实验四：卷积编码与Viterbi 译码

1. 实验目的：

理解信道编码解码，编写卷积编解码模块。在之前的实验的基础上，加上信道编解码模块，对整个链路进行编码仿真。用bertool生成未编码的以及编码的误码率信噪比曲线，分析编码增益。

1. 实验要求
2. 可支持的调制方式：BPSK,QPSK,16QAM.
3. 采用的信道编码：Convolutional Encoder/Viterbi Decoder
4. 所采用的码率：Coding Rate R=1/2
5. 卷积码编码方式：（2,1,7），R=1/2
6. 信道：AWGN

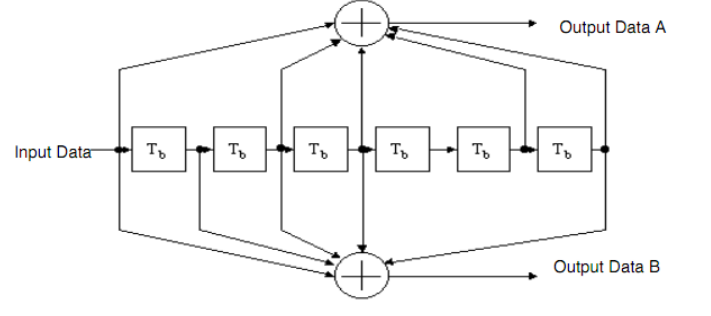


图1:卷积编码示意图，两位bit为133,171编码

6）信道译码（维特比译码原理图）

1. 实验过程
2. 编码函数的编写（Convolutional Encoder,用C++语言编写，然后利用mex将函数链接进入Matlab）

#include "mex.h"

//对输入数组进行编码，nd为输入数据

void convec(double \*input,double \*encoded,int nd)

{

double encoder[7] = { 0 };//编码器

for (int i = 0; i < nd; i++)

{

for (int j = 6; j >= 1; j--)encoder[j] = encoder[j - 1];//进行寄存器移位

encoder[0] = input[i];

double outA = (int)(encoder[0] + encoder[2] + encoder[3] + encoder[5] +

encoder[6]) % 2;

double outB = (int)(encoder[0] + encoder[1] + encoder[2] + encoder[3] +

encoder[6]) % 2;

encoded[2 \* i] = outA;

encoded[2 \* i + 1] = outB;

}

}

//C++到Matlab的接口封装函数

void mexFunction(int nlhs, mxArray \*plhs[], int nrhs, const mxArray \*prhs[])

{

int numofdata; //输入数组的大小

numofdata = mxGetN(prhs[0]);

plhs[0]=mxCreateDoubleMatrix(1,2\*numofdata,mxREAL);

double \*output=mxGetPr(plhs[0]);

double \*input=mxGetPr(prhs[0]);

convec(input,output,numofdata);

}

编码函数很简单，也很好写。出于简单化译码的考虑，没有将最后的1位输入bit进行移位编码至寄存器最后一位，故而会在后面的译码中产生错误。但是对于大量的数据而言，这12bit的编码值影响不大。

1. Vterbi Decoder（维特比译码函数，用C++编写，然后用mex函数作为接口，在Matlab中对数据做操作，此处采用的是寄存器交换法以及滑动窗译码的思想）

①根据维特比译码的原理，由于约束长度为K,总共有2^K-1个状态，对于（2,1,7）编码，总共有64个状态。状态之间的转换以及寄存器的交换如下图所示：

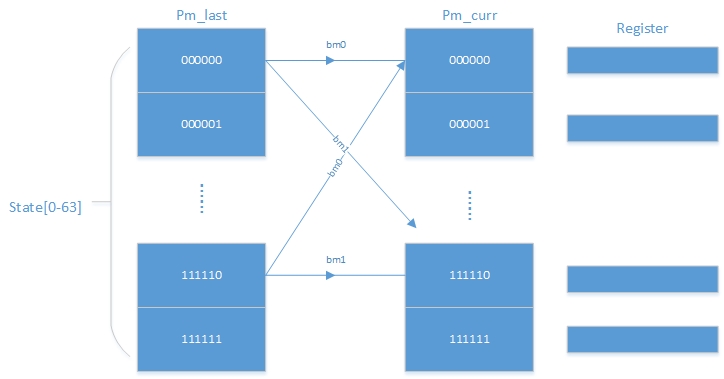


图2：状态迁移图

在寄存器交换时，每个状态需要知道其来源于哪个状态，并且需要知道其来源状态的路径度量与分支度量的和的较小者。，然后在两者之间选择较小者的寄存器，进行寄存器交换，再对当前状态进行译码。

②数据准备阶段：expect\_out[64][2],next\_state[64][2],regpos[64][2]

根据译码原理，每次译码都需要知道上一状态的期望转换状态，即输入0-1之后的状态，保存在next\_state数组中；同时，当前状态也需要知道此状态的来源状态，从而才能实现pm的比较以及寄存器的转换，用regpos数组记录；最后，也需要知道每个状态的期望输出，即输入0和1的时候的编码码值，这样才能通过与receive的信号做比较，得到分支度量，即码重。因此，在进行译码之前，写了一个prepare函数进行数据准备。函数代码如下：

void prepare(int expect\_out[][4], int next\_state[][2],int regpos[][2])

{

for (int i = 0; i < 64; i++)

{

int encoder\_1[7], encoder\_2[7];

int st\_1[6], st\_2[6];

//将状态数转换为二进制，保存在1-6为

dect2bin(i, encoder\_1); dect2bin(i, encoder\_2);

//编码器的首位输入，0位bit

encoder\_1[0] = 0; encoder\_2[0] = 1;

//每个状态的期望编码bit，也即期望输出bit

expect\_out[i][0] =(int) (encoder\_1[0] + encoder\_1[2] + encoder\_1[3] + encoder\_1[5]

+ encoder\_1[6]) % 2;

expect\_out[i][1] =(int) (encoder\_1[0] + encoder\_1[1] + encoder\_1[2] + encoder\_1[3]

+ encoder\_1[6]) % 2;

expect\_out[i][2] =(int) (encoder\_2[0] + encoder\_2[2] + encoder\_2[3] + encoder\_2[5]

+ encoder\_2[6]) % 2;

expect\_out[i][3] =(int) (encoder\_2[0] + encoder\_2[1] + encoder\_2[2] + encoder\_2[3]

+ encoder\_2[6]) % 2;

//当前状态的后一周期的分支状态，对应于输入为0,1的状态，此状态对应于

输入为0,1的分支度量

int state\_0, state\_1;

for (int m = 0; m < 6; m++)

{

st\_1[m] = encoder\_1[m];

st\_2[m] = encoder\_2[m];

}

state\_0 = bin2dect(st\_1, 6); state\_1 = bin2dect(st\_2, 6);

next\_state[i][0] = state\_0; next\_state[i][1] = state\_1;

}

//记录下一状态的寄存器的来源状态

for (int i = 0; i<state; i++)

{

int st= 0;

for (int j = 0; j<state; j++)

{

if (next\_state[j][0] == i)

{

regpos[i][0] = j; break;

}

if (next\_state[j][1] == i)

{

regpos[i][0] = j; break;

}

st += 1;

}

for (int j = st + 1; j<state; j++)

{

if (next\_state[j][0] == i)

{

regpos[i][1] = j; break;

}

if (next\_state[j][1] == i)

{

regpos[i][1] = j; break;

}

}

}

}

经过测试，expect\_out是正确的，next\_state数组的每个状态也是正确的，以及regpo

保存的也都是每个状态的正确来源状态。

③译码阶段：利用decode函数进行解码

关键变量的定义

int receive[2] = { 0 }; //存储接收的2bit信息

int bm[2][2] = { 0 }; //保存状态的期望分支度量

int pm\_last[64] = { 0 }; //保存每个状态上一时刻幸存路径的度量

int pm\_curr[64] = { 0 }; //保存每个状态当前时刻的幸存路径的度量

int reg[64][L] = { 0 }; //判决bit存储寄存器

int regtmp[64][L] = { 0 }; //寄存器中间变量，用于交换存储

int expect\_out[64][4] = { 0 }; //用于存储期望输出，0-1是输入0的期望输出,

2-3是输入1时的期望输出

int next\_state[64][2] = { 0 }; //每个状态的期望输出，0对应输入为1时的下一

状态，1对应于输入为1时的下一状态

int regpos[64][2] = { 0 }; //记录寄存器交换的对应关系

int tic\_now = 0; //当前状态的时钟数

int cnt = 0; //统计译码的bit数

int outcount = 0; //统计译码输出的bit数

具体代码见vitdecode.cpp文件，用到了很多工具函数，如二进制转十进制(bin2dect)，十进制转二进制(dect2bin)，指数运算(pow)，以及最小路径查找函数（findmin）均写在文件前面部分。另附Visual Studio 2013下的C++程序测试结果（数据量为1000,2000，10000,20000时均测过）：



图3：C++测试译码函数的结果

1. 实验结果：

BPSK：

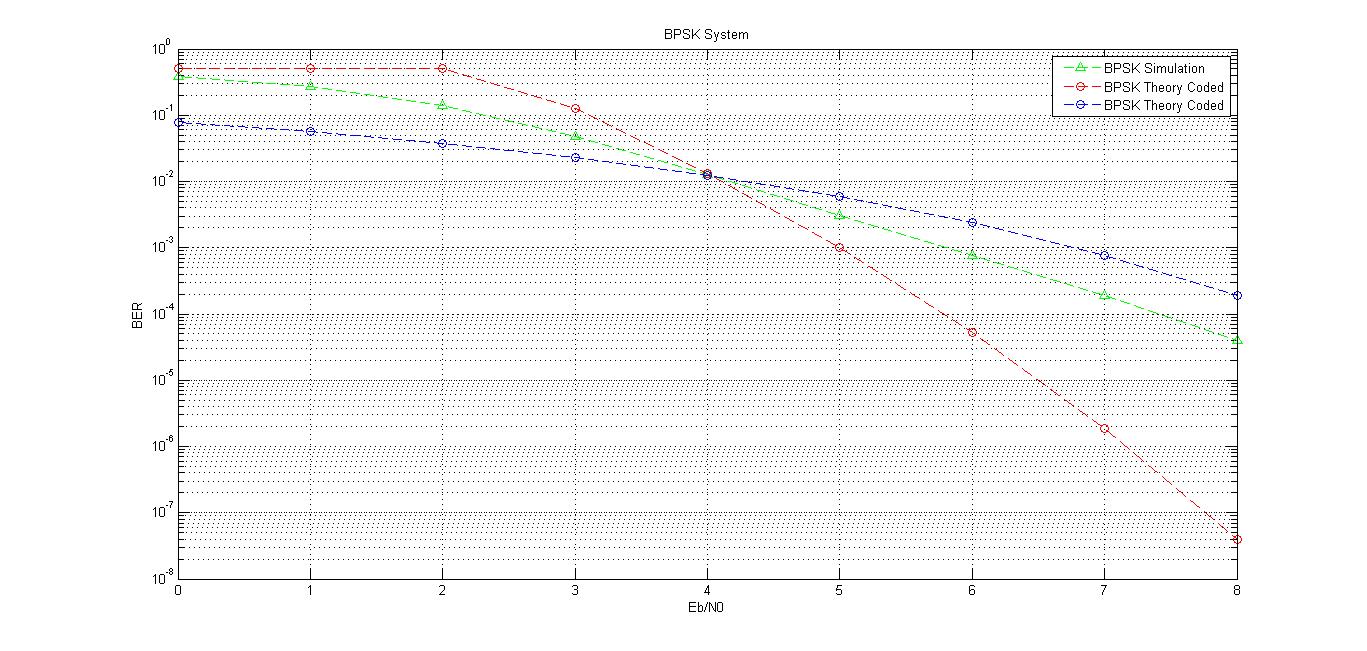


图4: BPSK的编码理论曲线（红色），未编码理论曲线（蓝色），

模拟仿真曲线（绿色）

QPSK:

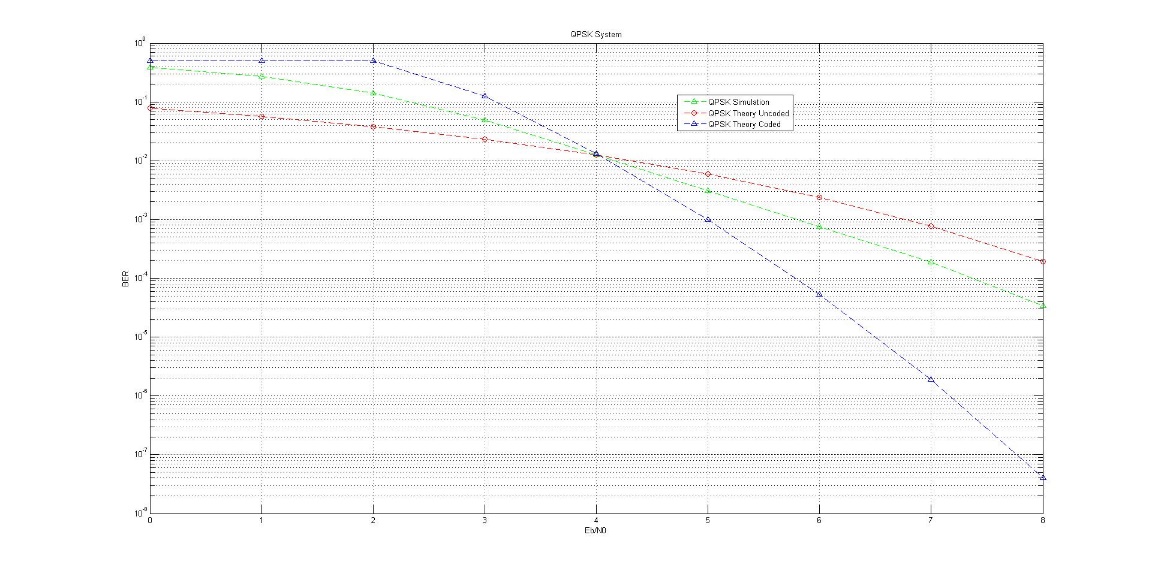


图:5：QPSK的编码理论曲线（蓝色），未编码理论曲线（红色），

模拟仿真曲线（绿色）

16QAM:

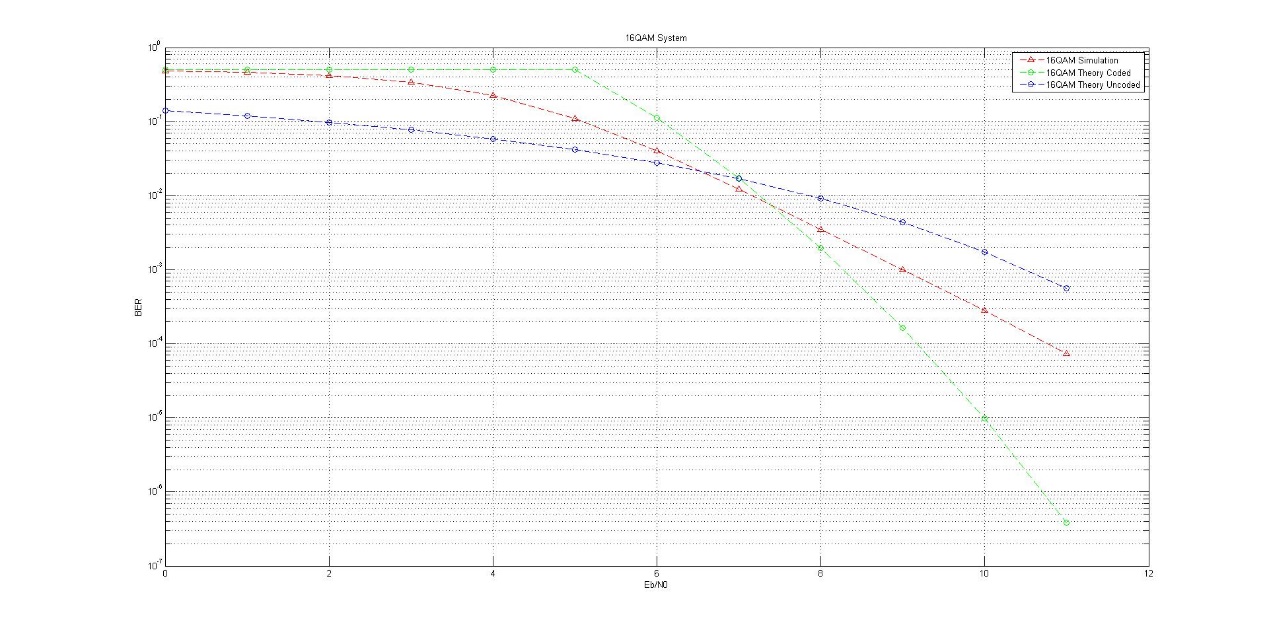


图6：16QAM的编码理论曲线（绿色），未编码理论曲线（蓝色），

模拟仿真曲线（红色）

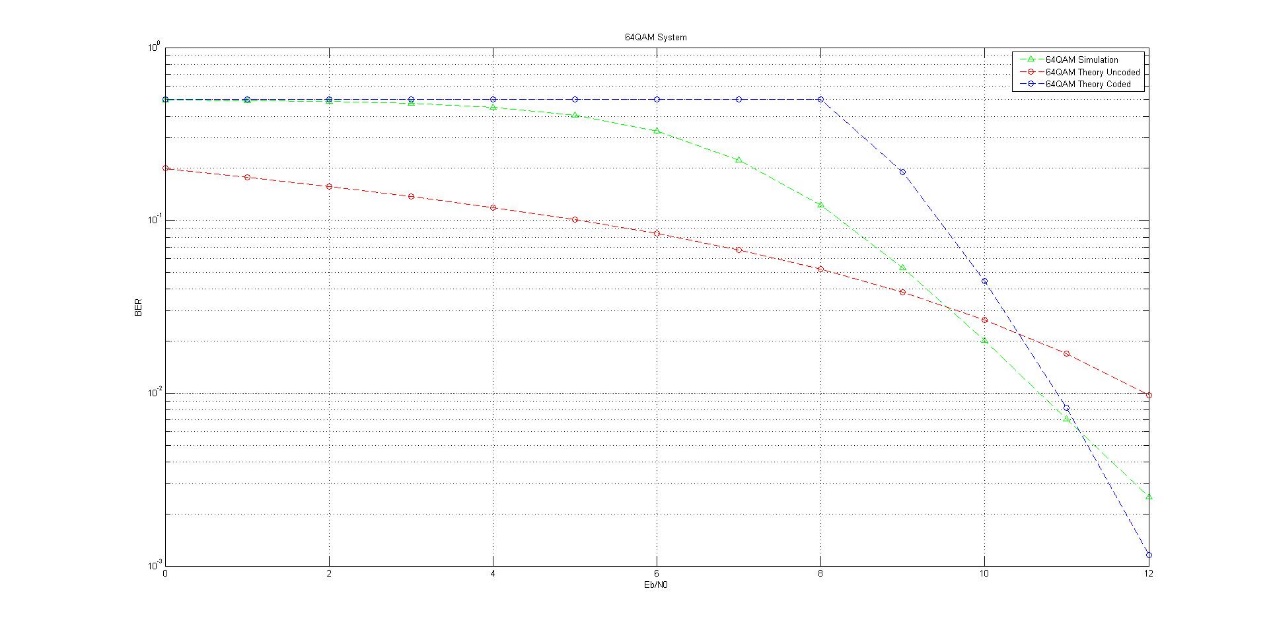
64QAM:

图6：64QAM的编码理论曲线（蓝色），未编码理论曲线（红色），

模拟仿真曲线（绿色）（Eb/N0=0:12）

64QAM:

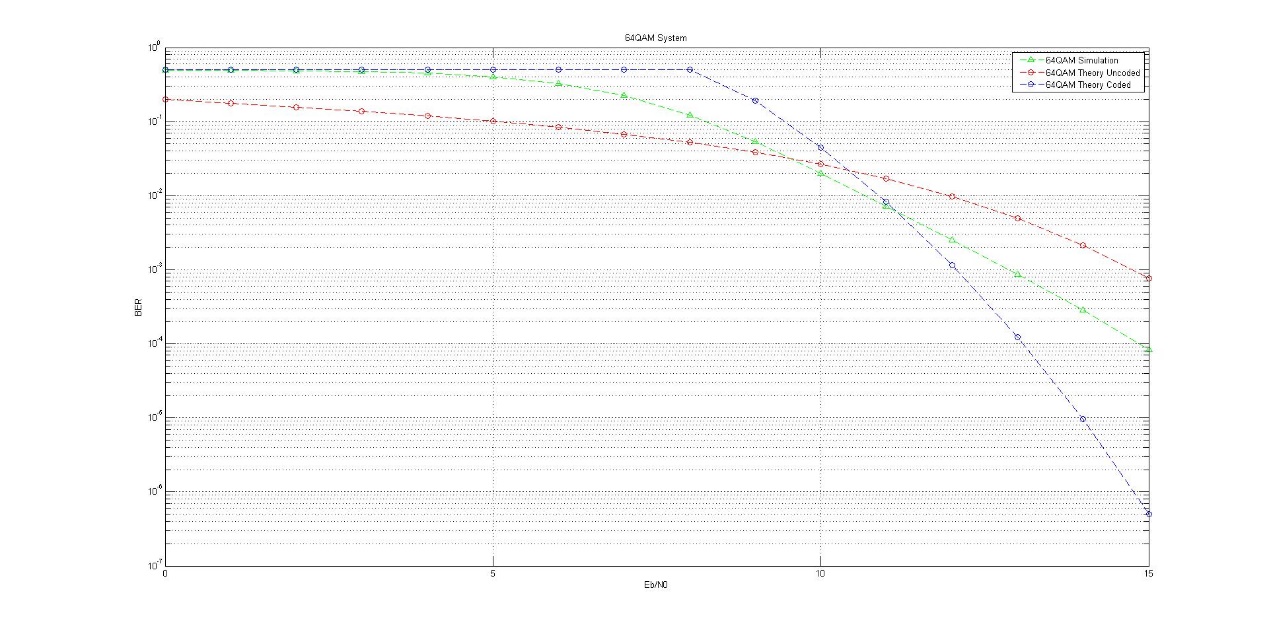


图6：64QAM的编码理论曲线（蓝色），未编码理论曲线（红色），

模拟仿真曲线（绿色）（Eb/N0=0:15）

1. 实验结果分析：

|  |  |  |
| --- | --- | --- |
| 调制方式 | 理论编码增益（Pb=10-3） | 模拟编码增益（Pb=10-3） |
| BPSK | 1.8dB | 1dB |
| QPSK | 1.8dB | 1dB |
| 16QAM | 1.9dB | 0.8dB |
| 64QAM | 1.5dB(Pb=10-2) | 1.7dB(Pb=10-2) |
| 64QAM | 3dB | 2dB |

表1：不同的调制方式下的编码增益

PS:由于64QAM调制在Eb/N0=12的时候尚未达到Pb=10-3而是10-2，同时为了取

Pb=10-3,因此取Eb/N0=15最大，从而得到编码增益，特此注明

1. 实验总结：

此次实验实现了信道的卷积编码以及维特比译码，同时通过对不同的调制方式下的编码增益的比较，可以知道信道编码解码确实可以在相同的信噪比下实现更低的误比特率。因此，在信息传输的过程中，加入信道编码，确实是很有好处的。

此次实验对C++有较高的要求，然而这些只是基础的东西，最关键的是对译码原理的理解。此次实验，由于自己一开始对译码的原理理解错误，每次寄存器交换的选择不是选pm+bm中的较小者，而是选每个状态的两个bm中的较小者，因此译出的结果一直不对。在与王敏与程仲麒同学交流之后，才发现自己的错误，将其改正之后很快就译出正确的结果了。此次实验，让自己体会到交流的重要性，还有做事情之前的思考的重要性。每次实验之前，都需要进行深入的思考，得到正确的逻辑，所谓磨刀不误砍柴工。

1. 思考题：

汉明距离的数值随着译码过程的进行而不断变大。那么对于流式的数据输入，用于保存汉明距离的数据单元终究会面对所保存值溢出的问题。请问这一问题是否会影响译码过程及结果，又如何解决或回避

答：由于此处用的是滑动窗加寄存器交换法译码算法，每次只保存窗长的PM，而且汉明距都是选取相对最小的值，是有限的，因而不会产生因为数据过大而溢出的问题。