LDPC 码编译码程序设计

学号：1901213306 姓名：杨庆龙

要求

编写(2016, 1008)LDPC 码的编码器与译码器，并搭建仿真系统统计误码性能。系统框图如图 Fig1 所示

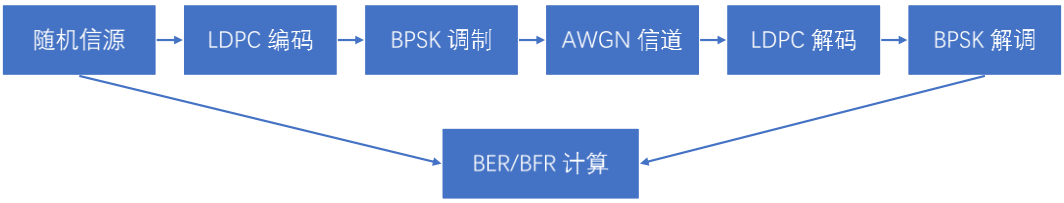

Fig1. 仿真系统框图

设计内容

该实验涉及的设计内容有以下几项，LDPC 码校验矩阵，随机信源，LDPC 编码器，BPSK 调制解调器，AWGN 信道，LDPC 码译码器。其中译码器共需要四种，分别为和积算法译码器，最小和算法译码器，归一化最小和算法译码器和偏置最小和算法译码器。

编解码原理

编码原理

本系统采用码长为 2016 比特，1/2 码率的系统码，其 H 矩阵如表达式如下所示.

$$H=\begin{bmatrix} H_{1,1} & H_{1,2} & \cdots & H_{1,n_b} \\ H_{2,1} & H_{2,2} & \cdots & H_{2,n_b} \\ \vdots & \vdots & \ddots & \vdots \\ H_{m_b,1} & H_{m_b,2} & \cdots & H_{m_b,n_b} \end{bmatrix}$$

对于系统码, H 矩阵可分为[H_p, H_s]两个部分, 其中, H_p 对应校验比特部分, 大小为 m_bz*m_bz ，H_s 对应信息比特部分，大小为 m_bz*k_bz，其中 $k_b=n_b-m_b$ 。H_p 具有如下所示的规则结构:

$$H_p=\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & 0 & a \\ 1 & 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 1 & 1 & & & 0 & 0 \\ \vdots & & \ddots & \ddots & & \vdots & \vdots \\ 0 & & & \ddots & 1 & 0 & 0 \\ 0 & 0 & & & 1 & 1 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 1 & 1 \end{bmatrix}$$

右上角 a 表示的矩阵的结构比较特殊，如下所示

$$\begin{bmatrix} 0 & 0 & \cdots & \cdots & 0 \\ 1 & 0 & \ddots & \ddots & \vdots \\ 0 & 1 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 & 0 \\ 0 & \cdots & 0 & 1 & 0 \end{bmatrix}$$

由于采用的是系统码，所以求出校验比特 p=[p_1,p_2,……p_{mbz}]之后，再与信息比特 s=[s_1,s_2,……p_{kbz}]拼接在一起即可得到编码结果。又考虑到校验矩阵结构特殊，所以采用如下所示的编码方法，以减小编码计算量。

步骤一：使用信息矢量 s，依据如下公式计算出中间结果 w

$$w = sH_s^T$$

步骤二：使用中间结果 w 计算出校验矢量 p，具体计算公式如下所示

$$p_i=\begin{cases} w_i & ,i=1 \\ w_i \oplus p_{(m_b-1)z+i-1} & ,1<i\le z \\ w_i \oplus p_{i-z} & ,i>z \end{cases}$$

解码原理

和积算法

初始化

每个比特从信道接收到的信道信息作为初始置信度，即变量节点的初始置信度 $u_{j->i}=u_{0->i}$，其中 $u_{0->i}$ 表示第 i 个变量节点从信道接收到的 LLR，如图 Fig2 所示。

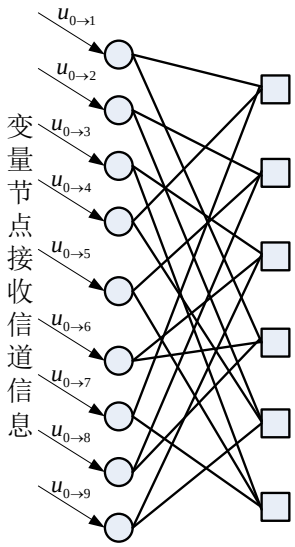

Fig2. 初始化节点图

置信度传递

每个变量节点要将自己的置信度传递给具有校验关系的校验节点。第 i 个变量节点传递给第 j 个校验节点的置信度计算方法如图 Fig3 所示，其中，d_i^v 表示第 i 个变量节点的度。

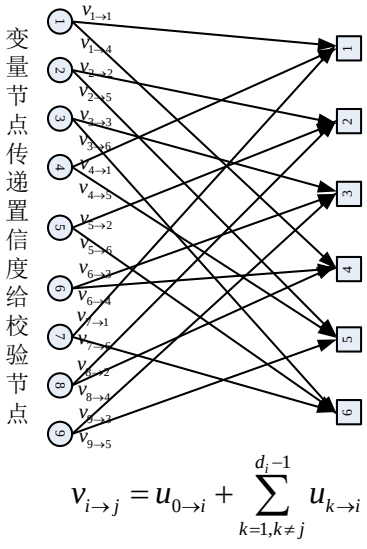

Fig3. 变量节点置信度传递图

校验节点依据变量节点传递来的置信度，更新每一个变量节点的置信度。第 j 个校验节点为第 i 个变量节点更新的置信度如图 Fig4 所示，其中 d_j^v 表示第 j 个校验节点的度。

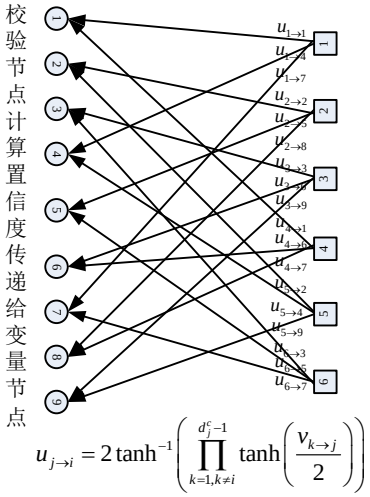

Fig4. 校验节点置信度传递图

判决

置信度更新完毕后依据如下关系进行硬判决。

$$\hat{x}_i=\begin{cases}1 & if \ \sum_{k=0}^{d_i^v}u_{k\to i}<0\\ 0 & if \ \sum_{k=0}^{d_i^v}u_{k\to i}\geq 0\end{cases}$$

如果所有的校验关系都得到满足，即满足公式 $Hx^t=0$。则译码结束，取出靠后的 k_{bz} 个比特即为信息位的估计值。若校验关系不满足，则返回迭代第二步继续迭代，直到迭代次数上限。

最小和算法

最小和算法的流程与和积算法完全一致，唯一的区别在于将和积算法的积运算简化为如下所示的最小和运算。

$$u_{j\to i}=\left(\prod_{k=1,k\neq i}^{d_j^c}sign\left(v_{k\to j}\right)\right)\left(\min_{k\in\{1,2\cdots d_j^c\}\backslash i}\left(\left|v_{k\to j}\right|\right)\right)$$

归一化最小和算法

归一化最小和算法建立在最小和算法之上，将较为粗糙的最小和算法更新为带有归一化系数α的最小和算法，具体公式如下所示，以减小更新幅度。

$$u_{j\to i}=\left(\prod_{k=1,k\neq i}^{d_j^c} sign\left(v_{k\to j}\right)\right)\left(\min_{k\in\{1,2\cdots d_j^c\}\backslash i}\left(\left|v_{k\to j}\right|\right)\right)*\alpha$$

偏置最小和算法

偏置最小和算法也是建立在最小和算法之上，在变量节点更新的置信度的基础上引入了偏置量β，具体公式如下所示

$$u_{j\to i}=\left(\prod_{k=1,k\neq i}^{d_j^c} sign\left(v_{k\to j}\right)\right)\left(max[\min_{k\in\{1,2\cdots d_j^c\}\backslash i}\left(\left|v_{k\to j}\right|\right)-\beta,0]\right)$$

程序设计与实现

语言选择

由于个人对 Matlab 并不是很熟悉，考虑到助教说过用 matlab 需要跑一整天，所以用 Python 可能要一周。加之编解码过程有大量的并行计算，所以使用和 C 语言性能类似，但对并行化支持更好的 golang 作为开发语言。

随机信源

随机信源模块位于 source 包内，该模块会等概率地产生给定长度的随机 01 序列。当序列长度为 1e5 时，可保证 0 或 1 在整个序列内的占比在 49.9%至 50.1%波动，基本符合信源条件。

稀疏矩阵

golang 没有提供现成的矩阵运算包，所以需要自行开发。又考虑到校验矩阵比较稀疏，所以在不调用底层 CPU 指令集的情况下，将其作为稀疏矩阵进行存储和计算将能够极大地减小运算开支，节约时间。
综上，开发了 matrix 包，以方便后续开发。该包支持稀疏矩阵的建立，行遍历，列遍历以及

矩阵乘法。

LDPC 编码器

LDPC 编码器位于 coder 模块的 encoder 文件中。创建编码器时，需要传入校验矩阵 H。调用 Encode 函数，传入特定长度的 01 序列，依据实验要求里所给的 LDPC 码编码过程，递推地得到所需结果。

调制解调器

调制解调器主要用于将 01 比特序列依据 BPSK 调制规则进行调制，再将解码器解码完成的浮点序列依据 BPSK 的规则进行解调处理。
调制部分较为简单，将比特 0 映射为 1，比特 1 映射为-1 即可。
解调部分将输入浮点数序列中的负数输入映射为-1，其余均映射为 0 即可。

信道

信道位于 channel 模块中，db 与线性值的互换也由该包以包函数的形式提供。
使用 BPSK 调制的一大好处是，每个信号的功率都是 1，在加噪过程中可以免去测量的部分。
加噪具体使用 math 包中的 NorRand 函数，生成符合标准正态分布的随机变量，再将其乘上噪声的标准差，最后将结果加到相对应的 BPSK 序列上即可。

LDPC 解码器

解码器位于 coder 模块的 decoder 文件中。LDPC 码的解码过程是一个节点迭代更新的过程，所以定义了 Node 类以实现节点更新的过程。又由于共有四种解码算法，所以需要一个泛型接口以减少工作量。综上，又定义了 Recorder 接口以解决这个问题，并实现了四个用于校验节点的 Recorder 和一个用于值节点的 Recorder。此外，在进行节点更新时若使用值节点更新校验节点的方法，将会出现竞争等问题，若使用锁，则会导致性能出现明显损失。因此，采用校验节点从值节点读取数值的方式进行更新。

仿真结果及分析

NMS 算法的α值选取

在 Eb/N0 为 1db 的信道条件下，从 0 到 1.0 测试不同的α值，结果如图 Fig5 所示。从图中可以看到，α为 0.7 时，性能最优。

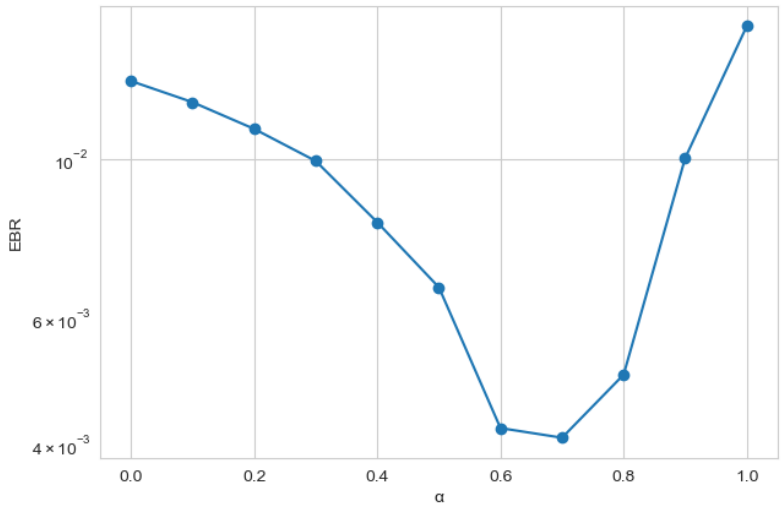

Fig5. NMS 算法不同 α 的纠错性能对比(Eb/N0=1db)

OMS 算法的β值选取

在 Eb/N0 为 1db 的信道条件下，从 0 到 1.0 测试不同的β值，结果如图 Fig6 所示。从图中可以看到，β为 0.5 时，性能最优。

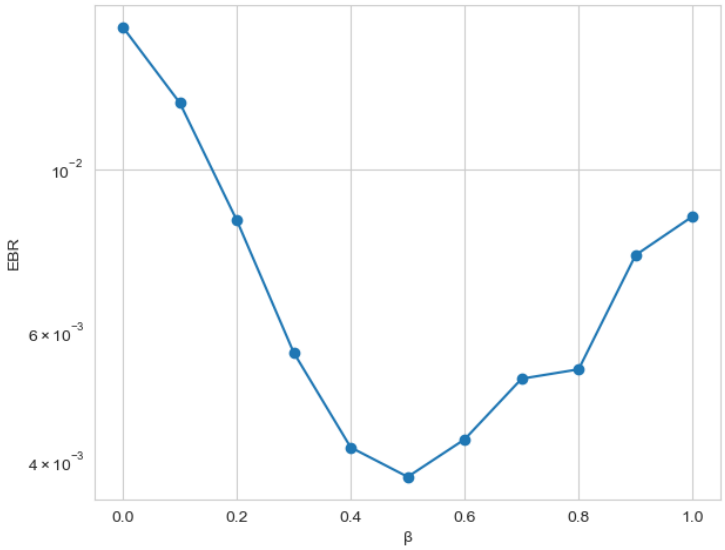

Fig6. OMS 算法不同 β 的纠错性能对比(Eb/N0=1db)

纠错性能分析

在[-1db, 2db]的信道条件下，运行四种解码算法，得到的结果如图 Fig7 所示。从图中可以看到，随着信道条件逐渐变好，四种算法的误码率也在逐渐降低。当 Eb/N0 大于 1db 后，SP 算法，NMS 算法和 OMS 算法的误码率更是显著下降。在 2db 的条件下，OMS 的 EBR 甚至优于 SP，这主要由于这两种译码算法均已基本不再出错，在这样的情况下，EBR 的值更

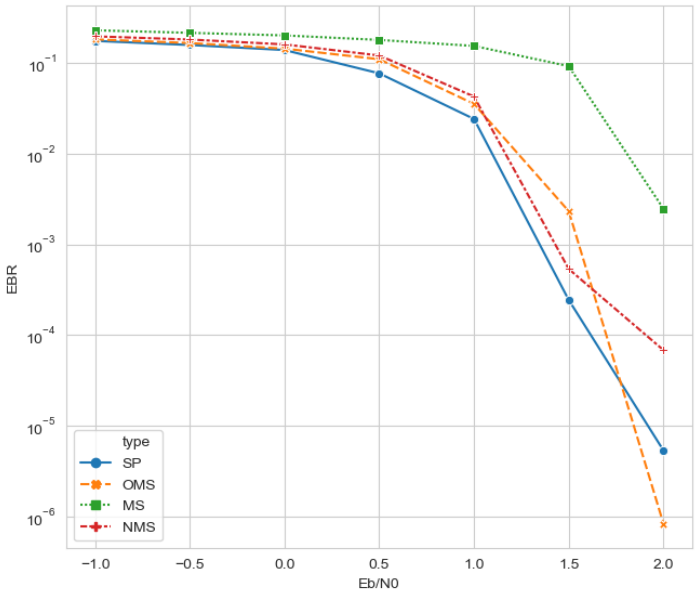

Fig7.四种算法在不同信道条件下的 EBR 对比

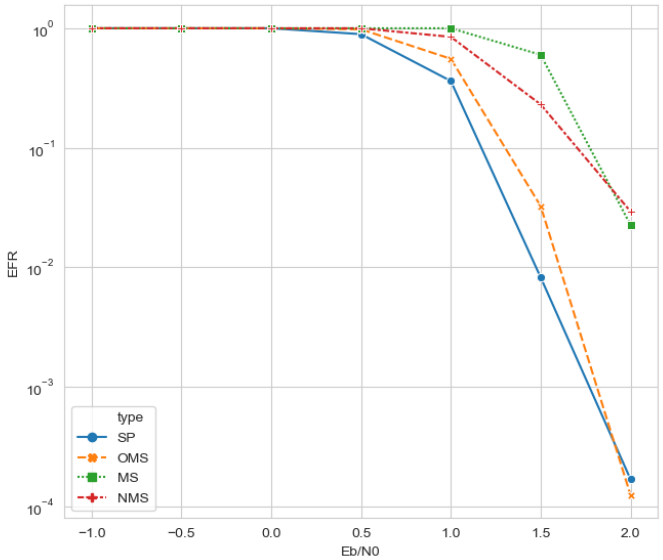

Fig8.四种算法在不同信道条件下的 EFR 对比

多是与测试使用的数据有关，与两者本身的纠错能力已经关系不大了。
对比来看，MS 算法的纠错性能最差，SP 算法的纠错性能稳定最优，OMS 和 NMS 在适当的参数选取下纠错性能也十分优秀。其中，OMS 算法更是与 SP 算法相差无几。考虑到 SP 算法需要计算较为复杂的三角函数，所以在实际的生产中，选用计算复杂度较低，而精度又没有太多损失的 OMS 算法作为解码算法也不失为一个不错的选择。

运行性能分析

使用 golang 编写的主要目的是为了节约运行时间，实际上，使用 golang 编写的编解码程序，在型号为 i5-8400F 的 CPU 上，运行所有的测试，共处理了 53770 个数据帧，用时 2652887ms 约为 44.2 分钟，比动辄几小时的 Matlab 略优。

项目链接

该大作业的代码与结果均已放到 Github 上，项目链接为 https://github.com/richsoap/Homeworks/tree/master/information/LDPC