点位于其母节点的左侧，这是由极化规律决定的。所以，R0在二叉树上对应的那个节点是不存在攀升的，“攀升”的定义在本讲义的第一章中。我们把长度为 N_v 的R0节点提供的全零比特序列填入 $C_{:,2l-1}^{[1+\log_2 N_v]}$ 这一段存储即可。

对于每个激活的SC译码器都执行上面的操作，我们就完成了R0节点的快速SCL译码。

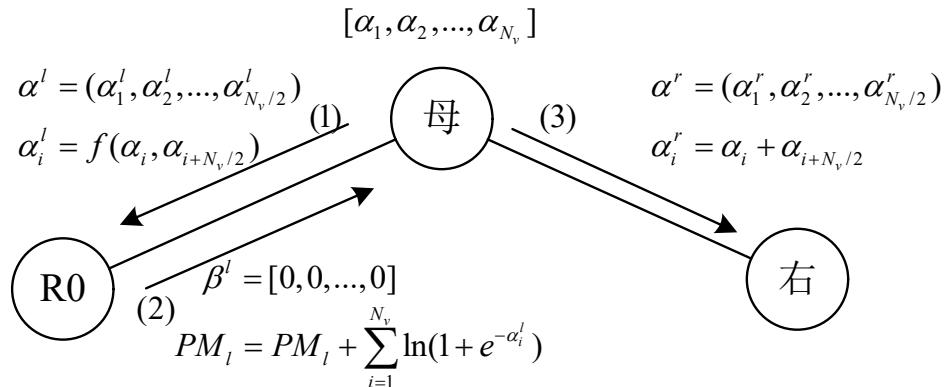


图 5.11 R0节点的快速SCL译码

5.3.4 Rep节点的快速译码

Rep节点的快速译码R0稍微难一点，因为译码Rep节点存在路径克隆。

对于列表数量为 L 的SCL译码器，任取一个激活的SC译码器 $SC_l, 1 \leq l \leq L$ ，当 SC_l 内的SC译码过程走到一个Rep节点时（至于如何判断走到了一个Rep节点，见上一章中的快速SC译

码器部分），计算Rep节点对应的LLR序列，如图5.12所示。然后我们知道，长度为 N_v 的Rep节点有两个合法码字，分别是长度为 N_v 的全0序列和长度为 N_v 的全1序列。这样， SC_l 想要同时保留这两个序列，这两个序列的度量值不同：长度为 N_v 的全0序列对应的路径度量增量是 $\Delta_{l,0} = \sum_{i=1}^{N_v} \ln(1 + e^{-\alpha_i^l})$ ，其中 $\Delta_{l,0}$ 中的 l 指的是第 l 个SC译码器 SC_l ；长度为 N_v 的全1序列对应的路径度量增量是 $\Delta_{l,1} = \sum_{i=1}^{N_v} \ln(1 + e^{\alpha_i^l})$ 。设 PM_l 是 SC_l 在译码该Rep节点前已经具有的度量值，那么 SC_l 的度量值两种可能，分别是 $PM_l + \Delta_{l,0}$ 和 $PM_l + \Delta_{l,1}$ 。

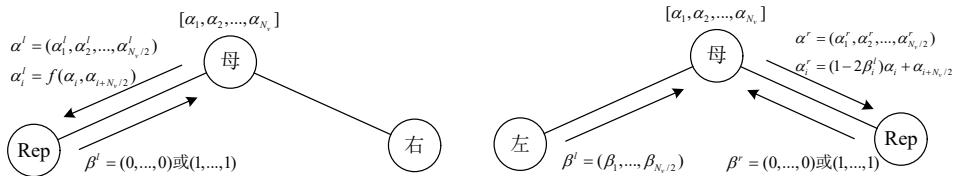


图 5.12 Rep 节点的SCL快速译码示意图。Rep节点不一定是其母节点的左孩子或右孩子，所以有两种情况。

每一个激活的SC译码器都执行如上的操作，只不过是数据的值不同。假设目前共有 S , $S \leq L$ 个激活的SC译码器，那么这 S 个译码器所有可能的度量值可以用下面的 $2 \times S$ 维矩阵表示，这个矩阵的第 s , $1 \leq s \leq S$ 列第1行表示 SC_{i_s} 保存全0序列时的度量值，第 s , $1 \leq s \leq S$ 列第2行表示 SC_{i_s} 保存全1序列时的度量值。之所以写出这样奇怪的下标是因为“到底哪些译码器是激活的”，是没有闭合表达式的，每次译码时都不同。

$$\mathbf{PM} \begin{bmatrix} PM_{i_1} + \Delta_{i_1,0} & \dots & PM_{i_s} + \Delta_{i_s,0} & \dots & PM_{i_S} + \Delta_{i_S,0} \\ PM_{i_1} + \Delta_{i_1,1} & \dots & PM_{i_s} + \Delta_{i_s,1} & \dots & PM_{i_S} + \Delta_{i_S,1} \end{bmatrix} \quad (5.19)$$

这时找到其中最小的 $\min\{L, 2S\}$ 个值，保存其对应的序列即可。此时，每个译码器 SC_{i_s} 的状态都可能不同。

- (a). 如果译码器 SC_{i_s} 对应的两个度量值均为入选最小的 $\min\{L, 2S\}$ 个值，那么它死亡了，收尸，将 i_s 压入堆栈 S_{sleep} 。
- (b). 如果译码器 SC_{i_s} 对应的两个度量中仅有有一个入选最小的 $\min\{L, 2S\}$ 个值，那么 SC_{i_s} 就保留这个入选的度量值对应的序列，记这个序列是“全 b 序列”， $b \in \{0, 1\}$ 。这时， SC_{i_s} 的度量值更新为 $PM_{i_s} = PM_{i_s} + \Delta_{i_s, b}$ 。

然后需要把全 b 序列放入C数组的对应位置。这个放置办法稍微复杂一点。设此时的Rep子码对应的最后一个叶节点是信源比特 u_i^1 ，把 $i - 1$ 做 $n = \log_2 N$ 位二进制展开得到 $< i_n, i_{n-1}, \dots, i_1 >$ ，记 $< i_n, i_{n-1}, \dots, i_1 >$ 低位连续的1的个数为 w 。然后去掉右侧低位连续

¹这个 u_i 就是Rep节点中唯一的信息比特。别慌， u_i 的索引号 i 可以通过我上一章中的节点搜索程序确定。

的 $\log_2 N_v$ 个1，得到一个二进制形式 $< i_n, i_{n-1}, \dots, i_{1+\log_2 N_v} >$ 。这时，看看 $i_{1+\log_2 N_v}$ 是零还是1。

如果 $i_{1+\log_2 N_v} = 0$ ，那么把全b序列放入 $C_{:,2i_s-1}^{[1+\log_2 N_v]}$ ，结束。对应的是图5.12左侧的情况。

如果 $i_{1+\log_2 N_v} = 1$ ，对应的是图5.12右侧的情况。把全b序列放入 $C_{:,2i_s}^{[1+\log_2 N_v]}$ ，然后用下式继续C数组的更新：

前 $w - \log_2 N_v - 1$ 次更新（赋值总是给C的右侧列）：

$$\begin{aligned} C_{1:2^{k+\log_2 N_v}-2,2i_s}^{[k+\log_2 N_v]} &= C_{:,2LC_{k-1+\log_2 N_v,i_s}-1}^{[k-1+\log_2 N_v]} \oplus C_{:,2i_s}^{[k-1+\log_2 N_v]}, \\ C_{2^{k+\log_2 N_v}-2+1:2^{k+\log_2 N_v}-1,2i_s}^{[k+\log_2 N_v]} &= C_{:,2i_s}^{[k-1+\log_2 N_v]}, \end{aligned} \quad (5.20)$$

$$2 \leq k \leq w - \log_2 N_v.$$

最后一次更新（赋值总是给C的左侧列）：

$$\begin{aligned} C_{1:2^{w-1},2i_s-1}^{[w+1]} &= C_{:,2LC_{w,i_s}-1}^{[w]} \oplus C_{:,2i_s}^{[w]}, \\ C_{2^{w-1}+1:2^w,2i_s-1}^{[w+1]} &= C_{:,2i_s}^{[w]}, \end{aligned} \quad (5.21)$$

(c). 如果译码器 SC_{i_s} 对应的两个度量全部入选最小的 $\min\{L, 2S\}$ 个值，那么从 S_{sleep} 弹出元素 j ，把Lazy Copy数组的第 i_s 列赋值给第 j 列，表示 SC_j 是由 SC_{i_s} 克隆来的。

原路径 SC_{i_s} 总是保留全0序列， SC_{i_s} 的度量值更新为 $PM_{i_s} = PM_{i_s} + \Delta i_s, 0$ ， SC_{i_s} 对应的C数组执行上面标号(b)的段落中的操作。

新路径 SC_j 总是保留全1序列， SC_j 的度量值更新为 $PM_j = PM_{i_s} + \Delta i_s, 1$ ， SC_j 对应的C数组执行上面标号(b)的段落中的操作。

5.3.5 R1节点的快速译码

首先我们断定R1节点总是位于它的母节点的右侧，如图5.13所示。这由极化信道的偏序关系决定的，具体原因留给读者作为练习。

长度为 N_v 的R1节点有 2^{N_v} 中可能的码字，所以穷举是不现实的。文献[24]提出了下面的策略。

在激活的SC译码器中任取一个 SC_l ，该译码器在译码目标R1节点前已经具有路径度量 PM_l 。 SC_l 译码器已经算出了目标R1节点需要的LLR序列 $\alpha_1, \alpha_2, \dots, \alpha_{N_v}$ （为了简明，扔掉了图5.13中的上角标 r ）。直接对 $\alpha_1, \alpha_2, \dots, \alpha_{N_v}$ 做硬判决得到比特序列 $\beta_1, \beta_2, \dots, \beta_{N_v}$ 。由上一章中的引理4.4可知， $\beta_1, \beta_2, \dots, \beta_{N_v}$ 是 $\alpha_1, \alpha_2, \dots, \alpha_{N_v}$ 的最大似然译码结果，且 $\beta_1, \beta_2, \dots, \beta_{N_v}$ 对应的路径没有任何惩罚，度量度量仍维持 PM_l 。但这还不够，我们希望得到 $\alpha_1, \alpha_2, \dots, \alpha_{N_v}$ 的一

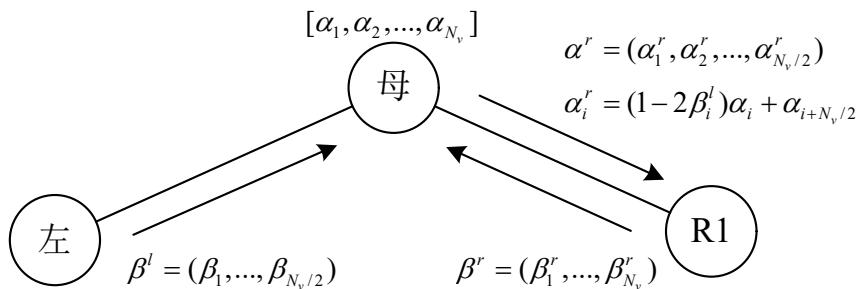


图 5.13 R1 节点的快速译码

些“次”最大似然译码结果，这些此最大似然译码结果的通过翻转 $\beta_1, \beta_2, \dots, \beta_{N_v}$ 中的若干个比特得到，每翻转一个比特 β_i 的值，这个次最大似然结果对应的路径度量加上惩罚值 $|\alpha_i|$ 。

我们给 $|\alpha_1|, |\alpha_2|, \dots, |\alpha_{N_v}|$ 排序，得到升序列 $|\alpha_{i_1}|, |\alpha_{i_2}|, \dots, |\alpha_{i_{N_v}}|, |\alpha_{i_j}| < |\alpha_{i_{j+1}}|$ 。如果我们仅考虑 SC_l 中的译码，暂时忽略其余 SC 译码器，然后我们马上注意到下面的事实。

译码结果 $\beta_1, \beta_2, \dots, \beta_{N_v}$ 的度量值仍维持 PM_l ， PM_l 就是该R1节点所有可能的译码结果中的最小度量值。除了 PM_l 之外，最小可能度量值就是 $PM_l + |\alpha_{i_1}|$ ，该值对应的比特序列为 $\beta_1, \dots, \beta_{i_1} \oplus 1, \dots, \beta_{N_v}$ 。因为我们的路径数量是 L ，所以找出前 L 个最小的度量值及其对应的序列就算我们赢了。

我们注意到 $PM_l + |\alpha_{i_L}|$ 绝对不会入选最小的 L 个度量值，这是因为 $PM_l, PM_l + |\alpha_{i_1}|, \dots, PM_l + |\alpha_{i_{L-1}}|$ 中的每一个都小于 $PM_l + |\alpha_{i_L}|$ 。所以最小的 L 个值就是 PM_l 以及 $PM_l + |\alpha_{i_1}|, \dots, PM_l + |\alpha_{i_{L-1}}|$ 的和，即我们不再考虑 $|\alpha_{i_L}|, \dots, |\alpha_{N_v}|$ 。图5.14中的二叉树结构可以轻松找出前 L 个最小的度量值，图5.14的绘制规律如图5.15所示。

注意到图5.14的分支数量是指数增加的，当 L 稍微较大时硬件是装不下的。我们的目标是找到图5.14的绿框中取值最小的 L 个表达式，及其对应的比特序列。因为 SCL 译码器的列表数是 L ，图5.14的分支数量就不能超过 L ，当分支数量到达 $2L$ 时，即到达 $|\alpha_{1+\log_2 L}|$ 指示的层时，我们要进行修剪。

修建的规则很简单，直接扔掉 $|\alpha_{1+\log_2 L}|$ 指示的层中具有 L 个最大值度量值的节点。这样的操作不会妨碍我们寻找图5.14的绿框中取值最小的 L 个表达式，因为图5.14的绿框中取值最小的 L 个表达式不可能是被修建掉的节点的叶节点。

原因在于：记上一段中被修建掉的 L 个度量值为 q_1, \dots, q_L ，保留的 L 个度量值为 p_1, \dots, p_L 。如果我们不修剪，并来到了下一层（ $|\alpha_{2+\log_2 L}|$ 指示的层），容易发现 q_1, \dots, q_L 对应的左孩子的度量值保持 q_1, \dots, q_L 不变，而 q_1, \dots, q_L 右孩子的度量值分别是 $q_1 + |\alpha_{2+\log_2 L}|, \dots, q_L + |\alpha_{2+\log_2 L}|$ ，无论是 q_1, \dots, q_L 还是 $q_1 + |\alpha_{2+\log_2 L}|, \dots, q_L + |\alpha_{2+\log_2 L}|$ ，它们中的任意一个值都大于 p_1, \dots, p_L 中的任意一个值，因此无论是 q_1, \dots, q_L 还是 $q_1 + |\alpha_{2+\log_2 L}|, \dots, q_L + |\alpha_{2+\log_2 L}|$ ，肯定不会入选前 L 个最小的值。

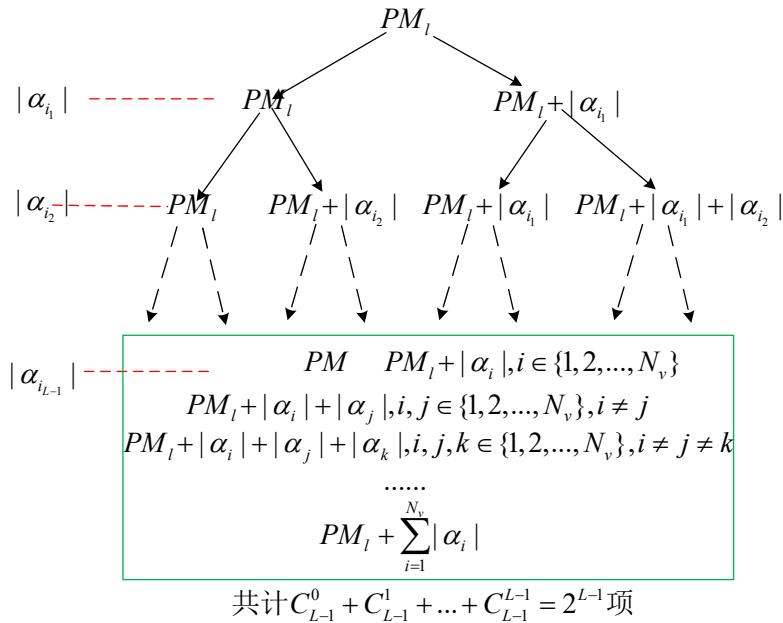


图 5.14 R1 节点的二叉树分裂。容易验证所有叶节点度量值的表达式如绿框内容所示。注意，这些表达式互不相同，不代表给定 $|\alpha_{i_1}|, \dots, |\alpha_{i_{L-1}}|$ 时，这些表达式的数值一定互不相同。

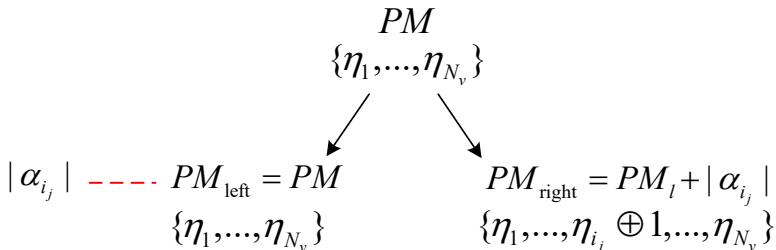


图 5.15 R1 节点的二叉树分裂规律。其中 PM 和 $(\eta_1, \dots, \eta_{N_v})$ 是图 5.14 中的一个非叶节点，意为比特序列 $(\eta_1, \dots, \eta_{N_v})$ 对应的度量值是 PM ， $|\alpha_{i_j}|$ 表示该层是根节点之下的第 j 层。左孩子总是不加改变地继承母节点的一切，右孩子的变化取决于 $|\alpha_{i_j}|$ ：右孩子总是把母节点的第 i_j 个比特翻转，然后对应加上惩罚值 $|\alpha_{i_j}|$ 。

综上，当图 5.14 中的分支数量达到 $2L$ 时，我们扔掉具有 L 个最大值度量值的节点。其后每层的分支数量都将是 $2L$ ，我们继续扔掉具有 L 个最大值度量值的节点，直到完成 $|\alpha_{i_{L-1}}|$ 指示的层中的操作。我们就找到了图 5.14 的绿框中取值最小的 L 个表达式（对应的比特序列只不过是在 $\beta_1, \beta_2, \dots, \beta_{N_v}$ 的基础上，翻转了对应表达式中的 α 下标位置上的比特值）。这样，在仅考虑一个激活的 SC 译码器的情况下，该 SC 译码器能够产生 L 个“较好的码字估计”，即最大似然译码结果及其在若干比特位置上的翻转。

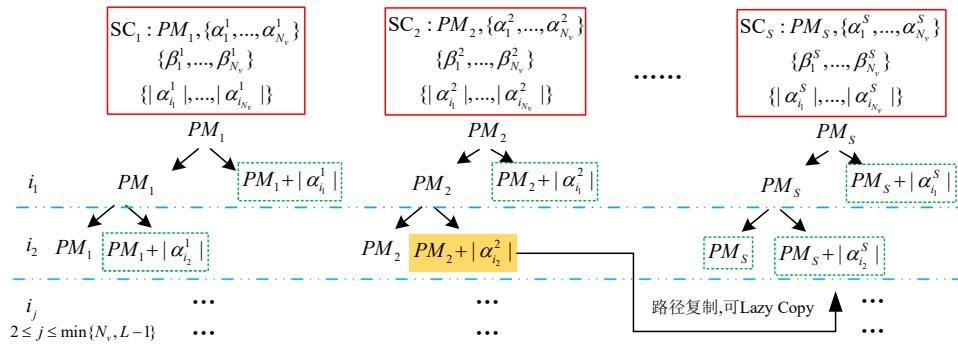


图 5.16 当遇到目标R1节点时SCL中已经存在S个激活的SC译码器的情况

当译码目标R1节点时已经存在 S , $2 \leq S \leq L$ 个激活的SC译码器时, 所需操作如图5.16所示。

图5.16中每一个SC译码器都收到了属于自己的LLR序列, 第 s 个SC译码器收到的LLR序列记为 $\{\alpha_1^s, \dots, \alpha_{N_v}^s\}$;

每一个译码器都对自己收到的LLR进行硬判决, 第 s 个SC译码器的判决序列记为 $\{\beta_1^s, \dots, \beta_{N_v}^s\}$;

每一个SC译码器都对属于自己的LLR序列进行绝对值排序, 第 s 个SC译码器LLR绝对值序列的升序排序记为 $\{|\alpha_{i_1}^s|, \dots, |\alpha_{i_{N_v}}^s|\}$;

图5.16中第一次分裂对应的层是由每个SC译码器收到的LLR序列中具有最小绝对值的LLR指示的, 例如第一次分裂是由最小绝对值索引 i_1 决定的。

第一次分裂中的每个左孩子完全继承母节点的数据, 而右孩子则翻转具有最小绝对值的LLR对应的比特值, 并加上相应的度量惩罚。这时共存在 $2S$ 个结果。我们假设 $2S > L$, 需要修剪, 仅保留具有最小的 L 个度量值的分支。绿色虚线框中的结果是我们扔掉的结果;

第二次分裂对应的层是由每个SC译码器收到的LLR序列中具有第二小绝对值的LLR指示的, 根据第一次分裂中的假设, 这时分支数必有 $2L$ 个, 仅保留具有最小的 L 个度量值的分支。绿色虚线框中的结果是我们扔掉的结果。此时, 注意到本例子中的第 S 个译码器的所有分支都被淘汰, 而第2个SC译码器有两个结果存留, 此时默认把左孩子保留在原路径中, 右孩子Lazy Copy给第 S 个译码器。这里的Lazy Copy比较简单, 只要告诉第 S 个译码器去读取第2个译码器的数据就行了: Lazy Copy数组初始化为 $[1, 2, \dots, S, \dots, L-1, L]$, 路径复制后变成 $[1, 2, \dots, 2, \dots, L-1, L]$ 。

剩余的分裂以此类推, 直到分支数超过 L , 这时保留其中 L 个具有最小度量值的结果, 扔掉其余结果, 直到完成具有第 $L-1$ 小绝对值的LLR指示的层中的操作。

我们注意到如果 $L-1 > N_v$, 上述操作进行到 $|\alpha_{N_v}|$ 指示的层就结束了, 因为没有更多的数据了。综上, 总计分裂的次数是 $\min\{N_v, L-1\}$ 。

5.3.6 SPC节点的快速译码算法

SPC节点的快速译码过程与R1节点类似。我们下面先考虑一条译码路径在Fast SCL译码器中的分裂方式，然后考虑 L 条译码路径的译码方式，因为当你遇到一个SPC节点时，SCL译码器中很可能已经存在 L 条译码路径了。

第 l 条路径中的SPC节点的快速译码， $1 \leq l \leq L$ ：

当长度为 N_v 的SPC节点收到它对应的LLR序列 $\alpha_1^{N_v}$ 时，对 $\alpha_1^{N_v}$ 做门限判决，得到SPC码字判决序列 $\hat{x}_1^{N_v}$ ，并且对 $\alpha_1^{N_v}$ 的绝对值序列 $|\alpha_1|^{N_v}$ 做升序排序，得到升序正实数序列 $\{\alpha_{i_1}, \alpha_{i_2}, \dots, \alpha_{i_{N_v}}\}$ 满足 $|\alpha_{i_j}| < |\alpha_{i_{j+1}}|$ 。

(a). 如果 $\bigoplus_{i=1}^{N_v} \hat{x}_i = 0$ ，意味着硬判决结果就是最大似然译码结果。这时，如果要进行比特翻转，根据SPC的限定条件，每次需要翻转偶数个比特，而引起最小变化的操作显然是只每次翻转两个比特。这样我们断定，度量惩罚项中带有 $|\alpha_{L+1}|$ 的路径绝对不会入选具有最小度量值的 L 个路径，因为带有 $|\alpha_{L+1}|$ 这一个惩罚项的度量值最小也是 $PM_l + |\alpha_1| + |\alpha_{L+1}|$ ，而 $PM_l, PM_l + |\alpha_{i_1}| + |\alpha_{i_2}|, PM_l + |\alpha_{i_1}| + |\alpha_{i_3}|, \dots, PM_l + |\alpha_{i_1}| + |\alpha_{i_L}|$ 这 L 项都小于 $PM_l + |\alpha_1| + |\alpha_{L+1}|$ 。因此，参与比较的LLR截止到 $|\alpha_L|$ ，SPC节点码字比特分裂如图5.17所示。

图5.17中，每层所加的惩罚总是由两个LLR指示，其中一个固定为 $|\alpha_{i_1}|$ ，另一个是 $|\alpha_{i_j}|, 2 \leq j \leq L$ 。在 $|\alpha_{i_1}| + |\alpha_{i_j}|$ 指示的层中，左孩子总是继承母节点的所有数据，而右孩子总是翻转比特 \hat{x}_{i_1} 和 \hat{x}_{i_j} 。例如图5.17中由 $(|\alpha_{i_1}|, |\alpha_{i_2}|)$ 指示的层中，左侧节点保持母节点的度量值 PM_l 和码字序列 $(\hat{x}_1, \dots, \hat{x}_{N_v})$ ¹，而右侧节点翻转了码字比特 \hat{x}_{i_1} 和 \hat{x}_{i_2} ，度量增添惩罚项 $|\alpha_{i_1}| + |\alpha_{i_2}|$ 。图5.17其余层的操作规律以此类推。我们也注意到 $|\alpha_{i_1}|$ 这一项将被反复地累加，当 $|\alpha_{i_1}|$ 累加偶数次时，没有惩罚；当 $|\alpha_{i_1}|$ 累加奇数次时，具有惩罚项 $|\alpha_{i_1}|$ 。这是因为累加偶数次 $|\alpha_{i_1}|$ 等价于 \hat{x}_{i_1} 维持原值不变。

我们很容易就能判断出图5.17中每一层的度量值的表达式²穷举了所有可能的情况： $|\alpha_{i_1}| + |\alpha_{i_j}|$ 指示的层共有 2^{j-1} 个节点，这些节点的度量值表达式互不相同，而 $C_j^0 + C_j^2 + \dots + C_j^{2\lfloor \frac{j}{2} \rfloor} = 2^{j-1}$ 代表了使用LLR序列 $\{|\alpha_{i_1}|, \dots, |\alpha_{i_j}|\}$ 时，所有可能的比特翻转情况。因此图5.17中的过程就是一个穷举的过程。

图5.17的分支数量指数增长。与R1节点的处理类似，当分支数量察达到 $2L$ 个时，对应的是 $|\alpha_{i_1}| + |\alpha_{i_{2+\log_2 L}}|$ 指示的层，此时需要扔掉度量值最大的 L 个分支，保留度量值最小的 L 个分支，且这样的修剪不会影响我们获得前 L 个“准最大似然码字”：就算本应被扔掉的分支存活到下一层，本应被扔掉的分支的后代的度量值也一定大于存活路径的度量。

(b). 如果 $\bigoplus_{i=1}^{N_v} \hat{x}_i = 1$ ，意味着硬判决结果非法，此时翻转 $|\alpha_{i_1}|$ 对应的比特就得到了最大似然译码结果。此时的SPC码字分裂如图5.18所示，图5.18和图5.17的规律完全一致，唯一的不同

¹ PM_l 是第 l 个SC译码器译码当前SPC节点前，已经具有的度量值

² 注意是表达式本身，而非将某次具体的实现值带入表达式计算出的数值

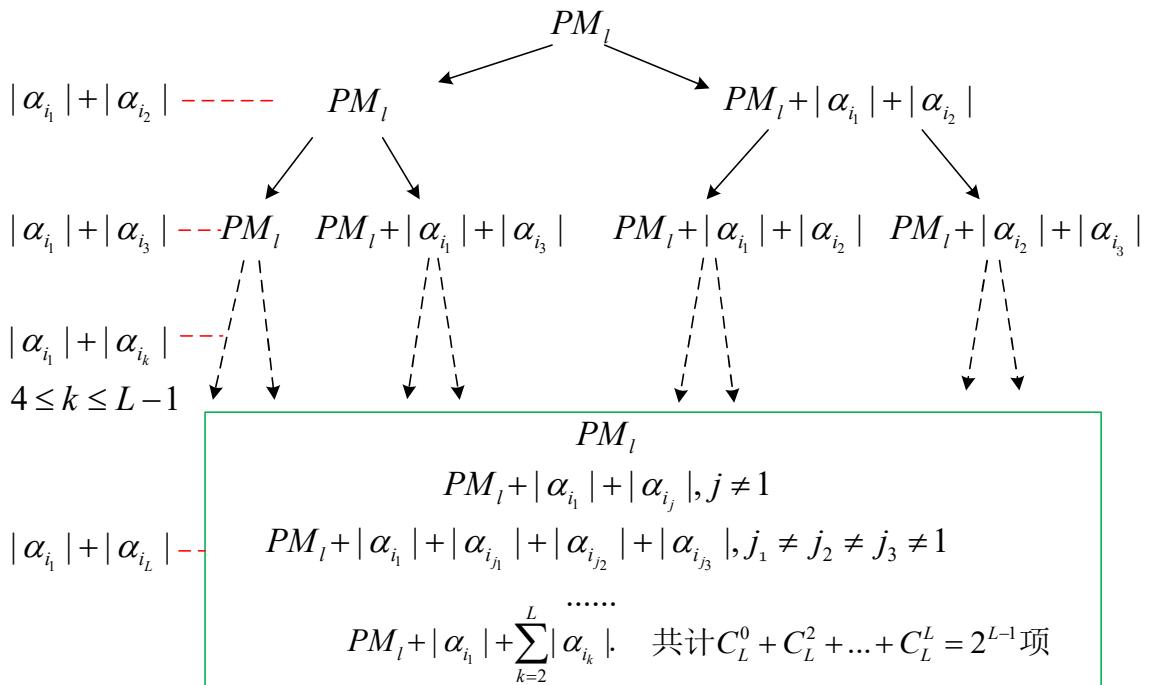


图 5.17 $\oplus_{i=1}^{N_v} \hat{x}_i = 0$ 时 SPC 节点码字比特分裂。总是假设 L 是 2 的幂，从而 L 也是偶数。这个假设也是实际中常见的，无论是学界或是工业界。

就是 5.18 中的母节点的数据变成了：度量值是 $PM_l + |\alpha_{i_1}|$ ，码字序列是 $(\hat{x}_1, \dots, \hat{x}_{i_1} \oplus 1, \dots, \hat{x}_{N_v})$ 。

注意当 $|\alpha_{i_1}|$ 累加偶数次时，没有惩罚；当 $|\alpha_{i_1}|$ 累加奇数次时，具有惩罚项 $|\alpha_{i_1}|$ 。

在大多数情况下，遇到一个 SPC 节点时，SCL 译码器中可能已经存在 S 个激活 SC 译码器，这些 SC 译码器一次标号为 $1, 2, \dots, S$ 。如图 5.19 所示。

每个 SC 译码器有自己的 LLR 序列 $\{\alpha_1^s, \dots, \alpha_{N_v}^s\}, 1 \leq s \leq S$ ；

每个 SC 译码器对自己的 LLR 序列绝对值排序得到 $\{|\alpha_{i_1}^s|, \dots, |\alpha_{i_{N_v}}^s|\}, 1 \leq s \leq S$ ；

每个 SC 译码器对自己的 LLR 序列做硬判决得到 $\{\beta_1^s, \dots, \beta_{N_v}^s\}, 1 \leq s \leq S$ ，这些判决结果或者落入上面(a) 段中的情况，或者落入上面(b) 段中的情况。

第一次分裂由索引 i_1, i_2 指示，每个 SC 译码器依照 (a) 段或 (b) 段中的情况，做出自己的分裂。如果分支数量超过了 L ，就仅保存具有最小度量值的 L 个分支，扔掉其余分支（绿色虚线框所示）。如果某个 SC 译码器的所有分支都被扔掉了，那么该译码暂时死亡，准备接收其它译码器复制来的数据。与 R1 节点类似，复制可用 Lazy Copy，即告诉克隆路径从哪里取出数据。例如，图 5.19 中，Lazy Copy 数组初始化为 $[1, 2, \dots, S, \dots, L]$ ，第 S 个 SC 译码器在 $|\alpha_{i_1}| + |\alpha_{i_3}|$ 指示的层中死亡，并接受了第 2 号 SC 译码器的数据，则 Lazy Copy 数组变成 $[1, 2, \dots, 2, \dots, L]$ 。

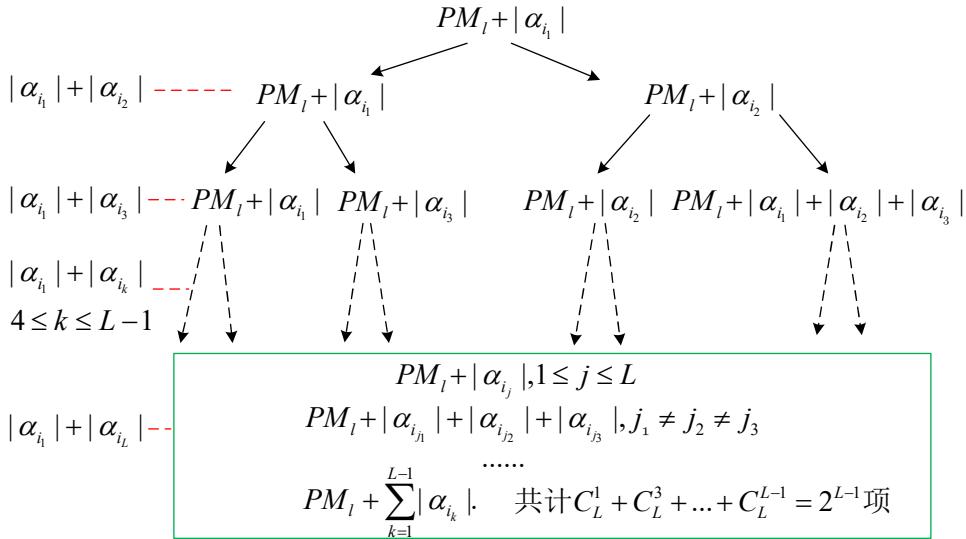
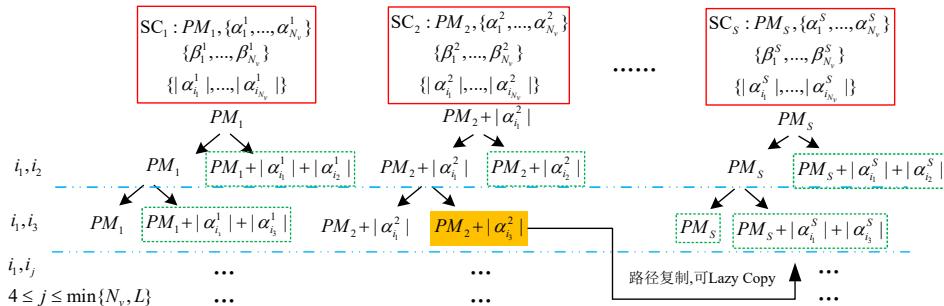
图 5.18 $\oplus_{i=1}^{N_v} \hat{x}_i = 1$ 时，SPC节点的码字比特分裂

图 5.19 遇到一个SPC节点时，已经多个激活的SC译码器的情况

根据极化信道的偏序关系，我们可以断定SPC节点总是其母节点右孩子。因此在更新SPC节点对应的C数组时，总是将码字比特的估计填入C数组的右侧一列，即落入Rep节点情况(b)的第四段的情况。

下面是Fast SCL译码器的MATLAB代码，输入参数和Fast SC译码器基本一致， L 是列表数量。这些代码的完整版我都列在封面上了，伸手党可自取。

```

1 function polar_info_esti = FastSCL_decoder ...
2 (1lr ,L, info_bits ,lambda_offset ,...

```

```
3 llr_layer_vec, bit_layer_vec, psi_vec, node_type_matrix
4 %LLR-based SCL deocoder
5 %Systematic polar code used
6 %4 nodes are identified and decoded, i.e., rate0, rate1, rep, spc,
7 %No BLER loss, while using fast algorithms.
8 %const
9 N = length(llr);
10 m = log2(N);
11 %memory declared
12 P = zeros(N - 1, L); %llr is public-used, so N - 1 is enough.
13 C = zeros(2 * N - 1, 2 * L);
14 PM = zeros(L, 1);
15 activepath = zeros(L, 1);
16 lazy_copy = zeros(m, L);
17 %initialize the 1st SC decoder
18 activepath(1) = 1;
19 lazy_copy(:, 1) = 1;
20 %decoding starts
21 %default: in the case of path clone in REP and leaf node,
22 %origianl always corresponds to bit 0s, while the new path bit 1s.
23 for i_node = 1 : size(node_type_matrix, 1)
24     current_index = node_type_matrix(i_node, 1);
25     llr_layer = llr_layer_vec(current_index);
26     M = node_type_matrix(i_node, 2);
27     %number of leaf nodes in this constituent node
28     reduced_layer = log2(M);
29     %For LLR recursion, this reduced layer denotes where to stop;
30     %For Bit recursion, this denotes where to start.
31     psi = psi_vec(i_node);
32     %This psi is used for bits recursion
33     psi_mod_2 = mod(psi, 2);
34     %To decide whether the bit recusion should continue.
35     %1 : continue; 0 : stop.
36     for l_index = 1 : L
37         if activepath(l_index) == 0
```

```
38         continue;
39
40     end
41
42     switch current_index
43
44     case 1
45         index_1 = lambda_offset(m);
46         for beta = 0 : index_1 - 1
47             P(beta + index_1, l_index) = ...
48             sign(11r(beta + 1)) *...
49             sign(11r(beta + index_1 + 1)) *...
50             min(abs(11r(beta + 1)), ...
51                 abs(11r(beta + index_1 + 1)));
52
53         end
54
55         for i_layer = m - 2 : -1 : reduced_layer
56             index_1 = lambda_offset(i_layer + 1);
57             index_2 = lambda_offset(i_layer + 2);
58             for beta = index_1 : index_2 - 1
59                 P(beta, l_index) = ...
60                 sign(P(beta + index_1, l_index)) *..
61                 sign(P(beta + index_2, l_index)) * ...
62                 min(abs(P(beta + index_1, l_index)), ...
63                     abs(P(beta + index_2, l_index)));
64
65         end
66
67     end
68
69     case N/2 + 1
70         index_1 = lambda_offset(m);
71         for beta = 0 : index_1 - 1
72             x_tmp = C(beta + index_1, 2 * l_index - 1);
73             P(beta + index_1, l_index) =..
74             (1 - 2 * x_tmp) * 11r(beta + 1) +...
75             11r(beta + index_1 + 1);
76
77         end
78
79         for i_layer = m - 2 : -1 : reduced_layer
80             index_1 = lambda_offset(i_layer + 1);
81             index_2 = lambda_offset(i_layer + 2);
82             for beta = index_1 : index_2 - 1
```

```
73         P(beta, l_index) = ...
74         sign(P(beta + index_1, l_index)) * ...
75         sign(P(beta + index_2, l_index)) * ...
76         min(abs(P(beta + index_1, l_index)), ...
77             abs(P(beta + index_2, l_index)));
78     end
79
80     otherwise
81         index_1 = lambda_offset(lr_layer + 1);
82         index_2 = lambda_offset(lr_layer + 2);
83         for beta = index_1 : index_2 - 1
84             %lazy copy
85             P(beta, l_index) = ...
86             (1 - 2 * C(beta, 2 * l_index - 1)) * ...
87             P(beta+index_1, lazy_copy(lr_layer+2,l_index))...
88             + P(beta+index_2, lazy_copy(lr_layer+2,l_index));
89         end
90         for i_layer = lr_layer - 1 : -1 : reduced_layer
91             index_1 = lambda_offset(i_layer + 1);
92             index_2 = lambda_offset(i_layer + 2);
93             for beta = index_1 : index_2 - 1
94                 P(beta, l_index) = ...
95                 sign(P(beta + index_1, l_index)) * ...
96                 sign(P(beta + index_2, l_index)) * ...
97                 min(abs(P(beta + index_1, l_index)), ...
98                     abs(P(beta + index_2, l_index)));
99             end
100        end
101    end
102 end%LLR calculations. You may fold thiis code block.
103 index_vec = M : 2 * M - 1;
104 switch node_type_matrix(i_node, 3)
105     case -1%****RATE 0*****
106         x = zeros(M, 1);
107         for l_index = 1 : L
```

```
108     if activepath(l_index) == 0
109         continue;
110     end
111     C(index_vec, psi_mod_2 + 2 * l_index - 1) = x;
112     PM(l_index) = PM(l_index) + ...
113         sum(log(1 + exp(-P(index_vec, l_index))));  
%Do not forget updating path metric in rate 0 nodes
114     %No copy in Rate 0 node
115
116 end
117 case 1%*****RATE 1*****
118 %input LLR: P(index_vec, :)
119 [PM, activepath, selected_subcodeword, lazy_copy]...
120 = Rate1(M,P(index_vec,:),activepath,PM,L,lazy_copy);
121 %output: subpolar codeword: 'selected_subcodeword'.
122 for l_index = 1 : L
123     if activepath(l_index) == 0
124         continue
125     end
126     C(index_vec, 2 * l_index - 1 + psi_mod_2) = ...
127         selected_subcodeword(:, l_index);
128 end
129 case 2%*****REP*****
130 [PM, activepath, lazy_copy, selected_subcodeword] = ...
131 Rep(PM, P(index_vec, :), activepath, L, M, lazy_copy);
132 for l_index = 1 : L
133     if activepath(l_index) == 0
134         continue
135     end
136     C(index_vec, 2 * l_index - 1 + psi_mod_2) = ...
137         selected_subcodeword(:, l_index);
138 end
139 case 3%*****SPC*****
140 [PM, activepath, selected_subcodeword, lazy_copy] = ...
141 SPC(index_vec,P(index_vec,:),activepath,PM,L,lazy_copy);
142 for l_index = 1 : L
```

```
143         if activepath(l_index) == 0
144             continue
145         end
146         C(index_vec, 2 * l_index - 1 + psi_mod_2) = ...
147         selected_subcodeword(:, l_index);
148     end
149 end
150 %Internal Bits Update for each active path
151 bit_layer = bit_layer_vec(current_index + M - 1);
152 for l_index = 1 : L
153     if activepath(l_index) == 0
154         continue;
155     end
156     if psi_mod_2 == 1%right child of its mother
157         for i_layer = reduced_layer : bit_layer - 1
158             index_1 = lambda_offset(i_layer + 1);
159             index_2 = lambda_offset(i_layer + 2);
160             for beta = index_1 : index_2 - 1
161                 C(beta + index_1, 2 * l_index) = ...
162                 mod(C(beta, 2*lazy_copy(i_layer+1,l_index)-1)+...
163                     C(beta, 2 * l_index), 2);
164                 C(beta + index_2, 2*l_index)=C(beta,2*l_index);
165             end
166         end
167         index_1 = lambda_offset(bit_layer + 1);
168         index_2 = lambda_offset(bit_layer + 2);
169         for beta = index_1 : index_2 - 1
170             C(beta + index_1, 2 * l_index - 1) = ...
171             mod(C(beta, 2*lazy_copy(bit_layer+1,l_index)-1)+...
172                 C(beta, 2*l_index), 2);
173             C(beta + index_2, 2 * l_index - 1) C(beta,2*l_index);
174         end
175     end
176 end
177 %lazy copy
```

```

178     if i_node < size(node_type_matrix, 1)
179         for i_layer = 1 : llr_layer_vec(current_index + M) + 1
180             for l_index = 1 : L
181                 lazy_copy(i_layer, l_index) = l_index;
182             end
183         end
184     end
185
186 %select the best path
187 [~, min_index] = min(PM);
188 x = C(end - N + 1 : end, 2 * min_index(1) - 1);
189 polar_info_esti = x(info_bits);%Systematic code
190 end

```

```

1 function [PM, activepath, lazy_copy, candidate_codeword]...
2 = Rep(PM, LLR_array, activepath, L, M, lazy_copy)
3 num_surviving_path = min(L, 2 * sum(activepath));
4 PM_pair = realmax * ones(2, L);
5 %In rep node, one exsiting path generates two candidates
6 candidate_codeword = zeros(M, L);
7 for l_index = 1 : L
8     if activepath(l_index) == 0
9         continue;
10    end
11    Delta0 = 0;
12    Delta1 = 0;
13    for i_llr = 1 : M
14        if LLR_array(i_llr, l_index) < 0
15            Delta0 = Delta0 - LLR_array(i_llr, l_index);
16        else
17            Delta1 = Delta1 + LLR_array(i_llr, l_index);
18        end
19    end
20    PM_pair(1, l_index) = PM(l_index) + Delta0;

```

```
21 PM_pair(2, 1_index) = PM(1_index) + Delta1;
22 end
23 PM_sort = sort(PM_pair(:));
24 PM_cv = PM_sort(num_surviving_path);
25 compare = zeros(2, L);
26 cnt = 0;
27 for j = 1 : L
28     for i = 1 : 2
29         if cnt == num_surviving_path
30             break;
31         end
32         if PM_pair(i, j) <= PM_cv
33             compare(i, j) = 1;
34             cnt = cnt + 1;
35         end
36     end
37 end
38 kill_index = zeros(L, 1);
39 kill_cnt = 0;
40 for i = 1 : L
41     if (compare(1, i) == 0)&&(compare(2, i) == 0)
42         %indicates that this path should be killed
43         activepath(i) = 0;
44         kill_cnt = kill_cnt + 1;%push stack
45         kill_index(kill_cnt) = i;
46     end
47 end
48
49 for l_index = 1 : L
50     if sum(compare(:, l_index)) == 0
51         continue;
52     end
53     path_state = compare(1, l_index) * 2 + compare(2, l_index);
54     switch path_state
55         case 1
```

```

56         candidate_codeword(:, 1_index) = 1;
57         PM(1_index) = PM_pair(2, 1_index);
58     case 2
59         PM(1_index) = PM_pair(1, 1_index);
60         %initialized values are 0s, so no update here.
61     case 3
62         new_index = kill_index(kill_cnt);
63         kill_cnt = kill_cnt - 1;
64         activepath(new_index) = 1;
65         lazy_copy(:, new_index) = lazy_copy(:, 1_index);
66         candidate_codeword(:, new_index) = 1;
67         PM(1_index) = PM_pair(1, 1_index);
68         PM(new_index) = PM_pair(2, 1_index);
69     end
70 end

```

```

1 function [PM, activepath, candidate_codeword, lazy_copy]...
2 = Rate1(M, LLR_array, activepath, PM, L, lazy_copy)
3 candidate_codeword = zeros(M, L);
4 llr_ordered = zeros(M, L);
5 sub_lazy_copy = 1 : L;%lazy_copy used in this function.
6 for l_index = 1 : L
7     if activepath(l_index) == 0
8         continue;
9     end
10    %sort length-Nv LLR in its absolute value.
11    [~, llr_ordered(:, l_index)] =...
12        sort(abs(LLR_array(:, l_index)));
13    candidate_codeword(:, l_index) =...
14        LLR_array(:, l_index) < 0;
15    %ML codeword loaded
16 end
17 for i_split = 1 : min(M, L - 1)%number of splits in codeword sense
18     PM_rate_1 = realmax * ones(2, L);

```

```
19 compare_rate_1 = zeros(2, L);
20 for l_index = 1 : L
21     if activepath(l_index) == 0
22         continue;
23     end
24     PM_rate_1(1, l_index) = PM(l_index);
25 %the first row remains the same.
26     PM_rate_1(2, l_index) = PM(l_index) + ...
27     abs(LLR_array(1lr_ordered(i_split, sub_lazy_copy(l_index)), ...
28     sub_lazy_copy(l_index)));
29 %the 2nd row is penalized.
30 end
31 num_surviving_path = min(L, 2 * sum(activepath));
32 PM_sorted = sort(PM_rate_1 (:));
33 PM_cv = PM_sorted(num_surviving_path);
34 cnt = 0;
35 for j = 1 : L
36     for i = 1 : size(PM_rate_1, 1) %****CAUTION!!!!*****
37         if cnt == num_surviving_path
38             break;
39         end
40         if PM_rate_1(i, j) <= PM_cv
41             compare_rate_1(i, j) = 1;
42             cnt = cnt + 1;
43         end
44     end
45 end
46 kill_index = zeros(L, 1);
47 kill_cnt = 0;
48 for l_index = 1 : L
49     if sum(compare_rate_1(:, l_index)) == 0
50         activepath(l_index) = 0;
51         kill_cnt = kill_cnt + 1;%push stack
52         kill_index(kill_cnt) = l_index;
53     end
```

```

54 end
55 for l_index = 1 : L
56     if activepath(l_index) == 0
57         continue
58     end
59     if sum(compare_rate_1(:, l_index)) == 2
60         new_index = kill_index(kill_cnt);
61         kill_cnt = kill_cnt - 1;
62         activepath(new_index) = 1;
63         sub_lazy_copy(new_index) = sub_lazy_copy(l_index);
64         codeword_tmp = candidate_codeword(:, l_index);
65         codeword_tmp(lr_ordered(i_split, sub_lazy_copy(l_index)))...
66             = mod(codeword_tmp(lr_ordered(i_split, sub_lazy_copy(l_index)))+1
67         candidate_codeword(:, new_index) = codeword_tmp;
68         PM(new_index) = PM(l_index) + ...
69             abs(LLR_array(lr_ordered(i_split, sub_lazy_copy(l_index)), ...
70                 sub_lazy_copy(l_index)));
71 %PM updated
72     end
73 end
74
75 for l_index = 1 : L
76     if activepath(l_index) == 0
77         continue
78     end
79     lazy_copy(:, l_index) = lazy_copy(:, sub_lazy_copy(l_index));
80 end
81 end

```

```

1 function [PM, activepath, candidate_codeword, lazy_copy]...
2 = SPC(index_vec, LLR_array, activepath, PM, L, lazy_copy)
3 M = length(index_vec);
4 candidate_codeword = zeros(M, L);
5 lr_ordered = zeros(M, L);

```

```
6 parity_check_track = zeros(L, 1);
7 %caution, default value must be 0s.
8 sub_lazy_copy = 1 : L;
9 %Lazy copy used in this function
10 for l_index = 1 : L
11     if activepath(l_index) == 0
12         continue;
13     end
14     abs_LLRLR = abs(LLR_array(:, l_index));
15     [~, llr_ordered(:, l_index)] = sort(abs_LLRLR);
16     %Ascending sorting
17     candidate_codeword(:, l_index) = LLR_array(:, l_index) < 0;
18     %quasi-ML codeword
19     if mod(sum(candidate_codeword(:, l_index)), 2) == 1
20         %Do not satisfy parity check
21         candidate_codeword(llr_ordered(1, l_index), l_index) =...
22         mod(candidate_codeword(llr_ordered(1, l_index), l_index)+1, 2);
23         %Least reliable Bit flip
24         parity_check_track(l_index) = 1;%initialize parity check
25         PM(l_index) = PM(l_index) + abs_LLRLR(llr_ordered(1, l_index));
26         %PM changes accordingly.
27     end
28 end
29 for t = 2 : min(M, L)
30     PM_spc = realmax * ones(2, L);
31     compare_spc = zeros(2, L);
32     for l_index = 1 : L
33         if activepath(l_index) == 0
34             continue;
35         end
36         LLR = LLR_array(:, sub_lazy_copy(l_index));
37         PM_spc(1, l_index) = PM(l_index);
38         %remain the same as original path
39         PM_spc(2, l_index) = PM(l_index) + ...
40         abs(LLR(llr_ordered(t, sub_lazy_copy(l_index)))) + ...
```

```
41      (1 - 2 * parity_check_track(l_index)) * ...
42      abs(LLR(1lr_ordered(1, sub_lazy_copy(l_index)))); 
43  end
44 num_surviving_path = min(L, 2 * sum(activepath));
45 PM_sorted = sort(PM_spc(:));
46 PM_cv = PM_sorted(num_surviving_path);
47 cnt = 0;
48 for j = 1 : L
49     for i = 1 : size(PM_spc, 1)
50         if cnt == num_surviving_path
51             break;
52         end
53         if PM_spc(i, j) <= PM_cv
54             compare_spc(i, j) = 1;
55             cnt = cnt + 1;
56         end
57     end
58 end
59 kill_index = zeros(L, 1);
60 kill_cnt = 0;
61 for i = 1 : L
62     if all(compare_spc(:, i) == 0)
63         activepath(i) = 0;
64         kill_cnt = kill_cnt + 1;%push stack
65         kill_index(kill_cnt) = i;
66     end
67 end
68 for l_index = 1 : L
69     if activepath(l_index) == 0
70         continue
71     end
72     path_state = sum(compare_spc(:, l_index));
73     switch path_state
74         case 2
75             new_index = kill_index(kill_cnt);
```

```

76         kill_cnt = kill_cnt - 1;
77         activepath(new_index) = 1;
78         sub_lazy_copy(new_index) = sub_lazy_copy(l_index);
79         %generate a new sub-codeword
80         codeword_tmp = candidate_codeword(:, l_index);
81         codeword_tmp(lr_ordered(t, sub_lazy_copy(l_index))) = ...
82             mod(codeword_tmp(lr_ordered(t, sub_lazy_copy(l_index)))+1,2);
83         %t-th index (after lr sorting)
84         codeword_tmp(lr_ordered(1, sub_lazy_copy((l_index)))) = ...
85             mod(codeword_tmp(lr_ordered(1, sub_lazy_copy(l_index)))+1,2);
86         %l-st index always
87         candidate_codeword(:, new_index) = codeword_tmp;
88         parity_check_track(new_index) = ...
89             mod(parity_check_track(l_index) + 1, 2);
90         %When you split path, the parity check sign
91         %should be fliped accordingly.
92         PM(new_index) = PM_spc(2, l_index);%PM updated
93     end
94 end
95
96 for l_index = 1 : L
97     if activepath(l_index) == 0
98         continue
99     end
100    lazy_copy(:, l_index) = lazy_copy(:, sub_lazy_copy(l_index));
101 end
102 end

```

§ 5.4 SCL译码的其它改进算法

5.4.1 循环冗余校验协助下的SCL译码器

循环冗余校验协助下的SCL（Cyclic redundancy check Aided SCL，CA-SCL）译码器是当今最流行的SCL译码器形式[20]。CA-SCL译码器使用CRC-polar级联码，运作过程如图5.20

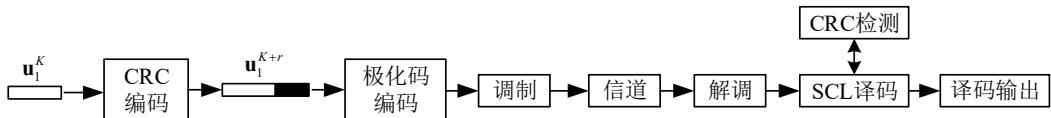


图 5.20 CA-SCL译码器的编译码过程

所示：数据比特为 (u_1, \dots, u_K) ，加长度为 r 的循环冗余校验后成为 $(u_1, \dots, u_K, v_1, \dots, v_r)$ ，对 $(u_1, \dots, u_K, v_1, \dots, v_r)$ 进行极化码编码，得到码字序列 (x_1, \dots, x_N) 。 (x_1, \dots, x_N) 对应的接收信号输入SCL译码器，SCL译码器给出 L 个信息比特估计，分别记为 $(\hat{u}_1, \dots, \hat{u}_K, \hat{v}_1, \dots, \hat{v}_r)_l, 1 \leq l \leq L$ ，对应路径度量分别为 $PM_l, 1 \leq l \leq L$ 。

如果 $(\hat{u}_1, \dots, \hat{u}_K, \hat{v}_1, \dots, \hat{v}_r)_l, 1 \leq l \leq L$ 中没有任何一个能通过CRC校验，或者说 $(\hat{u}_1, \dots, \hat{u}_K, \hat{v}_1, \dots, \hat{v}_r)_l, 1 \leq l \leq L$ 中没有一个是CRC码字，则宣布本次译码失败；

如果 $(\hat{u}_1, \dots, \hat{u}_K, \hat{v}_1, \dots, \hat{v}_r)_l, 1 \leq l \leq L$ 中仅有一个通过CRC校验，则选择该通过CRC校验的估计作为译码结果；

如果 $(\hat{u}_1, \dots, \hat{u}_K, \hat{v}_1, \dots, \hat{v}_r)_l, 1 \leq l \leq L$ 中有多个通过CRC校验，则在通过CRC校验的估计中，选择具有最小度量值的结果作为译码结果。

注意到CRC作为校验译码结果的手段，并不是唯一的，也可以使用偶校验码来校验SCL译码器的译码结果。对于使用偶校验码的SCL译码器，有兴趣的读者可以阅读参考文献[31][32]。

5.4.2 自适应CA-SCL译码器

自适应CA-SCL译码器运作过程如下[28]。

- (a). 使用CRC-polar级联码；
- (b). 先用SC译码器译码接收信号，如果SC译码器的译码结果通过CRC校验，则译码成功；如果SC译码器的结果未通过CRC校验，则选用 $L = 2$ 的CA-SCL译码器重新对同样的接收信号进行译码器；
- (c). 如果列表数为 L 的CA-SCL译码器存在译码结果通过CRC校验，则译码成功；否则，令 $L \leftarrow 2L$ ，重新用CA-SCL译码器对同样的接收信号译码。
- (d). 上面的过程一直持续，直到有存在译码结果通过CRC校验，或者 L 超过最大允许值。

5.4.3 剪枝CA-SCL译码器

剪枝CA-SCL译码器在译码过程中会关闭满足某些条件的且激活的SC译码器。

例如，路径度量值越小越好，在完成某个比特 u_i 的译码时，已经激活的 l 个SC译码器对应的度量值为 $\{PM_1, PM_2, \dots, PM_l\}$ ，记其中的最小值为 PM_{\min} ，大于 γPM_{\min} , $\gamma > 1$ 的度量值对应的SC译码器会被强行关闭，因为我们认为这样的SC译码器的正确率低，其中 γ 是人为选定的阈值。

有兴趣的读者可以阅读文献[29][30]。

第6章 简单MATLAB代码

本章主要给出我的代码的下载地址，帮助萌新入门。

§ 6.1 代码运行速度

代码在保证正确的基础上，运行速度就成了最重要的问题之一，有了运行速度快的代码，就能迅速验证思路，迅速得到结果。

我的代码都是全串行的，非常友好地不涉及多线程、多核运算，仅关注译码计算过程。那么这样的代码的运行速度如何？我想它能够满足大多数研究的要求，对于 $N = 2048$ 的码字，用普通的办公电脑，一天跑出 10^{-7} 的误码率（BLER）是没问题的。

我知道搞LPDC的人有时候需要跑出Error Floor， 10^{-7} 还不能满足需要。但是极化码在SC译码算法下不存在Error Floor，所以极化码一般不必仿真非常低的错误率。

下面的表是我在自己的普通台式机上的测试结果，时间单位是秒。BP译码器使用了[33]中的早期中止机制，最大迭代次数为50次，运行信噪比 $E_b/N_0 = 2.5\text{dB}$ 。

	$N = 256$	$N = 512$	$N = 1024$	$N = 2048$	$N = 4096$
编码速度	2.1×10^{-5}	4.1×10^{-5}	8.2×10^{-5}	1.8×10^{-4}	3.8×10^{-4}
SC译码	4.8×10^{-5}	1.1×10^{-4}	2.5×10^{-4}	5.2×10^{-4}	1.1×10^{-3}
CASCL译码器 $L = 8$	7.4×10^{-4}	1.5×10^{-3}	3.2×10^{-3}	6.7×10^{-3}	1.4×10^{-2}
CASCL译码器 $L = 32$	2.1×10^{-3}	4.6×10^{-3}	9.9×10^{-3}	2×10^{-2}	4.5×10^{-2}
BP译码器	3×10^{-4}	7×10^{-4}	2.2×10^{-3}	5×10^{-3}	1×10^{-2}

表 6.1 程序在普通PC上的运行速度，i5处理器，3.3GHz，8G内存。所有码字的码率都是 $R = 0.5$ ，由高斯近似在 $E_b/N_0 = 2.5\text{dB}$ 处构造

§ 6.2 传统CA-SCL译码器

下载地址: <https://github.com/YuYongRun/PolarCodeDecodersInMatlab/tree/master/PolarConventionalCASCL>

自带高斯近似码构造。

运行效果图:

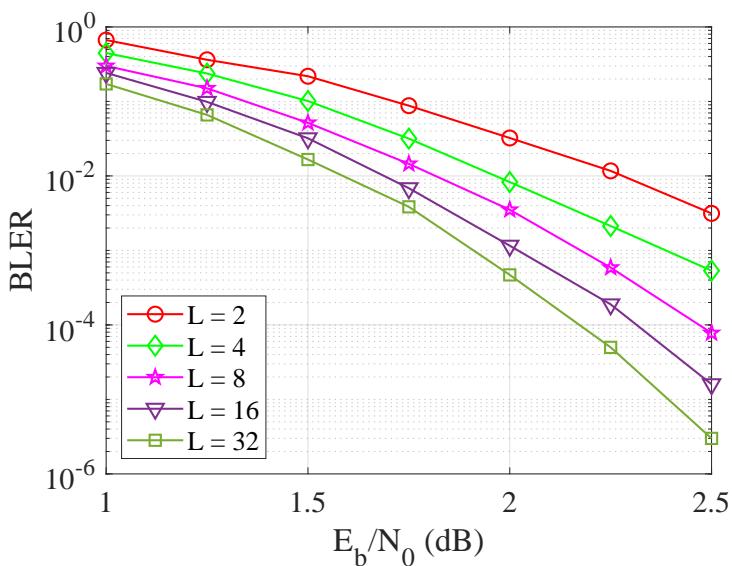


图 6.1 $N = 1024, R = 0.5$, CRC长度为16

§ 6.3 FastSCL译码器

下载地址: <https://github.com/YuYongRun/PolarCodeDecodersInMatlab/tree/master/PolarFastSCL>

该程序内有各种极化码构造算法和四种系统极化码的编码算法。

运行效果图:

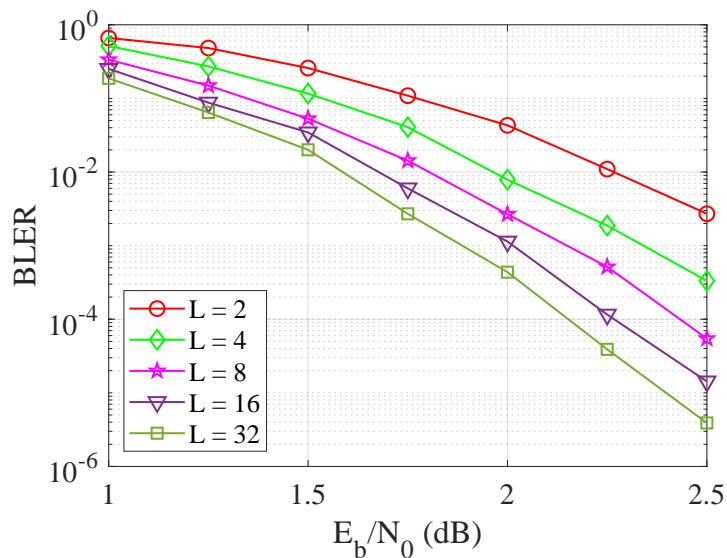


图 6.2 $N = 1024, R = 0.5$, CRC长度为16, 系统极化码

对比图6.1和图6.2可知, 快速译码算法没有任何BLER性能损失。

§ 6.4 BP译码器

下载地址: <https://github.com/YuYongRun/PolarCodeDecodersInMatlab/tree/master/PolarCodeBPdecoder>

运行效果图:

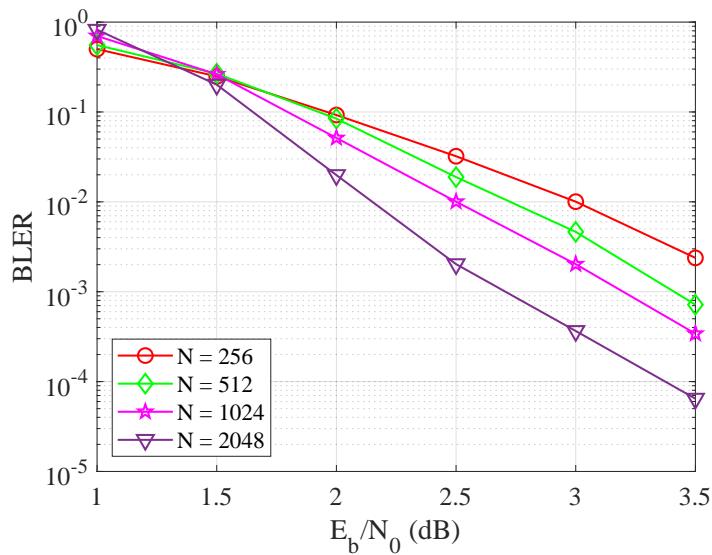


图 6.3 所有码字 $R = 0.5$, 最大迭代次数50, 早期中止机制来自文献[33]

参考文献

- [1] Lin S, Costello D J. 差错控制编码（第二版）[M]. 北京：机械工业出版社，2007. 120-123
- [2] Zhang Q, Liu A, Pan X, et al. CRC code design for list decoding of polar codes [J]. IEEE Communications Letters, 2017, 21(6): 1229-1232
- [3] Arikan E. Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels [J]. IEEE Transactions on Information Theory, 2009, 55(7): 3051-3073
- [4] Sarkis G, Tal I, Giard P, et al. Flexible and low-complexity encoding and decoding of systematic polar codes [J]. IEEE Transactions on Communications, 2016, 64(7): 2732-2745
- [5] Trifonov P. Efficient design and decoding of polar codes [J]. IEEE Transactions on Communications, 2012, 60(11): 3221-3227
- [6] Tal I, Vardy A. How to construct polar codes [J]. IEEE Transactions on Information Theory, 2013, 59(10): 6562-6582
- [7] He G, Belfiore J C, Land I, et al. Beta-expansion: A theoretical framework for fast and recursive construction of polar codes [C]. In: 2017 IEEE Global Communications Conference (GLOBECOM), Singapore, 2017. 1-6
- [8] Arikan E. Systematic polar coding [J]. IEEE Communications Letters, 2011, 15(8): 860-862
- [9] Li L, Zhang W. On the encoding complexity of systematic polar codes [C]. In: 2015 28th IEEE International System-on-Chip Conference (SOCC), Beijing, 2015. 415-420
- [10] Li L, Xu Z, Hu Y. Channel estimation with systematic polar codes [J]. IEEE Transactions on Vehicular Technology, 2018, 67(6): 4880-4889

- [11] Schürch C. A partial order for the synthesized channels of a polar code [C]. In: 2016 IEEE International Symposium on Information Theory (ISIT), Barcelona, 2016. 220-224
- [12] 卡林, 泰勒. 随机过程初级教程 (第2版) [M]. 北京: 人民邮电出版社, 2007.
- [13] Arikan E, Telatar E. On the rate of channel polarization [C]. In: 2009 IEEE International Symposium on Information Theory (ISIT), Seoul, 2009. 1493-1495
- [14] Alamdar-Yazdi A, Kschischang F R. A simplified successive-cancellation decoder for polar codes [J]. IEEE Communications Letters, 2011, 15(12): 1378-1380
- [15] . Increasing the throughput of polar decoders [J]. IEEE Communications Letters, 2013, 17(4): 725-728
- [16] Hanif M, Ardakani M. Fast successive-cancellation decoding of polar codes: identification and decoding of new nodes [J]. IEEE Communications Letters, 2017, 21(11): 2360-2363
- [17] Afisiadis O, Balatsoukas-Stimming A, Burg A. A low-complexity improved successive cancellation decoder for polar codes [C]. In: 2014 48th Asilomar Conference on Signals, Systems and Computers, Pacific Grove, 2014. 2116-2120
- [18] Zhang Z, Qin K, Zhang L, et al. Progressive bit-flipping decoding of polar codes over layered critical sets [C]. In: 2017 IEEE Global Communications Conference (GLOBECOM), Singapore, 2017. 1-6
- [19] Chandresris L, Savin V, Declercq D. Dynamic-SCFlip decoding of polar codes [J]. IEEE Transactions on Communications, 2018, 66(6): 2333-2345
- [20] Tal I, Vardy A. List decoding of polar codes [J]. IEEE Transactions on Information Theory, 2015, 61(5): 2213-2226
- [21] Balatsoukas-Stimming A, Parizi M B, Burg A. LLR-based successive cancellation list decoding of polar codes [J]. IEEE Transactions on Signal Processing, 2015, 63(19): 5165-5179
- [22] C. Leroux, A. J. Raymond, G. Sarkis and W. J. Gross, A semi-parallel successive-cancellation decoder for polar codes [J]. in IEEE Transactions on Signal Processing, vol. 61, no. 2, pp. 289-299, Jan.15, 2013.
- [23] Hashemi S A, Condo C, Gross W J. Simplified successive-cancellation list decoding of polar codes. In: the IEEE International Symposium on Information Theory (ISIT), Barcelona, 2016. 815-819

- [24] Hashemi S A, Condo C, Gross W J. Fast simplified successive-cancellation list decoding of polar codes [C]. In: 2017 IEEE Wireless Communications and Networking Conference Workshops (WCNCW), San Francisco, 2017. 1-6
- [25] Hashemi S A, Condo C, Gross W J. Fast and flexible successive-cancellation list decoders for polar codes [J]. IEEE Transactions on Signal Processing, 2017, 65(21): 5756-5769
- [26] M. Hanif, M. H. Ardakani and M. Ardakani, "Fast list decoding of polar codes: Decoders for additional nodes," 2018 IEEE Wireless Communications and Networking Conference Workshops (WCNCW), Barcelona, 2018, pp. 37-42.
- [27] H. Zhang et al., "A Flip-Syndrome-List Polar Decoder Architecture for Ultra-Low-Latency Communications," in IEEE Access, vol. 7, pp. 1149-1159, 2019.
- [28] Li B, Shen H, Tse D. An adaptive successive cancellation list decoder for polar codes with cyclic redundancy check [J]. IEEE Communications Letters, 2012, 16(12): 2044-2047
- [29] Chen K, Li B, Shen H, et al. Reduce the complexity of list decoding of polar codes by treepruning [J]. IEEE Communications Letters, 2016, 20(2): 204-207
- [30] Chen J, Fan Y Z, Xia C Y, et al. Low-complexity list successive-cancellation decoding of polar codes using list pruning [C]. In: 2016 IEEE Global Communications Conference (GLOBECOM), Washington DC, 2016. 1-6
- [31] Wang T, Qu D, Jiang T. Parity-check-concatenated polar codes [J]. IEEE Communications Letters, 2016, 20(12): 2342-2345
- [32] Zhang H Z, Li R, Wang J, et al. Parity-check polar coding for 5G and beyond [C]. In: 2018 IEEE International Conference on Communications (ICC), Kansas City, 2018. 1-7
- [33] Yuan B, Parhi K K. Early stopping criteria for energy-efficient low-latency beliefpropagation polar code decoders [J]. IEEE Transactions on Signal Processing, 2014, 62(24): 6496-6506