# FEC 入门导引

V1.0
蔡中恒/c00194582

在写 FEC 的入门导引之前，我心里其实很忐忑。一方面 FEC 和 DSP 相比，差不多相当于两个截然不同的系统，彼此的思维相差非常大。而要认真写起来的话，恐怕几本书都不够，而且我现在也没有这种能力。所以现时我也只能根据自己这段时间的学习和总结，将其记录下来分享给大家。等到后面有了新的感悟，也许还能再在这份文档的基础上再更新一些。

# DAY1：入门书籍推荐

算法的本质是数学。想要从 DSP 进入 FEC，一些数学入门书籍是必不可少的。而在这些入门书籍中，着重要看的是和代数数论相关的书籍。本着由易到难的原则，这里也依次向大家推荐一些我看过的数学书籍。

## 数学女孩2


作者: [日] 结城浩
出版社: 人民邮电出版社
副标题: 费马大定理
译者: 丁 灵
出版年: 2015-12
页数: 368
定价: 42.00元
装帧: 平装
丛书: 图灵新知
ISBN: 9787115411112

豆瓣评分
8.9
56人评价
5星 62.5%
4星 30.4%
3星 7.1%
2星 0.0%
1星 0.0%

首先是结城浩写的《数学女孩 2》这本书，其实这个系列一共有三本，和 FEC 关联最紧密的是第二本。在阅读这本书之前，千万不要以为这本书是日本的轻小说（虽然的确书中有很多少男少女三角恋的描写），但书中使用的公式和各种证明推导可能会超出你的想像，所以一定要保持严肃的态度来阅读这本优秀的科普书籍。本书阅读完之后，应该能够学到的东西有如下几点：

1、 **无穷递降法的证明思路以及费马大定理在 n=4 的证明过程；**
2、 **群环域的基本概念，以及有限域在数学上的作用；**
3、 **费马大定理完整证明的大概思路。**

最后再补充一下，如果看这本书都觉得吃力，里面的公式和证明推导大部分都看不懂的话，那么接下来的几本书，建议就不要再看了——毕竟这本书已经是我推荐的参考书里面最简单的一本。

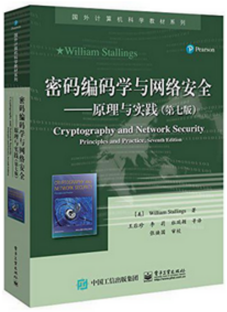
## 程序员的数学2


作者: 平冈和幸 / 堀玄
出版社: 人民邮电出版社
副标题: 概率统计
译者: 陈筱烟
出版年: 2015-8-1
页数: 405
定价: CNY 79.00
装帧: 平装
丛书: 程序员的数学
ISBN: 9787115400512

豆瓣评分
8.7
87人评价
5星 43.7%
4星 35.6%
3星 14.9%
2星 4.6%
1星 1.1%

FEC 涉及到大量的概率统计知识，所以一本好的概率统计书籍非常重要，虽然大家可能都在大学期间学习过概率统计这门课，但我还是要推荐《程序员的数学 2：概率统计》这本书，优点是图例充分，每一点都讲得很细。讲述兼顾了传统的概率统计理论和概率统计的现代应用。这本书阅读完之后，基本上 FEC 相关的概率统计知识，就都没问题了。

## 密码编码学与网络安全-原理与实践




作者: 王后珍 / 威廉·斯托林斯 (William Stallings) / 等
出版社: 电子工业出版社
出版年: 2017-12-1
页数: 284
定价: CNY 95.00
装帧: 平装
ISBN: 9787121329210
传送链接:百度盘 / 微盘 / mLook / Library Genesis / ebook3000 / Torrentseeker / 新浪爱问 / Readfree / 周读

豆瓣评分
目前无人评价

搞定概率统计之后，接下来还需要深入学习一下 FEC 相关的群环域概念，尤其要掌握伽罗华域的作用和原理。这部分倒不用专门去看近世代数的书，看和 FEC 相关的就可以了。其实接班上所有讲到 FEC 的教材，都会在开篇专门拿出一章来将伽罗华域；但实话实说，在我看过的书里面，能够把伽罗华域讲清楚了的，只有这本《密码编码学与网络安全》。这本书只用看**第二章数论基础**和**第五章有限域**即可。看完这两个章节，相信大家能够明白伽罗华域和扩域的概念、本原多项式的由来，以及如何根据本原多项式设计伽罗华域的扩域。

## 代数数论简史


作者: 冯克勤
出版社: 哈尔滨工业大学出版社
出版年: 2015-1-1
页数: 189
定价: 28
装帧: 平装
ISBN: 9787560349602
传送链接:百度盘 / 微盘 / mLook / Library Genesis / ebook3000 / Torrentseeker / 新浪爱问 / Readfree / 周读 / 我的小书屋 / 逛电驴 / 读秀@RUC / 云海电子图书馆 /

豆瓣评分
目前无人评价

读完上面那些书之后，其实 FEC 所需要的基本代数知识，就了解得差不多了。如果想要更进一步，在数学方面打下更坚实的基础，还可以尝试先看看这本《代数数论简史》，了解代数数论的发展简史。看这本书基本上只有两种感受：一、卧槽这个证明真牛逼！二、卧槽这居然也能证明！

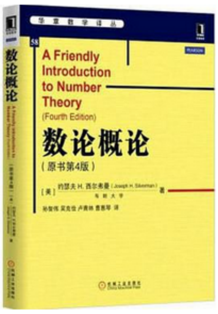
友情提示：

**只看你看得懂的部分！**

**只看你看得懂的部分！**

**只看你看得懂的部分！**

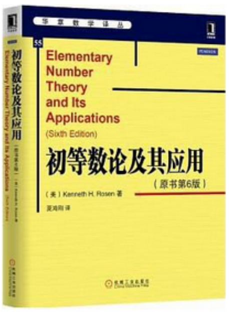
## 数论概论（原书第4版）




作者: [美] 约瑟夫H.西尔弗曼
出版社: 机械工业出版社
译者: 孙智伟 / 吴克俭 / 卢青林 / 曹惠琴
出版年: 2016-1-1
页数: 287
定价: 59.00元
装帧: 平装
丛书: 华章数学译丛
ISBN: 9787111522003
传送链接:百度盘 / 微盘 / mLook / Library Genesis / eb

豆瓣评分
评价人数不足

## 初等数论及其应用（原书第6版）




作者: [美] Kenneth H·Rosen
出版社: 机械工业出版社
译者: 夏鸿刚
出版年: 2015-3-1
页数: 489
定价: 89.00元
装帧: 平装
丛书: 华章数学译丛
ISBN: 9787111486978

豆瓣评分
评价人数不足

对于学有余力的同学，还可以看看这两本代数数论的教材，这两本也是非常有趣（nan）味（du）的书，认真学习这两本书可以进一步体会到数学的美（kong）妙（bu）之处，也能弥补《代数数论简史》中大段公式看不懂的尴尬。如果学完这两本书仍然觉得还不过瘾，那我真诚地建议下一步可以直接上**抽象代数**。

*

我写文章都是按照 DAY 来划分章节，今天的信息量大了一些。稍微总结一下，大概是这样的脉络：

高斯曾经说过："数学是科学的皇后，而数论则是数学的皇后。"FEC 就是建立在数论上的一个成果。数论本身充满了美感，为什么呢？因为数论上的很多证明，包括群环域的概念，基本都是数学界的天才用灵感铺就而成。为什么数学家要创造群环域这种概念？——我个人的理解是**这些概念逼近了运算的本质**。加法、乘法、除法这些运算，都是人类在生产活动中自然而然发现的，但是越往后就越会发现，光靠这些运算解决不了很多问题，所以数学家才会努力去挖掘"运算"的本质。群环域都涉及到对运算的定义，通过设计不同的运算，我们可以发现很多数学猜想中的内在联系，从而在看似完全不相干的命题中找到捷径，从而去证明或者否定。这种捷径，往往就是数学的本质。

数字信号处理这种事情，坦率说非常枯燥。枯燥的原因在于我们只看到了成果，只去运用成果，而没有从根本上去思考数学界的前辈为什么要想出来这些规则。如果真的花点心思去看看当中的数学原理，相信很大程度上能够提升自己对 FEC 的兴趣。**数学天才们建造了一座厨房，而我们只能用里面的微波炉来加热昨晚的剩饭。**多去看看其余的领域，才能更好地理解我们现在做的事情到底在整个数学体系中所在怎样的地位。

DAY1 至此结束，祝大家晚安。最后留一个问题，大家看完上面的书后可以思考一下：**为什么 FEC 和密码学，都必须用有限域，而不用无限域？**

# DAY2：置信传播算法原理推导

介绍 LDPC 码的书有很多，在这里我并不打算重复，只根据自己的理解来简单讲一下 LDPC 的码的原理。大家可以这么想，我们发送信息 bit，目的是为了将信息无损地传递到接收方。信息的传递需要介质，无论是电磁波，或者光波，都可以用来承载信息。当然，这些介质也不可避免地会对传输的信号造成劣化，从而在接收端出现误码。从某种角度而言，数字通信的目的就是为了消除误码，这方面 DSP 会做各种各样的补偿和均衡，但这远远是不够的——**因为 DSP 的各种处理无法消除白噪**。DSP 能够做到的最好程度，就是将完美的符号+白噪声的信号送给 FEC，然后让 FEC 来做最后的处理。

所以这就要求 FEC 必须有一定程度的纠错能力，但是数字通信系统发送的是随机码流，就算出现差错，我们在接收端也是不知道的。这个时候，我们就需要对发端做编码处理，增加一些冗余的开销，来对随机的 bit 之间人为添加约束。举一个最简单的例子，假设我们有 c0 和 c1 两个 bit，为了保证传输不出错，我们可以加一个校验位，组成一个校验方程。

c0+c1+c2 = 0;

注意到这里应该是在伽罗华域里面的运算（现在明 y 白为什么要用有限域而不是用无限域了吧，发送的随机符号取值是有限的，要让校验方程发挥作用，加法运算就必须在有限域下进行，无限域没办法做校验方程），所以这里的加法是模 2 加法，这样三个 bit 就存在联系了。三个 bit 经过传输后，如果我们接收到的符号分别是 y0、y1、y2，那么硬判后按照校验方程，我们可以计算出 y0+y1+y2 的值，如果这个值是 0，那我认为 y0 和 y1 的可靠性就很高；如果这个值是 1，那 y0、y1、y2 里面肯定有存在错误。

接下来大家肯定会想，只知道错误没用啊，我们需要知道哪个 bit 错了，这样才能达到译码目的。这个时候，我们可以从条件概率的公式出发，来看看到底是哪个 bit 错了。

## 1. 只有一个校验方程的情况

将上面的码字 bit 扩展到更一般的情况，还是用上面的例子，在随机发送码流中，假设我们接收到的 y0 存在这样的关系：

$$P(c_0 = 1|y_0) > 0.5$$

这个公式意思是说，在接收到 y0 码字的前提下，c0 为 1 的概率大于 0.5。对应到数字通信系统中，比方说在 BPSK 信号中，如果接收到 y0 大于 0，那我们有足够理由推测发送信号 c0 就是 bit1（在白噪下信号分布服从高斯分布，具体的推导大家可以自行查阅概率统计的教材）。当然这一个条件还不够，我们既然在发送端设计了校验方程，那这个地方肯定要把校验方程也用起来。假设涉及到 c0 的其中一个校验方程 S 是 c0+c1+c2 = 0，那么我们还需要考虑校验方程几个码字之间的相关性。那么根据条件概率的公式，我们可以得到：

$$P\big(c_0 = 1\big|(S|\{y_0, y_1, y_2\})\big) = \frac{P(c_0 = 1, (S|\{y_0, y_1, y_2\}))}{P(S|\{y_0, y_1, y_2\})}$$

满足 S 的条件，就意味着 c0+c1+c2 应该等于 0，这样我们可以进一步展开，获取到 c0=1 且满足 S 的各种情况：

$$P\big(c_0 = 1\big|(S|\{y_0, y_1, y_2\})\big) = \frac{P(c_0 = 1, c_1 = 1, c_2 = 0|\{y_0, y_1, y_2\}) + P(c_0 = 1, c_1 = 0, c_2 = 1|\{y_0, y_1, y_2\})}{\sum_{\substack{q_0, q_1, q_2 \in \{0,1\} \\ q_0 + q_1 + q_2 = 0}} P(c_0 = q_0, c_1 = q_1, c_2 = q_2|\{y_0, y_1, y_2\})}$$

这里令$P(c_0 = 1|y_0) = p_0$，其余以此类推。那我们可以进一步得到：

$$P\big(c_0 = 1 \big| (S|\{y_0, y_1, y_2\})\big) = \frac{p_0p_1(1-p_2)+p_0(1-p_1)p_2}{p_0p_1(1-p_2)+p_0(1-p_1)p_2+(1-p_0)(1-p_1)(1-p_2)+(1-p_0)p_1p_2}$$

这样我们就得到了完整的接收端译码过程。但是校验方程可能不止一个，这些方程当中的某些 bit 又包含在了其它校验方程中，最终组成一种相互约束的网状结构。这种情况下，又应该如何处理呢？

## 2. 有两个校验方程的情况

Gallarger 在 1962 年提出了概率译码软判决算法，置信传播算法（BP）正是在概率译码的基础上发展起来的。

这里我们使用一个稍微复杂一点的校验矩阵，有 9 行，15 列。信息位是 6bit，校验位是 9bit。如下图所示。

$$H = \begin{bmatrix} 0&1&0&1&0&0&0&1&0&0&0&0&0&0&0 \\ 0&0&1&0&1&0&0&0&1&0&0&0&0&0&0 \\ 1&0&0&0&0&1&1&0&0&0&0&0&0&0&0 \\ 0&0&0&1&0&0&1&0&0&1&0&0&0&0&0 \\ 0&0&0&0&1&0&0&1&0&0&1&0&0&0&0 \\ 0&0&0&0&0&1&0&0&1&0&0&1&0&0&0 \\ 1&0&0&0&0&0&0&0&0&1&0&0&1&0&0 \\ 0&1&0&0&0&0&0&0&0&0&1&0&0&1&0 \\ 0&0&1&0&0&0&0&0&0&0&0&1&0&0&1 \end{bmatrix}$$

那么我们的码字 bit 也可以写成矩阵形式：

$$c^T = \begin{bmatrix} c1 \\ c2 \\ c3 \\ c4 \\ c5 \\ c6 \\ c7 \\ c8 \\ c9 \\ c10 \\ c11 \\ c12 \\ c13 \\ c14 \\ c15 \end{bmatrix}$$

**这里多说一下，DSP 里面的书如果给出一个一维向量，一般默认是列向量；但 FEC 的书里面给出一个一维向量，一般默认就是行向量。**

校验方程和码字做矩阵乘法，乘积应该是一个全为 0 的向量：

$$Hc^T = \begin{cases} c2 + c4 + c8 = 0 \\ c3 + c5 + c9 = 0 \\ c1 + c6 + c7 = 0 \\ c4 + c7 + c10 = 0 \\ c5 + c8 + c11 = 0 \\ c6 + c9 + c12 = 0 \\ c1 + c10 + c13 = 0 \\ c2 + c11 + c14 = 0 \\ c3 + c12 + c15 = 0 \end{cases}$$

当然这里的加法仍然是模 2 加法。

我们先看 c1。c1 包含在 2 个校验方程中，分别是：

$$\boldsymbol{c1} + c6 + c7 = 0$$

$$\boldsymbol{c1} + c10 + c13 = 0$$

要按照之前我们推导的公式来计算 c1 为 1 的概率，显然我们需要先知道 c6、c7、c10、c13 的概率。假设我们已经知道它们为 1 的概率，根据其在校验方程中的位置，我们将其命名为：

$$P(c_6 = 1|(S_3|y_6)) = P_{3,2}$$

解释一下，$S_3$是从上往下数第 3 个校验方程，$P_{3,2}$是第 3 个校验方程里面第 2 个 bit 为 1 的概率，这样的话，其余的 c7、c10、c13 也可以依次得出：

$$P(c_7 = 1|(S_3|y_7)) = P_{3,3}$$

$$P(c_{10} = 1|(S_7|y_{10})) = P_{7,2}$$

$$P(c_{13} = 1|(S_7|y_{13})) = P_{7,3}$$

那么对 c1 来说，c1 参与了两个校验方程，那么 c1 在两个校验方程中为 1 的概率应该分别表示为：

$$P(c_1 = 1|(S_3|y_1))$$

$$P(c_1 = 1|(S_7|y_1))$$

但我们真正要求解的是$P(c_1 = 1|y_1)$，**在$\boldsymbol{S_3}$和$\boldsymbol{S_7}$相互独立的情况下**，根据概率统计的知识，可以得到：

$$P(c_1 = 1|y_1) = P(c_1 = 1|(S_3|y_1)) \cdot P(c_1 = 1|(S_7|y_1))$$

这就是说，我们只需要分别求解$P(c_1 = 1|(S_3|y_1))$和$P(c_1 = 1|(S_7|y_1))$，然后两者相乘就能求解出 c1 的概率。而对于每个独立的校验方程，在上一小节里面我们已经做了充分分析，套用相应公式即可。

## 3. 多个校验方程的情况

接下来我们要推广到更一般的情况了。假设我们要求解的码字是$c_d$，这个码字包含在 j 个校验方程中，我们假设每个校验方程都由 m 个码字组成。在讨论之前，我们先证明一个结论。

假设一个二进制序列的长度为$m$，bit 相互独立，其中第$l$个 bit 为 1 的概率为$P_l$，那么整个序列中包含偶数个 1 的概率为：

$$P_{even} = \frac{1 + \sum_{l=1}^{m}(1 - 2P_l)}{2}$$

证明如下：

这个序列和二项分布有些不同，二项分布通常是将每个随机变量等于 1 的概率都看做相同，而这里因为我们的校验方程里每个 bit 为 1 的概率有所不同，所以专门做了设定：第$l$个 bit 为 1 的概率为$P_l$。

但在这里我们还是可以借用二项分布概率的推导原理来推导该式子。在二项分布情况下，假设每个 bit 为 1 的概率都是$p$，根据概率统计的相关知识，从**组合**的角度，我们可以得到这个序列中有$k$个 1 的概率为：

$$P(k) = C_m^k p^k (1 - p)^{m-k}$$

而二项分布的概率公式又可以用牛顿多项式来表达：

$$[(1 - p) + pt]^m$$

将$t$看做自变量，将这个多项式展开，序列中有$k$个 1 的概率恰好就是$t^k$的系数

$C_m^k p^k (1-p)^{m-k}$。

我们将二项分布推广到更一般的情况，第$l$个 bit 为 1 的概率为$P_l$。这种情况下，牛顿多项式就应该变为：

$$f(t) = \prod_{l=1}^{m} [(1-P_l) + P_l t]$$

这里面$t^k$的系数，也恰好就是：**一个二进制序列的长度为$m$，bit 相互独立，其中第$l$个 bit 为 1 的概率为$P_l$，那么整个序列中包含$k$个 1 的概率。**

这样达到我们的要求了吗？没有。我们要得到的是**包含偶数个 1 的概率**。

那这样就要在牛顿多项式上再动动脑筋了。

我们可以再构造一个式子，在$f(t)$上稍微做一点变形，得到$g(t)$：

$$g(t) = \prod_{l=1}^{m} [(1-P_l) - P_l t]$$

对$g(t)$来说，当$k$为偶数的时候，$t^k$的系数和$f(t)$相同；当$k$为奇数的时候，$t^k$的系数和$f(t)$相反。如果我们把$f(t)$和$g(t)$相加再除以 2，就可以得到：

$$f(t) + g(t) = \frac{\prod_{l=1}^{m}[(1-P_l) + P_l t] + \prod_{l=1}^{m}[(1-P_l) - P_l t]}{2}$$

如果我们令$t=1$，那么$f(1)+g(1)$就是我们要的**包含偶数个 1 的概率。**

$$P_{even} = \frac{\prod_{l=1}^{m}[(1-P_l) + P_l] + \prod_{l=1}^{m}[(1-P_l) - P_l]}{2} = \frac{1 + \prod_{l=1}^{m}(1-2P_l)}{2}$$

自然也可以得到：

$$P_{odd} = 1 - P_{even} = \frac{1 - \prod_{l=1}^{m}(1-2P_l)}{2}$$

证明完毕。

我们再来看多个校验方程的情况，根据条件概率的定义，我们可以对$c_d$的概率写出如下公式：

$$P(c_d = 0|(S_i|y)) = \frac{P(c_d = 0, S_i|y)}{P(S_i|y)}$$

其中$S_i$是包含$c_d$的第$i$个校验方程。同样也可以得到：

$$P(c_d = 1|(S_i|y)) = \frac{P(c_d = 1, S_i|y)}{P(S_i|y)}$$

两个式子，其实只要获取到一个，我们就可以得到$c_d$的准确值，但是仔细看看就会发现，$P(S_i|y)$这个式子并不好求，但是我们可以用折中的方法来消去这个式子，这就是：

$$\frac{P(c_d = 0|(S_i|y))}{P(c_d = 1|(S_i|y))} = \frac{P(c_d = 0, S_i|y)}{P(c_d = 1, S_i|y)}$$

$\frac{P(c_d=0|(S_i|y))}{P(c_d=1|(S_i|y))}$的值，只要大于 1，我们就认为$c_d$是 0；反之则认为$c_d$是 1。继续运用条件概率的知识来展开式子：

$$\frac{P(c_d = 0|(S_i|y))}{P(c_d = 1|(S_i|y))} = \frac{P(c_d = 0, S_i|y)}{P(c_d = 1, S_i|y)} = \frac{P(c_d = 0, S_i, y)}{P(c_d = 1, S_i, y)} = \frac{P(c_d = 0, y)P(S_i|c_d = 0, y)}{P(c_d = 1, y)P(S_i|c_d = 1, y)}$$
$$= \frac{P(y)P(c_d = 0|y)P(S_i|c_d = 0, y)}{P(y)P(c_d = 1|y)P(S_i|c_d = 1, y)} = \frac{P(c_d = 0|y)P(S_i|c_d = 0, y)}{P(c_d = 1|y)P(S_i|c_d = 1, y)}$$

这里为了让式子看起来方便一些，我们可以令$P(c_d=1|y)=P_d$，那么$P(c_d=0|y)=1-P_d$。

那么根据我们此前用楷体字推导的结论，如果$c_d=0$，那么$S_i$剩余的码字必定有偶数个1；如果$c_d=1$，那么$S_i$剩余的码字必定有奇数个1。这样可以得到：

$$P(S_i|c_d=0,y)=\frac{1+\prod_{l=1}^{m-1}\left(1-2P_{i,l}\right)}{2}$$

$$P(S_i|c_d=1,y)=\frac{1-\prod_{l=1}^{m-1}\left(1-2P_{i,l}\right)}{2}$$

其中$P_{i,l}$表示第$i$个校验方程里面第$l$个码字为 1 的概率。综合所有$j$个校验方程，就可以得到$c_d$的最终概率：

$$P(S|c_d=0,y)=\prod_{i=1}^{j}\frac{1+\prod_{l=1}^{m-1}\left(1-2P_{i,l}\right)}{2}$$

$$P(S|c_d=1,y)=\prod_{i=1}^{j}\frac{1-\prod_{l=1}^{m-1}\left(1-2P_{i,l}\right)}{2}$$

代入后可以得到：

$$\frac{P(c_d=0|(S_i|y))}{P(c_d=1|(S_i|y))}=\frac{1-P_d}{P_d}\prod_{i=1}^{j}\frac{1+\prod_{l=1}^{m-1}\left(1-2P_{i,l}\right)}{1-\prod_{l=1}^{m-1}\left(1-2P_{i,l}\right)} \tag{1}$$

这就是说，我们只要知道了$P_{i,l}$的值，我们就可以把$c_d$的条件概率算出来。问题是$P_{i,l}$又该如何求得？如果我们强行令$P_{i,l}=P\left(c_{i,l}=1|y\right)$，那这种设定是不够准确的，因为这种设定没有考虑$c_{i,l}$自己所参与的校验方程对其的约束。

所以，稳妥的做法是使用(1)式来计算$P_{i,l}$，注意到在计算$P_{i,l}$的校验方程时，$P_{i,l}$和$c_d$共有的校验方程不能纳入其中，因为我们的目的是利用$P_{i,l}$来求得$c_d$，如果在计算$P_{i,l}$的时候就已经用了$c_d$了，那就会陷入循环利用的怪圈。

这样一层层地往上追溯，只要这些校验方程没有环路，最后总能沿着不同的校验方程追溯到源头，在尽头的时候，我们就可以仅考虑接收信号的值，而不用考虑 bit 之间的校验方程关系（相关的校验方程在追溯的过程中已经用过，就不能再重复利用了）。我们还是以之前的校验矩阵为例，来说明如何求解$c1$。

$$Hc^T=\begin{cases}c2+c4+c8=0\\c3+c5+c9=0\\c1+c6+c7=0\\c4+c7+c10=0\\c5+c8+c11=0\\c6+c9+c12=0\\c1+c10+c13=0\\c2+c11+c14=0\\c3+c12+c15=0\end{cases}$$

$c1$参与的校验方程如下所示：

$$\boldsymbol{c1}+c6+c7=0$$

$$\boldsymbol{c1}+c10+c13=0$$

第一步，我们先看如何求解$c6$，那就先看看$c6$参与的校验方程（排除和$c1$相关的校验方程）：

$$c6+\boldsymbol{c9}+\boldsymbol{c12}=0$$

那这里我们还需要先求出$c9$和$c12$，分别看两者参与的校验方程：

$$c3+c5+\boldsymbol{c9}=0$$

$$c3 + \boldsymbol{c12} + c15 = 0$$

这里面$c15$只参与了这一个校验方程，所以我们可以令$P_{15} = P(c_{15} = 1|y)$，由此得到一个尽头。

而另外的$c3$在校验方程中构成了一个环路（$c3, c9, c12$）。这种环路按道理来说无法严格用式子(1)来计算概率，因为这三个校验方程相互嵌套，彼此并不独立，而$\prod_{i=1}^{j}$　这个式子的前提就是各校验方程相互独立。但是在实际应用中，只要环路够长，一般来说强行使用式子(1)的结果虽然无法做到最佳，但也还是能接受。

剩下码字的概率求解方法可以以此类推，这种层层追溯的过程，可以画出一个树状的图形，叫做校验集合树。

至此，概率译码算法的步骤可以描述如下：对每一个码字 bit，画出相应的校验集合树，从最高层的节点开始，应用式子(1)逐层计算出各节点的后验概率分布，直到求出根节点的后验概率分布，根据该后验概率分布判决该 bit 是 1 还是 0。

# 4. LLR 是怎么来的？

在工程运算中，可以注意到式子(1)需要很多乘法，为了简化运算，我们可以将其取自然对数：

$$\ln\frac{P(c_d = 0|(S_i|y))}{P(c_d = 1|(S_i|y))} = \ln\frac{1 - P_d}{P_d} + \sum_{i=1}^{j}\ln\frac{1 + \prod_{l=1}^{m-1}(1 - 2P_{i,l})}{1 - \prod_{l=1}^{m-1}(1 - 2P_{i,l})}$$

为了简化公式，我们直接标记$\ln\frac{P(X=0)}{P(X=1)}$为 LLR，这样就可以得到：

$$LLR_{c_d\_final} = LLR_{c_d\_init} + \sum_{i=1}^{j}\ln\frac{1+\prod_{l=1}^{m-1}(1-2P_{i,l})}{1-\prod_{l=1}^{m-1}(1-2P_{i,l})} \tag{2}$$

我们再来看看$\sum_{i=1}^{j}\ln\frac{1+\prod_{l=1}^{m-1}(1-2P_{i,l})}{1-\prod_{l=1}^{m-1}(1-2P_{i,l})}$这个式子，如果也要做成 LLR 的形式，可以这样来看：

令$\ln\frac{P(c_{i,l}=0|(S|y))}{P(c_{i,l}=1|(S|y))} = LLR_{i,l}$，那么$P_{i,l} = \frac{1}{1+e^{LLR_{i,l}}}$，代入$\prod_{l=1}^{m-1}(1 - 2P_{i,l})$后得到：

$$\prod_{l=1}^{m-1}(1 - 2P_{i,l}) = \prod_{l=1}^{m-1}\left(1 - \frac{2}{1 + e^{LLR_{i,l}}}\right) = \prod_{l=1}^{m-1}\left(\frac{e^{LLR_{i,l}} - 1}{e^{LLR_{i,l}} + 1}\right) = \prod_{l=1}^{m-1}\left(\frac{e^{\frac{LLR_{i,l}}{2}} - e^{-\frac{LLR_{i,l}}{2}}}{e^{\frac{LLR_{i,l}}{2}} + e^{-\frac{LLR_{i,l}}{2}}}\right)$$

根据双曲正切函数的定义$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$，可以得到：

$$\prod_{l=1}^{m-1}(1 - 2P_{i,l}) = \prod_{l=1}^{m-1}\tanh(\frac{LLR_{i,l}}{2})$$

代入式子(2)中：

$$LLR_{c_d\_final} = LLR_{c_d\_init} + \sum_{i=1}^{j}\ln\frac{1 + \prod_{l=1}^{m-1}\tanh(\frac{LLR_{i,l}}{2})}{1 - \prod_{l=1}^{m-1}\tanh(\frac{LLR_{i,l}}{2})}$$

又根据双曲反正切函数的定义：$\tanh^{-1}(x) = \frac{1}{2}\ln\frac{1+x}{1-x}$，我们又可以进一步简化为：

$$LLR_{c_{d\_final}} = LLR_{c_{d\_init}} + \sum_{i=1}^{j} 2\tanh^{-1}(\prod_{l=1}^{m-1}\tanh(\frac{LLR_{i,l}}{2})) \quad (3)$$

最后我们还要将公式修正一下，在第 3 节的讨论中，已经说过了在计算$P_{i,l}$的时候，不能使用同时包含$c_{i,l}$和$c_d$的校验方程，所以这里我们要把式子(3)写成更一般的形式，这样就得到式子(4)：

$$LLR_{c_{d\_new}} = LLR_{c_{d\_init}} + \sum_{m' \in M(l)}\left[2\tanh^{-1}(\prod_{l' \in L(m')\backslash l}\tanh(\frac{LLR_{m'l'}}{2}))\right] \quad (4)$$

解释一下，$LLR_{c_d}$，其中$c_d$表示码字自己的编号顺序，以上面的校验矩阵为例，$c_d$就表示$c1, c2, c3, \ldots$这些码字，而当中的$m'$是校验方程的编号，因为一个码字可能参与好几个校验方程，比方说$c1$参与了第 3 个和第 7 个校验方程，这里的$m'$就是 3 和 7。$l$表示在第$m'$个校验方程中的第$l$个码字。

这就是置信传播算法（BP）最基本的公式，置信传播算法又叫做和积算法，从这个公式自然就能看出来和在哪里，积在哪里。这里再补充说明一下：

$L(m')$表示和校验方程$S_{m'}$相连的码字$c_d$的集合，$L(m')\backslash l$就表示集合$L(m')$去掉$l$。这个操作的原因是：校验方程$S_{m'}$中$c_d$为 1 的概率，应该等于$S_{m'}$中**其余的码字**出现奇数个 1 的概率，所以自然需要将$l$自身去掉。

$M(l)$表示和码字$c_d$（其中$c_d$处在该校验方程的第$l$个位置上）相连的所有校验方程的集合。式子(4)中还有一点需要注意，在计算$LLR_{m'l'}$的时候，需要排除同时包含有$c_{m'l'}$和$c_d$的校验方程。这一点约束在式子(4)中不太好表现出来，但在下一节里面，我们将用迭代运算的方式，专门来讲述。

# 5. 如何迭代？

根据之前的推导，似乎我们只需要运用公式(4)，对每个码字一层层地做一次条件概率运算，就可以得到正确的结果。——显然这是不可能的。为什么呢？在第 3 节里面，我们一层层往上追溯直到顶点，这个时候我们强行令$P_{15} = P(c_{15} = 1|y)$。其实这样假设是不太合适的，因为$c_{15}$本身也受到校验方程的约束，不能简单根据接收信号直接判定其概率。——这样局面就尴尬了。为了打破这个局面，我们就需要在运用中采用迭代译码的方式。

迭代译码的思路是这样的：

最开始计算各节点概率时，我们手头能用的只有接收信号的概率（例如$P_{15} = P(c_{15} = 1|y)$），这个概率肯定不够准确，但是也能用。我们可以先用接收信号的概率作为初始的 LLR 值，先把所有码字都算一遍，得到新的 LLR 值。经过第一次迭代后，我们就可以认为新的 LLR 值比初始的 LLR 值更接近真实概率。既然每一次迭代都可以更接近真实概率，那么只要我们迭代多次，最终得到的 LLR 值就一定可以逼近真实的概率。

这个思路也是 Gallarger 想出来的，LLR 就是每个节点的置信度，迭代的过程就是置信度在校验方程这个大网络中不断传播的过程，所以叫做置信传播算法。坦率说前面几节的推导非常具有逻辑性，只要一步步推导，一般都能搞定。但是迭代算法这个东西，确实需要灵光一现才能想出来。这就是顺着别人的路往下走，和自己开辟新道路的差别。

接下来我们来分拆式子(4)，完整写出一遍迭代流程：

**Step 1：根据接收信号的初始 LLR 值，计算每个校验方程中每个码字更新后的概率。**

我们用$m'$表示第$m'$个校验方程，$l$表示第$i$个校验方程中第$l$个码字，这样我们将公式的一部分写成如下的形式：

$$C_{m'l}=2\tanh^{-1}(\prod_{l'\in L(m')\backslash l}\tanh(\frac{LLR_{m'l'}}{2}))$$

还是用之前的校验矩阵来举例。

$$Hc^T=\begin{cases} c2+c4+c8=0 \\ c3+c5+c9=0 \\ c1+c6+c7=0 \\ c4+c7+c10=0 \\ c5+c8+c11=0 \\ c6+c9+c12=0 \\ c1+c10+c13=0 \\ c2+c11+c14=0 \\ c3+c12+c15=0 \end{cases}$$

根据校验方程，如果我们要计算$C_{11}$的值，那对应的$LLR_{m'l'}$就是$c4$和$c8$的 LLR 值，如果是计算$C_{12}$的值，那对应的$LLR_{m'l'}$就是$c2$和$c8$的 LLR 值。其余以此类推，按照矩阵将所有的$C_{m'l}$都算一遍，得到的$C$矩阵和校验方程的维度相同，都是 9 行 3 列。

**Step 2：根据计算出的$C_{m'l}$，更新每个码字的 LLR。**

这样新的公式就是：

$$LLR_{c_d\_new}=LLR_{c_d\_init}+\sum_{m'\in M(l)}C_{m'l}$$

上一步计算出来的$C$矩阵中，$C_{11}$和$C_{81}$都是码字$c2$参与的校验方程，所以要计算$c2$新的 LLR 值，那就应该是：

$$LLR_{c2\_new}=LLR_{c2\_init}+C_{11}+C_{81}$$

这里取$d=2$，$m'$为 1 和 8，$l$为 1。

其余码字以此类推。

其实在这个时候，我们已经可以将$LLR_{c2_new}$做硬判直接得到结果，但是一次迭代是不保险的，我们还需要多迭代几次，逐渐逼近最佳值。那么，是不是将 step1 和 step2 循环使用就可以了呢？不是的，**我们还没有排除同时包含有$c_{m'l'}$和$c_l$的校验方程**，直接循环会让信息重复发送，所以这里还需要多花一个步骤。

**Step 3：减去重复发送的信息**

我们先看看，如果只有两个步骤的话，会是什么局面：

Step 1：通过$LLR_{m'l'}$来计算$C_{m'l}$；其中$C_{m'l}=2\tanh^{-1}(\prod_{l'\in L(m')\backslash l}\tanh(\frac{LLR_{m'l'}}{2}))$；

Step 2：通过得到的$C_{m'l}$来计算$LLR_{c_d_new}=LLR_{c_d_init}+\sum_{m'\in M(l)}C_{m'l}$

然后循环进行。

这里的问题在于，我在 step1 计算$C_{m'l}$的时候，这里面的$LLR_{m'l'}$有一部分信息来自于上一次迭代给过来的$C_{m'l}$，这个地方就形成了信息的重复传递。所以，我们在 step1 之前，需要先做一个 step0，将上一次传递过来的$C_{m'l}$信息减掉。

**这样我们就得到了 BP 算法的最终流程，这里重写如下：**

**Step 0：根据接收信号的初始 LLR 值，减去重复信息，得到$V_{m'l}$。**

这里的$V$矩阵和$C$矩阵相同维度，因为码字的 LLR 值只有一个，但是同一个码字需要参与多个校验方程，所以针对每个不同的校验方程，$V_{m'l}$也不同，公式如下。

$$V_{m'l} = LLR_{c_d\_old} - C_{m'l\_old}$$

注意，如果是第一次迭代，那么$C_{m'l_old} = 0$，$LLR_{c_d\_old} = LLR_{c_d\_init}$

然后根据上面的矩阵再举例说明：

$$Hc^T = \begin{cases} c2 + c4 + c8 = 0 \\ c3 + c5 + c9 = 0 \\ c1 + c6 + c7 = 0 \\ c4 + c7 + c10 = 0 \\ c5 + c8 + c11 = 0 \\ c6 + c9 + c12 = 0 \\ c1 + c10 + c13 = 0 \\ c2 + c11 + c14 = 0 \\ c3 + c12 + c15 = 0 \end{cases}$$

如果我们要计算$V_{11}$和$V_{81}$的值，那公式就是：

$$V_{11} = LLR_{c2\_old} - C_{11\_old}$$
$$V_{81} = LLR_{c2\_old} - C_{81\_old}$$

其余码字以此类推。

**Step 1：根据接收信号的初始 LLR 值，计算每个校验方程中每个码字更新后的概率。**

我们用$m'$表示第$m'$个校验方程，$l$表示第$i$个校验方程中第$l$个码字的$V$值，这样我们将公式的一部分写成如下的形式：

$$C_{m'l} = 2\tanh^{-1}\Big(\prod_{l' \in L(m') \backslash l} \tanh\big(\frac{V_{m'l}}{2}\big)\Big)$$

还是用之前的校验矩阵来举例。

$$Hc^T = \begin{cases} c2 + c4 + c8 = 0 \\ c3 + c5 + c9 = 0 \\ c1 + c6 + c7 = 0 \\ c4 + c7 + c10 = 0 \\ c5 + c8 + c11 = 0 \\ c6 + c9 + c12 = 0 \\ c1 + c10 + c13 = 0 \\ c2 + c11 + c14 = 0 \\ c3 + c12 + c15 = 0 \end{cases}$$

根据校验方程，如果我们要计算$C_{11}$的值，那对应的$LLR_{m'l'}$就是$c4$和$c8$的 LLR 值，如果是计算$C_{12}$的值，那对应的$LLR_{m'l'}$就是$c2$和$c8$的 LLR 值。其余以此类推，按照矩阵将所有的$C_{m'l}$都算一遍，得到的$C$矩阵和校验方程的维度相同，都是 9 行 3 列。

**Step 2：根据计算出的$C_{m'l}$，更新每个码字的 LLR。**

这样新的公式就是：

$$LLR_{c_d\_new} = LLR_{c_d\_old} + \sum_{m' \in M(l)} C_{m'l}$$

上一步计算出来的$C$矩阵中，$C_{11}$和$C_{81}$都是码字$c2$参与的校验方程，所以要计算$c2$新的 LLR 值，那就应该是：

$$LLR_{c2\_new} = LLR_{c2\_old} + C_{11} + C_{81}$$

其余码字以此类推。

此时对$LLR_{c2_new}$做硬判，观察得到的结果是否满足所有校验方程，如果满足，迭代过程

停止；如果不满足，重复 step0~step2，直到满足校验方程。

DAY2 至此结束，祝大家晚安。

# DAY3：分组 LDPC 编译码介绍

这一章我们会用之前的校验矩阵做例子，完整走一遍分组码的编译码流程，最后形成浮点代码的形式。

首先重写一遍校验矩阵 H 和校验方程。信息位是 6bit，校验位是 9bit。

$$H=\begin{bmatrix}0&1&0&1&0&0&0&1&0&0&0&0&0&0&0\\0&0&1&0&1&0&0&0&1&0&0&0&0&0&0\\1&0&0&0&0&1&1&0&0&0&0&0&0&0&0\\0&0&0&1&0&0&1&0&0&1&0&0&0&0&0\\0&0&0&0&1&0&0&1&0&0&1&0&0&0&0\\0&0&0&0&0&1&0&0&1&0&0&1&0&0&0\\1&0&0&0&0&0&0&0&0&1&0&0&1&0&0\\0&1&0&0&0&0&0&0&0&0&1&0&0&1&0\\0&0&1&0&0&0&0&0&0&0&0&1&0&0&1\end{bmatrix}$$

校验方程如下：

$$Hc^T=\begin{cases}c2+c4+c8=0\\c3+c5+c9=0\\c1+c6+c7=0\\c4+c7+c10=0\\c5+c8+c11=0\\c6+c9+c12=0\\c1+c10+c13=0\\c2+c11+c14=0\\c3+c12+c15=0\end{cases}$$

分组码的编码，显然应该是每一组进入 6 个新码字，然后根据 6 个新码字来计算 9 个 bit 的校验码字。这里把 H 矩阵也分为信息位和校验位两部分：$H=[H_I,H_p]$。同时也将编码后的码字划分为信息位和校验位两部分：$c=[m,p]$。根据校验关系：

$$Hc^T=0$$

所以：

$$H_Im^T+H_pp^T=0$$
$$H_Im^T=H_pp^T=u$$

所以在校验的时候，我们可以先根据信息位的$H_Im^T$求出$u$，然后根据$H_p$求出$p$，这样就完成了一次编码过程。这里可以注意到，在运算过程中我们需要对$H_p$求逆矩阵。在硬件电路中，求逆矩阵的操作是非常难实现的，为了简化硬件计算，$H_p$一般采用的是单位矩阵循环移位的形式，这样矩阵的求逆计算就可以变为数组的循环移位，从而简化硬件电路，此处不细讲，大家自行思考。

译码的方法就是上一章讲过的方法，当然口说无凭，我们还是直接上代码吧。首先是设定区域，最后生成的 H 就是文档中写的校验矩阵。

```matlab
clear
clc
close all
rng(266);

%% setting
bit_grp      = 1000;
k            = 6;
m            = 9;
n            = k + m;
noise_scale = 2.4;% 调整噪声功率的比值，目的是让BPSK信号既出现纠前误码又不至于超出纠前门限
iter_num     = 5; % 译码迭代次数
dec_grp      = bit_grp;
dec_mode     = 0; % 0: atanh mode; 1: minsum mode.
dec_out      = zeros(n,dec_grp);% 硬判输出结果

Hb           = [1 0 1 -1 -1; -1 0 0 0 -1;0 -1 -1 0 0];
Z            = eye(3);
H            = zeros(size(Hb,1)*length(Z),size(Hb,2)*length(Z));

for idx_row = 1:size(Hb,1)
    for idx_col = 1:size(Hb,2)
        if(Hb(idx_row,idx_col)==-1)
            H((idx_row-1)*length(Z)+(1:length(Z)),(idx_col-1)*length(Z)+(1:length(Z))) = zeros(size(Z));
        else
            H((idx_row-1)*length(Z)+(1:length(Z)),(idx_col-1)*length(Z)+(1:length(Z))) = circshift(Z,[0 Hb(idx_row,idx_col)]);
        end
    end
end

HI = H(:,1:k);
HP = H(:,k+1:end);
```

接下来先计算好两个用于指示 idx 的矩阵，方便在后面译码的时候用。其中 C2V_slct 矩阵计算出来是这样的，恰好就是校验方程的形式：

| | | |
|---|---|---|
| 2 | 4 | 8 |
| 3 | 5 | 9 |
| 1 | 6 | 7 |
| 4 | 7 | 10 |
| 5 | 8 | 11 |
| 6 | 9 | 12 |
| 1 | 10 | 13 |
| 2 | 11 | 14 |
| 3 | 12 | 15 |

而 V2C_slct 矩阵是这样的：

| | |
|---|---|
| 3 | 7 |
| 1 | 8 |
| 2 | 9 |
| 1 | 4 |
| 2 | 5 |
| 3 | 6 |
| 3 | 4 |
| 1 | 5 |
| 2 | 6 |
| 4 | 7 |
| 5 | 8 |
| 6 | 9 |
| 7 | 7 |
| 8 | 8 |
| 9 | 9 |

每一行表示一个码字，每列表示该码字参与的校验方程。比方说第 2 行的 1 和 8，就表示$c2$参与了第 2 个和第 8 个校验方程。

```
% 找出H矩阵中的非0元素，方便迭代用。
[node_row,node_col] = find(H==1);
node_array          = [node_row node_col];
clear node_row node_col
% C2V表示校验方程。
C2V_slct            = zeros(size(H,1),length(find(node_array(:,1)==1)));
for idx=1:size(C2V_slct,1)
    % 找到H矩阵每行中为1的位置
    Q_adress        = find(node_array(:,1)==idx);
    % 将该位置存入Q_slct矩阵中
    C2V_slct(idx,:) = node_array(Q_adress,2);
end

% V2C，行为变量节点，列为和该变量节点相连结的校验节点
V2C_slct            = zeros(size(H,2),2);
for idx=1:size(V2C_slct,1)
    Q_adress        = find(node_array(:,2)==idx);
    V2C_slct(idx,:) = node_array(Q_adress,1);
end
```

接下来是编码、做 BPSK 映射，加噪声的过程，对 BPSK 近似用幅度来计算其 LLR 值。

```
%% create the bit data in
bit_in = randi([0 1],k*bit_grp,1);
bit_in = reshape(bit_in,k,[]);

%% use LDPC to encode
u = mod(HI*bit_in,2);
p = mod(mod(inv(HP),2)*u,2);
% check code
% H_check = mod(HI*bit_in + HP*p,2);

enc_out = [bit_in;p];

%% mapping and calc llr
llr_data = 2*double(~enc_out)-1;
% 因为llr=ln(APP(X=0)/APP(X=1))，这里为了简化llr的计算，所以对enc_out做了一个取反动作，然后直接映射成BPSK信号，作为LLR值。
%% add noise
noise    = randn(size(llr_data))+1j*randn(size(llr_data));
noise    = noise/noise_scale;
tmp_data = llr_data + noise;

scatterplot(tmp_data(:))
llr_data = real(llr_data + noise);
```

接下来是译码流程，每一组码字译码前，先将相关矩阵清零，然后开始迭代处理。第一步先更新 V 矩阵，根据之前 BP 算法的流程，更新 V 矩阵的时候需要将 APP 值（APP 就是迭代过程中保存的 LLR 值）减去对应的 C，去除重复传递的信息。idx 大于 12 的时候只减一个，是因为 c13、c14、c15 都只参与了一个校验方程。

```
for grp_idx=1:dec_grp
    tmp_llr_data = llr_data_in(:,grp_idx);
    V            = zeros(15,2);
    APP          = tmp_llr_data;
    C            = zeros(9,3);
    tmp_dec_out  = zeros(15,1);

    for iter_idx=1:iter_num
        % 更新V的概率信息
        for idx=1:n
            [R_row,R_col] = find(C2V_slct==idx);
            C_slct        = [R_row,R_col];
            if(idx<=12)
                V(idx,1) = APP(idx) - C(C_slct(1,1),C_slct(1,2));
                V(idx,2) = APP(idx) - C(C_slct(2,1),C_slct(2,2));
            else
                V(idx,1) = APP(idx) - C(C_slct(1,1),C_slct(1,2));
            end
        end
```

接下来更新 C 矩阵，这里提供了两种方法，一种是上一章推导过的直接使用双曲正切和反双曲正切函数来计算 C 矩阵；另外一种叫最小和算法，是对双曲正切在硬件实现上的改进，最小和算法在这份文档中不讲，大家可以自行查阅相关材料，也可以自己思考一下。

```
    % 校验节点信息更新：Z1到Z9
    for idx=1:m
        idx_array_1 = V2C_slct(C2V_slct(idx,1),:);
        new_idx_1   = find(idx_array_1==idx,1);
        idx_array_2 = V2C_slct(C2V_slct(idx,2),:);
        new_idx_2   = find(idx_array_2==idx,1);
        idx_array_3 = V2C_slct(C2V_slct(idx,3),:);
        new_idx_3   = find(idx_array_3==idx,1);

        if(dec_mode==0)
            C(idx,1) = 2*atanh(tanh(V(C2V_slct(idx,2),new_idx_2)/2)*tanh(V(C2V_slct(idx,3),new_idx_3)/2));
            C(idx,2) = 2*atanh(tanh(V(C2V_slct(idx,1),new_idx_1)/2)*tanh(V(C2V_slct(idx,3),new_idx_3)/2));
            C(idx,3) = 2*atanh(tanh(V(C2V_slct(idx,1),new_idx_1)/2)*tanh(V(C2V_slct(idx,2),new_idx_2)/2));
        else
            C(idx,1) = sign(V(C2V_slct(idx,2),new_idx_2))*sign(V(C2V_slct(idx,3),new_idx_3))*min(abs(V(C2V_slct(idx,2),new_idx_2)),abs(V(C2V_slct(idx,3),new_idx_3)));
            C(idx,2) = sign(V(C2V_slct(idx,1),new_idx_1))*sign(V(C2V_slct(idx,3),new_idx_3))*min(abs(V(C2V_slct(idx,1),new_idx_1)),abs(V(C2V_slct(idx,3),new_idx_3)));
            C(idx,3) = sign(V(C2V_slct(idx,1),new_idx_1))*sign(V(C2V_slct(idx,2),new_idx_2))*min(abs(V(C2V_slct(idx,1),new_idx_1)),abs(V(C2V_slct(idx,2),new_idx_2)));
        end
    end
```

最后更新 APP 值，并做硬判和校验；等到迭代次数完成后，就输出最终的译码值，并且将收发端的 bit 信息做对比，如果 148 行的 sum(sum(check))输出为 0，就表明译码成功。

```
124             % 变量节点信息更新：c1到c15
125 -           for idx=1:n
126 -               [R_row,R_col] = find(C2V_slct==idx);
127 -               C_slct        = [R_row,R_col];
128 -               if(idx<=12)
129 -                   APP(idx)  = APP(idx) + C(C_slct(1,1),C_slct(1,2)) + C(C_slct(2,1),C_slct(2,2));
130 -               else
131 -                   APP(idx)  = APP(idx) + C(C_slct(1,1),C_slct(1,2));
132 -               end
133                 % 硬判
134 -               if(APP(idx)>=0)
135 -                   tmp_dec_out(idx) = 0;
136 -               else
137 -                   tmp_dec_out(idx) = 1;
138 -               end
139 -           end
140 -           llr_array(:,iter_idx) = APP;
141 -           check_out             = mod(H*tmp_dec_out,2);
142 -           check_out_sum         = sum(check_out);
143 -       end
144 -       dec_out(:,grp_idx)        = tmp_dec_out;
145 -   end
146
147 -   check = abs(dec_out(1:n,:) - enc_out(1:n,1:dec_grp));
148 -   sum(sum(check))
```

这里也留下一道思考题，大家有兴趣可以思考一下：

1、 采用双曲正切的译码方式，目前代码中使用 5 次迭代；如果把迭代次数增加到 16，运行代码会出现怎样的情况？出现异常又该如何解决呢？

DAY3 至此结束，祝大家晚安。

# DAY4：用 layer-decoding 来实现 LDPC 译码

从 DAY1 到 DAY3，我们已经可以写一个简易的分组 LDPC 编译码浮点代码。但是这种浮点代码和具体的芯片实现相比，在实现架构上还是有差距的。我们再来仔细观察一下传统的 BP 算法流程，需要将所有码字的 V 值更新完成后，才能计算 C 值，同样地，需要将所有 C 值计算完成后，才能更新 APP 值，从原理上来说当然是这样没错。但这样做也有不好的地方，那就是芯片存储资源，会变得非常非常大。毫无疑问我们要想办法尽量减小芯片的存储资源，鉴于此，我们可以根据校验矩阵的特点，在硬件实现上做一些资源简化。

首先来看 H 矩阵，H 矩阵如果平均分为 3 大行，5 大列的话，我们就可以看出每个小块都可以写成两种形式：

$$H=\begin{bmatrix}0&1&0&1&0&0&0&1&0&0&0&0&0&0&0\\0&0&1&0&1&0&0&0&1&0&0&0&0&0&0\\1&0&0&0&0&1&1&0&0&0&0&0&0&0&0\\0&0&0&1&0&0&1&0&0&1&0&0&0&0&0\\0&0&0&0&1&0&0&1&0&0&1&0&0&0&0\\0&0&0&0&0&1&0&0&1&0&0&1&0&0&0\\1&0&0&0&0&0&0&0&0&1&0&0&1&0&0\\0&1&0&0&0&0&0&0&0&0&1&0&0&1&0\\0&0&1&0&0&0&0&0&0&0&0&1&0&0&1\end{bmatrix}$$

单位矩阵的循环移位，或者全 0 矩阵。

既然这样，我们就可以用一个新的矩阵 Hb 和 Z 来表示校验矩阵 H：

$$H_b=\begin{bmatrix}1&0&1&-1&-1\\-1&0&0&0&-1\\0&-1&-1&0&0\end{bmatrix},\ \text{Z=3}$$

Hb 里面每个元素，对应 H 的每个 3×3 的方阵（所以令 Z=3）。Hb 中的 1 代表单位矩阵向右循环移移 1 位，0 代表单位矩阵（就是不移位），-1 代表全 0 矩阵。这样通过 Hb 和 Z，我们就能完整表达校验矩阵。

现在有 Hb 了，我们再来看看如何节省资源，将 H 矩阵分拆成 3×3 方阵，并且都可以表示为单位矩阵的循环移位，好处是什么？

**——好处是做校验的时候，我们可以只对码字做循环移位就可以了，这种操作在硬件上很容易实现啊！**

来来来举个例子。Hb(1,1)是 1，表示的矩阵是：

$$\begin{bmatrix}0&1&0\\0&0&1\\1&0&0\end{bmatrix}\cdot\begin{bmatrix}c1\\c2\\c3\end{bmatrix}=\begin{bmatrix}c2\\c3\\c1\end{bmatrix}$$

就相当于将码字做一个循环移位。那这样校验方程按行就可以分为 3 层（layer），我们可以逐层使用 BP 算法，这样不但方便做循环移位，也节省了很多资源。

大家也可以想一想，如果我们把资源节省到极致，按照校验矩阵的每一行来做 BP 算法，那就没办法在芯片上写成循环移位的形式，反而会增加额外的资源。

所以说，按层来做 BP 算法，在芯片实现上最节省，同时也是最规则的，这种译码方式，叫做 layer decoding。

$$c2+c4+c8=0$$
$$c3+c5+c9=0$$
$$c1+c6+c7=0$$
$$c4+c7+c10=0$$
$$c5+c8+c11=0$$
$$c6+c9+c12=0$$
$$c1+c10+c13=0$$
$$c2+c11+c14=0$$
$$c3+c12+c15=0$$

那么，校验方程分成 3 个部分依次更新 BP 算法。

第一次是： $\begin{matrix} c2+c4+c8=0 \\ c3+c5+c9=0 \\ c1+c6+c7=0 \end{matrix}$，按照 step0~step2 的流程走一遍。

第二次是： $\begin{matrix} c4+c7+c10=0 \\ c5+c8+c11=0 \\ c6+c9+c12=0 \end{matrix}$，按照 step0~step2 的流程走一遍。

第三次是： $\begin{matrix} c1+c10+c13=0 \\ c2+c11+c14=0 \\ c3+c12+c15=0 \end{matrix}$，按照 step0~step2 的流程走一遍。

这样就完成了一次迭代。

接下来我们还是用代码来说明，首先仍然是定义区。

```
clear
clc
close all
rng(257);

%% setting aera
bit_grp      = 1000;
k            = 6;
m            = 9;
n            = k + m;
noise_scale = 2.4;% 调整噪声功率的比值，目的是让BPSK信号既出现纠前误码又不至于超出纠前门限
iter_num     = 5; % 迭代次数
dec_grp      = bit_grp;
dec_mode     = 0; % 0: atanh mode; 1: minsum mode.
u            = zeros(size(Hb,1)*length(Z),bit_grp);
dec_out      = zeros(n,dec_grp);% 硬判输出结果

Hb           = [1 0 1 -1 -1; -1 0 0 0 -1;0 -1 -1 0 0];
HIb          = Hb(:,1:2);
Z            = eye(3);
H            = zeros(size(Hb,1)*length(Z),size(Hb,2)*length(Z));
```

```
for idx_row = 1:size(Hb,1)
    for idx_col = 1:size(Hb,2)
        if(Hb(idx_row,idx_col)==-1)
            H((idx_row-1)*length(Z)+(1:length(Z)),(idx_col-1)*length(Z)+(1:length(Z))) = zeros(size(Z));
        else
            H((idx_row-1)*length(Z)+(1:length(Z)),(idx_col-1)*length(Z)+(1:length(Z))) = circshift(Z,[0 Hb(idx_row,idx_col)]);
        end
    end
end

HI = H(:,1:k);
HP = H(:,k+1:end);
```

然后是编码过程。这次我们换一种方式，用 Hb（只看信息位的话就是 Mb 这个矩阵）加循环移位来完成编码过程。看起来这种方式在 matlab 里面要写很长的代码，其实在芯片里面反而更加好实现。

```
%% create the bit data in
bit_in = randi([0 1],k*bit_grp,1);
bit_in = reshape(bit_in,k,[]);

%% use LDPC to encode
for grp_idx = 1:size(bit_in,2)
    Mb     = reshape(bit_in(:,grp_idx),length(Z),size(bit_in,1)/length(Z));
    tmp_u = zeros(size(Hb,1)*length(Z),2);
    for row_idx = 1:size(Mb,1)
        for col_idx = 1:size(Mb,2)
            if(HIb(row_idx,col_idx)==-1)
                tmp_u((1:length(Z))+length(Z)*(row_idx-1),col_idx) = zeros(length(Z),1);
            else
                tmp_u((1:length(Z))+length(Z)*(row_idx-1),col_idx) = circshift(Mb(:,col_idx),-HIb(row_idx,col_idx));
            end
        end
    end
    tmp_u         = mod(sum(tmp_u,2),2);
    u(:,grp_idx)  = tmp_u;
end
% u = mod(HI*bit_in,2); u的计算过程也可以简写为这个式子
p = mod(mod(inv(HP),2)*u,2);
% check code
% H_check = mod(HI*bit_in + HP*p,2);
enc_out = [bit_in;p];
```

然后是 map 映射，加噪声。

```

%% mapping and calc llr
llr_data = 2*double(~enc_out)-1;
% 因为llr=ln(APP(X=0)/APP(X=1))，这里为了简化llr的计算，所以对enc_out做了一个取反动作，然后直接映射成BPSK信号，作为LLR值。
%% add noise
noise     = randn(size(llr_data))+1j*randn(size(llr_data));
noise     = noise/noise_scale;
tmp_data = llr_data + noise;

scatterplot(tmp_data(:))
llr_data = real(llr_data + noise);
```

接下来是重头戏译码。注意到我们这次新增加了 row_idx 的循环，这个循环就是按层进行 BP 算法的操作，注意在计算的时候需要排除方阵为全零矩阵的情况。

```
%% decode
llr_data_in = llr_data(:,1:dec_grp);
llr_array   = zeros(size(llr_data_in,1),iter_num);

for grp_idx=1:dec_grp
    tmp_llr_data = llr_data_in(:,grp_idx);
    V            = zeros(size(Hb,1),3); %列为校验矩阵的最大行重3
    APP          = reshape(tmp_llr_data,length(Z),size(tmp_llr_data,1)/length(Z));
    tmp_dec_out  = zeros(15,1);
    C            = zeros(size(Hb,1),3);
    C_total      = repmat(C,length(Z),1);%每组数据独立进行，所以需要先将C_total清零
    for iter_idx=1:iter_num
        C = zeros(size(Hb,1),3);%每次迭代之前需要先将C清零
        % 更新V的概率信息
        for row_idx = 1:size(Hb,1)% 按Hb逐行更新
            V      = zeros(size(Hb,1),3); %列为校验矩阵的最大行重3，因为是不规则的ldpc校验矩阵，所以在每次使用V之前都要先清0一把
            V_idx = 0;
             %% 按Hb的层将APP的值进行循环移位，送给V矩阵
            for col_idx=1:size(Hb,2)
                if(Hb(row_idx,col_idx)~=-1)
                    V_idx       = V_idx + 1;
                    V(:,V_idx) = circshift(APP(:,col_idx),-Hb(row_idx,col_idx));
                    % 在V矩阵里面减去上一次迭代送过来的C矩阵
                    V(:,V_idx) = V(:,V_idx) - C_total((1:length(Z))+length(Z)*(row_idx-1),V_idx);
                else
                end
            end
```

接下来也是逐层更新 C 和 APP。每更新一次 C 矩阵，就会把这一层的 C 矩阵存入 C_total 矩阵中，方便下次迭代的时候减去相应信息。

```
        %% 按Hb的层计算C矩阵
        if(dec_mode==0)
            C(:,1) = 2*atanh(tanh(V(:,2)/2).*tanh(V(:,3)/2));
            C(:,2) = 2*atanh(tanh(V(:,1)/2).*tanh(V(:,3)/2));
            C(:,3) = 2*atanh(tanh(V(:,1)/2).*tanh(V(:,2)/2));
        else
            C(:,1) = sign(V(:,2)).*sign(V(:,3)).*min(abs(V(:,2)),abs(V(:,3)));
            C(:,2) = sign(V(:,1)).*sign(V(:,3)).*min(abs(V(:,1)),abs(V(:,3)));
            C(:,3) = sign(V(:,1)).*sign(V(:,2)).*min(abs(V(:,1)),abs(V(:,2)));
        end

        C_total((1:length(Z))+length(Z)*(row_idx-1),:) = C;

        %% 将计算出来的C矩阵回加到V矩阵上
        tmp_V = V + C;
        %% 在V矩阵上按照Hb的行进行循环反移位
        % 找出该row_idx下Hb不为-1的列数。
        return_col_idx = find(Hb(row_idx,:)~=-1);
        V_idx          = 0;
        for col_idx = return_col_idx
            V_idx      = V_idx + 1;
            V(:,V_idx) = circshift(tmp_V(:,V_idx),Hb(row_idx,col_idx));
        end

        %% 将更新后的V放回llr的ram APP上，对应回填
        APP(:,return_col_idx) = V;
    end
```

最后仍然是硬判和输出。

```
        serial_APP = APP(:);
        for idx=1:length(serial_APP)
            if(serial_APP(idx)>=0)
                tmp_dec_out(idx) = 0;
            else
                tmp_dec_out(idx) = 1;
            end
        end
        llr_array(:,iter_idx) = serial_APP;
    end

    %% 按照Hb逐行更新完成后，对APP进行硬判
    check_out          = mod(H*tmp_dec_out,2);
    check_out_sum      = sum(check_out);
    dec_out(:,grp_idx) = tmp_dec_out;

end
check = abs(dec_out(1:n,:) - enc_out(1:n,1:dec_grp));
sum(sum(check))
```

到这里，我们就将编译码的简易（但是完整）的流程讲了一遍。根据 LDPC 校验矩阵自身的特点，我们可以采用 layer decoding 的方式做硬件处理上的简化，虽然多花了一些时间，但是节省了中间存储的资源。考虑到实际使用的 LDPC 矩阵的行数都是以千计，就可以看出 layer decoding 的价值。DAY3 和 DAY4 的代码我都有提供，大家也可以自行运行修改调试。这里还是留一个思考题：

Layer decoding 和普通的 BP 算法迭代译码，在中间过程的输出 bit 上，是否能做到完全等效？

最后再补充一点，之前我们都没有讨论这个 H 矩阵是如何得来的。似乎我们随随便便地设计很多校验方程拼凑在一起，都可以达到纠错目的。——感觉 FEC 的实现也很好做嘛，对么？

不是的，仔细深入下去，就会发现很多问题？比如：

校验方程的个数应该如何选取？

参与校验方程的数据 bit 应该如何分布？信息位和校验位的比例怎样才合适？

每种确定的校验方程，纠前门限到底能到多少？

校验方程应该运行在伽罗华域还是伽罗华域的扩域上，这两者有什么区别？

这里面每个问题，如果认真去研究，都可以花费数年时光。当然在这篇文档里面我们不会去探讨这么深入的问题，大家有兴趣可以自行去找 FEC 的相关教材来学。

DAY4 至此结束，祝大家晚安。