**目录**

[1. 系统设计 2](#_Toc61527549)

[1.1. 设计目标 2](#_Toc61527550)

[1.2. 设计方案 2](#_Toc61527551)

[2. 系统实现 2](#_Toc61527552)

[2.1. 信源 2](#_Toc61527553)

[2.2. 信道 7](#_Toc61527554)

[2.3. 信源编解码 10](#_Toc61527555)

[2.4. 信道编解码 18](#_Toc61527556)

[3. 系统仿真与分析 29](#_Toc61527557)

[3.1. 理想情况下 29](#_Toc61527558)

[3.2. 信源非理想情况下 32](#_Toc61527559)

[3.3. 信道非理想情况下 40](#_Toc61527560)

[3.4. 一般非理想情况下 50](#_Toc61527561)

[4. 系统改进【可选】 76](#_Toc61527562)

[5. 总结 77](#_Toc61527563)

[5.1. 设计完成情况 77](#_Toc61527564)

[5.2. 经验与收获 77](#_Toc61527565)

[6. 附录——相关技术简介 78](#_Toc61527566)

# 系统设计

## 设计目标

使用计算机编程语言，完整实现一个包含信源、信道、信源编解码、信道编解码等模块的信息传输系统模型，并通过与理论推导的结果进行比对，验证系统实现的正确性。再进一步通过多种参数组合下的仿真，模拟不同通信条件下的信息传递，掌握信息论与编码理论的理论和实践基础。

## 设计方案

整体系统的原理图如下：

信源

信源编码

信道

信源解码

信宿

信道编码

信道解码

噪声源

一个完整的系统，应该是先从信源产生信源序列，然后经过信源编码，接着进行信道编码，在进入有噪声源的信道，出来依次进行信道解码和信源解码，就是接收方得到的序列，即信宿。

# 系统实现

## 信源

### 二元离散无记忆信源（DMS）模块

程序API如下：

byteSource.exe INPUT LEN OUTPUT

– INPUT：信源消息概率分布P(0)

– LEN：消息序列的长度（字节为单位，一般至少10K字节长）

– OUTPUT：信源输出消息序列文件的路径

关键代码如下

def getInfoSouece(symbol\_prob, msgLength):

    '''生成指定大小的概率分布的数据

    通过读入指定概率分布以及信息源长度，通过蒙特卡罗法，

    生成符合输入条件的数据的数组返回

    Args:

        symbol\_prob (array): 指定字节符号的概率分布

        msgLength (int): 信息源的长度

    Returns:

        msg (array): 指定字节符号的概率分布的数据

    '''

    # 计算给定概率分布 symbol\_prob 的累积概率分布 symbol\_cumsum

    symbol\_cumsum = symbol\_prob.cumsum()

    # 生成 [0,1] 之间均匀分布的随机实数 symbol\_random

    symbol\_random = random.uniform(size=msgLength)

    # 输出符合条件的消息符号 symbol\_random,插入 symbol\_cumsum 指定位置

    msg = searchsorted(symbol\_cumsum, symbol\_random)

    return msg

### 信源指标计算模块

程序API如下：

byteSource\_calc.exe INPUT OUTPUT

– INPUT：信源输出消息序列文件的路径

– OUTPUT：存放计算结果的路径

每次运行byteSource\_calc.exe之后，将会在OUTPUT文件后添加计算结果（注意：不是覆盖），其中第一列为信源输出消息序列文件字节取值，第二列为字节概率分布，第三列为数据比特概率分布，第四列为二元DMS的信息熵，第五列为二元DSM的信源冗余度。

关键代码如下：

def genPDistribution(Input\_Data):

    '''由输入文件数据，得到两种概率分布

    Args:

        input\_Data (list): 输入数据

    Returns:

        P\_list (list): 字节符号的概率分布

        P0 (float): 数据比特概率分布

    '''

    # 获得字节符号的概率分布

    (hist, bin\_edges) = histogram(Input\_Data, bins=range(257))

    P = hist / Input\_Data.size

    P\_list = list(P)

    # 获得数据比特概率分布

    P0 = 1 - P\_list[-1]\*\*(1/8)

    return P\_list, P0

def genEntropy(P0):

    '''由数据比特概率分布计算信息熵以及信源冗余度

    Args:

        P0 (float): 数据比特概率分布

    Returns:

        I (float): 二元DMS的信息熵（信息比特/二元消息）

        R (float): 二元DSM的信源冗余度

    '''

    # 获得二元DMS的信息熵

    I = P0 \* log2(1 / P0) + (1 - P0) \* log2(1 / (1 - P0))

    # 获得二元DSM的信源冗余度

    Imax = log2(1 / 0.5)

    R = I / Imax

    return I, R

### 单元测试

* byteSource部分：

理论推导：

设信源消息概率分布，消息序列的长度为1MB，则，即二元离散无记忆信源生成0和1的概率相等。而每个字节相当于二元DMS的8次扩展，设该二元DMS的8次扩展信源为Y，则Y的概率空间为：



其中分别由8位二元离散无记忆信源符号（0和1）组成，取十进制值0~255；为对应的概率，设中二元离散无记忆信源符号0的个数为，则

故当二元离散无记忆信源生成0和1的概率相等时，其8次扩展信源的每一个值的取值概率都相等，即信源输出消息序列文件的字节概率是均匀分布的。

测试结果：

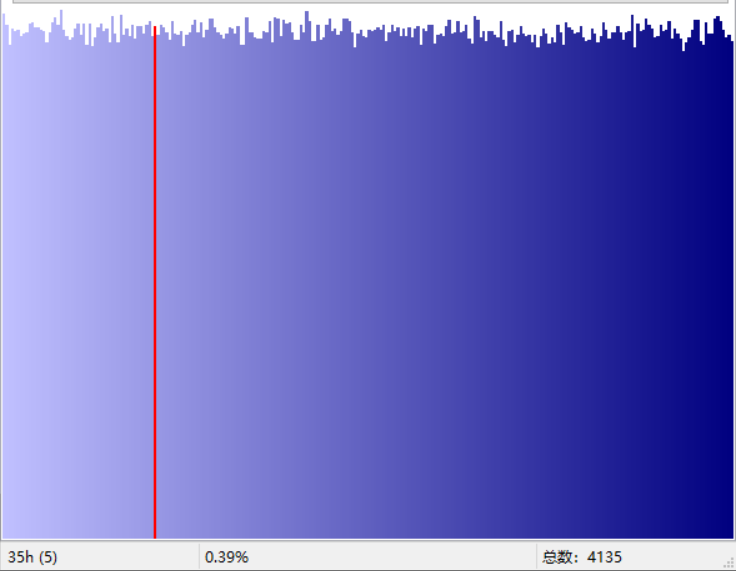
在cmd中输入命令：

python byteSource.py 0.5 1048576 unit-test\P0=0.5.LEN=1MB.OUTPUT.dat

生成的信源输出消息序列文件如下：



可以看到文件大小确实为1MB，在16进制文件编辑器HxD中打开该文件，使用统计功能显示字节概率分布如下：



可以看到字节概率分布基本均匀，均在0.39%~0.40%左右，与理论推导结果比较接近；而不能完全符合理论推导的均匀分布的原因是byteSource程序是根据蒙特卡罗法随机生成字节的，因此在一定文件大小下实际字节概率分布与理论会有微小的差距，而随着文件大小的增加，实际字节概率分布会趋于理论的均匀分布。

* byteSource\_calc部分：

理论推导：

根据byteSource部分设定的输入参数，信源指标理论计算结果应如下：

1. 数据比特概率分布（即二元DMS的概率分布统计）：
2. 字节概率分布（即二元DMS的8次扩展信源的概率分布统计）：
3. 二元DMS的信息熵：
4. 二元DSM的信源冗余度：，其中为信源的实际熵，则；为信源最大熵，此二元离散无记忆信源生成0和1的概率相等，故信息熵即为最大熵，，故

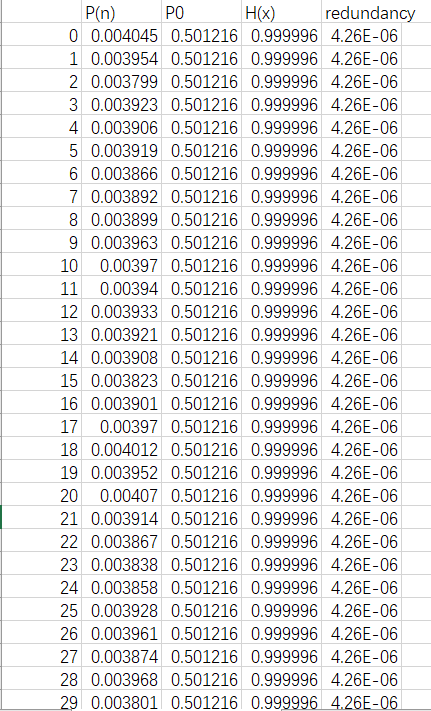
测试结果：

在cmd中输入命令：

python byteSource\_calc.py unit-test\P0=0.5.LEN=1MB.OUTPUT.dat unit-test\P0=0.5.LEN=1MB.calc.CSV

生成的信源指标计算结果如下：





其中第一列为信源输出消息序列文件字节取值，第二列为字节概率分布，第三列为数据比特概率分布，第四列为二元DMS的信息熵，第五列为二元DSM的信源冗余度。可以看到四个实际信源指标均与理论推导结果相接近，在一定误差允许范围内可以认为byteSource\_calc程序能正确计算信源指标，通过单元测试。

## 信道

### 二元对称信道（BSC）模块

程序API如下：

byteChannel.exe INPUT P OUTPUT

– INPUT：信道输入消息序列文件

– P：错误传递概率p

– OUTPUT：信道输出消息序列文件

关键代码如下：

def byteChannel(inputData, noiseData):

    '''将噪声作用于输入文件

    将噪声文件中的每一位与原文件对应位进行异或运算，

    从而得到被噪声作用过的输出文件

    Args:

        inputData (list): 输入文件数据

        noiseData (list): 噪声文件数据

    Returns:

        outputData (list): 输出文件数据

    '''

    # 输入文件数据列表长度

    msg\_len = len(inputData)

    # 生成按位异或后的列表

    outputData = [noiseData[i] ^ inputData[i] for i in range(msg\_len)]

    return outputData

### 信道指标计算模块

程序API如下：

byteChannel\_calc.exe X Y OUTPUT

– X：信道输入消息序列文件的路径

– Y：信道输出消息序列文件的路径

– OUTPUT：信道计算结果文件的路径

计算结果追加的格式为

"输入消息序列的信息熵","输出消息序列的信息熵","平均互信息量"

关键代码如下：

def workflow(x\_file\_name, y\_file\_name, out\_file\_name, verbose=False):

    """The main workflow."""

    # Number of binary bits in one symbol.

    N = 8

    x = read\_file\_as\_bytes(x\_file\_name)

    y = read\_file\_as\_bytes(y\_file\_name)

    ## --- Core computation: begin

    start\_time = time.time()

    joint\_p\_xy = calc\_joint\_p\_xy(x, y)

    H\_x = calc\_H\_p(calc\_p\_x(joint\_p\_xy)) / N

    H\_y = calc\_H\_p(calc\_p\_y(joint\_p\_xy)) / N

    joint\_H\_xy = calc\_joint\_H\_xy(joint\_p\_xy) / N

    cond\_H\_xy = calc\_cond\_H\_xy(joint\_p\_xy) / N

    cond\_H\_yx = calc\_cond\_H\_yx(joint\_p\_xy) / N

    I\_xy = H\_x - cond\_H\_xy

    # Calculate error probability of the BSC.

    err = np.bitwise\_xor(x, y)

    p\_BSC = count\_binary\_1(err) / (x.size \* N)

    elapsed\_time = time.time() - start\_time

    ## --- Core computation: end

    write\_results(out\_file\_name, [

                  x\_file\_name, y\_file\_name, H\_x, H\_y, I\_xy])

    return H\_x

### 单元测试

* byteChannel部分：

理论推导：

设错误传递概率，信道输入消息X为的概率分布，信道输出消息为Y，重用【实验3.1 二元对称信道（BSC）仿真】中制作的信道指标计算excel表格，在自变量中输入，得到理论推导结果如下红框所示：



其中输入消息序列的信息熵H(X)=1，输出消息序列的信息熵H(Y)=1，X和Y的联合熵H(XY)=2，损失熵H(X|Y)=1，噪声熵H(Y|X)=1，平均互信息量I(X;Y)=0。

测试结果：

运行byteChannel.py程序，在cmd中输入如下命令：

python byteChannel.py unit-test\P0=0.5.LEN=1MB.OUTPUT.dat 0.5 unit-test\p=0.5.P0=0.5.LEN=1MB.Channel\_OUTPUT.dat

生成的信道输出消息序列文件如下：



重用【实验3.1 二元对称信道（BSC）仿真】中教师提供的信道指标计算程序calcBSCInfo.exe，其程序API如下：

calcBSCInfo.exe X Y OUTPUT

X path to the channel input file

Y path to the channel output file

OUTPUT path to the output file to append results

在cmd中输入如下命令：

calcBSCInfo.exe ..\P0=0.5.LEN=1MB.OUTPUT.dat ..\p=0.5.P0=0.5.LEN=1MB.Channel\_OUTPUT.dat ..\Channel\_calcBSCInfo\_result.CSV

程序运行后信道指标计算结果如下：



可以看到实际测试的计算结果和理论推导的结果相接近，在一定误差允许范围内可以认为是正确的，byteChannel程序单元测试完成。

* byteChannel\_calc部分：

理论推导：

同上byteChannel部分。

测试结果：

运行byteChannel\_calc.py程序，将在cmd中输入如下命令：

python byteChannel\_calc.py unit-test\P0=0.5.LEN=1MB.OUTPUT.dat unit-test\p=0.5.P0=0.5.LEN=1MB.Channel\_OUTPUT.dat unit-test\Channel\_calc\_result.CSV

得到如下CSV文件和结果：





该计算结果与使用程序calcBSCInfo.exe计算得到的结果相同，且与理论推导的结果相接近，可以认为该结果是正确的，byteChannel\_calc单元测试完成。

## 信源编解码

### 霍夫曼编码模块

程序API如下：

byteSourceEncoder.exe PMF INPUT OUTPUT

– PMF： path to probability mass function CSV file

– INPUT：path to the encoder input file

– OUTPUT：path to the encoder output file

关键代码如下

def encode(pmf\_file\_name, in\_file\_name, out\_file\_name):  
 """  
 @description: use to encode  
 @param: pmf\_file\_name: the path of pmf file  
 @param: in\_file\_name: the path of input file  
 @param: out\_file\_name: the path of output file  
 @return: (len(source), len(encoded)): (the length of source, the length of encoded source)  
 """  
  
 '''  
 打开 CSV 文件并读取，然后保存为字典，  
 将第一列以 uint8 保存为键，  
 第二列以 float 保存为值  
 '''  
 with open(pmf\_file\_name, newline='') as csv\_file:  
 pmf = dict([(uint8(row[0]), float(row[1]))  
 for row in reader(csv\_file)])  
  
 # 构建赫夫曼树  
 # 将EOF符号设置为“frequencies”中的第一个符号，  
 # 这样“dahuffman”在构建Huffman树时不会添加新的EOF符号  
 # 递归实现  
 codec = HuffmanCodec.from\_frequencies(pmf)  
 # 以 uint8 读取输入文件中的数据  
 source = fromfile(in\_file\_name, dtype='uint8')  
 # 使用source作为编码方案编码  
 # str.encode(encoding=source, errors='strict')  
 encoded = codec.encode(source)  
  
 # 返回编码后的码书  
 codebook = codec.get\_code\_table()  
 # 高低位标志，little 表示反序，左边为低位右边为高位  
 byteorder = 'little'  
 # 返回一个字节数组，  
 # 对应 header\_size，头部字节数，uint16格式  
 header = bytearray(2)  
 # 使用尾插法向字节数组 header 添加元素 len(codebook)-1  
 # 对应 symbol\_count，码书中符号个数减一，uint8 格式  
 header.append(len(codebook)-1)  
 # 扩展列表，追加序列 len(source).to\_bytes(4, byteorder)  
 # 对应 source\_len，信源符号个数，uint32 格式  
 header.extend(len(source).to\_bytes(4, byteorder))  
 # 向字节数组追加符号、码字长度以及码字  
 for symbol, (word\_len, word) in codebook.items():  
 (word\_len, word) = codebook[symbol]  
 # 返回 word\_len/8 向上取整的值，用于指定保存这个码字所需的字节数  
 word\_bytes = int(ceil(word\_len / 8))  
 header.append(symbol)  
 header.append(word\_len)  
 # 将码字转为二进制追加到 header  
 header.extend(word.to\_bytes(word\_bytes, byteorder))  
 # 将 header 首部三位转为二进制，即 Header 部分  
 header[0:2] = len(header).to\_bytes(2, byteorder)  
  
 '''  
 以二进制方式把数据写入输出文件  
 首先写入首部  
 然后写入霍夫曼编码  
 '''  
 with open(out\_file\_name, 'wb') as out\_file:  
 out\_file.write(header)  
 out\_file.write(encoded)  
  
 # 返回信源长度，编码后的信源长度  
 return (len(source), len(encoded))

### 霍夫曼解码模块

程序API如下：

byteSourceDecoder.exe INPUT OUTPUT

– INPUT：path to the decoder input file

– OUTPUT：path to the decoder output file

关键代码如下：

def decode(in\_file\_name, out\_file\_name):  
 """  
 @description: use to decode  
 @param: in\_file\_name: the path of input file  
 @param: out\_file\_name: the path of output file  
 @return: (len(encoded), len(decoded)): (the length of encoded source, the length of decoded source)  
 """  
  
 # 高低位标志，little 表示反序，左边为低位右边为高位  
 byteorder = 'little'  
  
 '''  
 以二进制读取文件  
 '''  
 with open(in\_file\_name, 'rb') as in\_file:  
 # 第二位是首部长度(header\_size)，以反序读取，转为 int  
 header\_size = int.from\_bytes(in\_file.read(2), byteorder)  
 # 操作二进制数据，需使用BytesIO  
 # 以二进制读取首部(Header) header\_size-2 位部分  
 header = BytesIO(in\_file.read(header\_size-2))  
 # 读取全部文件数据  
 encoded = in\_file.read()  
  
 codebook = {}  
 # 获取 symbol\_count，位于header第一位的第零位  
 symbol\_count = header.read(1)[0]  
 # 获取 source\_len，位于header第四位，以反序读取后转为int  
 source\_len = int.from\_bytes(header.read(4), byteorder)  
 # 将编码放入字典 codebook 中  
 for k in range(symbol\_count+1):  
 # 读取符号  
 symbol = uint8(header.read(1)[0])  
 # 读取码字长度  
 word\_len = header.read(1)[0]  
 # 码字字节数  
 word\_bytes = int(ceil(word\_len / 8))  
 # 码字  
 word = int.from\_bytes(header.read(word\_bytes), byteorder)  
 # 将码字长度、码字存入字典codebook中  
 codebook[symbol] = (word\_len, word)  
  
 # 将字典作为参数初始化一个HuffmanCodec类用于译码  
 codec = HuffmanCodec(codebook)  
 # 译码  
 # np.asarray()，将数据转化为ndarray但不占用新内存  
 decoded = asarray(codec.decode(encoded))[:source\_len]  
 # 存入输出文件  
 decoded.tofile(out\_file\_name)  
  
 # 返回编码后长度，译码后长度  
 return (len(encoded), len(decoded))

### 信源编码指标计算模块

程序API如下：

codding\_effect.exe X Y OUTPUT

– X：编码前的文件的路径

– Y：编码后的文件的路径

– OUTPUT：信源编码计算结果文件的路径

会在OUTPUT结果文件中添加如下内容：

"压缩比","平均码长","编码效率"," 编码前的文件的信息熵","编码后的文件的信息熵"

关键代码如下：

def H\_s(BS):  
 '''计算信息熵  
 根据用户输入的文件计算编码前文件信息熵和编码后文件信息熵  
 Args:  
 BS (BitStream): 文件的比特流  
   
 Returns:  
 H(BS1)编码前文件的信息熵 (信息比特/字节)  
 H(BS2)编码后文件的信息熵 (信息比特/字节)  
 '''  
 #由于一个字节由八个二进制字符组成，因此可视为二进制字符信源的八次拓展  
 count = 0  
 for i in range(len(BS)):  
 if BS[i]==1:  
 count=count+1 #数出二进制串1的个数即可  
 P1=count/len(BS)  
 P0=1-P1  
 Hs\_bit=-P1\*log2(P1)-P0\*log2(P0) #计算平均1bit的信息熵  
 Hs\_byte=Hs\_bit\*8 #乘以8即等于平均一个字节的信息熵  
 return Hs\_byte,P0  
  
def compress\_ratio(BS1,BS2):  
 '''计算压缩比  
 根据用户输入的文件计算压缩比  
 Args:  
 BS1 (BitStream): 文件1的比特流  
 BS2 (BitStream): 文件2的比特流  
 Returns:  
 compress(float)压缩比  
 '''  
 #用编码前文件大小除以编码后文件大小即可得到压缩比  
 compress = len(BS1)/len(BS2)  
 return compress  
  
def l(in\_file\_name,P0):  
 '''计算平均码长  
 根据用户输入的文件计算平均码长  
 Args:  
 in\_file\_name(string): 输入文件  
   
 Returns:  
 l(float)平均码长 (码字数据比特/信源字节)  
 '''  
 byteorder = 'little'  
 with open(in\_file\_name, 'rb') as in\_file:  
 # 第二位是首部长度(header\_size)，以反序读取，转为 int  
 header\_size = int.from\_bytes(in\_file.read(2), byteorder)  
 # 操作二进制数据，需使用BytesIO  
 # 以二进制读取首部(Header) header\_size-2 位部分  
 header = BytesIO(in\_file.read(header\_size-2))  
 # 读取全部文件数据  
 encoded = in\_file.read()  
  
 codebook = {}  
 # 获取 symbol\_count，位于header第一位的第零位  
 symbol\_count = header.read(1)[0]  
 # 获取 source\_len，位于header第四位，以反序读取后转为int  
 source\_len = int.from\_bytes(header.read(4), byteorder)  
 # 将编码放入字典 codebook 中  
 len\_code=0  
 for k in range(symbol\_count+1):  
 # 读取符号  
 symbol = uint8(header.read(1)[0])  
 # 读取码字长度  
 word\_len = header.read(1)[0]  
 # 码字字节数  
 word\_bytes = int(ceil(word\_len / 8))  
 # 码字  
 word = int.from\_bytes(header.read(word\_bytes), byteorder)  
 # 将码字长度、码字存入字典codebook中  
 codebook[symbol] = (word\_len, word)  
 c=bin(symbol)[2:].rjust(8,'0')  
   
 cou = 0  
 for b in range(len(c)):  
 if c[b]=='0':  
 cou=cou+1  
 len\_code = len\_code+word\_len\*pow(P0,cou)\*pow(1-P0,8-cou)  
   
 return len\_code  
  
def coding\_efficiency(Hs,L):  
 '''计算编码效率  
 根据用户输入的文件计算编码效率以及编码前文件信息熵和编码后文件信息熵  
 Args:  
 BS1 (BitStream): 文件1的比特流  
 BS2 (BitStream): 文件2的比特流  
   
 Returns:  
 effict(float)编码效率  
 '''  
 n = Hs/L  
 return n

### 单元测试

* byteSourceEncoder/ byteSourceDecoder部分：

理论推导：

注意测试时避免将未经验证的channelEncoder和channelDecoder程序串联起来自我验证。

测试结果：

待编码的原文件如下：



1. 编码测试：

在cmd中输入如下命令进行5长的重复码编码：

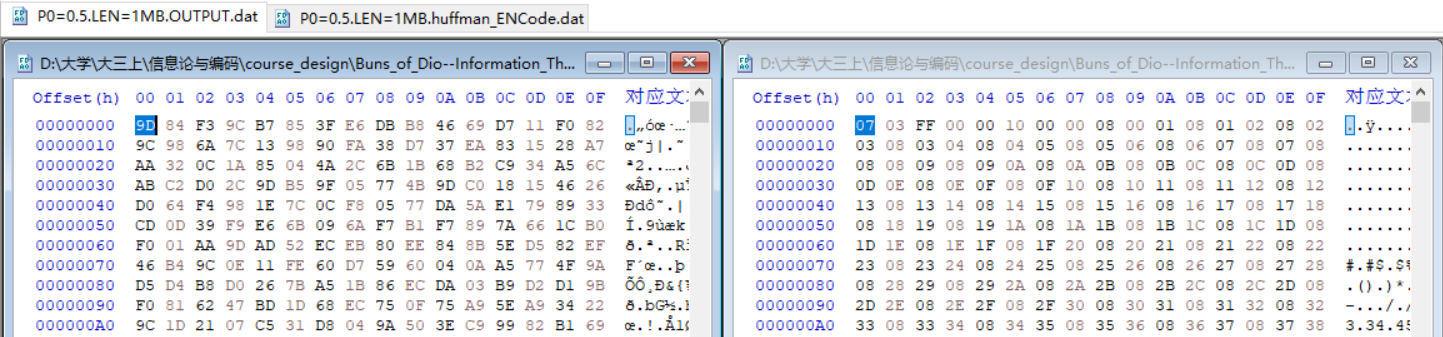
python byteSourceEncoder.py unit-test\pmf.byte.p0=0.5.csv unit-test\P0=0.5.LEN=1MB.OUTPUT.dat unit-test\P0=0.5.LEN=1MB.huffman\_ENCode.dat

程序运行完毕后得到如下文件：



可以看到文件大小与编码前的原文件大小不同，甚至比原文件要大一些，这是因为原文件是由等概率的二元DMS生成，字节概率分别是比较均匀的，而霍夫曼编码基于变长码的思想，因此对信源符号分布较均匀的序列的压缩效果微弱甚至反而会增大文件大小。

使用HxD的文件比较功能对两者的数据进行比较：



可以看到两者数据亦不相同，说明编码器是对原文件进行了处理的。

1. 解码测试：

在cmd中输入如下命令，利用解码器对编码后的文件进行解码：

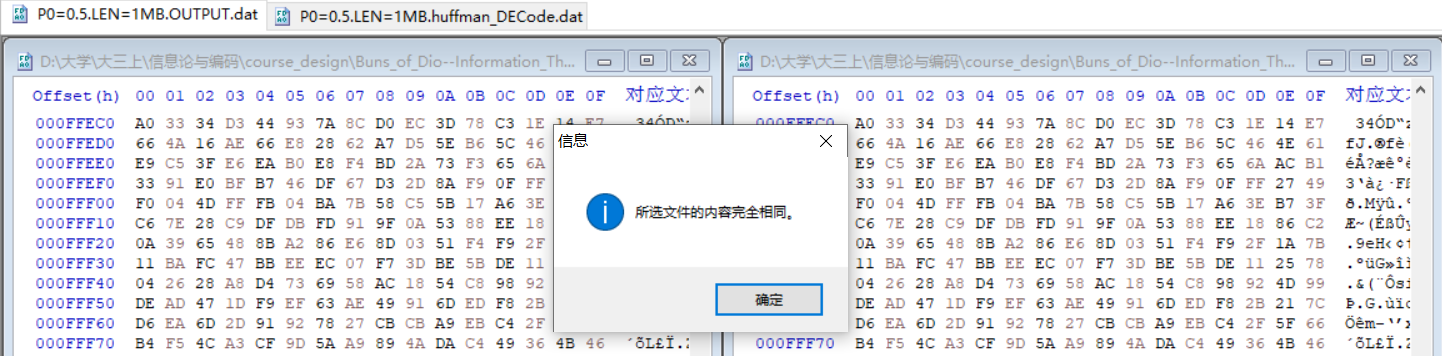
python byteSourceDecoder.py unit-test\P0=0.5.LEN=1MB.huffman\_ENCode.dat unit-test\P0=0.5.LEN=1MB.huffman\_DECode.dat

运行完毕后得到的文件如下：



可以看到文件大小与编码前的原文件大小相同。

使用HxD的文件比较功能对解码后的文件和原文件进行比较：



HxD弹窗提示所选文件的内容完全相同，说明解码后可以完整无失真地还原原文件，单元测试通过。

* codding\_effect部分：

理论推导：

1. 压缩比

编码前文件长度为l=1048576字节，编码后文件长度为L=1049351字节，故压缩比

1. 编码前的文件的信息熵

信源消息概率分布，故每二元消息的信息熵为

每字节有8数据比特，则编码前的文件的信息熵为

1. 平均码长

因为霍夫曼码是在信源给定情况下的最佳码，所以其平均码长的界限为

其中代表r元信源的信息熵，单元测试中为二元信源即r=2，且单元测试中编码前文件的每个字节相当于二元DMS的8次扩展（拓展次数N=8），同样采取霍夫曼编码方法时，因为霍夫曼码是紧致码，所以编码后单个数据比特所编得的平均码长随拓展次数N的增加很快接近于极限值——二元DMS的信息熵H，故理想条件下编码后单个数据比特的平均码长为

进而每个信源符号（8个数据比特）编码后的理论平均码长为

1. 编码效率

根据编码效率计算公式，编码效率为

1. 编码后的文件的信息熵

编码后的每二元消息的信息熵

编码后的文件的信息熵为

测试结果：

在cmd中输入如下命令，运行codding\_effect：

python codding\_effect.py unit-test\P0=0.5.LEN=1MB.OUTPUT.dat unit-test\P0=0.5.LEN=1MB.huffman\_ENCode.dat unit-test\huffman\_calresult.CSV

运行完毕后得到如下结果：





可以看到实际测试结果与理论推导结果相接近，在一定误差允许范围内可以认为是正确的，codding\_effect程序单元测试完成。

## 信道编解码

### 信道编码模块

程序API如下：

channelEncoder.exe INPUT TYPE OUTPUT N

– INPUT： 编码前的文件

– TYPE ： 编码的类型，“-r”为重复码，“-l”为线性分组码

– OUTPUT：编码后的文件

– N：若为重复码，“5”，“7”，“9”分别代表N是5，7，9的重复码

若为线性分组码，“3”，“4”，“5”分别代表（7，4），（15，11）（31，26）线性分组码

关键代码如下：

def gen\_header(method, var, BS\_len, BS):

    '''

    生成文件头

    Args:

        method (int): 编码方式，0 为重复码 1 为 线性分组码

        var (int): 重复码的码字长度 或 线性分组码的奇偶校验长度

        BS\_len (int): 编码前序列的长度

        BS (bitstream): 编码后文件比特流

    Returns:

        source (bitstream): 最终输出文件比特流

    '''

    method = BitStream(uint=method, length=8)

    factor = BitStream(uint=var, length=8)

    source\_length = Bits(uint=BS\_len, length=32)

    headers = encode\_repeat(method + factor + source\_length, 9)

    source = headers + BS

    return source

def addZero(arr, n):

    '''

    根据需要在数组后面补足 0

    Args：

        arr (array): 指定的数组

        n (int): 需要添加的最长长度

    '''

    lack0 = len(arr) % n

    if lack0 != 0:

        arr = hstack((arr, array([0] \* (n - lack0))))

    return arr

def encode\_repeat(BS\_encode, n):

    '''

    重复码编码

    Args:

        BS\_encode (bitstream): 输入文件比特流

        n (int): 重复码的码字长度

    Returns:

        BS\_encode\_repetition (bitstream): 编码后的比特流

    '''

    # 判断是否是规定的重复码的码字长度

    if (n in (3, 5, 7, 9)) == False:

        return

    # 转成二进制便于编码

    BS\_encode\_bin = BS\_encode.bin

    BS\_encode\_rep = ''

    for i in BS\_encode\_bin:

        BS\_encode\_rep += n \* i

    BS\_encode\_repetition = BitStream(bin=BS\_encode\_rep)

    return BS\_encode\_repetition

def encode\_linear(BS\_encode, j):

    '''

    线性分组码编码

    Args:

        BS\_encode (bitstream): 输入文件比特流

        j (int): 奇偶校验码的码字长度

    Returns:

        C (array): 编码后的数组

    '''

    if (j in (3, 4, 5))== False:

        return

    # 获取生成矩阵

    G = genG(j)

    # 将比特流转换为数组形式

    BS\_bin = BS\_encode.bin

    BS\_len = len(BS\_bin)

    BS\_bin\_arr = array([int(BS\_bin[i], 2) for i in range(BS\_len)])

    # 将数组转换为指定样式（由奇偶校验长度j决定）的矩阵

    if j == 3:

        BS\_bin\_arr = addZero(BS\_bin\_arr, 4)

        BS\_mat = BS\_bin\_arr.reshape(-1, 1).reshape(-1, 4)

    elif j == 4:

        BS\_bin\_arr = addZero(BS\_bin\_arr, 11)

        BS\_mat = BS\_bin\_arr.reshape(-1, 1).reshape(-1, 11)

    elif j == 5:

        BS\_bin\_arr = addZero(BS\_bin\_arr, 26)

        BS\_mat = BS\_bin\_arr.reshape(-1, 1).reshape(-1, 26)

    # 矩阵相乘相加，得到编码后的数组

    C = dot(BS\_mat, G).ravel() % 2

    C = addZero(C, 8)

    C\_str = ''.join(str(i) for i in C)

    C\_BS = BitStream(bin=C\_str)

    return C\_BS

### 信道解码模块

程序API如下：

channelDecoder.exe INPUT OUTPUT

– INPUT：解码前的文件路径

– OUTPUT：解码后的文件路径

关键代码如下：

def decode\_repeat(BS\_decode, BDRT, method='-a'):

    '''

    重复码解码，兼有解码文件头功能

    Args:

        BS\_dncode (bitstream): 待解码的输入文件比特流

        BDRT (int): 重复码的码字长度

        method (str): -a 表示解码文件本身、-h 表示解码文件头，默认为 '-a'

    Returns:

        BS\_encode\_repetition (str): 解码后生成的字符串

    '''

    # 三个字符串备用

    BS\_encode\_repetition = '0b'

    BS = ''

    BS\_decode\_rep = ''

    if method == '-h':

        # 解码文件头

        BS\_decode\_rep = BS\_decode.bin[0:432]

    else:

        # 解码文件

        BS\_decode\_rep = BS\_decode.bin[432:]

    # 解码程序本题

    for x in BS\_decode\_rep:

        BS += x

        if len(BS) == BDRT:

            if BS.count('1') > BS.count('0'):

                BS\_encode\_repetition += '1'

            elif BS.count('1') < BS.count('0'):

                BS\_encode\_repetition += '0'

            BS = ''

    # 若空文件则返回 None

    if BS\_encode\_repetition == '0b':

        return

    return BS\_encode\_repetition

def decode\_linear(C\_decode, j):

    '''

    线性分组码解码

    Args:

        C\_dncode (bitstream): 待解码的输入文件比特流

        j (int): 奇偶校验长度

    Returns:

        BS\_info\_mat\_str (str): 解码后生成的字符串

    '''

    # 根据奇偶校验长度，得到(n, k)线性分组码的 n, k

    if j == 3:

        n = 7

        k = 4

    elif j == 4:

        n = 15

        k = 11

    else:

        n = 31

        k = 26

    # 去掉文件头部分开始解码

    BS\_bin = C\_decode.bin[432:]

    # 还原成 n列 的矩阵形式

    BS\_len = len(BS\_bin)

    overflow0 = BS\_len % n

    if overflow0 != 0:

        BS\_bin = BS\_bin[:(BS\_len - overflow0)]

    BS\_bin\_arr = array([int(BS\_bin[i], 2) for i in range(len(BS\_bin))])

    BS\_data\_mat = BS\_bin\_arr.reshape(-1, n)

    # 获得信息组

    BS\_info\_mat = BS\_data\_mat[:, :k]

    # 获得校验矩阵

    GT = genG(j)[:, k:].T

    H = hstack((GT, identity(GT.shape[0])))

    # 获得伴随式

    S = dot(BS\_data\_mat, H.T) % 2

    # 纠错

    HT = array(H.T)

    pos = []

    for i in range(len(S)):

        for j in range(len(HT)):

            if (S[i] == HT[j]).all():

                pos.append((i, j))

    BS\_info\_mat = linear\_correct(BS\_info\_mat, pos, k)

    # 将矩阵转换为字符串

    BS\_info\_mat\_ravel = BS\_info\_mat.ravel()

    BS\_info\_mat\_str = ''.join(str(i) for i in BS\_info\_mat\_ravel)

    return BS\_info\_mat\_str

def linear\_correct(BS\_info\_mat, pos, n):

    '''

    线性分组码纠错

    Args:

        BS\_info\_mat (array): 信息组

        pos (liat): 误码位置

        n (int): 信息组的长度，与监督元相区别

    Returns:

        BS\_info\_mat (array): 纠错后的信息组

    '''

    for i in range(len(pos)):

        if pos[i][1] < n:

            BS\_info\_mat[pos[i]] = not(BS\_info\_mat[pos[i]]) + 0

    return BS\_info\_mat

def gen\_header(headers):

    '''

    获取文件头信息

    Args:

        headers (bitstream): 文件头

    Returns:

        method (int): 编码方式，0 为重复码 1 为 线性分组码

        factor (int): 重复码的码字长度 或 线性分组码的奇偶校验长度

        source\_length (int): 编码前序列的长度

    '''

    method = int(headers.bin[0:8], 2)

    factor = int(headers.bin[8:16], 2)

    source\_length = int(headers.bin[16:], 2)

    return method, factor, source\_length

### 信道编解码指标计算模块

程序API如下：

channelCoder\_calc.exe INPUT1 INPUT2 OUTPUT

– INPUT1：编码前的文件的路径

– INPUT2：编码后的文件的路径

– INPUT3: 解码后的文件的路径

– OUTPUT：信源编码计算结果文件的路径

调用后会在计算结果加入以下内容：

"误码率","编码前的信源信息传输率","编码后的信源信息传输率"," 压缩比"

关键代码如下：

def gen\_BER(FBE, FD):

    '''获取误码率

    Args:

        FBE (str): 编码前文件比特流转换的字符串

        FD (str): 解码后文件比特流转换的字符串

    Returns:

        error\_rate (float): 误码率

    '''

    error\_bit = list(map(lambda x, y: x == y, FBE, FD)).count(False)

    error\_rate = error\_bit / len(FBE)

    return error\_rate

def gen\_Rs(method, factor):

    '''获取编码前信息传输率以及编码后信息传输率

    Args:

        method (str): 解码方式 0 为重复码 1 为 线性分组码

        factor (str): 重复码的码字长度 或 线性分组码的奇偶校验长度

    Returns:

        R\_before (float): 编码前信息传输率

        R\_after (float): 编码后信息传输率

    '''

    R\_before = log2(2) / 1

    if method == 0:

        R\_after = log2(2) / factor

    elif method == 1:

        if factor == 3:

            R\_after = log2(2) / 7

        elif factor == 4:

            R\_after = log2(2) / 15

        elif factor == 5:

            R\_after = log2(2) / 31

        else:

            return

    else:

        return

    return R\_before, R\_after

def gen\_compression\_ratio(source\_length, FAE):

    '''获取压缩比

    Args:

        source\_length (int): 编码前文件长度

        FAE (str): 编码后文件比特流转换的字符串

    Returns:

        (float): 压缩比

    '''

    return source\_length / len(FAE)

### 单元测试

* channelEncoder/channelDecoder部分：

理论推导：

注意测试时避免将未经验证的channelEncoder和channelDecoder程序串联起来自我验证。

测试结果：

待编码的原文件如下：



对6种方案分别进行测试，以5长的重复码为例

1. 编码测试：

在cmd中输入如下命令进行5长的重复码编码：

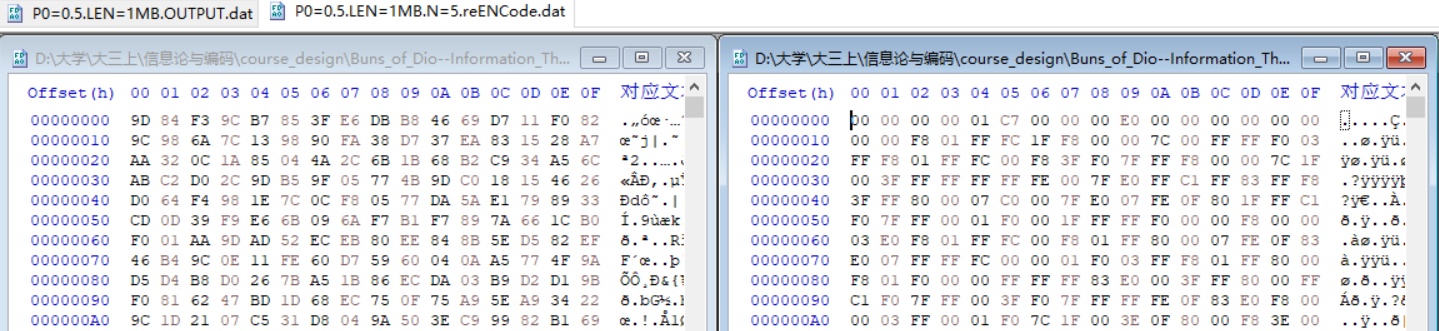
python channelEncoder.py -r unit-test\P0=0.5.LEN=1MB.OUTPUT.dat unit-test\P0=0.5.LEN=1MB.N=5.reENCode.dat 5

程序运行完毕后得到如下文件：



可以看到文件大小与编码前的原文件大小不同，N=5长的重复编码后的文件约比原文件大5倍。

使用HxD的文件比较功能对两者的数据进行比较：



可以看到两者数据亦不相同，说明编码器是对原文件进行了处理的。

1. 解码测试：

在cmd中输入如下命令，利用解码器对编码后的文件进行解码：

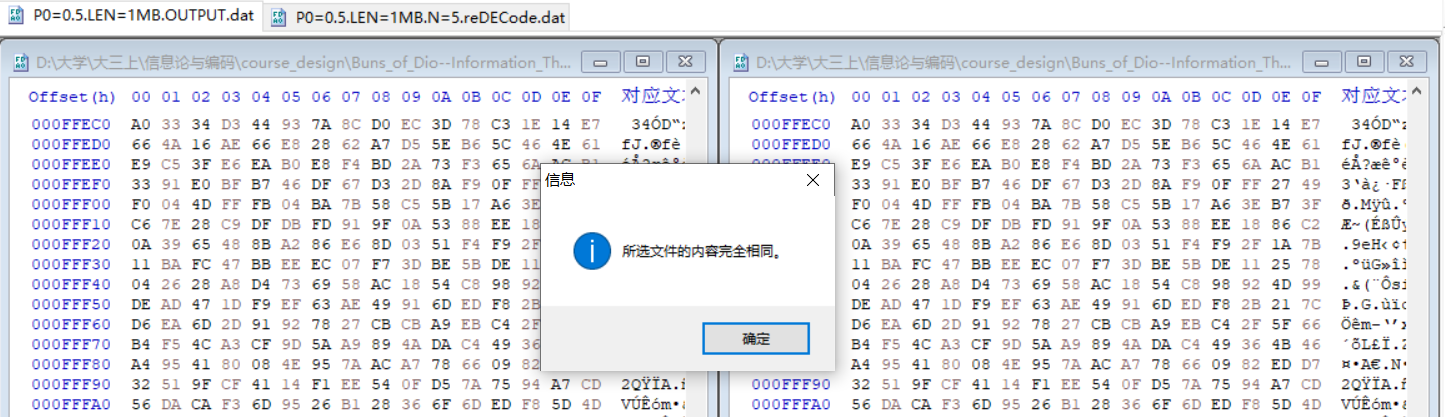
python channelDecoder.py unit-test\P0=0.5.LEN=1MB.N=5.reENCode.dat unit-test\P0=0.5.LEN=1MB.N=5.reDECode.dat

运行完毕后得到的文件如下：



可以看到文件大小与编码前的原文件大小相同。

使用HxD的文件比较功能对解码后的文件和原文件进行比较：



HxD弹窗提示所选文件的内容完全相同，说明解码后可以完整无失真地还原原文件，5长重复码单元测试通过。

7/9长的重复码和(n, k)线性分组码的测试步骤与5长重复码的步骤相同，不再赘述，经测试均通过。

* channelCoder\_calc部分：

理论推导：

1. 误码率

由于原文件编码后未经信号传输而直接解码，没有噪声干扰，因此相当于无失真编解码，误码率

1. 编码前的信源信息传输率

信源为等概率的二元DMS，消息个数，因此编码前的信源信息传输率

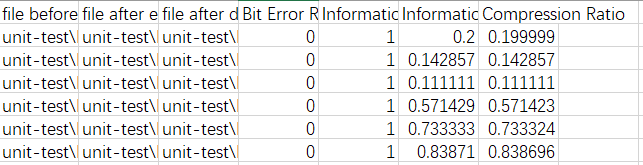
1. 编码后的信源信息传输率

设编码是将每n位二元消息编成N位码长的码字，则

1. 对于5长的重复码，n=1，N=5，故编码后的信源信息传输率
2. 对于7长的重复码，n=1，N=7，故编码后的信源信息传输率
3. 对于9长的重复码，n=1，N=9，故编码后的信源信息传输率
4. 对于(7, 4)线性分组码，n=4，N=7，故编码后的信源信息传输率
5. 对于(15, 11)线性分组码，n=11，N=15，故编码后的信源信息传输率
6. 对于(31, 26)线性分组码，n=26，N=31，故编码后的信源信息传输率
7. 压缩比
8. 对于5长的重复码，编码前文件长度为l=1048576字节，编码后文件长度为L=5242898字节，故压缩比
9. 对于7长的重复码，编码前文件长度为l=1048576字节，编码后文件长度为L=7340050字节，故压缩比
10. 对于9长的重复码，编码前文件长度为l=1048576字节，编码后文件长度为L=9437202字节，故压缩比
11. 对于(7, 4)线性分组码，编码前文件长度为l=1048576字节，编码后文件长度为L=1835026字节，故压缩比
12. 对于(15, 11)线性分组码，编码前文件长度为l=1048576字节，编码后文件长度为L=1429895字节，故压缩比
13. 对于(31, 26)线性分组码，编码前文件长度为l=1048576字节，编码后文件长度为L=1250245字节，故压缩比

测试结果：

根据不同编码类型输入相应参数，运行channelCoder\_calc，运行完毕后得到如下结果：



前三列分别为：编码前文件路径、编码后文件路径、解码后文件路径；后四列分别为：误码率、编码前的信源信息传输率、编码后的信源信息传输率、压缩比。可以看到实际测试结果与理论推导结果相接近，可以认为是正确的，channelCoder\_calc程序单元测试完成。

# 系统仿真与分析

## 理想情况下

### 条件设定

理想情况下，信源为等概率二元DMS，信源概率分布为，

则其信息熵（每数据比特）为

当其信源数据率为*v*（数据比特每秒）时，其每秒信源信息率为，单位为信息比特每秒,由于信道为无噪无损的离散无记忆信道，即噪声熵和损失熵都为0，本题定义信源数据率为1数据比特/秒。

**信源概率分布****，即等概率二元DMS，信道为无噪无损的离散无记忆信道，不使用信源编码，不使用信道编码**

由于信源概率分布，信息熵为:



定义信源数据率为1数据比特/秒，所以信源信息率为：

。

无信源编码，则信道的输入信息率等于信源信息率；

无信道编码，信道无噪无损，所以信道的输出信息率与等于信道的输入信息率，不存在误码率。理论推导结果为:

a.信源的信息率:



其中M为是总字节数，由于本题中设定信源的数据率为1数据比特/秒，传输时间为1s，故M=1信息比特。

b.信道的数据率数据比特/秒

c.信道的输入信息率=1信息比特/秒

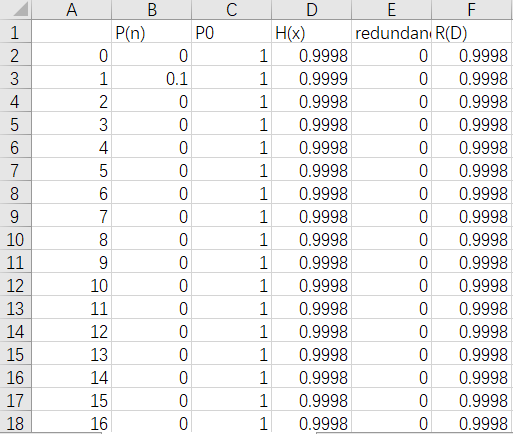
d.信道的输出信息率=1信息比特/秒

e.信宿的误码率=0

### 仿真

**信源概率分布****，即等概率二元DMS，信道为无噪无损的离散无记忆信道，不使用信源编码，不使用信道编码：**

先用程序byteSource.py生成一个概率分布为离散无记忆信源(大小为1KB)，调用程序byteSource\_calc.py可以计算其信源信息率的大小(数据率为1bit/s)(文件名为：P0=0.5.LEN=10KB.calc.CSV)：

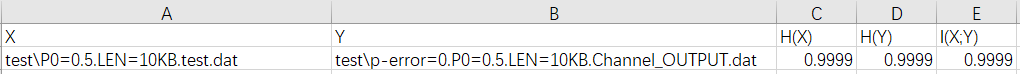


从上图可以知道计算的信源信息率为1信息比特/秒。

再通过程序byteChannel.py生成过信道的序列：



再调用byteChannel\_calc.py的函数进行对其过信道的序列进行信道的输入与的信息熵的计算，如下(文件名：P0=0.5.Channel\_not\_source\_calc\_result.CSV)：



由于取信源的数据率为1数据比特/秒，由于不使用信源编码，因此，信道的数据率为数据比特/s。再与上图中的信息熵分别相乘，即可得到信道的输入与输出的信息率，经过计算可以得到信道的输出信息率=输入信息率=1.0信息比特/秒。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 信源信息率  （信息比特/秒） | 信道数据率  （数据比特/秒） | 信道输入信息率（信息比特/秒） | 信道输出信息率（信息比特/秒） | 误码率 |
| 理论值 | 1 | 1 | 1 | 1 | 0 |
| 仿真值 | 0.9998 | 0.9998 | 0.9999 | 0.9999 | 0 |

### 分析

对比上述理论分析和仿真结果可以看到，仿真结果大致与理论推导结果一致，对于相应的仿真值与理论值相比，理论值较大，原因可能与计算时的保留小数位有关。因为理论值计算得到的信息熵为1，但是实际上由于是生成一定概率分布的信源，长度也只有10KB，故计算出来的信息熵只会趋近1，但不完全等于1，导致后面许多计算相对理论值小一点，但这结果基本在可接受的误差范围以内，故可判断该理论推导和仿真结果基本正确。

## 信源非理想情况下

### 条件设定

### 理论推导

假设信源概率分布为,则其信息熵（每数据比特）为。当其信源数据率为v（数据比特每秒）时，其信源信息率为，单位为信息比特每秒,由于信道为无噪无损的离散无记忆信道，即噪声熵和损失熵都为0，本题定义信源数据率为1数据比特/秒。

**(1).信源概率分布，不使用信源编码**

由于信源概率分布****，信息熵为:



定义信源数据率为1数据比特/秒，所以信源信息率为

。

无信源编码，则信道的输入信息率等于信源信息率；

无信道编码，信道无噪无损，所以信道的输出信息率与等于信道的输入信息率，不存在误码率。理论推导结果为:

a.信源的信息率:



其中M为是总字节数，由于本题中设定信源的数据率为1数据比特/秒，传输时间为1s，故M=1信息比特。

b.信道的数据率=数据比特/秒

c.信道的输入信息率=0.4690信息比特/秒

d.信道的输出信息率=0.4690信息比特/秒

e.信宿的误码率=0

**(2).信源概率分布，使用信源编码**

由于信源概率分布****，信息熵为:



定义信源数据率为1数据比特/秒，所以信源信息率为

。

无信道编码，信道无噪无损，所以信道的输出信息率与等于信道的输入信息率，不存在误码率。

有信源编码：由于我们是在每次的信源生成消息后，根据当前信源概率分布生成霍夫曼码书，所以每次的霍夫曼码书可能略有不同，我们输出对应的每次的平均码长，由于信道传输消息的时间是取决于信源生成消息序列的时间，所以信道数据率为，即8×信源信息率/输出序列的平均码长。

由于采用了霍夫曼编码，信源实际上被压缩，信息率提高，所以信道的输入信息率会大于等于信源的信息率。

理论推导结果为:

a.信源的信息率:



其中M为是总字节数，由于本题中设定信源的数据率为1数据比特/秒，传输时间为1s，故M=1信息比特。



**(3).信源概率分布，不使用信源编码**

由于信源概率分布****，信息熵为:



定义信源数据率为1bit/s，所以信源信息率为

。

无信源编码，则信道的输入信息率等于信源信息率；

无信道编码，信道无噪无损，所以信道的输出信息率与等于信道的输入信息率，不存在误码率。理论推导结果为:

a.信源的信息率:



其中M为是总字节数，由于本题中设定信源的数据率为1数据比特/秒，传输时间为1s，故M=1信息比特。

b.信道的数据率数据比特/秒

c.信道的输入信息率=0.8813信息比特/秒

d.信道的输出信息率=0.8813信息比特/秒

e.信宿的误码率=0

**(4).信源概率分布，使用信源编码**

由于信源概率分布****，信息熵为:



定义信源数据率为1数据比特/秒，所以信源信息率为

。

无信道编码，信道无噪无损，所以信道的输出信息率与等于信道的输入信息率，不存在误码率。

有信源编码：由于我们是在每次的信源生成消息后，根据当前信源概率分布生成霍夫曼码书，所以每次的霍夫曼码书可能略有不同，我们输出对应的每次的平均码长，由于信道传输消息的时间是取决于信源生成消息序列的时间，所以信道数据率为，即8×信源信息率/输出序列的平均码长。

由于采用了霍夫曼编码，信源实际上被压缩，信息率提高，所以信道的输入信息率会大于等于信源的信息率。

理论推导结果为:

a.信源的信息率:



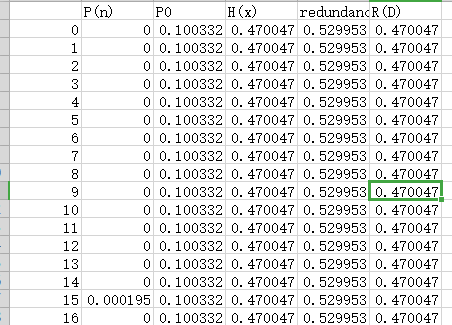
其中M为是总字节数，由于本题中设定信源的数据率为1数据比特/秒，传输时间为1s，故M=1信息比特。



### 仿真

**(1).信源概率分布时，不使用信源编码：**

先用程序byteSource.py生成一个概率分布为离散无记忆信源(大小为1KB)，调用程序byteSource\_calc.py可以计算其信源信息率的大小(数据率为1bit/s)(文件名为：P0=0.1.LEN=10KB.calc.CSV)：

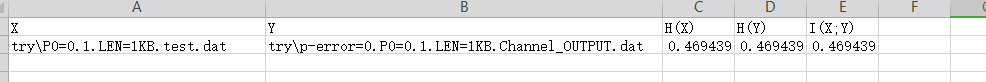


从上图的第六列可以知道计算的信源信息率为0.4700信息比特/秒(保留四位小数)。

再通过程序byteChannel.py生成过信道的序列，如下：



再调用byteChannel\_calc.py的函数进行对其过信道的序列进行信道的输入与的信息熵的计算，如下(文件名：P0=0.1.Channel\_not\_source\_calc\_result.CSV)：



由于取信源的数据率为1数据比特/秒，由于不使用信源编码，因此，信道的数据率为数据比特/s。再与上图中的信息熵分别相乘，即可得到信道的输入与输出的信息率，经过计算可以得到信道的输出信息率=输入信息率=0.4694信息比特/秒(保留四位小数)。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 信源信息率  （信息比特/秒） | 信道数据率  （数据比特/秒） | 信道输入信息率（信息比特/秒） | 信道输出信息率（信息比特/秒） | 误码率 |
| 理论值 | 0.4690 | 0.4690 | 0.4690 | 0.4690 | 0 |
| 仿真值 | 0.4700 | 0.4694 | 0.4694 | 0.4694 | 0 |

**(2).信源概率分布时，使用信源编码：**

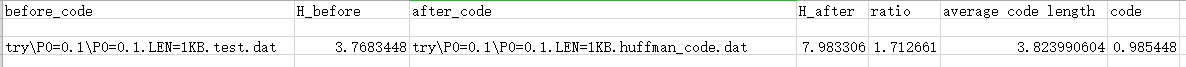
将之前得到的信源文件先进行霍夫曼编码，即通过程序：byteSourceEncoder.py实现，(其中的霍夫曼码书由程序Doublelisan.exe生成)得到使用编码后的信源文件(调用命令见程序API)：



将得到的霍夫曼编码不进行信道编码后，通过信道，即调用程序：byteChannel.py可以获得一个通过信道后的文件：



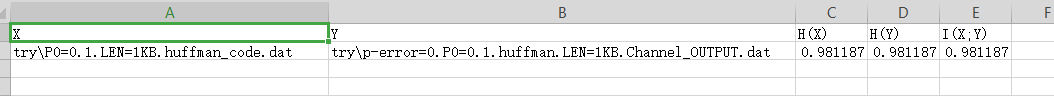
调用程序codding\_effect.py可以计算8次扩展后的所得未编码前的信源信息熵，编码后的信源信息熵(即输入信道前的信息熵)以及输出序列的平均码长：

 根据上图数据，可以计算下列实际值：

仿真的信源信息率：

仿真的信道数据率：

调用程序byteChannel\_calc.py可以得到下列数据(文件：P0=0.1.Channel\_source\_huffman\_calc\_result.csv)：



从上式可得信道输入与输出的信息熵，可以认为是仿真的输入与输出的信道的输入信息率以及信道的输出信息率。

理论推导的结果如下：

信源信息率=

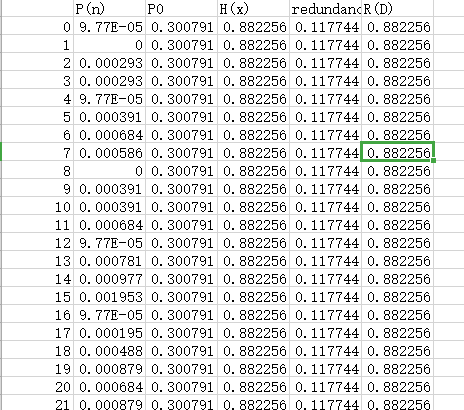
信道数据率=

信道输入信息率=信道输出信息率=信道数据率=0.9812数据比特

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 信源信息率  （信息比特/秒） | 信道数据率  （数据比特/秒） | 信道输入信息率（信息比特/秒） | 信道输出信息率（信息比特/秒） | 误码率 |
| 理论值 | 0.4690 | 0.9812 | 0.9812 | 0.9812 | 0 |
| 仿真值 | 0.4710 | 0.9979 | 0.9979 | 0.9979 | 0 |

**(3).信源概率分布时，不使用信源编码：**

先用程序byteSource.py生成一个概率分布为离散无记忆信源(大小为1KB)，调用程序byteSource\_calc.py可以计算其信源信息率的大小(数据率为1bit/s)(文件名为：P0=0.3.LEN=10KB.calc.CSV)：

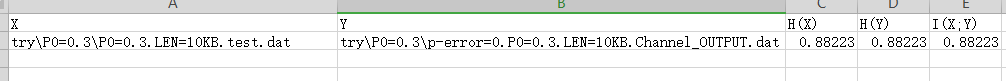


从上图的第六列可以知道计算的信源信息率为0.8823信息比特/秒(保留四位小数)。

再通过程序byteChannel.py生成过信道的序列，如下：



再调用byteChannel\_calc.py的函数进行对其过信道的序列进行信道的输入与的信息熵的计算，如下：



由于取信源的数据率为1数据比特/秒，由于不使用信源编码，因此，信道的数据率为数据比特/s。再与上图中的信息熵分别相乘，即可得到信道的输入与输出的信息率，经过计算可以得到信道的输出信息率=输入信息率=0.8823信息比特/秒(保留四位小数)。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 信源信息率  （信息比特/秒） | 信道数据率  （数据比特/秒） | 信道输入信息率（信息比特/秒） | 信道输出信息率（信息比特/秒） | 误码率 |
| 理论值 | 0.8813 | 0.8813 | 0.8813 | 0.8813 | 0 |
| 仿真值 | 0.8823 | 0.8823 | 0.8823 | 0.8823 | 0 |

**(4).信源概率分布时，使用信源编码：**

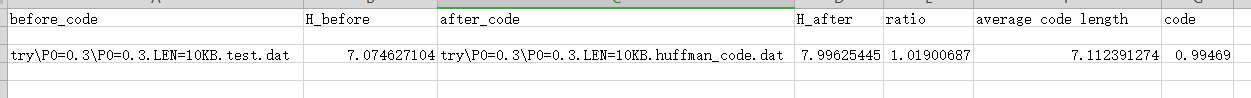
将之前得到的信源文件先进行霍夫曼编码，即通过程序：byteSourceEncoder.py实现，(其中的霍夫曼码书由程序Doublelisan.exe生成)得到使用编码后的信源文件(调用命令见程序API)：



将得到的霍夫曼编码不进行信道编码后，通过信道，即调用程序：byteChannel.py可以获得一个通过信道后的文件：



调用程序codding\_effect.py可以计算8次扩展后的所得未编码前的信源信息熵，编码后的信源信息熵(即输入信道前的信息熵)以及输出序列的平均码长：

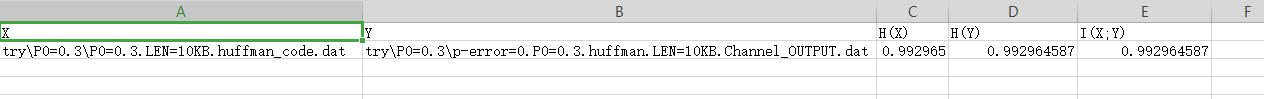


根据上图数据，可以计算下列实际值：

仿真的信源信息率：

仿真的信道数据率：

调用程序byteChannel\_calc.py可以得到下列数据(文件：P0=0.1.Channel\_source\_huffman\_calc\_result.csv)：



从上式可得信道输入与输出的信息熵，可以认为是仿真的输入与输出的信道的输入信息率以及信道的输出信息率。

理论推导的结果如下：

信源信息率=

信道数据率=

信道输入信息率=信道输出信息率=信道数据率=0.9912数据比特

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 信源信息率  （信息比特/秒） | 信道数据率  （数据比特/秒） | 信道输入信息率（信息比特/秒） | 信道输出信息率（信息比特/秒） | 误码率 |
| 理论值 | 0.8813 | 0.9912 | 0.9912 | 0.9912 | 0 |
| 仿真值 | 0.8843 | 0.9995 | 0.9995 | 0.9995 | 0 |

### 分析

从以上4种情况的理论推导与仿真结果可以看出，仿真结果大致与理论推导结果一致，对于相应的仿真值与理论值相比，仿真值较大，原因可能与计算时的保留小数位有关。可以从以上情况看出，对于存在信源编码时，通过无噪无损离散无记忆信道且不经过信道编码的信道输入信息率(输出信息率)要大于信源信息率，原因在于huffman编码为非等长编码，计算时需要考虑平均码长。

## 信道非理想情况下

### 条件设定

信道非理想情况下的固定条件：

信源的信息率：1（信息比特/秒）

信源：等概率二元DMS

信源编码：无

根据选用的可变条件不同，总共有9种情况，即：

1.无信道编码，错误传递概率为0.01

2.无信道编码，错误传递概率为0.2

3.无信道编码，错误传递概率为0.5

4.N长的重复码，N为5，错误传递概率为0.01

5.N长的重复码，N为5，错误传递概率为0.2

6.N长的重复码，N为5，错误传递概率为0.5

7.(7,4)线性分组码，错误传递概率为0.01

8.(7,4)线性分组码，错误传递概率为0.2

9.(7,4)线性分组码，错误传递概率为0.5

### 理论推导

假设信道错误传递概率为，其中，令

首先推导在所给的所有条件下都相同的几个待测指标：

信源的数据率都为1（数据比特/秒）,且信源为等概率二元DMS,则信源的信息率均为

（信息比特/秒），其中r为码符号的个数

即三种信道错误概率下信源的信息率（信息比特/秒）

因为没有经过信源编码，所以信道的数据率（数据比特/秒）

即三种信道错误概率下信道的数据率（数据比特/秒）

接着分情况讨论在可变条件下的待测指标：

情况一，无信道编码：

编码后信道的信息传输率，其中M为输入消息（许用码字）的个数，n是编码后码字的长度（码元的个数），

此时无信道编码，则M=2,n=1,可以得到（信息比特/数据比特）

则信道的输入信息率（信息比特/秒）

即三种信道错误概率下信道的输入信息率为（信息比特/秒）

输出的每个码字包含的信息量和输入的码字包含的信息量是一样的，所以输出信息率等于输入信息率，即（信息比特/秒）

三种信道错误概率下信道的输入信息率为（信息比特/秒）

假设等概率二元DMS中，其中一个的概率为，另一个的概率为，都为0.5

当信道错误概率为，误码率为



则信宿关于信源的信息率（信息比特/秒）

当信道错误概率为，误码率为



则信宿关于信源的信息率（信息比特/秒）

当信道错误概率为，误码率为



则信宿关于信源的信息率（信息比特/秒）

情况二，N长的重复码：N=5：

编码后信道的信息传输率，其中M为输入消息（许用码字）的个数，n是编码后码字的长度（码元的个数），

此时无信道编码，则M=2,n=5,可以得到（信息比特/数据比特）

则信道的输入信息率（信息比特/秒）

即三种信道错误概率下信道的输入信息率为（信息比特/秒）

输出的每个码字包含的信息量和输入的码字包含的信息量是一样的，所以输出信息率等于输入信息率，即（信息比特/秒）

三种信道错误概率下信道的输入信息率为（信息比特/秒）

当信道错误概率为，误码率为



则信宿关于信源的信息率（信息比特/秒）

当信道错误概率为，误码率为



则信宿关于信源的信息率（信息比特/秒）

当信道错误概率为，误码率为



则信宿关于信源的信息率（信息比特/秒）

情况三，(7,4)线性分组码：

编码后信道的信息传输率，其中M为输入消息（许用码字）的个数，n是编码后码字的长度（码元的个数），

此时无信道编码，则M=7,n=16,可以得到（信息比特/数据比特）

则信道的输入信息率（信息比特/秒）

即三种信道错误概率下信道的输入信息率为（信息比特/秒）

输出的每个码字包含的信息量和输入的码字包含的信息量是一样的，所以输出信息率等于输入信息率，即（信息比特/秒）

三种信道错误概率下信道的输入信息率为（信息比特/秒）

当信道错误概率为，误码率为



则信宿关于信源的信息率（信息比特/秒）

当信道错误概率为，误码率为



则信宿关于信源的信息率（信息比特/秒）

当信道错误概率为，误码率为



则信宿关于信源的信息率（信息比特/秒）

### 仿真

运行byteSource.py程序，在cmd中输入命令：

python byteSource.py 0.5 1048576 channel\_not\_ideal\P0=0.5.LEN=1MB.OUTPUT.dat

生成的信源输出消息序列文件如下：



信源指标计算；

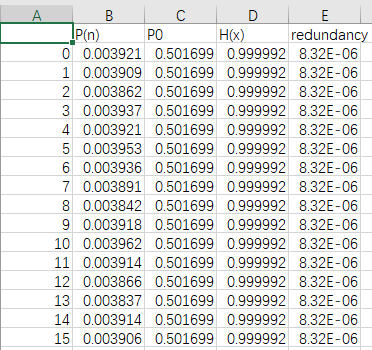
在cmd中输入命令：

python byteSource\_calc.py channel\_not\_ideal\P0=0.5.LEN=1MB.OUTPUT.dat channel\_not\_ideal\P0=0.5.LEN=1MB.calc.CSV

生成的信源指标计算结果如下：



打开如下：

****

不经过信道编码：

1.错误传递概率为0.01

运行byteChannel.py程序，在cmd中输入如下命令：

python byteChannel.py channel\_not\_ideal\P0=0.5.LEN=1MB.OUTPUT.dat 0.01 channel\_not\_ideal\p=0.01.P0=0.5.LEN=1MB.Channel\_OUTPUT.dat

生成的信道输出消息序列文件如下：



2.错误传递概率为0.2

运行byteChannel.py程序，在cmd中输入如下命令：

python byteChannel.py channel\_not\_ideal\P0=0.5.LEN=1MB.OUTPUT.dat 0.2 channel\_not\_ideal\p=0.2.P0=0.5.LEN=1MB.Channel\_OUTPUT.dat

生成的信道输出消息序列文件如下：



3.错误传递概率为0.5

运行byteChannel.py程序，在cmd中输入如下命令：

python byteChannel.py channel\_not\_ideal\P0=0.5.LEN=1MB.OUTPUT.dat 0.5 channel\_not\_ideal\p=0.5.P0=0.5.LEN=1MB.Channel\_OUTPUT.dat

生成的信道输出消息序列文件如下：



信道编码指标的计算：

把不同错误传递概率的指标计算的结果写入到同一个CSV文件中，便于比较，即写入P0=0.5.LEN=1MB.Channel\_calc\_result.CSV文件中

1.错误传递概率为0.01

运行byteChannel\_calc.py程序，将在cmd中输入如下命令：

python byteChannel\_calc.py channel\_not\_ideal\P0=0.5.LEN=1MB.OUTPUT.dat channel\_not\_ideal\p=0.01.P0=0.5.LEN=1MB.Channel\_OUTPUT.dat channel\_not\_ideal\P0=0.5.LEN=1MB.Channel\_calc\_result.CSV

2.错误传递概率为0.2

运行byteChannel\_calc.py程序，将在cmd中输入如下命令：

python byteChannel\_calc.py channel\_not\_ideal\P0=0.5.LEN=1MB.OUTPUT.dat channel\_not\_ideal\p=0.2.P0=0.5.LEN=1MB.Channel\_OUTPUT.dat channel\_not\_ideal\P0=0.5.LEN=1MB.Channel\_calc\_result.CSV

3.错误传递概率为0.5

运行byteChannel\_calc.py程序，将在cmd中输入如下命令：

python byteChannel\_calc.py channel\_not\_ideal\P0=0.5.LEN=1MB.OUTPUT.dat channel\_not\_ideal\p=0.5.P0=0.5.LEN=1MB.Channel\_OUTPUT.dat channel\_not\_ideal\P0=0.5.LEN=1MB.Channel\_calc\_result.CSV

运行结果如下



N长的重复码，N为5：

在cmd中输入如下命令进行N长为5的重复码编码：

python channelEncoder.py -r channel\_not\_ideal\P0=0.5.LEN=1MB.OUTPUT.dat channel\_not\_ideal\P0=0.5.LEN=1MB.N=5.reENCode.dat 5

生成的经过N长为5的重复码编码文件如下：



(7,4)线性分组码：

在cmd中输入如下命令进行(7,4)线性分组码的编码：

python channelEncoder.py -l channel\_not\_ideal\P0=0.5.LEN=1MB.OUTPUT.dat channel\_not\_ideal\P0=0.5.LEN=1MB.7.4.linearENCode.dat 3

生成的经过(7,4)线性分组码的编码文件如下



N长为5的重复码编码，通过不同错误传递概率的信道：

1.错误传递概率为0.01

运行byteChannel.py程序，在cmd中输入如下命令：

python byteChannel.py channel\_not\_ideal\P0=0.5.LEN=1MB.N=5.reENCode.dat 0.01 channel\_not\_ideal\p=0.01.P0=0.5.LEN=1MB.N=5.reENCode.Channel\_OUTPUT.dat

生成的信道输出消息序列文件如下：



在cmd中输入如下命令，利用解码器对得到的文件进行解码：

python channelDecoder.py channel\_not\_ideal\p=0.01.P0=0.5.LEN=1MB.N=5.reENCode.Channel\_OUTPUT.dat channel\_not\_ideal\p=0.01.P0=0.5.LEN=1MB.N=5.reDECode.dat

解码后得到的文件如下：



2.错误传递概率为0.2

运行byteChannel.py程序，在cmd中输入如下命令：

python byteChannel.py channel\_not\_ideal\P0=0.5.LEN=1MB.N=5.reENCode.dat 0.2 channel\_not\_ideal\p=0.2.P0=0.5.LEN=1MB.N=5.reENCode.Channel\_OUTPUT.dat

生成的信道输出消息序列文件如下：



在cmd中输入如下命令，利用解码器对得到的文件进行解码：

python channelDecoder.py channel\_not\_ideal\p=0.2.P0=0.5.LEN=1MB.N=5.reENCode.Channel\_OUTPUT.dat channel\_not\_ideal\p=0.2.P0=0.5.LEN=1MB.N=5.reDECode.dat

解码后得到的文件如下：



3.错误传递概率为0.5

运行byteChannel.py程序，在cmd中输入如下命令：

python byteChannel.py channel\_not\_ideal\P0=0.5.LEN=1MB.N=5.reENCode.dat 0.2 channel\_not\_ideal\p=0.5.P0=0.5.LEN=1MB.N=5.reENCode.Channel\_OUTPUT.dat

生成的信道输出消息序列文件如下：



在cmd中输入如下命令，利用解码器对得到的文件进行解码：

python channelDecoder.py channel\_not\_ideal\p=0.5.P0=0.5.LEN=1MB.N=5.reENCode.Channel\_OUTPUT.dat channel\_not\_ideal\p=0.5.P0=0.5.LEN=1MB.N=5.reDECode.dat

这里错误传递概率为0.5时,解码解不出来,可能是信道的干扰太严重了(虽然信道编码的文件头已经用N=9的重复码保护了)

（7，4）线性分组码，经过不同错误传递概率的信道

1.错误传递概率为0.01

运行byteChannel.py程序，在cmd中输入如下命令：

python byteChannel.py channel\_not\_ideal\P0=0.5.LEN=1MB.7.4.linearENCode.dat 0.01 channel\_not\_ideal\p=0.01.P0=0.5.LEN=1MB.7.4.linearENCode.Channel\_OUTPUT.dat

生成的信道输出消息序列文件如下：



在cmd中输入如下命令，利用解码器对得到的文件进行解码：

python channelDecoder.py channel\_not\_ideal\p=0.01.P0=0.5.LEN=1MB.7.4.linearENCode.Channel\_OUTPUT.dat channel\_not\_ideal\p=0.01.P0=0.5.LEN=1MB.7.4.linearDECode.dat

解码后得到的文件如下：



2.错误传递概率为0.2

运行byteChannel.py程序，在cmd中输入如下命令：

python byteChannel.py channel\_not\_ideal\P0=0.5.LEN=1MB.7.4.linearENCode.dat 0.2 channel\_not\_ideal\p=0.2.P0=0.5.LEN=1MB.7.4.linearENCode.Channel\_OUTPUT.dat

生成的信道输出消息序列文件如下：



在cmd中输入如下命令，利用解码器对得到的文件进行解码：

python channelDecoder.py channel\_not\_ideal\p=0.2.P0=0.5.LEN=1MB.7.4.linearENCode.Channel\_OUTPUT.dat channel\_not\_ideal\p=0.2.P0=0.5.LEN=1MB.7.4.linearDECode.dat

解码后得到的文件如下：



3. 错误传递概率为0.5

运行byteChannel.py程序，在cmd中输入如下命令：

python byteChannel.py channel\_not\_ideal\P0=0.5.LEN=1MB.7.4.linearENCode.dat 0.5 channel\_not\_ideal\p=0.5.P0=0.5.LEN=1MB.7.4.linearENCode.Channel\_OUTPUT.dat

生成的信道输出消息序列文件如下：



在cmd中输入如下命令，利用解码器对得到的文件进行解码：

python channelDecoder.py channel\_not\_ideal\p=0.5.P0=0.5.LEN=1MB.7.4.linearENCode.Channel\_OUTPUT.dat channel\_not\_ideal\p=0.5.P0=0.5.LEN=1MB.7.4.linearDECode.dat

解码后得到的文件如下



1M的信源文件，经过(7,4)线性分组码的信道编码,进入错误传递概率为0.5的信道,解码出来只有66kB,可见受到的干扰比较严重(信道编码文件头是用N=9的重复码保护的)

整理上述结果可以得到

无信道编码时：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 无信道编码时理论和实际结果 | | | | | | |
| 错误传递概率p | 数值 | 信源信息率（信息比特/秒） | 信道数据率（数据比特/秒） | 信道输入信息率（信息比特/秒） | 信道输出信息率（信息比特/秒） | 误码率 |
| p=0.01 | 理论值 | 1 | 1 | 1 | 1 | 0.01 |
| 实际值 | 0.999992 | 0.999992 | 0.999992 | 0.999992 | 0.019908 |
| p=0.2 | 理论值 | 1 | 1 | 1 | 1 | 0.2 |
| 实际值 | 0.999992 | 0.999992 | 0.999992 | 0.999992 | 0.283507 |
| p=0.5 | 理论值 | 1 | 1 | 1 | 1 | 0.5 |
| 实际值 | 0.999992 | 0.999992 | 0.999992 | 0.999992 | 0.5656 |

N长的重复码，N=5时：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| N长的重复码，N=5时理论和实际结果 | | | | | | |
| 错误传递概率p | 数值 | 信源信息率（信息比特/秒） | 信道数据率（数据比特/秒） | 信道输入信息率（信息比特/秒） | 信道输出信息率（信息比特/秒） | 误码率 |
| p=0.01 | 理论值 | 1 | 0.2 | 0.2 | 0.2 | 9.8506e-6 |
| 实际值 | 0.999992 | 0.179543 | 0.179543 | 0.179543 | 0.000319 |
| p=0.2 | 理论值 | 1 | 0.2 | 0.2 | 0.2 | 0.05792 |
| 实际值 | 0.999992 | 0.179543 | 0.179543 | 0.179543 | 0.0699 |
| p=0.5 | 理论值 | 1 | 0.2 | 0.2 | 0.2 | 0.5 |
| 实际值 | 0.999992 | 0.179543 | 0.179543 | -------- | ------- |

上表中没有数据的空是因为错误传递概率在高到0.5的时候，信道解码解不出来（文件头已经用N=9的重复码保护了，但还是生成不了解码文件），这和理论上其他条件不变情况下，误码率随着错误传递概率的增加而增加是符合的。

（7，4）线性分组码时：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| （7，4）线性分组码时理论和实际结果 | | | | | | |
| 错误传递概率p | 数值 | 信源信息率（信息比特/秒） | 信道数据率（数据比特/秒） | 信道输入信息率（信息比特/秒） | 信道输出信息率（信息比特/秒） | 误码率 |
| p=0.01 | 理论值 | 1 | 0.571429 | 0.571429 | 0.571429 | 0.002031 |
| 实际值 | 0.999992 | 0.429698 | 0.429698 | 0.429698 | 0.003945 |
| p=0.2 | 理论值 | 1 | 0.571429 | 0.571429 | 0.571429 | 0.423283 |
| 实际值 | 0.999992 | 0.429698 | 0.429698 | 0.429698 | 0.537574 |
| p=0.5 | 理论值 | 1 | 0.571429 | 0.571429 | 0.571429 | 0.9375 |
| 实际值 | 0.999992 | 0.429698 | 0.429698 | ------- | ------- |

上表中没有填写数据的空是因为,在错误传递概率为0.5时,即使有了信道编码(信道编码的文件头用N=9的重复码保护),但是1M的信源文件经过这一系列条件最终解码出来只有66kB,我觉得用这个文件算误码率已经没有价值了(干扰太严重了)

### 分析

对于上述的仿真以及比较,基本符合理论推导,但是在信道错误传递概率比较高的时候(比如等于0.5),有经过信道编码,而没有信源编码的情况下,无论是N=5的重复码的信道编码还是(7,4)线性分组码的信道编码,都会出现不能正确信道解码的情况(信道编码的文件头已经用N=9的重复码保护了)。

我觉得原因可能是：

1.文件头的保护还不够到位,可能要换用更长的N的重复码去保护文件头，或者换用其它的编码方式来进行保护。

2.可能是程序本身有问题，信道程序对于噪声的加入还不是很完善。

## 一般非理想情况下

### 条件设定

合理观察，列出了以下可变条件以实现对每一环节的比较

拟计算：

（1）信源概率分布0.1，信道错误传递概率0.01，无信源编码，无信道编码

（2）信源概率分布0.1，信道错误传递概率0.01，无信源编码，重复码码长5

（3）信源概率分布0.1，信道错误传递概率0.01，无信源编码，(7,4)线性分组码

（4）信源概率分布0.1，信道错误传递概率0.5，无信源编码，无信道编码

（5）信源概率分布0.1，信道错误传递概率0.2，无信源编码，重复码码长5

（6）信源概率分布0.1，信道错误传递概率0.2，无信源编码，(7,4)线性分组码

（7）信源概率分布0.1，信道错误传递概率0.01，有信源编码，无信道编码

（8）信源概率分布0.1，信道错误传递概率0.01，有信源编码，重复码码长5

（9）信源概率分布0.1，信道错误传递概率0.01，有信源编码，(7,4)线性分组码

### 理论推导

假设信源的概率分布的概率空间为，则其信息熵(信息比特/符号)为：

由于固定条件中给出信源的数据率，而在这个例子中信源符号为0或1，一个符号就是一个数据比特，所以信源信息率为：

**（1）信源概率分布0.1，信道错误传递概率0.01，无信源编码，无信道编码**

信源概率分布，不使用信源编码，不使用信道编码，信道错误传递概率0.01。

由于信源概率分布，信息熵为：

因为信源的数据率，代入公式，计算得：

由于使用的是无信源编码，所以信道的输入信息率等于信源的信息率

由于信道错误传递概率0.01，对于二元对称信道来说即：

因此信道传输率为

所以得到信道的数据率

信道的输出信息率等于：

信宿关于信源的信息率,这里

所以信宿关于信源的信息率

由于不使用信道编码，信道错误传递概率为0.01

所以信宿的误码率

因此理论推导结果为：

a.信源的信息率

b.信道的数据率，其中

c.信道的输入信息率

d.信道的输出信息率

e. 信宿关于信源的信息率

f. 信宿的误码率.01

**（2）信源概率分布0.1，信道错误传递概率0.01，无信源编码，重复码码长5**

信源概率分布，不使用信源编码，使用信道编码，信道错误传递概率0.01。

由于信源概率分布，信息熵为：

因为信源的数据率，代入公式，计算得：

由于使用的是无信源编码，但却使用了信道编码，重复码码长为5，所以信道的输入信息率等于信源的信息率乘以信道编码后的信息传输率

由于信道错误传递概率0.01，对于二元对称信道来说即：

信道重复码编码后即编码后0、1的概率分布不变

因此信道传输率为

所以得到信道的数据率

信道的输出信息率等于：

信宿关于信源的信息率,这里

所以信宿关于信源的信息率

由于使用信道编码，重复码长为5，信道错误传递概率为9.8506×

所以信宿的误码率

因此理论推导结果为：

a.信源的信息率

b.信道的数据率，其中

c.信道的输入信息率

d.信道的输出信息率

e. 信宿关于信源的信息率

f. 信宿的误码率

**（3）信源概率分布0.1，信道错误传递概率0.01，无信源编码，(7,4)线性分组码**

信源概率分布，不使用信源编码，使用信道编码，信道错误传递概率0.01。

由于信源概率分布，信息熵为：

因为信源的数据率，代入公式，计算得：

由于使用的是无信源编码，但却使用了(7,4)线性分组码，n=4，N=7，所以信道编码后的信源信息传输率

由于信道错误传递概率0.01，对于二元对称信道来说即：

信道编码前有

因此信道编码前信道传输率为

所以得到编码后信道的数据率

信道的输出信息率等于：

信宿关于信源的信息率,这里

所以信宿关于信源的信息率

由于使用信道编码，(7,4)线性分组码，信道错误传递概率为

所以信宿的误码率

因此理论推导结果为：

a.信源的信息率

b.信道的数据率，其中

c.信道的输入信息率

d.信道的输出信息率

e. 信宿关于信源的信息率

f. 信宿的误码率

**（4）信源概率分布0.1，信道错误传递概率0.5，无信源编码，无信道编码**

信源概率分布，不使用信源编码，不使用信道编码，信道错误传递概率0.5。

由于信源概率分布，信息熵为：

因为信源的数据率，代入公式，计算得：

由于使用的是无信源编码，所以信道的输入信息率等于信源的信息率

由于信道错误传递概率0.5，对于二元对称信道来说即：

因此信道传输率为

所以得到信道的数据率

信道的输出信息率等于：

信宿关于信源的信息率,这里

所以信宿关于信源的信息率

由于不使用信道编码，信道错误传递概率为0.5

所以信宿的误码率

因此理论推导结果为：

a.信源的信息率

b.信道的数据率，其中

c.信道的输入信息率

d.信道的输出信息率

e. 信宿关于信源的信息率

f. 信宿的误码率.5

**（5）信源概率分布0.1，信道错误传递概率0.2，无信源编码，重复码码长5**

信源概率分布，不使用信源编码，使用信道编码，信道错误传递概率0.2。

由于信源概率分布，信息熵为：

因为信源的数据率，代入公式，计算得：

由于使用的是无信源编码，但却使用了信道编码，重复码码长为5，所以信道的输入信息率等于信源的信息率乘以信道编码后的信息传输率

由于信道错误传递概率0.2，对于二元对称信道来说即：

信道重复码编码后即编码后0、1的概率分布不变

因此信道传输率为

所以得到信道的数据率

信道的输出信息率等于：

信宿关于信源的信息率,这里

所以信宿关于信源的信息率

由于使用信道编码，重复码长为5，信道错误传递概率为0.05792

所以信宿的误码率

因此理论推导结果为：

a.信源的信息率

b.信道的数据率，其中

c.信道的输入信息率

d.信道的输出信息率

e. 信宿关于信源的信息率

f. 信宿的误码率

**（6）信源概率分布0.1，信道错误传递概率0.2，无信源编码，(7,4)线性分组码**

信源概率分布，不使用信源编码，使用信道编码，信道错误传递概率0.2。

由于信源概率分布，信息熵为：

因为信源的数据率，代入公式，计算得：

由于使用的是无信源编码，但却使用了(7,4)线性分组码，n=4，N=7，所以信道编码后的信源信息传输率

由于信道错误传递概率0.2，对于二元对称信道来说即：

信道编码前有

因此信道编码前信道传输率为

所以得到编码后信道的数据率

信道的输出信息率等于：

信宿关于信源的信息率,这里

所以信宿关于信源的信息率

由于使用信道编码，(7,4)线性分组码，信道错误传递概率为

所以信宿的误码率

因此理论推导结果为：

a.信源的信息率

b.信道的数据率，其中

c.信道的输入信息率

d.信道的输出信息率

e. 信宿关于信源的信息率

f. 信宿的误码率

**（7）信源概率分布0.1，信道错误传递概率0.01，有信源编码，无信道编码**

信源概率分布，不使用信源编码，使用信道编码，信道错误传递概率0.2。

由于信源概率分布，信息熵为：

因为信源的数据率，代入公式，计算得：

由于采用了霍夫曼编码，信源实际上被压缩，信息率提高，所以信道的输入信息率会大于等于信源的信息率。

编码后输入信道的信息率即编码后平均每个码符号所携带的信息量：

,此处N为文件字节长度即8，为编码后的平均码字长度

因为霍夫曼码是在信源给定情况下的最佳码，所以其平均码长的界限为

因此我们可以认为，在概率P0=p下，理想状态下的理论霍夫曼平均码长为Hr(S)，此处H(S)是概率P0=p情况下的二元离散无记忆信息熵，而r一般取2，意为二元码的霍夫曼编码平均码长的界限，由于文件存储格式为byte，即8个bit，可认为是二元码的8次拓展，所以计算时Hr(S)=8×H(S)===3.736

所以编码后信道的输入信息率

由于信道错误传递概率0.01，对于二元对称信道来说即：

信源编码后即编码后0、1的概率分布相当于等概率分布

因此信道传输率为

所以得到信道的数据率

信道的输出信息率等于：

信宿关于信源的信息率,这里

所以信宿关于信源的信息率

由于不使用信道编码，信道错误传递概率为0.01

所以信宿的误码率

因此理论推导结果为：

a.信源的信息率

b.信道的数据率，其中

c.信道的输入信息率

d.信道的输出信息率

e. 信宿关于信源的信息率

f. 信宿的误码率

**（8）信源概率分布0.1，信道错误传递概率0.01，有信源编码，重复码码长5**

信源概率分布，不使用信源编码，使用信道编码，信道错误传递概率0.2。

由于信源概率分布，信息熵为：

因为信源的数据率，代入公式，计算得：

由于采用了霍夫曼编码，信源实际上被压缩，信息率提高，所以信道的输入信息率会大于等于信源的信息率。

编码后输入信道的信息率即编码后平均每个码符号所携带的信息量：

,此处N为文件字节长度即8，为编码后的平均码字长度

因为霍夫曼码是在信源给定情况下的最佳码，所以其平均码长的界限为

因此我们可以认为，在概率P0=p下，理想状态下的理论霍夫曼平均码长为Hr(S)，此处H(S)是概率P0=p情况下的二元离散无记忆信息熵，而r一般取2，意为二元码的霍夫曼编码平均码长的界限，由于文件存储格式为byte，即8个bit，可认为是二元码的8次拓展，所以计算时Hr(S)=8×H(S)===3.736

因为使用了信道编码，重复码码长为5，所以信道的输入信息率等于信源的信息率乘以信道编码后的信息传输率

所以编码后信道的输入信息率

由于信道错误传递概率0.01，对于二元对称信道来说即：

信源编码后变为等概率分布，信道重复码编码后即编码后0、1的概率分布不变

因此信道传输率为

所以得到信道的数据率

信道的输出信息率等于：

信宿关于信源的信息率,这里

所以信宿关于信源的信息率

由于使用信道编码，重复码长为5，信道错误传递概率为

所以信宿的误码率

因此理论推导结果为：

a.信源的信息率

b.信道的数据率，其中

c.信道的输入信息率

d.信道的输出信息率

e. 信宿关于信源的信息率

f. 信宿的误码率

**（9）信源概率分布0.1，信道错误传递概率0.01，有信源编码，(7,4)线性分组码**

信源概率分布，不使用信源编码，使用信道编码，信道错误传递概率0.2。

由于信源概率分布，信息熵为：

因为信源的数据率，代入公式，计算得：

由于采用了霍夫曼编码，信源实际上被压缩，信息率提高，所以信道的输入信息率会大于等于信源的信息率。

编码后输入信道的信息率即编码后平均每个码符号所携带的信息量：

,此处N为文件字节长度即8，为编码后的平均码字长度

因为霍夫曼码是在信源给定情况下的最佳码，所以其平均码长的界限为

因此我们可以认为，在概率P0=p下，理想状态下的理论霍夫曼平均码长为Hr(S)，此处H(S)是概率P0=p情况下的二元离散无记忆信息熵，而r一般取2，意为二元码的霍夫曼编码平均码长的界限，由于文件存储格式为byte，即8个bit，可认为是二元码的8次拓展，所以计算时Hr(S)=8×H(S)===3.736

由于使用了(7,4)线性分组码，n=4，N=7，所以信道编码后的信源信息传输率

由于信道错误传递概率0.01，对于二元对称信道来说即：

信道编码前有

因此信道编码前信道传输率为

所以得到编码后信道的数据率

信道的输出信息率等于：

信宿关于信源的信息率,这里

所以信宿关于信源的信息率

由于使用信道编码，(7,4)线性分组码，信道错误传递概率为

所以信宿的误码率

因此理论推导结果为：

a.信源的信息率

b.信道的数据率，其中

c.信道的输入信息率

d.信道的输出信息率

e. 信宿关于信源的信息率

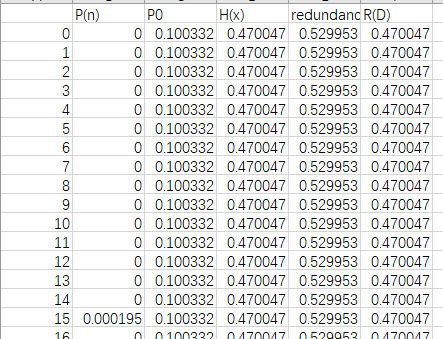
f. 信宿的误码率

### 仿真

**（1）信源概率分布0.1，信道错误传递概率0.01，无信源编码，无信道编码**

- 先用byteSource.exe生成一个概率分布为的离散无记忆信源（大小为10KB）



- 然后调用byteSource\_calc.exe程序计算信源文件的信息量大小

此处第四列显示信源的信息量为H(S)=0.470(信息比特/符号)，所以信源信息率

- 再通过程序byteChannel.py生成过信道的序列，错误概率为0.01



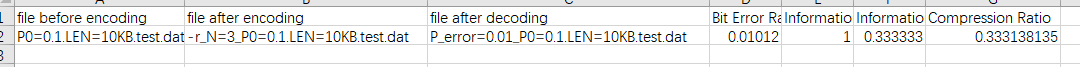
- 再调用byteChannel\_calc.exe程序进行对其过信道的序列进行信道的输入与输出的信息熵的计算



由于信源的数据率为1数据比特/秒 且不使用信源编码，因此实际的信道数据率为，其中

由于程序API要求，程序存在缺陷，若要使用channelCoder\_calc.exe计算误码率，则需先生成一个信道编码后的文件同信源文件一齐输入才能与经过错误传递后的文件进行比较，无奈只能先调用channelEncoder.exe生成重复码码长为5的信道编码文件。



- 使用channelCoder\_calc.exe计算误码率

计算得误码率为0.01012

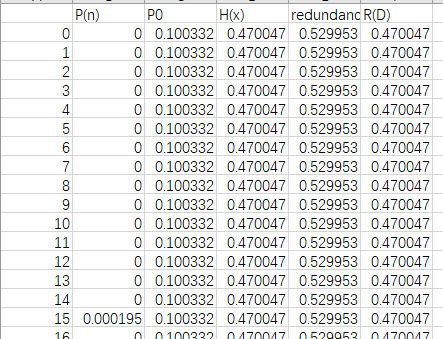
比较结果如下：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 信源信息率  （信息比特/秒） | 信道数据率  （数据比特/秒） | 信道输入信息率（信息比特/秒） | 信道输出信息率（信息比特/秒） | 误码率 |
| 理论值 | 0.467 | 0.413 | 0.467 | 0.493 | 0.01 |
| 仿真值 |  | 0.416 | 0.469 | 0.494 | 0.01 |

**（2）信源概率分布0.1，信道错误传递概率0.01，无信源编码，重复码码长5**

- 先用byteSource.exe生成一个概率分布为的离散无记忆信源（大小为10KB）



- 然后调用byteSource\_calc.exe程序计算信源文件的信息量大小

此处第四列显示信源的信息量为H(S)=0.470(信息比特/符号)，所以信源信息率

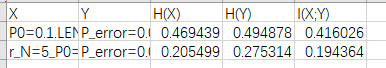
- 使用channelEncoder.exe 生成信道编码后的文件，重复码编码，码长为5



- 再通过程序byteChannel.exe生成过信道的序列，错误概率为0.01



- 再调用byteChannel\_calc.exe程序进行对其过信道的序列进行信道的输入与输出的信息熵的计算



这里看到程序计算编码前和编码后信息量不一致，因为重复码使得文件的字节排列顺序被打乱，计算出错但是我们能从这个结果中算出正确的

运用这个结果我们可以算出仿真的信道输出和信道传输率

- 使用channelDecoder.exe解码错误传递后的文件，生成解码后文件



- 使用channelCoder\_calc.exe计算误码率



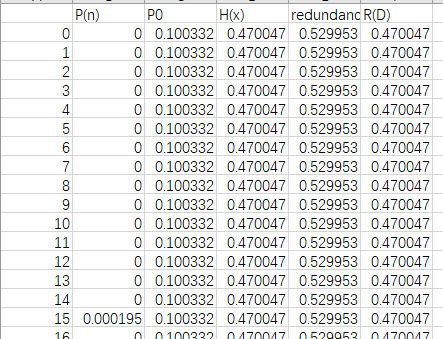
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 信源信息率  （信息比特/秒） | 信道数据率  （数据比特/秒） | 信道输入信息率（信息比特/秒） | 信道输出信息率（信息比特/秒） | 误码率 |
| 理论值 | 0.467 |  |  |  | 0.01 |
| 仿真值 |  |  |  | 0.0987 | 0 |

**（3）信源概率分布0.1，信道错误传递概率0.01，无信源编码，(7,4)线性分组码**

- 先用byteSource.exe生成一个概率分布为的离散无记忆信源（大小为10KB）



- 然后调用byteSource\_calc.exe程序计算信源文件的信息量大小



此处第四列显示信源的信息量为H(S)=0.470(信息比特/符号)，所以信源信息率

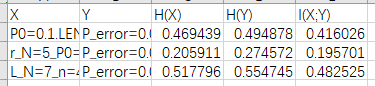
- 使用channelEncoder.exe 生成信道编码后的文件，（7，4）线性分组码



- 再通过程序byteChannel.exe生成过信道的序列，错误概率为0.01



- 再调用byteChannel\_calc.exe程序进行对其过信道的序列进行信道的输入与输出的信息熵的计算



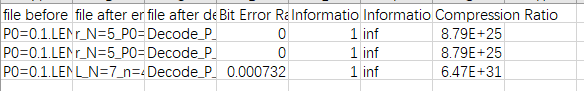
第三行可见，由于线性分组码改变了文件的0、1分布，因此信息熵发生改变,而且byteChannel\_calc.exe程序是以字节为单位来计算信息熵的，线性分组码打乱了原来的字节序，所以这样计算的结果对仿真没有什么意义，但是我们能从这个结果中算出正确的

以及输入输出信息量运用这个结果我们可以算出仿真的信道输出和信道传输率.

- 使用channelDecoder.exe解码错误传递后的文件，生成解码后文件



- 使用channelCoder\_calc.exe计算误码率



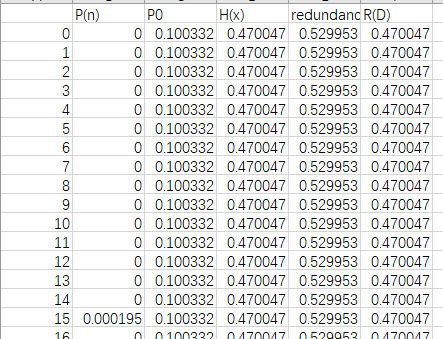
第三行可见误码率为0.000732

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 信源信息率  （信息比特/秒） | 信道数据率  （数据比特/秒） | 信道输入信息率（信息比特/秒） | 信道输出信息率（信息比特/秒） | 误码率 |
| 理论值 | 0.467 |  |  |  |  |
| 仿真 |  |  |  |  | 0.000732 |

**（4）信源概率分布0.1，信道错误传递概率0.5，无信源编码，无信道编码**

- 先用byteSource.exe生成一个概率分布为的离散无记忆信源（大小为10KB）



- 然后调用byteSource\_calc.exe程序计算信源文件的信息量大小

此处第四列显示信源的信息量为H(S)=0.470(信息比特/符号)，所以信源信息率

- 再通过程序byteChannel.py生成过信道的序列，错误概率为0.5



- 再调用byteChannel\_calc.exe程序进行对其过信道的序列进行信道的输入与输出的信息熵的计算

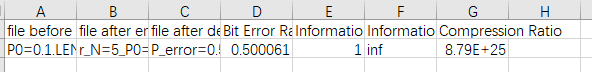


由于信源的数据率为1数据比特/秒 且不使用信源编码，因此实际的信道数据率为，其中

由于程序API要求，程序存在缺陷，若要使用channelCoder\_calc.exe计算误码率，则需先生成一个信道编码后的文件同信源文件一齐输入才能与经过错误传递后的文件进行比较，无奈只能先调用channelEncoder.exe生成重复码码长为5的信道编码文件。



- 使用channelCoder\_calc.exe计算误码率



计算得误码率为0.5

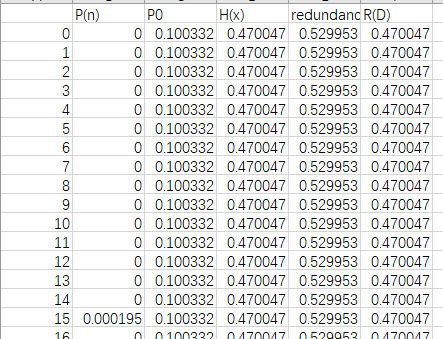
比较结果如下：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 信源信息率  （信息比特/秒） | 信道数据率  （数据比特/秒） | 信道输入信息率（信息比特/秒） | 信道输出信息率（信息比特/秒） | 误码率 |
| 理论值 | 0.467 | 0 | 0.467 | 1 | 0.5 |
| 仿真值 |  | 0.09 | 0.469 | 0.9975 | 0.5 |

**（5）信源概率分布0.1，信道错误传递概率0.2，无信源编码，重复码码长5**

- 先用byteSource.exe生成一个概率分布为的离散无记忆信源（大小为10KB）



- 然后调用byteSource\_calc.exe程序计算信源文件的信息量大小

此处第四列显示信源的信息量为H(S)=0.470(信息比特/符号)，所以信源信息率

- 使用channelEncoder.exe 生成信道编码后的文件，重复码编码，码长为5



- 再通过程序byteChannel.exe生成过信道的序列，错误概率为0.2

- 再调用byteChannel\_calc.exe程序进行对其过信道的序列进行信道的输入与输出的信息熵的计算



这里看到程序计算编码前和编码后信息量不一致，因为重复码使得文件的字节排列顺序被打乱，计算出错但是我们能从这个结果中算出正确的

运用这个结果我们可以算出仿真的信道输出和信道传输率.

- 使用channelDecoder.exe解码错误传递后的文件，生成解码后文件

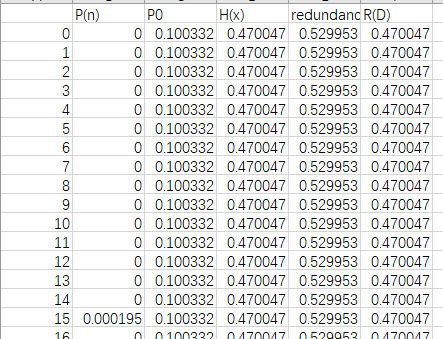
发现，错误概率0.2对我们的程序来说太大了，虽然对头文件使用了长度为9的重复码编码，但仍然会改变我们的头文件，使得解码错误，因此程序无法胜任。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 信源信息率  （信息比特/秒） | 信道数据率  （数据比特/秒） | 信道输入信息率（信息比特/秒） | 信道输出信息率（信息比特/秒） | 误码率 |
| 理论值 | 0.467 |  |  |  |  |
| 仿真值 |  |  |  | 0.0987 | 未知 |

**（6）信源概率分布0.1，信道错误传递概率0.2，无信源编码，(7,4)线性分组码**

- 先用byteSource.exe生成一个概率分布为的离散无记忆信源（大小为10KB）



- 然后调用byteSource\_calc.exe程序计算信源文件的信息量大小

此处第四列显示信源的信息量为H(S)=0.470(信息比特/符号)，所以信源信息率

- 使用channelEncoder.exe 生成信道编码后的文件，（7，4）线性分组码



- 再通过程序byteChannel.exe生成过信道的序列，错误概率为0.2



- 再调用byteChannel\_calc.exe程序进行对其过信道的序列进行信道的输入与输出的信息熵的计算



可见，由于线性分组码改变了文件的0、1分布，因此信息熵发生改变,而且byteChannel\_calc.exe程序是以字节为单位来计算信息熵的，线性分组码打乱了原来的字节序，所以这样计算的结果对仿真没有什么意义，但是我们能从这个结果中算出正确的

以及输入输出信息量运用这个结果我们可以算出仿真的信道输出和信道传输率。

- 使用channelDecoder.exe解码错误传递后的文件，生成解码后文件

发现，错误概率0.2对我们的程序来说太大了，虽然对头文件使用了长度为9的重复码编码，但仍然会改变我们的头文件，使得解码错误，因此程序无法胜任。

- 但我们仍能进行错误概率为0.1的编码译码，此处进行测试

- 通过程序byteChannel.exe生成过信道的序列，错误概率为0.1



- 使用channelDecoder.exe解码错误传递后的文件，生成解码后文件



- 使用channelCoder\_calc.exe计算误码率

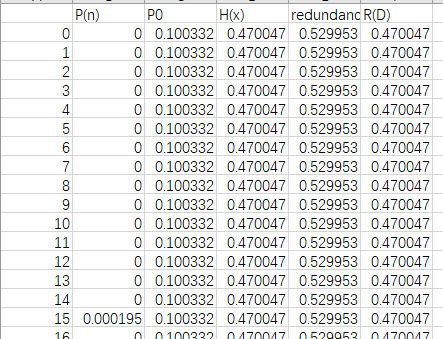


|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 信源信息率  （信息比特/秒） | 信道数据率  （数据比特/秒） | 信道输入信息率（信息比特/秒） | 信道输出信息率（信息比特/秒） | 误码率 |
| 理论值 | 0.467 |  |  |  |  |
| 仿真值 |  |  |  | 0.427 | 0.067  (p=0.1) |

**（7）信源概率分布0.1，信道错误传递概率0.01，有信源编码，无信道编码**

- 先用byteSource.exe生成一个概率分布为的离散无记忆信源（大小为10KB）



- 然后调用byteSource\_calc.exe程序计算信源文件的信息量大小

此处第四列显示信源的信息量为H(S)=0.470(信息比特/符号)，所以信源信息率

- 利用byteSourceEncoder.exe对信源进行编码



- 利用codding\_effect.exe程序计算信源编码指标



- 将编码后文件通过错误信道，错误概率为0.01



- 使用byteChannel\_calc.exe程序进行对其过信道的序列进行信道的输入与输出的信息熵的计算



- 使用channelCoder\_calc.exe计算误码率

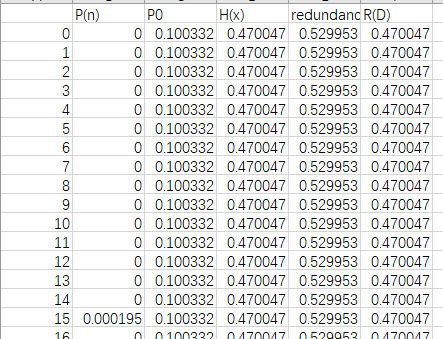


|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 信源信息率  （信息比特/秒） | 信道数据率  （数据比特/秒） | 信道输入信息率（信息比特/秒） | 信道输出信息率（信息比特/秒） | 误码率 |
| 理论值 | 0.467 |  |  |  |  |
| 仿真值 |  |  |  | 0.982502 | 0.01 |

**（8）信源概率分布0.1，信道错误传递概率0.01，有信源编码，重复码码长5**

- 先用byteSource.exe生成一个概率分布为的离散无记忆信源（大小为10KB）



- 然后调用byteSource\_calc.exe程序计算信源文件的信息量大小

此处第四列显示信源的信息量为H(S)=0.470(信息比特/符号)，所以信源信息率

- 利用byteSourceEncoder.exe对信源进行编码



- 利用codding\_effect.exe程序计算信源编码指标



- 使用channelEncoder.exe生成信道编码后文件，重复码长为5

- 将信道编码后文件通过错误信道，错误概率为0.01

- 使用byteChannel\_calc.exe程序进行对其过信道的序列进行信道的输入与输出的信息熵的计算



这里看到程序计算编码前和编码后信息量不一致，因为重复码使得文件的字节排列顺序被打乱，计算出错但是我们能从这个结果中算出正确的

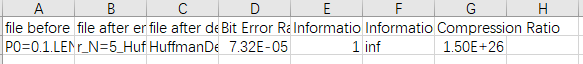
运用这个结果我们可以算出仿真的信道输出和信道传输率.

- 使用channelDecoder.exe解码错误传递后的文件，生成解码后文件



- 使用byteSourceDecoder.exe解码霍夫曼编码后文件

- 使用channelCoder\_calc.exe计算误码率



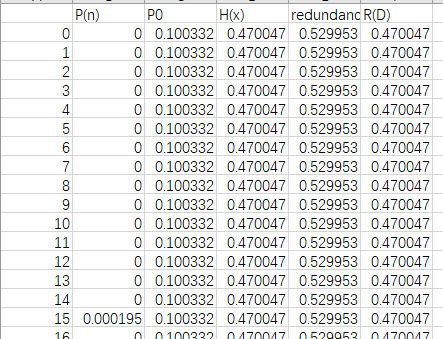
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 信源信息率  （信息比特/秒） | 信道数据率  （数据比特/秒） | 信道输入信息率（信息比特/秒） | 信道输出信息率（信息比特/秒） | 误码率 |
| 理论值 | 0.467 |  |  |  |  |
| 仿真值 |  |  |  |  | 7.32E-05 |

**（9）信源概率分布0.1，信道错误传递概率0.01，有信源编码，(7,4)线性分组码**

- 先用byteSource.exe生成一个概率分布为的离散无记忆信源（大小为10KB）



- 然后调用byteSource\_calc.exe程序计算信源文件的信息量大小



此处第四列显示信源的信息量为H(S)=0.470(信息比特/符号)，所以信源信息率

- 利用byteSourceEncoder.exe对信源进行编码



- 利用codding\_effect.exe程序计算信源编码指标



- 使用channelEncoder.exe生成信道编码后文件,使用(7,4)线性分组码



- 将信道编码后文件通过错误信道，错误概率为0.01



- 使用byteChannel\_calc.exe程序进行对其过信道的序列进行信道的输入与输出的信息熵的计算



可见，由于线性分组码改变了文件的0、1分布，因此信息熵发生改变,而且byteChannel\_calc.exe程序是以字节为单位来计算信息熵的，线性分组码打乱了原来的字节序，所以这样计算的结果对仿真没有什么意义，但是我们能从这个结果中算出正确的

以及输入输出信息量运用这个结果我们可以算出仿真的信道输出和信道传输率。

- 使用channelDecoder.exe解码错误传递后的文件，生成解码后文件



- 使用byteSourceDecoder.exe解码霍夫曼编码后文件



- 使用channelCoder\_calc.exe计算误码率



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 信源信息率  （信息比特/秒） | 信道数据率  （数据比特/秒） | 信道输入信息率（信息比特/秒） | 信道输出信息率（信息比特/秒） | 误码率 |
| 理论值 | 0.467 |  |  |  |  |
| 仿真值 |  |  |  |  | 0 |

### 分析

从结果可以看出，信源编码对信息进行了压缩，使得有用信息增加，因此提高了信息传输率，而信道编码虽然增加了信息的冗余度，降低了信息传输率，但却能降低信道的错误传递概率，其中使用线性分组码在同等误码率的条件下信息传输率要比重复码的高，对于不同的错误概率，两种编码都能胜任且降低错误概率，将信源编码与信道编码相结合能够在降低错误概率的同时尽量增加信息传输率。

从上面一般非理想情况的仿真可以看到，理论结果与仿真结果大致相同，出现误差的原因有以下几点：

一、在使用byteSource.exe生成文件时，由于根据信源概率空间生成的Source文件里面符号0和1的分布不一定是严格满足的，具有随机性，所以计算出来的信源信息率与理论计算值有误差。

二、在使用byteChannel\_calc.exe程序进行对其过信道的序列进行信道的输入与输出的信息熵的计算时，由于计算信息熵的方法不同，老师提供的方法避免了计算log0时nan的出现，因此造成了信源信息率和信道输入信息率的不匹配。

三、由于程序自身问题，在计算过信道序列时线性分组码由于改变了文件的0、1分布概率，因此导致了信息熵的改变，而线性分组码和重复码都因为打乱了原本程序的字节序，即原来的byteChannel\_calc.exe计算信息熵的方法是以单个字节看作一个符号来计算的，因此导致编码后计算出的信息熵有误，无奈之下我们只能结合部分公式对可用的结果进行推导，这一过程可能产生了误差。

四、在计算误码率时，程序没有考虑到log0时nan的出现，所以有一定误差。

# 系统改进【可选】

无

# 总结

## 设计完成情况

|  |  |
| --- | --- |
| 已完成的内容 | 未完成的内容 |
| 1.系统设计 | 4.可选部分 |
| 2.系统实现 |  |
| 3.系统仿真与分析 |  |
| 5.总结 |  |
| 6. 附录——相关技术简介 |  |

## 经验与收获

|  |  |  |
| --- | --- | --- |
| 序号 | 已完成的内容 | 经验与收获 |
| 2.1 | 信源单元测试：  可自由设定信源消息概率分布  可自由设定生成的消息序列的长度  信源概率分布统计表的生成  离散无记忆信源的信息熵的计算  信源冗余度的计算 | 在实际运行该模块时，发现离散无记忆信源的信息熵与生成消息序列的长度无关，只与各字节的概率分布相关。 |
| 2.2 | 信道单元测试：  可自由设定信源消息概率分布  可自由设定生成的消息序列的长度  可自由设定信道的错误传递概率  信道的输入信息熵的计算  信道的输出信息熵的计算  平均互信息量的计算  信道容量的计算 | 当信源等概率分布时，平均互信息量最大，此时的平均互信息量为信道容量。 |
| 2.3 | 信源编解码单元测试：  可自由设定信源消息概率分布  可自由设定生成的消息序列的长度  霍夫曼码书构造  霍夫曼编码  霍夫曼解码  平均码长的计算  编码效率的计算 | 霍夫曼编码是变长码，考虑了信源的统计特性，而且是最佳码，对于同一个信源，编码并非唯一，但平均码长都相同。 |
| 2.4 | 信道编解码单元测试：  可自由设定信源消息概率分布  可自由设定生成的消息序列的长度  可自由设定信道的错误传递概率  汉明码编码  汉明码解码  误码率*R*error（汉明失真）的计算 | 汉明码编解码误码率较低 |
| 3.1 | 理想情况下的系统仿真与设计：  可自由设定生成的消息序列的长度  信源的信息率的计算  信道的数据率的计算  信道的输入信息率的计算  信道的输出信息率的计算  信宿的误码率的计算 | 理想情况下信道的输入输出信息率相等，误码率为0 |
| 3.2 | 信源非理想情况下的系统仿真与设计：  可自由设定信源消息概率分布  可自由设定生成的消息序列的长度  可自由设定是否使用信源编码  信源的信息率的计算  信道的数据率的计算  信道的输入信息率的计算  信道的输出信息率的计算  信宿的误码率的计算 | 信源非理想时，无论是否使用信源编码，信道的输入输出信息率相等，误码率为0 |
| 3.3 | 信道非理想情况下的系统仿真与设计：  可自由设定生成的消息序列的长度  可自由设定信道的错误传递概率  可自由设定是否使用信道编码  信源的信息率的计算  信道的数据率的计算  信道的输入信息率的计算  信道的输出信息率的计算  信宿的误码率的计算 | 信道非理想时，在信源分布概率和错误传递概率相同的条件下，使用信道编码比不使用信道编码会提高信道的输入信息率，错误传递概率越小，误码率越低 |
| 3.4 | 一般非理想情况下的系统仿真与分析：  可自由设定生成的消息序列的长度  给定两种信源概率分布  给定三种信道的错误传递概率  无信源编码、有信源编码  无信道编码、重复码、(*n*,*k*)线性分组码  信源的信息率的计算  信道的数据率的计算  信道的输入信息率的计算  信道的输出信息率的计算  信宿的误码率的计算 | 一般非理想情况的通讯条件设定较复杂，信源编码对信息进行了压缩，使得有用信息增加，因此提高了信息传输率，而信道编码虽然增加了信息的冗余度，降低了信息传输率，但却能降低信道的错误传递概率，其中使用线性分组码在同等误码率的条件下信息传输率要比重复码的高，对于不同的错误概率，两种编码都能胜任且降低错误概率，将信源编码与信道编码相结合能够在降低错误概率的同时尽量增加信息传输率。 |

# 附录——相关技术简介

计算机语言

- python

- 官方名称、版本号、官网URL

python、Python 3.8.7 - Dec. 21, 2020、https://www.python.org/

主要函数包/模组

- numpy

- 官方名称、版本号、官网URL

- NumPy、NumPy v1.19.4、https://numpy.org/

- 在本设计中用途：用其数学模组来简化程序中的计算，但因为最终打包时含有numpy库会导致文件过大，于是尽量少使用。

- scipy

- 官方名称、版本号、官网URL

- Scientific computing tools for Python、SciPy 1.5.4、https://www.scipy.org/

- 在本设计中用途：最优化、线性代数功能，同样，因为最终打包时含有scipy库会导致文件过大，于是尽量少使用。

- bitString

- 官方名称、版本号、官网URL

- bitstring、bitstring 3.1.7、https://bitstring.readthedocs.org

- 在本设计中用途：将文件从二进制数据流读取并将其解释为二进制数据流，使得我们可以用二进制流的方式处理文件。

- sys

- 官方名称、版本号、官网URL

- System-specific parameters and functions、sys 3.8.1、

- 在本设计中用途：用于实现交互功能，即文件输入输出参数。

工具软件

- visual studio code

- 官方名称、版本号、官网URL

- visual studio code、version 1.43.1、https://code.visualstudio.com/

- 在本设计中用途：用于代码的编写与开发。

- PyInstaller

- 官方名称、版本号、官网URL

- PyInstaller、PyInstaller 4.1、https://www.pyinstaller.org/

- 在本设计中用途：将Python程序打包为标准可执行文件，这些文件可以在未安装Python的计算机上运行