LAB3 卷积码编解码实验

王清训 519021910593

2022 年 5 月 25 日

目录

1 实验目标 **1**

2 实验环境 **1**

3 实验结果 **2**

3.1 卷积码编码 2

3.1.1 实现细节 2

3.1.2 验证结果 3

3.2 Viterbi 译码 3

3.2.1 实现细节 3

3.2.2 补充说明 6

3.2.3 验证结果 6

3.3 Viterbi 译码仿真结果与分析 10

4 思考题 **16**

4.1 凿孔带来了部分监督比特的缺失，在使用 Viterbi 译码时该如何调整译码流程 (如何处理凿孔位置的分支度量)? 16

4.2 在低信噪比区域，为何未编码系统的 BER 性能更好? 16

5 实验总结与思考 **16**

1 实验目标

理解信道编码与凿孔在通信系统中的作用，掌握卷积码编码原理以及中最大似然译码流程；编写（2，1，7）卷积码编码模块与 ViterBi 译码模块（支持凿孔）；完成整个链路的仿真以及相关性能分析。

2 实验环境

对于编码功能的实现由于比较简单，因此在 matlab 中直接实现，不会对运行速度造成影响，但是由于 Viterbi 译码是一个严格顺序执行的过程，2160 的序列长度导致了在 matlab 中运行起

来速度很慢，严重影响实验进程，因此经过与助教的交流选择用 C 语言实现译码模块，并利用 mex 转化编译，使其能够在 matlab 中作为一个函数使用。

3 实验结果

卷积码编解码实验结果如下：

3.1 卷积码编码

首先是编码实现与验证。

3.1.1 实现细节

IEEE 802.11n协议中规定卷积编码使用的八进制生成多项式为 $[133 \quad 171]_O$ ，即(2,1,7)卷积码：

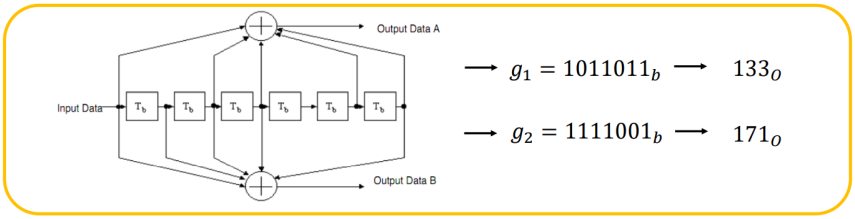

图 1: 编码原理

根据上图给出的编码原理，可以根据输入是 0 或 1 和当前状态得到下一个状态和输出，即可以构建基于输入的状态转移矩阵和输出矩阵。而在 matlab 中示例代码中已然包括了这两个矩阵 ($TxRx.Code.trellis.nextStates$ 和 $TxRx.Code.trellis.outputs$)，其中 $TxRx.Code.trellis.outputs$ 中以十进制保存输出，因此为了转化为编码序列，要将 0/1/2/3 分别转化为 00/01/10/11，关键代码列出如下：

```
if(in == 1)
    out = trenlis.outputs(reg+1,2);
    reg = trenlis.nextStates(reg+1,2);
else
    out = trenlis.outputs(reg+1,1);
    reg = trenlis.nextStates(reg+1,1);
end
%将输出表中的0，1，2，3转化成二进制
if(out == 0)
    coded = [coded,[0 0]];
elseif(out == 1)
    coded = [coded,[0 1]];
elseif(out == 2)
    coded = [coded,[1 0]];
else
    coded = [coded,[1 1]];
end
```

图 2: 编码关键代码

编码前要将所有寄存器清零；编码最后要在序列尾部最后补上 K-1 个 0，确保信息序列尾部也能完全移出寄存器。另外在编码后通过凿孔提高码率，降低发射功耗，以上部分示例代码中都

已包括，不做赘述。

3.1.2 验证结果

```
%% -- encoding
[info_bits, tx_bits,tx_bits_my] = CC_encoder(TxRx); % tx_bits_my是自己写的函数的编码结果，tx_bits是自带函数编码结果
encoder_error = encoder_error + sum(tx_bits~=tx_bits_my);%记录自己写的函数和自带函数编码不同的总位数
```

图 3: 验证代码

将 encoder 模块进行改写，令输出参数包括系统自带函数的编码结果 tx_bits 和我自己实现的函数的编码结果 tx_bits_my。用 $encoder\_error$ 变量记录几百次循环中所有编码不一致的 bit 数量

```
947/10000 packets and 2045520 bits passed;
Configuration: modulated by 4-QAM, CC decoder is Hard-Viterbi decoder, Current BER is:
Hard-Viterbi:
   3.6736e-01   3.1449e-01   2.4858e-01   1.7850e-01   1.1241e-01   6.3868e-02   3.2769e-02   1.3255e-02   5.5179e-03   2.1146e-03   5.7003e-04

>> encoder_error

encoder_error =

     0
```

图 4: 验证结果

经过 947 次循环，947 次编码的结果均与系统自带函数完全一致，验证了编码功能的正确性。

3.2 Viterbi 译码

接下来是 Viterbi 译码部分的实现与验证，我所实现的是硬编码。前文有提，由于运行速度的问题，最终译码用 C 语言实现，但是 matlab 写的代码也保存在 $decoder\_CC$ 中，分别选择 $myViterbi\_c$ 和 $myViterbi\_mt$ 两种译码方式即可分别选择 C 语言和 matlab 语言。

3.2.1 实现细节

加-比-选 迭代寻找最短路径：
初始化：令$\Gamma_0(0)=0$，其余$\Gamma_0(s')=-\infty, \forall s'\in\{1,\dots,63\}$ (编码前所有寄存器置为零状态)
for $l=1$ **to** L
1. 根据卷积码的状态转移图，计算所有可能的分支度量$\lambda_l(s',s)$
2. 对于第$(l-1)$时刻的每个状态s'，以及其所有可能抵达的第l时刻的状态s，计算路径度量 $\Gamma_l(s',s)=\Gamma_{l-1}(s')+\lambda_l(s',s)$ **//加**
3. 对于第l时刻的每个状态s，比较所有的$\Gamma_l(s',s)$得到幸存路径度量$\Gamma_l(s)$，即 $\Gamma_l(s)=\min_{s'}\Gamma_l(s',s)$ **//比**
并将这些幸存路径(状态和度量)选出，保存在幸存路径矩阵中 **//选**
end for
回溯译码：
由于信息序列末尾补零，末尾时刻寄存器强制为零状态，最大似然路径是第L时刻状态为0的幸存路径。从该路径的第L时刻回溯至第1时刻，即可得到最大似然码字

图 5: Viterbi 译码伪代码

Viterbi 译码的流程总结而言就是加比选最后回溯译码，伪代码如上图，接下来分别介绍各个环节的实现细节。

```
switch (output0)   //计算路径度量（=分支度量+先前状态的路径度量）
{
case 0:
    transfer[x+nextstate0*64] = (int)(!(rx_bits[l*2]==0)) + (int)(!(rx_bits[l*2+1]==0)) + F[x];
    input[x+nextstate0*64] = 0;
    break;
case 1:
    transfer[x+nextstate0*64] = (int)(!(rx_bits[l*2]==0)) + (int)(!(rx_bits[l*2+1]==1)) + F[x];
    input[x+nextstate0*64] = 0;
    break;
case 2:
    transfer[x+nextstate0*64] = (int)(!(rx_bits[l*2]==1)) + (int)(!(rx_bits[l*2+1]==0)) + F[x];
    input[x+nextstate0*64] = 0;
    break;
case 3:
    transfer[x+nextstate0*64] = (int)(!(rx_bits[l*2]==1)) + (int)(!(rx_bits[l*2+1]==1)) + F[x];
    input[x+nextstate0*64] = 0;
    break;
default:
transfer[x+nextstate0*64] = 100000;
    break;
}
if(l<2154)//最后六个input为0
    switch (output1)
    {
    case 0:
        transfer[x+nextstate1*64] = (int)(!(rx_bits[l*2]==0)) + (int)(!(rx_bits[l*2+1]==0)) + F[x];
        input[x+nextstate1*64] = 1;
        break;
    case 1:
        transfer[x+nextstate1*64] = (int)(!(rx_bits[l*2]==0)) + (int)(!(rx_bits[l*2+1]==1)) + F[x];
        input[x+nextstate1*64] = 1;
        break;
    case 2:
        transfer[x+nextstate1*64] = (int)(!(rx_bits[l*2]==1)) + (int)(!(rx_bits[l*2+1]==0)) + F[x];
        input[x+nextstate1*64] = 1;
        break;
    case 3:
        transfer[x+nextstate1*64] = (int)(!(rx_bits[l*2]==1)) + (int)(!(rx_bits[l*2+1]==1)) + F[x];
        input[x+nextstate1*64] = 1;
        break;
    default:
        transfer[x+nextstate1*64] = 100000;
        break;
    }
```

图 6: 加-代码

首先是加，根据输入是 0 或者 1，通过状态转移矩阵和输出矩阵找到对应的下一个状态和输出，将输出与输入序列对应位置 bit 比较，计算汉明距得到分支度量，然后与先前状态的路径度量 F(x) 相加的到新的路径度量暂存到 transfer 中。

```
for(j = 0; j<2 ;j++)
{
    y[j] = -100;
}
for(j = 0; j<64 ; j++)//找到所有指向此状态的上一状态（最多两个）
{
    if(count_y > 1) break;
    if(transfer[j+x*64] != 100000)
    {
        y[count_y] = j;
        count_y ++;
    }
}
if(count_y==1)
{
    state = y[0];
    q = transfer[state+x*64];
    b = input[state+x*64];
}
else if(count_y==2)//若有两条可行路径，比较路径度量并选择更小的一条路径
{
    int a1 = y[0];
    int a2 = y[1];
    int t1 = transfer[a1+64*x];
    int t2 = transfer[a2+64*x];
    if(t1 > t2)
    {
        q = t2;
        b = input[a2+64*x];
        state = a2;
    }
    else
    {
        q = t1;
        b = input[a1+64*x];
        state = a1;
    }
```

图 7: 比-代码

对于 64 个状态最多只能存在 64 条幸存路径，每条路径以某一个状态结束，每个路径会产生两条新的路径，对应到 64 个下一状态最多有两条新路径指向同一个状态，即比较最多是两条路径进行比较，因此在实现比较时，首先判断有几条路径指向了此状态，若只有一条那不必筛选，若有两条则选择路径度量更加小的那一条。

```
if(state>=0)//将幸存路径保存下来
{
    if(l == 0)
        {windows[tblen*x] = b;}
    else
    {
        windows[l+tblen*x] = b;
        memcpy(windows+x*tblen,windows_copy+state*tblen,l*sizeof(int));
    }
}
if(q<10000000)//更新以x状态为结尾的路径的度量
    F[x] = q;
}
```

图 8: 选-代码

选出路径度量更小的一条路径后，作为幸存路径更新其结束状态为当前状态，即以当前状态结束的路径更新为这条幸存路径。最后更新每个状态对应的路径度量 F(x)

最后一步回溯译码比较简单，在 2160 个 bit 全部处理完成后，由于最终结束状态一定为 000000，所以将最终结束状态为 000000 的路径拿出来就是译码序列。

3.2.2 补充说明

在第一版代码中，使用了 pdf 中要求的滑动窗，但是对于超出滑动窗长度的 bit 进行判决时使用的方法不唯一，因此最终的结果也不尽相同，最终我选择了当前路径度量最小的一条路径的第一个 bit 判决为正确输出 bit，但是误码率仍然高于系统自带函数，咨询助教过后，发现是系统自带函数的滑动窗实现方式和 pdf 中讲的不同，所以结果也不可能一致，因此得到助教同意后，最终没有使用 pdf 中要求的滑动窗，得到了很好的结果。

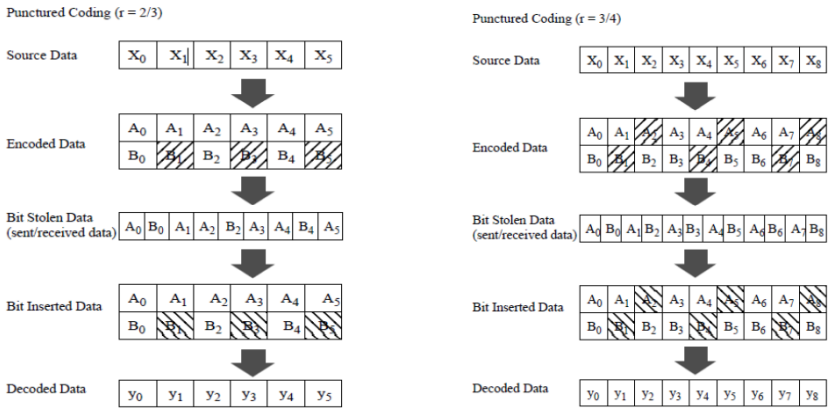

图 9: 凿孔

在译码中对于凿孔的处理与上图是结合起来的。若不凿孔编码序列长度 4320，输出序列长度 2160，凿孔之后，编码序列中少了一部分 bit，因此第一步是找到输出序列的每一位对应的是编码序列中的哪一位或者哪两位，具体做法是 $bit_out \times 2(+1) - bit_zaokong$，也就是先按照未凿孔找到对应编码序列中的 2bits，再减去这 2bits 前被凿掉的 bit 数就找到了真正对应的编码序列中的 bits，然后以 2/3 为例，若是输出序列中的奇数 bit 则只用第一位作汉明距，若是偶数位则与不凿孔相同。

3.2.3 验证结果

以 4-QAM 为调制方式分别对 1/2 码率、2/3 码率、3/4 码率的译码功能正确性进行验证。验证方式为与系统自带 $Hard\_Viterbi$ 的结果进行比较。首先将实现的函数与系统自带函数两者译码后得到的序列进行对比，统计不同 bit 的数量，越少说明译码效果越好。然后，分别用两个函数跑出 BER 曲线，对误码率进行比较，若曲线基本重合或者甚至比系统自带函数误码率更小说明译码效果很好。

```
if (TxRx.Code.Rate == 1/2) % no puncturing
        info_bits_hat1 = vitdec(rx_bits, TxRx.Code.trellis, TxRx.Code.tblen, 'term', 'hard');
else % punturing is required
        info_bits_hat1 = vitdec(rx_bits, TxRx.Code.trellis, TxRx.Code.tblen, 'term', 'hard', TxRx.Code.Puncturing.Pattern);
end
info_bits_hat1 = logical(info_bits_hat1);
info_bits_hat3 = Hard_Viterbi_wqx(rx_bits,TxRx.Code.trellis.nextStates,TxRx.Code.trellis.outputs,2160,[1 0]);

error = sum(info_bits_hat1~=info_bits_hat3)
```

图 10: 1/2 码率译码测试代码

```
>> test_12

error =

     0
```

图 11: 1/2 码率译码测试结果

```
988/10000 packets and 2134080 bits passed;
Configuration: modulated by 4-QAM, CC decoder is Hard-Viterbi myViterbi_c decoder, Current BER is:
Hard-Viterbi:
   3.7075e-01   3.1420e-01   2.5151e-01   1.8191e-01   1.1733e-01   6.5199e-02   3.2691e-02   1.4933e-02   5.9106e-03   1.6785e-03   5.3466e-04

myViterbi_c:
   3.6948e-01   3.1541e-01   2.5061e-01   1.8215e-01   1.1615e-01   6.4654e-02   3.2573e-02   1.4976e-02   5.8851e-03   1.6740e-03   5.3466e-04
```

图 12: 1/2 码率 BER 性能数据

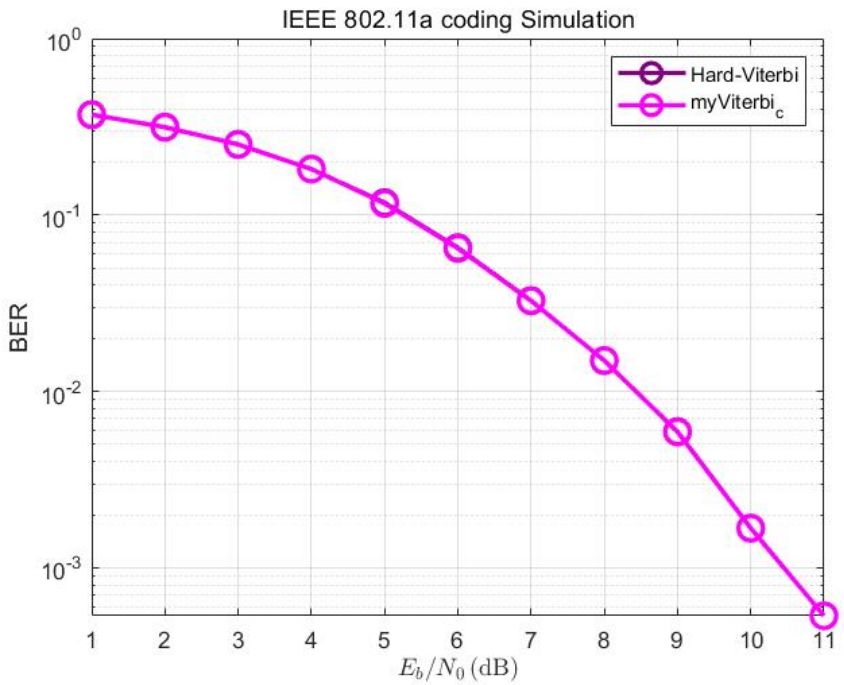

图 13: 1/2 码率 BER 性能曲线对比

```
if (TxRx.Code.Rate == 1/2) % no puncturing
        info_bits_hat1 = vitdec(rx_bits, TxRx.Code.trellis, TxRx.Code.tblen, 'term', 'hard');
else % punturing is required
        info_bits_hat1 = vitdec(rx_bits, TxRx.Code.trellis, TxRx.Code.tblen, 'term', 'hard', TxRx.Code.Puncturing.Pattern);
end
info_bits_hat1 = logical(info_bits_hat1);
puncturing_pattern = [1 1 0];
info_bits_hat3 = Hard_Viterbi_wqx(rx_bits,TxRx.Code.trellis.nextStates,TxRx.Code.trellis.outputs,2160,puncturing_pattern);
error = sum(info_bits_hat3~=info_bits_hat1)
```

图 14: 2/3 码率译码测试代码

```
error =

     1
```

图 15: 2/3 码率译码测试结果

```
614/10000 packets and 1326240 bits passed;
Configuration: modulated by 4-QAM, CC decoder is Hard-Viterbi myViterbi_c decoder, Current BER is:
Hard-Viterbi:
   4.2143e-01   3.8559e-01   3.3664e-01   2.7399e-01   2.0129e-01   1.3198e-01   7.5850e-02   3.7016e-02   1.4273e-02   5.4444e-03   1.7901e-03

myViterbi_c:
   4.2296e-01   3.8564e-01   3.3463e-01   2.6870e-01   1.9263e-01   1.2438e-01   6.7450e-02   3.2236e-02   1.1574e-02   4.6373e-03   1.4545e-03
```

图 16: 2/3 码率 BER 性能数据

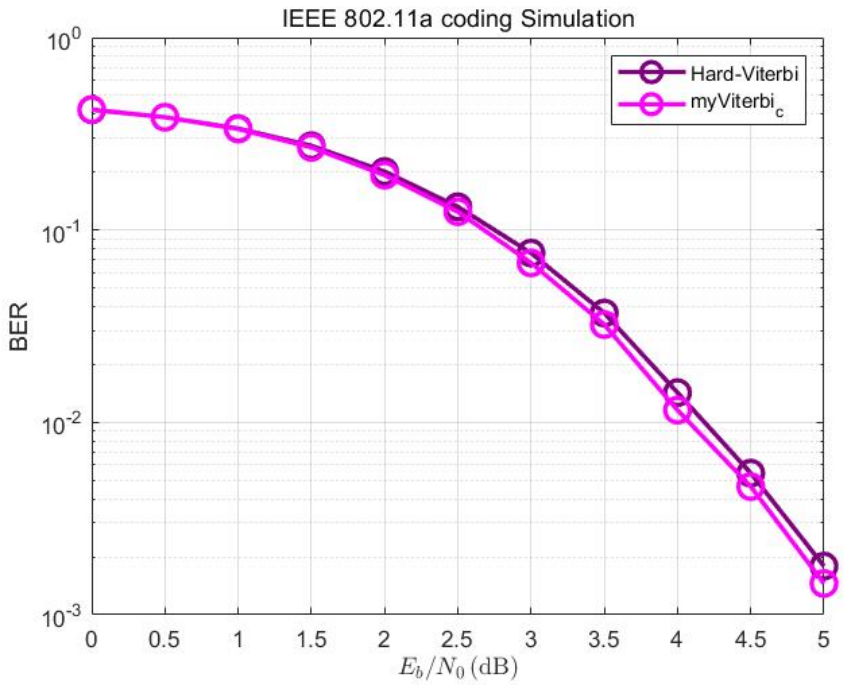

图 17: 2/3 码率 BER 性能曲线对比

```
if (TxRx.Code.Rate == 1/2) % no puncturing
       info_bits_hat1 = vitdec(rx_bits, TxRx.Code.trellis, TxRx.Code.tblen, 'term', 'hard');
else % punturing is required
       info_bits_hat1 = vitdec(rx_bits, TxRx.Code.trellis, TxRx.Code.tblen, 'term', 'hard', TxRx.Code.Puncturing.Pattern);
end
info_bits_hat1 = logical(info_bits_hat1);
puncturing_pattern = [1 1 1 0];
info_bits_hat3 = Hard_Viterbi_wqx(rx_bits,TxRx.Code.trellis.nextStates,TxRx.Code.trellis.outputs,2160,puncturing_pattern);
error = sum(info_bits_hat1~=info_bits_hat3)
```

图 18: 3/4 码率译码测试代码

```
error =

     0
```

图 19: 3/4 码率译码测试结果

```
355/10000 packets and 766800 bits passed;
Configuration: modulated by 4-QAM, CC decoder is Hard-Viterbi myViterbi_c decoder, Current BER is:
Hard-Viterbi:
   4.4987e-01   4.2323e-01   3.8415e-01   3.3107e-01   2.6556e-01   1.9098e-01   1.2050e-01   6.6971e-02   3.2707e-02   1.3481e-02   5.1161e-03

myViterbi_c:
   4.4871e-01   4.2491e-01   3.8382e-01   3.2860e-01   2.6217e-01   1.8747e-01   1.1662e-01   6.3069e-02   3.1121e-02   1.2378e-02   4.5696e-03
```

图 20: 3/4 码率 BER 性能数据

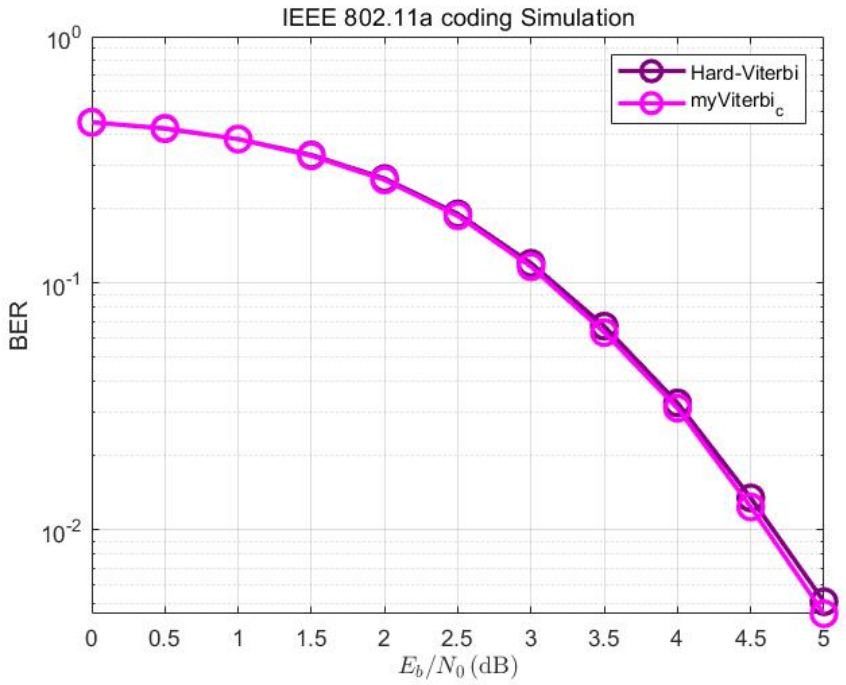

图 21: 3/4 码率 BER 性能曲线对比

可以看到译码结果不同 bit 的数目所有码率均基本为 0，2/3 码率存在一个 bit 误差其实是因为译码性能比系统自带函数更好，因此这个 1bit 的误差带来了更好的性能，从误码率中就可以看出，1/2 码率时，两条曲线完全重合，而 2/3 和 3/4 码率则是自己实现的译码函数性能优于系统自带函数。究其原因，应该是因为没有使用滑动窗的情况下相当于滑动窗长度为 2160，滑动窗越

长带来的是误码率的降低，系统自带函数也没有这么长的滑动窗，在 1/2 码率时，系统自带函数尚能保证优秀的译码，当码率升高，误码率随之升高，具有更大滑动窗的实现方式的性能自然就高于被滑动窗限制的系统自带函数。

3.3 Viterbi 译码仿真结果与分析

因为我所实现的是硬译码，因此分别对 1/2、2/3、3/4 码率的 Viterbi 硬译码进行了仿真，调制方式分别为 4-QAM、16-QAM、64-QAM，共九个对比图，对比了我自己实现的硬判决译码与系统自带硬判决译码的仿真性能。其中 4-QAM 以 0.5db 为间隔跑了 0-5db，其中 16-QAM 以 0.5db 为间隔跑了 0-7db，其中 64-QAM 以 0.5db 为间隔跑了 0-9db。另外，对于 1/2 码率还利用 BERTOOL 绘制了对应调制方式的性能上界渐进曲线。

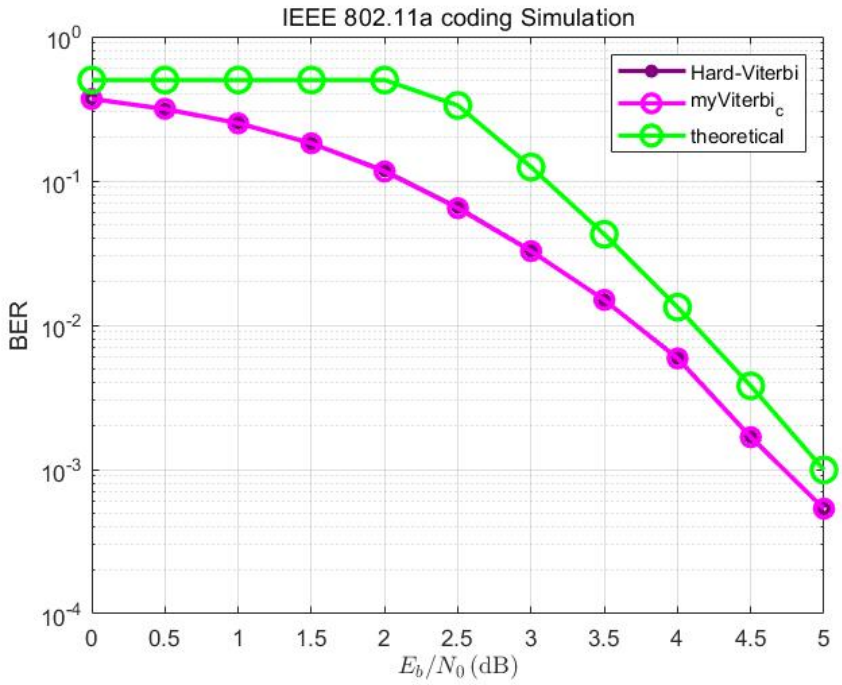

图 22: 1/2 码率 4-QAM Viterbi 译码仿真结果

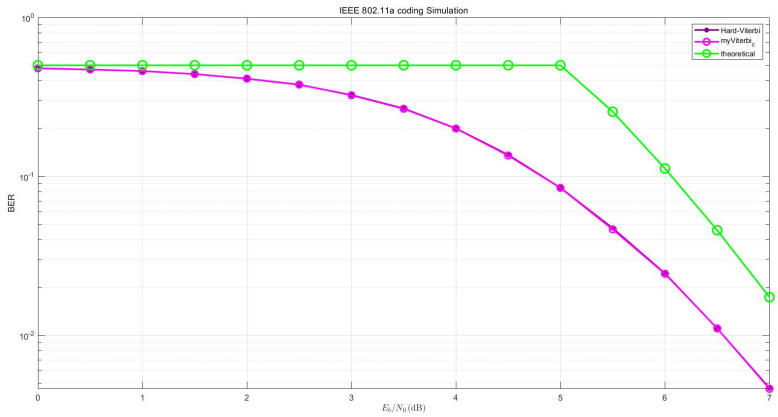

图 23: 1/2 码率 16-QAM Viterbi 译码仿真结果

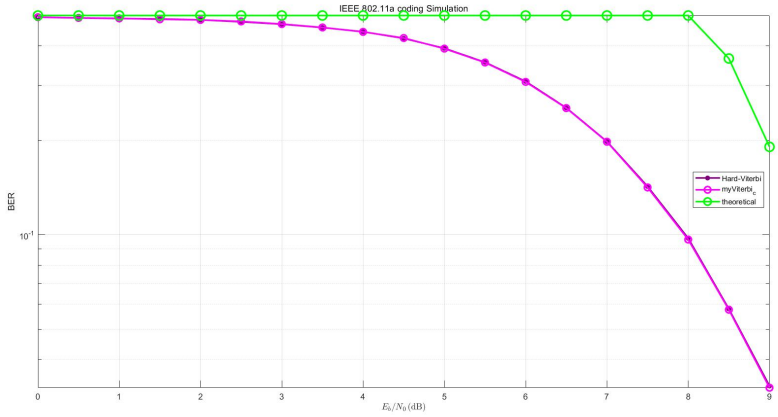

图 24: 1/2 码率 64-QAM Viterbi 译码仿真结果

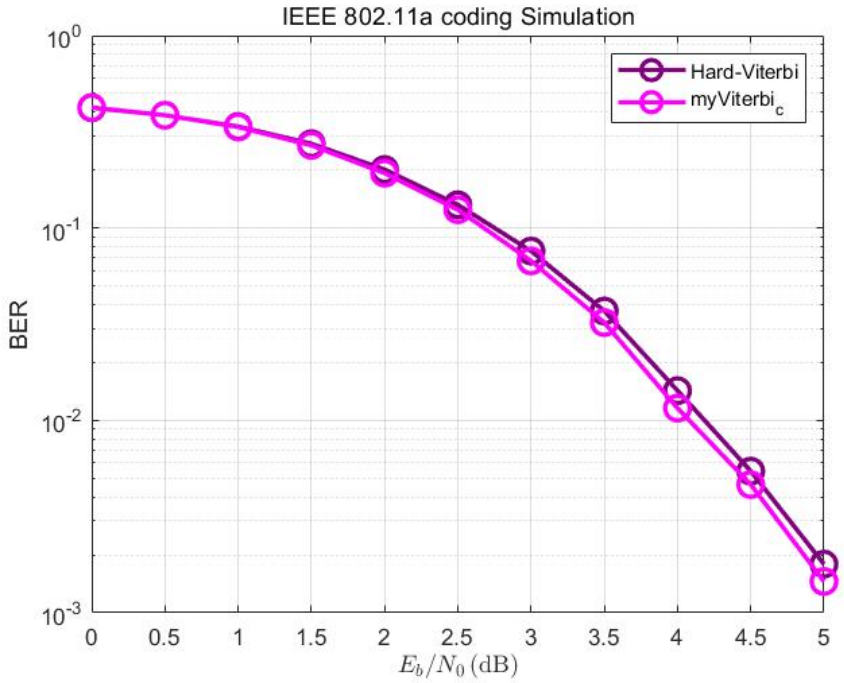

图 25: 2/3 码率 4-QAM Viterbi 译码仿真结果

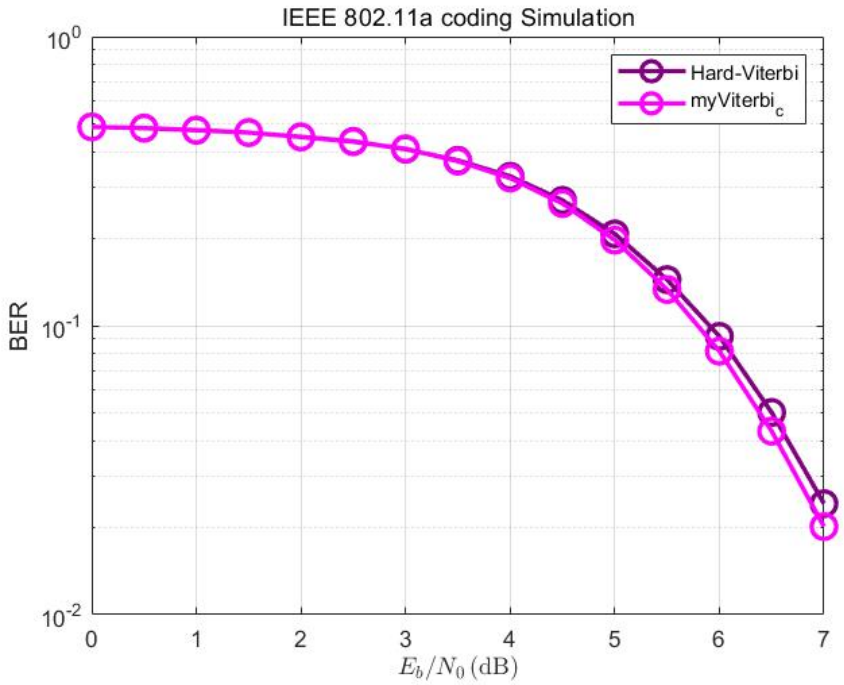

图 26: 2/3 码率 16-QAM Viterbi 译码仿真结果

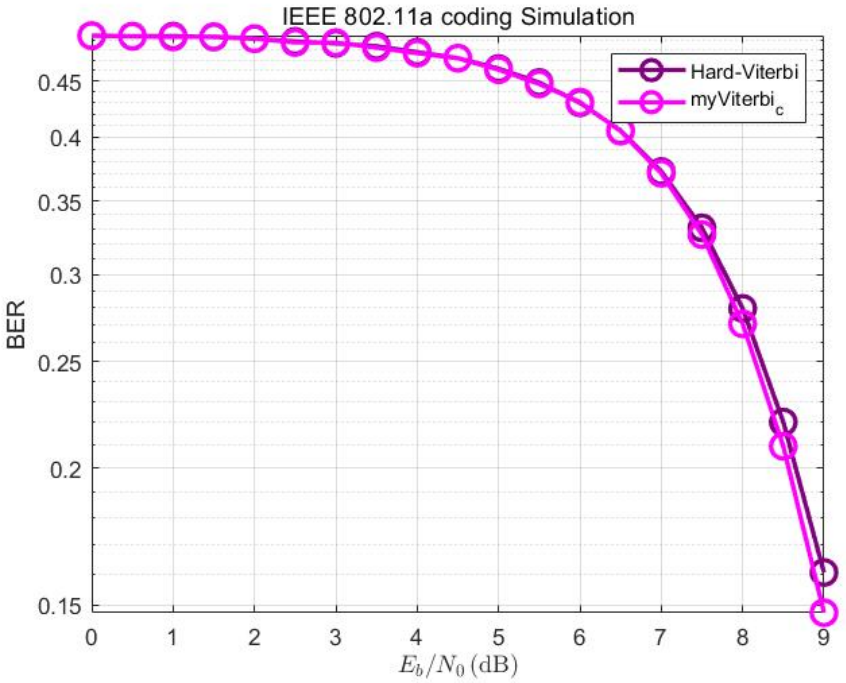

图 27: 2/3 码率 64-QAM Viterbi 译码仿真结果

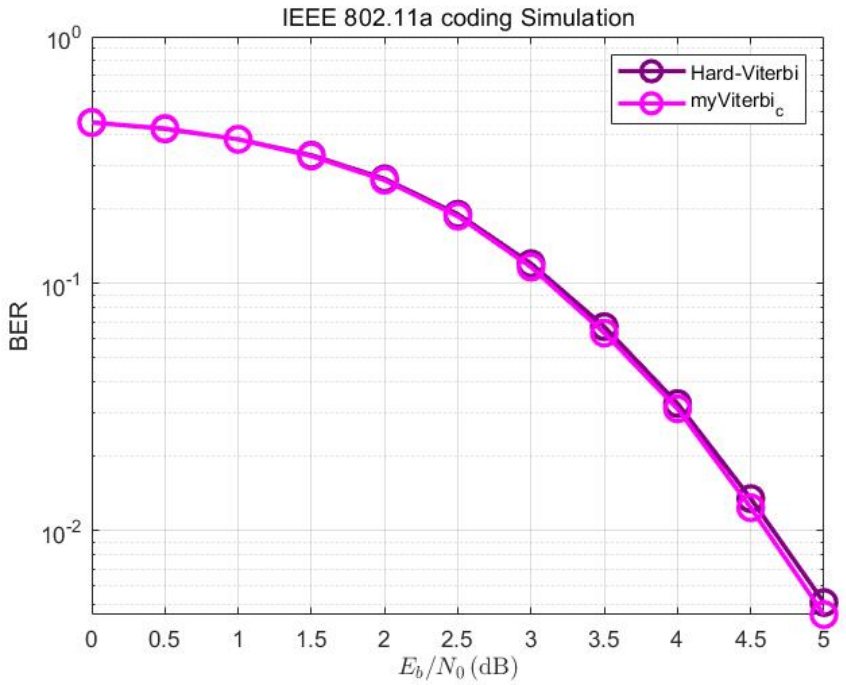

图 28: 3/4 码率 4-QAM Viterbi 译码仿真结果

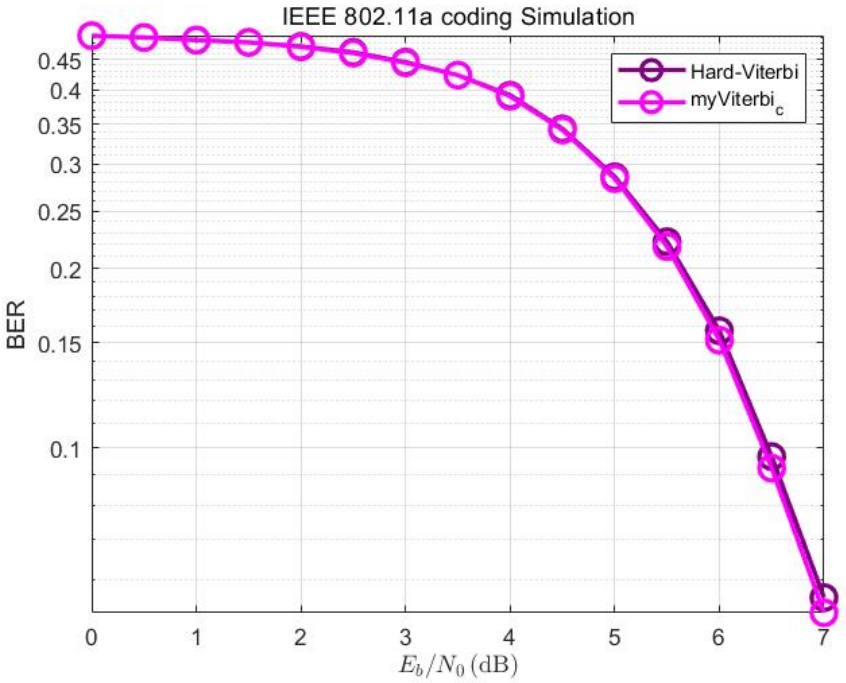

图 29: 3/4 码率 16-QAM Viterbi 译码仿真结果

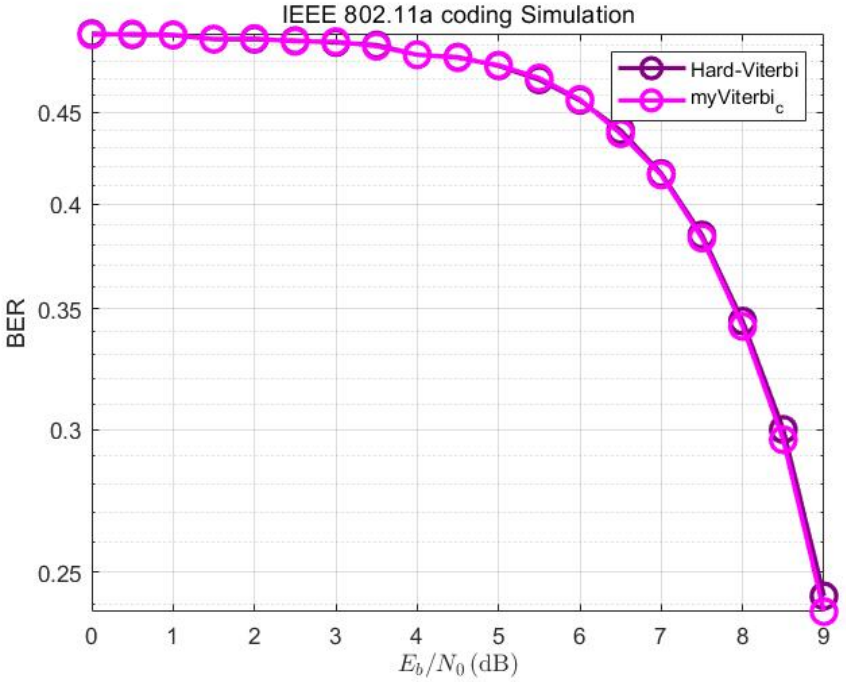

图 30: 3/4 码率 64-QAM Viterbi 译码仿真结果

可见，对于 1/2 码率，无论是怎样的调制方式，两条曲线均基本重合，且都在上界渐进曲线的下方，即都没有超过上界，进一步验证了功能的正确性。而对于 2/3 和 3/4 码率，自己实现的硬判决译码性能则均稍高于系统自带硬判决译码。原因在前文已经提过。

此外，虽然未对软判决进行实现，但仍然对软硬判决的 BER 性能曲线在 4-QAM 调制方式下进行了对比。

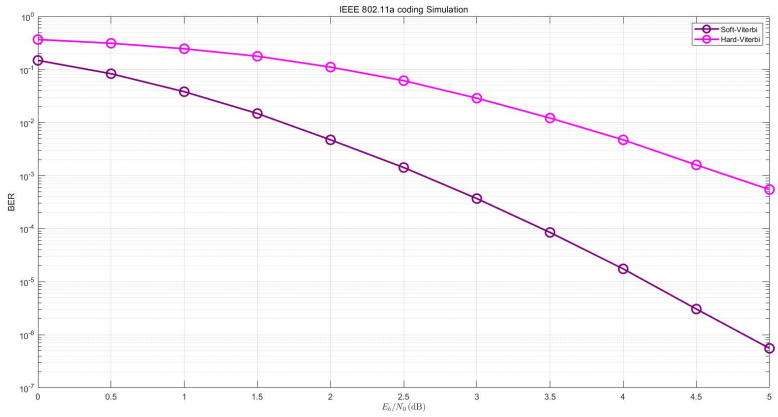

图 31: 1/2 码率软硬判决对比

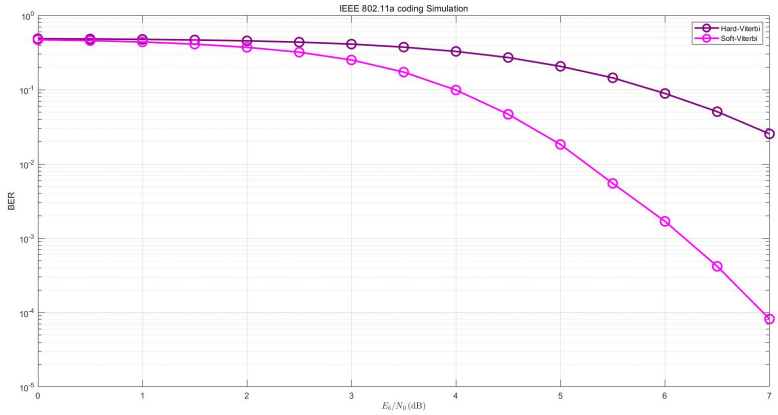

图 32: 2/3 码率软硬判决对比

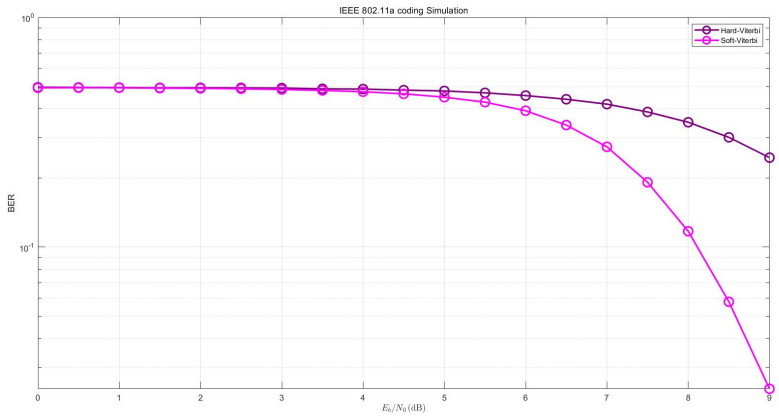

图 33: 3/4 码率软硬判决对比

可以看到 1/2 码率的软判决优于硬判决 2db 多，2/3 码率的软判决优于硬判决差不多刚好 2db，3/4 码率的软判决优于硬判决略小于 2db。

4 思考题

4.1 凿孔带来了部分监督比特的缺失，在使用 Viterbi 译码时该如何调整译码流程（如何处理凿孔位置的分支度量）？

在处理凿孔位置的时候，分支度量的计算方式发生了变化，而分支度量等于汉明距离（硬判决）和欧式距离（软判决），对于硬判决，原本每一个 bit 对应编码序列 2bits，即要计算 2bits 的汉明距离等于 $r_1 \oplus c_1 + r_2 \oplus c_2$，若存在凿孔，则可能这 2bits 中会被凿掉 1bit，以第二个 bit 被凿掉为例，此时汉明距离等于 $r_1 \oplus c_1$。对于软判决。原本每 1bit 对应编码序列 2 位，即要计算 2 位的欧式距离等于 $(r_1 - c_1)^2 + (r_2 - c_2)^2$，若存在凿孔，则可能这 2 位中会被凿掉 1 位，以第二个 bit 被凿掉为例，此时汉明距离等于 $(r_1 - c_1)^2$。

4.2 在低信噪比区域，为何未编码系统的 BER 性能更好？

编码的过程是给待编码的信息增加一些冗余位，以牺牲带宽提升纠错能力，获得编码增益。所以在信噪比较低时，接收信号冗余所引起的能量损失的负面影响超过了其纠错带来的好处，这是因为信噪比太低，所产生的误码率超出了纠错码的纠错能力（此时纠错码可能也会出错，且信噪比较低时容易出现连续误码，这对纠错码的纠错能力影响很大），所以越纠越错，导致误码率反而增大。

5 实验总结与思考

此次实验是大学以来耗时最长的一次实验，由于各种各样的原因导致中间出现了很多问题，首先对于 matlab 的不熟悉且执着于滑动窗导致第一版代码就花了两天时间，然后带着滑动窗及其造成的误差以及运行速度的下降，我将 matlab 代码花了一天转化为 C 语言，但是 bug 频出，经过一天调完 bug 后对性能不够满意促使我在得知可以不用滑动窗后将滑动窗给删掉了，然后在

确定了偶尔的 matlab 崩溃是电脑问题后终于一帆风顺完成了实验，此次实验令我印象深刻，最令我感触的一点是当跑一次代码的时间成本上去之后，容错率就会大大下降，比较马虎的代码导致了大量的时间消耗，这与集成电路的流片亦有相似，只有仔细与反复验证才能节省成本。