Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

PROYECTO FINAL DE CARRERA

Decodificación de Canal Asistida por Protocolo
(Protocol Assisted Channel Decoding)

Estudios: Ingeniería Superior de Telecomunicaciones
Autor: Daniel Moreno Lozano
Director: Francisco José Rico Novella
Año: 2016

Abstract

This document describes how the channel decoder at Physical layer of a communication system can benefit from information brought either by careful examination of the standard, or by already received packets.

In all communication systems we have a stablished protocol. That helps communication between sender and receiver more understanding. This fixed structure usually leads to the systematic use of headers and other repetitive fields in the frame structure. What if we can take profit of all this redundancy of the standard to improve the decoding process? That's the main idea of Protocol-Assisted Channel Decoding.

Resumen

La presente tesis describe como el decodificador de canal en la capa física de un sistema de comunicación se puede beneficiar de información obtenida por un detenido estudio del estándar, o por paquetes recibidos anteriormente.

En todo sistema de comunicación existe un protocolo establecido. Esto ayuda a que haya un entendimiento entre el emisor y el receptor durante la comunicación. Esta estructura fija normalmente conlleva el uso sistemático de cabeceras y campos repetitivos en la estructura de los paquetes. ¿Y si podemos aprovechar de esa redundancia del estándar para mejorar el proceso de decodificación? Esa es la principal idea de la Decodificación de Canal Asistida por Protocolo.

Resum

La present tèsi descriu com el decodificador de canal a la capa física d'un sistema de comunicació es pot beneficiar de l'informació obtinguda per un rigurós estudi de l'estàndard, o de paquets rebuts anteriorment. A tot sistema de comunicació hi existeix un protocol establert.

Això ajuda a que hi hagi una entesa entre l'emissor i el receptor durant la comunicació. Aquesta estructura fixa normalment implica l'ús sistemàtic de capçaleres i camps repetitius a l'estructura dels paquets. I si podem aprofitar aquesta redundància de l'estàndard per tal de millorar el procés de decodificació? Aquesta és la principal idea de la Decodificació de Canal Assistida per Protocol.

Agradecimientos

Quiero dedicar este trabajo especialmente a mis padres, ellos saben que sin ellos haber llegado hubiera sido imposible. Gracias por vuestro apoyo, paciencia y confianza.

Sobretodo me gustaría mostrar mi gratitud a la Universidad ETSETB por la oportunidad de formarme y aprender de los mejores. Pero especialmente a mi tutor, el profesor Francisco José Rico, por la oportunidad de trabajar con él, por su ayuda y gran disponibilidad

Y por supuesto a mi prometida Jeannine, con tu apoyo y energía todo es más fácil.

Índice

Abstract III

Resumen IV

Resum V

Agradecimientos VI

Lista de Figuras IX

Nomenclatura X

1. Introducción 1

2. Fundamentos de transmisión de datos 2
- 2.1. Esquema general 2
- 2.2. Codificación de fuente 3
- 2.3. Concepto de información 5
- 2.4. Entropía 7
- 2.5. Codificación de fuente 8
- 2.6. Canal de comunicación 10
  - 2.6.1. Límites fundamentales de transmisión de datos 10
  - 2.6.2. Capacidad de canal 12
  - 2.6.3. Canal discreto sin memoria DMC 13
- 2.7. Capacidad de un canal discreto sin memoria simétrico 15

3. Codificación de canal 16
- 3.1. Introducción 16
- 3.2. Códigos convolucionales 17
  - 3.2.1. Demodulación firme o Hard-decision Decode 20
  - 3.2.2. Algoritmo de Viterbi 23
  - 3.2.3. Demodulación indecisa o Soft-decision Decode 24

4. Estructura Genérica de Paquete en Capa Física 26

5. Algoritmo BCJR 28
- 5.1. Descripción 28
- 5.2. El problema general 28
- 5.3. Aplicación a códigos convolucionales 32

6. Protocol-Assisted Channel Decoding 35
- 6.1. Estimador Óptimo 35
- 6.2. Estimador Subóptimo 35

7. Aplicación al estándar 802.11a **37**
7.1. Encapsulación y codificación 37
7.2. Descripción de la capa MAC 37
7.3. Descripción de la capa PHY 39
7.4. Resultados de la Simulación 39

8. Conclusiones **43**

A. Apéndice **44**
A.1. Código desarrollado 44
A.2. Resultados simulación 72

Índice de figuras

1. OSI model 1
2. Esquema general de transmisión de datos 3
3. Proporcionalidad entre probabilidad y cantidad de información . 5
4. Función logarítmica decreciente 6
5. Entropía de fuente binaria. 8
6. Formula de Shannon de capacidad de canal 11
7. Diagrama de parámetros de canal 12
8. Diagrama de bloques de canal 12
9. Relación información de X e Y 13
10. Diagrama de canal discreto 13
11. Diagrama canal BSC 15
12. Diagrama de generación de código bloque 17
13. Diagrama de bloque de código continuo. 17
14. Diagrama de generación de código continuo. 18
15. Ejemplo diagrama de codificador continuo 18
16. Diagrama de estados continuo k/n=1/3 (formato: entrada/salida) 19
17. Diagrama Trellis del codificador. 20
18. Diagrama distancia libre de código. 20
19. Diagrama estados código convolucional r=1/2 21
20. Evaluación con estado inicial 0. 22
21. Evaluación con estado inicial 1. 23
22. Algoritmo de Viterbi 24
23. Diagrama esquemático de un sistema de transmisión 28
24. Diagrama de transición de estados y salidas de una fuente de Markov de 3 estados. 29
25. Diagrama de Trellis de la Fig. 24. 30
26. Codificador de tasa 1/2 y diagrama de Trellis 33
27. Tramas PHY y MAC para 802.11a 38
28. Trama PHY para 802.11a 39
29. Header Error Rate simulación 41
30. Frame Error Rate simulación 42

Nomenclatura

Abreviaciones

APP	A Posteriori Probabilities
BSC	Binary Symmetric Channel
CRC	Cyclic Redundancy Check
DMC	Discrete Memoryless Channel
MAC	Media Access Control
OSI	Open Systems Interconnection
PHY	PHYsical layer
ROHC	RObust Header Compression

Símbolos Matemáticos

$\lambda_t(m)$	APP of the states
$\sigma_t(m', m)$	APP of the transitions
p_t(m|m')	Transition probability from state m' to state m at time t
q_t(X|m,m')	Output probability given the transition m' to m
S_t	State at time t
$S_t^{t'}$	Symbol sequence from time t to t'; $S_t^{t'} = S_t, S_{t+1}, ..., S_{t'}$
X_t	Output symbol at time t

1. Introducción

Como debe saber, el extendido modelo de capas OSI [1] nos brinda una caracterización y estandarización de cada función presente en un sistema de comunicación. Separa cada función de la red en capas lógicas perfectamente diferenciadas.

Esta abstracción de capas hace más fácil el diseño de redes, ya que cada una de ellas es funcionalmente independiente. No deben preocuparse por la información proveniente de otras capas. Esto permite trabajar con contenidos heterogéneos dentro de la misma red de comunicación. Cada capa asume que su capa inferior le entrega información libre de errores, entonces se centra en entregar información perfecta a las capas superiores. Con tal de conseguirlo, códigos detectores de errores, como CRC o checksums, han sido introducidos en algunos puntos de la pila de protocolo estándar además de mecanismos de retransmisión en casa de paquetes dañados.

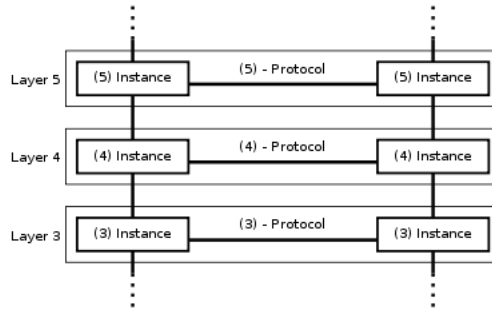


Figura 1: OSI model

Como se ha comentado anteriormente, las diferentes capas o niveles trabajan independientemente, pero en ocasiones necesitan conocer la misma información o una correlación de ésta. Esto implica que que habrá una cierta redundancia, especialmente en las cabeceras de los paquetes. Esta peculiaridad ya ha sido utilizada para minimizar la longitud de las cabeceras en ROHC [2]. De todas maneras, podemos utilizar esa información presente en las cabeceras para mejorar las tareas que otras capas deben llevar a cabo, así como para subsecuentes paquetes. Podemos sacar provecho de esta redundancia del protocolo. Podemos pensar en esta redundancia como una interacción entre capas, por lo tanto el enfoque puede asumir como un joint approach, más que un cross-layer approach [3, 6] en el cual la intención es afinar conjuntamente los parámetros de distintas capas.Obviamente, en join approaches las capas tienen una división menos estricta con tal de optimizar el rendimiento y utilización de recursos. Ahora las capas tienen acceso a información útil presente en otras capas. El inconveniente de las joint approaches es la pérdida de coherencia en la arquitectura, la cual era el principal motivo de las tener capas diferenciadas.

Joint Protocol and Channel Decoding (JPCD) intenta hacer el mejor uso posible de los datos recibidos explotando la redundancia presente en las capas de protocolo así como a la redundancia introducida por el codificador de canal con tal de obtener un rendimiento óptimo [7]. Esta técnica sería muy útil en el contexto de comunicaciones mixtas cableadas e inalámbricas. JPCD ya ha sido utilizado para mejorar la eficiencia de diversas capas de la pila de protocolo: para una recuperación más fidedigna de las cabeceras [8] o para una robusta recuperación de paquetes agregados [9, 13].

El objetivo de este proyecto es demostrar y validar que la decodificación de canal también puede beneficiarse de redundancia perfectamente conocida presente en la pila de protocolo por medio de protocol-assisted channel decoding. En nuestro caso, el rendimiento la capa física PHY puede ser incrementado haciendo uso de la redundancia presente en la capa de enlace MAC y en capas de protocolo superiores. Cuando se efectúa la decodificación de canal en la capa física, uno estima el contenido de los paquetes PHY decodificados. Debido a la redundancia presente en la pila de protocolo, se puede ver que algunas partes de la cabecera y el payload de los paquetes físicos son perfectamente conocidos, y pueden tener un rol de símbolos piloto, de la misma manera que en [14]. En otros casos, solamente puede obtenerse información a priori sobre los valores de estos bits. Estos bits piloto permiten una decodificación mejorada de la secuencia codificada. Contrariamente a los habituales símbolos piloto, los cuales deben ser introducidos antes de efectuar la codificación, estos "símbolos piloto" se encuentran en muchas pilas de protocolo sin ninguna modificación de los estándares ni ninguna adición de bits extra que deban ser transferidos.

En la sección 4 introduciremos la estrategia para identificar los bits utilizaremos como bits piloto. Esto será posible analizando detalladamente la arquitectura del paquete PHY acorde al protocolo. Seguidamente la sección 5presentará el algoritmo BCJR para una decodificación óptima. Tras haber definido las bases y herramientas necesarias, la sección 6 define el decodificador de canal asistido por protocol propuesto. Un estimador óptimo en primer lugar, y seguidamente, por motivos de implementabilidad, un estimador subóptimo. Finalmente la sección 7 muestra los resultados de la simulación hecha en el contexto de comunicaciones inalámbricas utilizando el estándar 802.11a[15]. El código ha sido desarrollado en el lenguaje C++ haciendo uso de la librería IT++ [23] y puede encontrarse adjunto en el apéndice A.

2. Fundamentos de transmisión de datos

2.1. Esquema general

El principal objetivo de un sistema de transmisión es completar el envío de información del emisor al receptor de manera eficiente: lo más rápidamente y con el menor numero de errores posibles. Para completar esa meta se llevan a cabo una serie de técnicas de tratamiento de datos en diferentes etapas. Cada una de ellas aplicando distintas funciones sobre la información discreta, típicamente

bits. De esta manera podemos entender todo el sistema emisor-receptor como un conjunto de bloques concatenados como se observa en la Figura 2

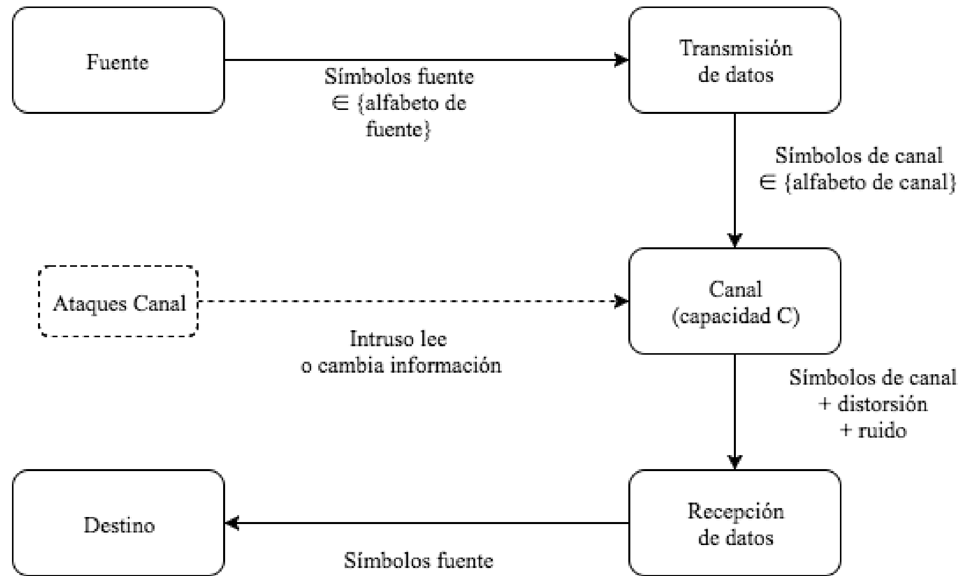


Figura 2: Esquema general de transmisión de datos

A continuación se listan los bloques implicados y sus funciones:

- Codificación de fuente
    - Adaptación de los alfabetos de fuente y canal.
    - Minimización de la longitud de los mensajes.
- Codificación de canal
    - Control de errores.
    - Detección de símbolos en el receptor
- Criptografía
    - Seguridad de la comunicación

2.2. Codificación de fuente

Las fuentes de información emiten mensajes en la forma de una concatenación de símbolos de fuente S_f.

Los símbolos de fuente forman el alfabeto de la fuente. Los mensajes generados por una fuente contienen una gran cantidad de información redundante. De

esta manera, el principal objetivo de la codificación de fuente es reducir esa redundancia a fin de explotar el canal de comunicación de la manera más eficiente posible.

La compresión ideal de los mensajes de la fuente se obtiene a partir de la estadística de la misma. La codificación asigna palabras código a símbolos de fuente siguiendo el siguiente algoritmo:

$$\text{Símbolos de fuente frecuentes} \Longleftrightarrow \text{Palabras código cortas}$$

$$\text{Símbolos de fuente poco probables} \Longleftrightarrow \text{Palabras código largas}$$

Sirva el siguiente ejemplo como muestra:

Partiendo de la tablas ASCII, dónde cada carácter es representado por 8 bits. El carácter '0' corresponde al código ASCII 00110000 (48). Dada una fuente con los símbolos $\{'1','2',...,'6'\}$ se proponen 2 posibles codificaciones:

Codificación binaria

S_f	P_c
'1'	00
'2'	01
'3'	10
'4'	11
'5'	000
'6'	001

Codificación ternaria

S_f	P_c
'1'	00
'2'	01
'3'	02
'4'	10
'5'	11
'6'	12

Las codificaciones propuestas anteriormente se utilizan típicamente como técnicas de compresión de datos. De las cuales se derivan dos aproximaciones:

1. Con pérdidas
   - *a*) La información decodificada puede resultar diferente de la original.
   - *b*) Se aumenta la compresión al asumir pérdida de información o distorsión.
   - *c*) Aplicación habitual en servicios audiovisuales: JPEG, MPEG, MP3, ...
2. Sin pérdidas
   - *a*) La información decodificada es exactamente igual a la original.
   - *b*) Aplicaciones en servicios de datos: Huffman, Lempel-Ziv (zip, 7z, ...)

2.3. Concepto de información

Podemos entender la información como una métrica medible. Según la teoría de la información, la cantidad de información que proporciona un suceso es inversamente proporcional la probabilidad de que éste tenga lugar.

Un ejemplo muy sencillo para entender esta medida sería pensar en la previsión meteorológica. Concretamente imaginemos la previsión meteorológica en el desierto. Si la predicción indica que mañana será un día soleado, esta noticia no aporta apenas información, ya que así es prácticamente cada día en el desierto. En el otro extremo, si la predicción indicara que va hay previsión de lluvia, esta noticia aportaría una gran cantidad de información debido a que es un evento extraordinario y poco frecuente. Ver figura 3 para entender la relación.

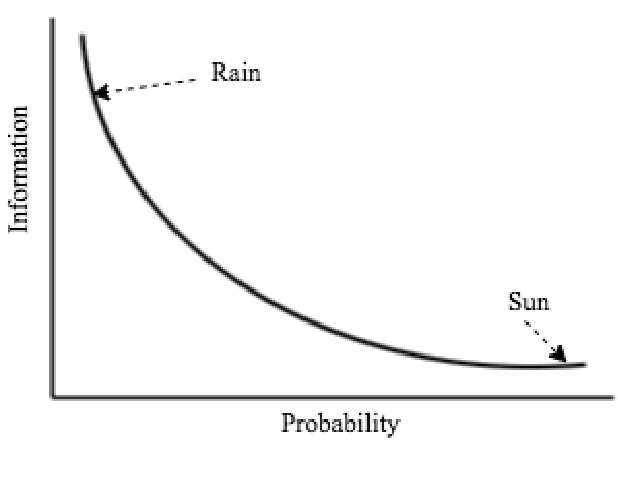


Figura 3: Proporcionalidad entre probabilidad y cantidad de información

Por lo tanto, un suceso s_i aporta más información I cuanto menor sea la probabilidad p_i de que este ocurra. Así tenemos,

$$I(s_i) \quad = \quad f(p_i) \geq 0$$

Definiendo una función decreciente que relación la información del suceso y su probabilidad de ocurrir.

Estudiando la propiedades de esta función respecto a dos eventos obtenemos:

$$I(s_i, s_j) \quad = \quad f(p(s_i, s_j))$$

Si s_i, s_j son independientes $\Rightarrow$ p(s_i,s_j)=$p_i p_j$

$$\begin{aligned} I(s_i, s_j) &= f(p_i p_j) \\ &= I(s_i) + I(s_j) = f(p_i) + f(p_j) \\ f(p_i p_j) &= f(p_i) + f(p_j) \end{aligned}$$

Dada es la relación, la función debe ser logarítmica.

$$f \Rightarrow log_a$$

Por lo tanto, la función debe cumplir la propiedad logarítmica y ser decreciente.

$$\begin{aligned} I(s_i) &= -log_a(p_i) \\ &= log_a(\frac{1}{p_i}) \end{aligned}$$

Representada en la figura 4.

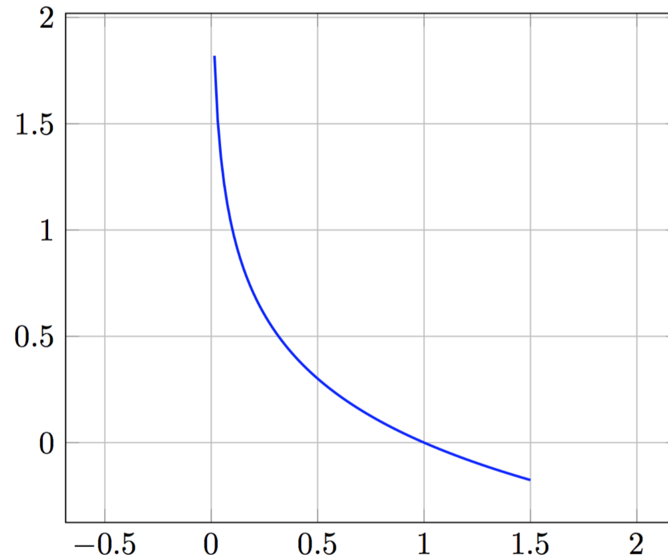


Figura 4: Función logarítmica decreciente

Para nuestro caso de estudio $a = 2$, debido a la lógica binaria. Otras opciones utilizadas serían:

- $a = 2 \Rightarrow$ unidades bits

- $a = e \Rightarrow$ unidades nats
- $a = 10 \Rightarrow$ unidades hartleys

Finalmente:

$$\begin{aligned} I(s_i) &= log_a(\frac{1}{p_i}) \\ &= \frac{1}{log_2 a} log_2(\frac{1}{p_i}) \end{aligned}$$

2.4. Entropía

La entropía H es la información media de una fuente. Concretamente es el número media de unidades de información por cada uno de los símbolos de la fuente. De esta manera, una fuente de sin memoria de k símbolos tendrá la siguiente entropía:

$$\begin{aligned} H &= E[I(s_i)] = p_1 I(s_1) + \ldots + p_k I(s_k) \\ &= p_1 log_a(\frac{1}{p_1}) + \ldots + p_k log_a(\frac{1}{p_k}) \\ &= \sum_{i=1}^{k} p_i log(\frac{1}{p_i}) \end{aligned}$$

Propiedades de la entropía:

- H es un valor intrínseco de la fuente, siendo independiente de la codificación. Sólo depende de la estadística de los símbolos de la fuente.
- H es siempre un valor mayor o igual que cero, llegando a su máximo cuando todos los símbolos son equiprobables.

$$0 \leq H \leq log_a k$$

Considerando una fuente binaria; por ejemplo una moneda. Habría dos posibles símbolos: cara (C) y cruz (X), con probabilidades p y $1-p$ respectivamente. Obteniendo,

$$H(moneda) = plog_a(\frac{1}{p}) + (1-p) log_a(\frac{1}{1-p})$$

Implicando,

$$H(p) \quad = \quad H(1-p)$$

Cierto para fuentes con símbolos de Bernouilli (fuentes con 2 símbolos posibles)

Evidentemente la máxima cantidad de información de este sistema sería 1 bit, en caso de ser una fuente de símbolos equiprobables ($p = 0{,}5$)

$$0 \quad \leq \quad H(p) \leq log_2 2 = 1bit$$

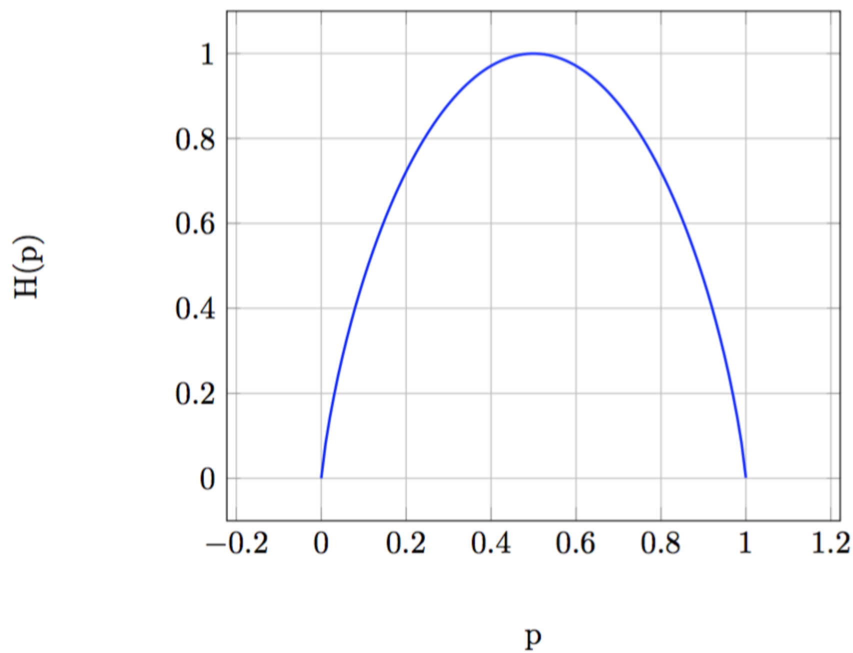


Figura 5: Entropía de fuente binaria.

2.5. Codificación de fuente

El proceso de codificación de fuente trata de asociar palabras código a símbolos o cadenas de símbolos de la fuente.

Su finalidad es minimizar la longitud media L de la palabra código que emite el codificador. Intentando tender lo máximo posible a la entropía de la fuente cumpliendo la siguiente ecuación (en caso de codificaciones sin pérdidas).

$$L = \sum_{i=1}^{k} p_i l_i \geq H$$

Siendo l_i la longitud de la palabra código asociada al símbolo s_i cuya probabilidad es p_i.

Por lo tanto conviene que:

$$l_i \simeq log_a \frac{1}{p_i}$$

Siendo a el tamaño de alfabeto del canal o valores posibles de los dígitos de la palabra código.

Así se puede definir la eficiencia de un código como:

$$E \triangleq \frac{H}{L}$$

A continuación se presenta un ejemplo con dos codificaciones distintas para ver la implicaciones. Consideremos una fuente con 4 símbolos posibles {A, B, C, D} y probabilidades 1/2, 1/4, 1/8 y 1/8 respectivamente. Aplicaremos una codificación con longitud de código constante L_c y otra con longitud variable L_v. Obteniendo la cantidad de información como $I(s_i) = log_2(p_i)$.

Símbolo s_i	Probabilidad	Código L_c	Código L_v	$I(s_i)$
A	1/2	00	0	1
B	1/4	01	10	2
C	1/8	10	110	3
D	1/8	11	111	3

Cuadro 1: Ejemplo longitud media de código.

Obteniendo la entropía de la fuente y el rango de longitud media de código.

$$H = 1,75 \text{ bits/símbolo}$$
$$L \geq 1,75 \text{ dígitos binarios/símbolo}$$

De esta manera calculamos la longitud medía de palabra código para cada codificación resultando,

$$\begin{aligned} L_c &= 2 \\ L_v &= 1,75 \end{aligned}$$

Se puede ver como en este caso la codificación variable es más eficiente que la constante, suprimiendo completamente toda redundancia que pueda introducir la codificación.

$$\begin{aligned} E_c &= 87,5\,\% \\ E_v &= 100\,\% \end{aligned}$$

2.6. Canal de comunicación

El canal de comunicación se define como el medio de transmisión por el cual viajan las señales que contienen la información en su viaje desde el emisor al receptor.

Habitualmente se trata de un medio hostil que puede degradar la señal, introducir interferencias o distorsionar la misma. Afectado por muchas variables como la propia naturaleza del canal, si éste es compartido y las fuentes de ruido que pueda contener.

Cabe destacar que cada canal de transmisión es adecuado para algunas señales concretas y no todos sirven para cualquier tipo de señal. Por ejemplo, la señal eléctrica se propaga bien por canales con propiedades conductoras como el cobre, oro, etc. Por otro lado, el comportamiento es diferente en el caso de señales ópticas, que requerirían el uso de canales de fibra óptica construidos a partir de dieléctricos como el plástico o cristales.

Un canal de comunicación puede ser caracterizado por sus propiedades físicas: naturaleza de la señal que se va a transmitir, velocidad de transmisión, ancho de banda, nivel de ruido que genera, modo de inserción de emisores y receptores, etc.

2.6.1. Límites fundamentales de transmisión de datos

Come se explica en la sección anterior, un canal queda caracterizado por si propia naturaleza.

De esta manera Shannon definió los límites de la capacidad de canal con la fórmula que lleva su nombre:

$$C = Wlog_2(1 + S/N)\ bps$$

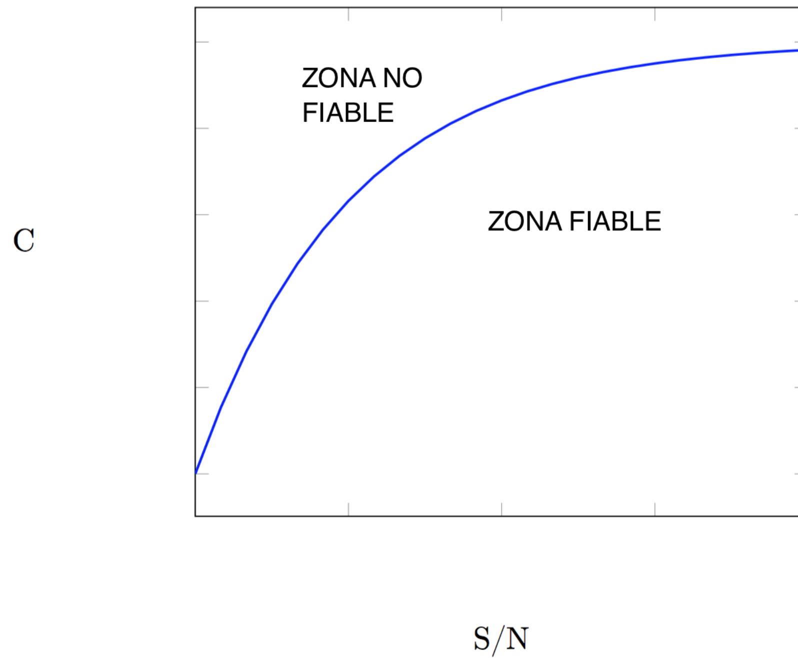


Figura 6: Formula de Shannon de capacidad de canal

Consideremos los siguientes parámetros:

v_t : velocidad de transmisión
C : capacidad de canal (bps)
W : ancho de banda (Hz)
S/N : relación señal a ruido lineal
A : número de niveles de codificación
v_m : velocidad de modulación
T_F : tiempo de generación de un símbolo de fuente
E : eficiencia espectral (C/W)

Representados en la figura 7.

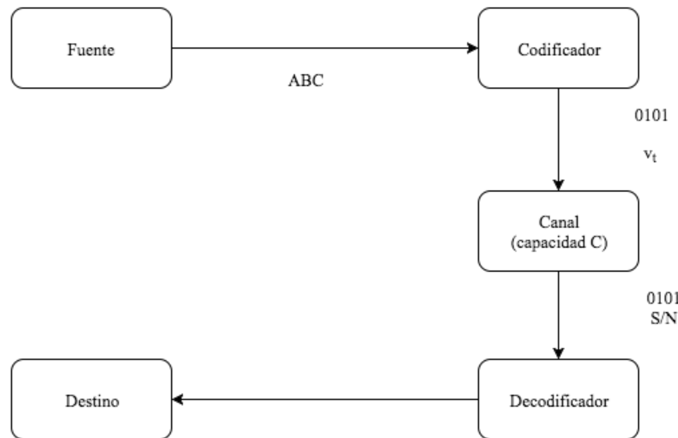


Figura 7: Diagrama de parámetros de canal

La velocidad de transmisión es proporcional a la velocidad de modulación acorde al número de niveles de la codificación,

$$v_t \quad = \quad v_m log_2 A$$

Para un canal fiable tenemos que siempre:

$$v_t \quad \leq \quad C$$

2.6.2. Capacidad de canal

Definiremos la capacidad de canal de una canal discreto sin memoria (DMC) como:

$$C \quad = \quad \max_{p(x)} I(X;Y)$$

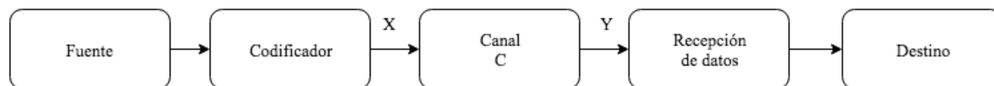


Figura 8: Diagrama de bloques de canal

Es decir, la capacidad queda definida por la fuente que es capaz de transferir más información al destino para un canal determinado.

Se trata de hallar la distribución de probabilidades $p(x)$ de los símbolos de una fuente X que hace más parecida la salida a la entrada del canal. Por lo tanto, esta fuente aprovecha mejor el canal.

Teniendo en cuanta que se conoce Y a la salida del canal y la cantidad de información que nos queda por estimar es $H(X|Y)$ que debe de ser la menor posible.

$$\begin{aligned} H(X|Y) &= H(X) - I(X;Y) \\ I(X;Y) &= H(X) - H(X|Y) \end{aligned}$$

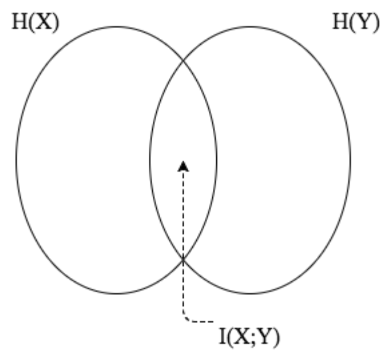


Figura 9: Relación información de X e Y

2.6.3. Canal discreto sin memoria DMC

Para caracterizar un canal se utiliza una matriz estocástica de probabilidades condicionales Q para expresar la relación entre símbolos de canal a la entrada y a la salida.

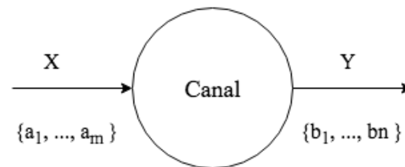


Figura 10: Diagrama de canal discreto

$$P(b_j|a_i) \triangleq Prob\,[Y = b_j|X = a_i]$$

$$Q = \begin{pmatrix} p(b_1|a_1) & p(b_2|a_1) & \cdots & p(b_n|a_1) \\ p(b_1|a_2) & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ p(b_1|a_m) & \cdots & \cdots & p(b_n|a_m) \end{pmatrix}$$

Ejemplos de canales:
a) Canal binario sin ruido

$$X \left\{ \begin{array}{ccc} 1 & \longrightarrow & 1 \\ 0 & \longrightarrow & 0 \end{array} \right\} \text{Y} \qquad Q = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$\begin{array}{rcl} H(X|Y) & = & 0 = H(X) - I(X;Y) \\ C = \max\limits_{p(x)} I(X;Y) = \max\limits_{p(x)} H(X) & = & log_2 m = 1 bit/simb \\ & \uparrow & \\ símbolos & & equiprobables \end{array}$$

b) Canal ruidos con salidas no solapadas

$$-1 \begin{array}{l} \overset{1/2}{\nearrow} \; -1,5 \\ \\ \underset{1/2}{\searrow} \; -0,5 \end{array} \qquad\qquad 1 \begin{array}{l} \overset{1/2}{\nearrow} \; 0,5 \\ \\ \underset{1/2}{\searrow} \; 1,5 \end{array}$$

$$\begin{array}{ccc} Q & = & \begin{pmatrix} 1/2 & 1/2 & 0 & 0 \\ 0 & 0 & 1/2 & 1/2 \end{pmatrix} \begin{array}{l} \leftarrow_{a_1=-1} \\ \leftarrow_{a_2=1} \end{array} \\ & & \nearrow \quad \nearrow \quad \nearrow \quad \uparrow \\ & & {\scriptstyle b_1=-1,5 \quad b_2=-0,5 \quad b_3=0,5 \quad b_4=1,5} \end{array}$$

$$\begin{array}{rcl} H(X|Y) & = & 0 \\ C = \max\limits_{p(x)} [H(X) - H(X|Y)] = \max\limits_{p(x)} H(X) & = & log_2 m = 1 bit/simb \end{array}$$

c) Canal BSC

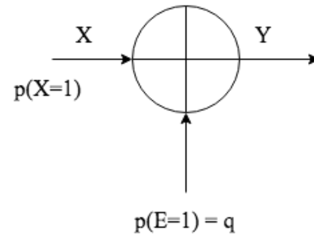


Figura 11: Diagrama canal BSC

$$H(Y|X) = E$$

$$\begin{aligned}
C &= \max_{p(x)} I(X;Y) \\
&= \max_{p(x)} [H(Y) - H(Y|X)] \\
&= \max_{p(x)} [H(Y) - H(E)] \\
&= log_2 n - H(E) = 1 - H(1) \\
&\nwarrow \textit{símbolos equiprobables}
\end{aligned}$$

$$X \left\{ \begin{array}{ccc} 0 & \xrightarrow{1-q} & 0 \\ & \overset{q}{\searrow} & \\ & \nearrow & \\ 1 & \underset{1-q}{\overset{q}{\longrightarrow}} & 1 \end{array} \right\} Y \qquad Q = \begin{pmatrix} 1-q & q \\ q & 1-q \end{pmatrix}$$

$$\left\{ \begin{array}{ccc} si\ q = 0 & \Longrightarrow & C = 1 bit/simb \\ si\ q = 1/2 & \Longrightarrow & C = 0 \end{array} \right\}$$

2.7. Capacidad de un canal discreto sin memoria simétrico

Un canal discreto sin memoria es simétrico si y sólo si:

- Las filas de Q son permutaciones entre sí.

- Las columnas de Q son permutaciones entre sí.

Ejemplo:

$$Q = \begin{pmatrix} 1/2 & 1/4 & 1/4 \\ 1/4 & 1/2 & 1/4 \\ 1/4 & 1/4 & 1/2 \end{pmatrix}$$

Obteniendo la siguiente capacidad:

$$\begin{aligned} C &= \max_{p(x)} I(X;Y) \\ &= \max_{p(x)} [H(Y) - H(Y|X)] \\ &= log_2 n - H(fila\ de\ Q) \end{aligned}$$

$$\begin{aligned} H(Y|X) \quad) &= \sum_{a_i} p(X=i) \sum_{b_j} p(Y=j) log_2 \frac{1}{p(Y=j|X=i)} \\ &= H(fila\ de\ Q) \end{aligned}$$

3. Codificación de canal

3.1. Introducción

Con el objetivo de combatir los retos que hemos visto que representa el canal, se introduce la codificación de canal.

Recordemos los objetivos principales de la comunicación:

- Transmisión rápida $\Rightarrow$ Codificación de fuente
- Transmisión segura $\Rightarrow$ Criptografía
- Transmisión fiable $\Rightarrow$ Codificación de canal

La meta de la codificación de canal es proteger la información frente a las degradaciones del canal (ruido y distorsiones). Lo hace añadiendo redundancia para realizar un control del enlace de datos.

En la codificación se detectan errores, además de poder corregirlos también.

Podemos clasificar las técnicas de codificación en dos categorías:

- En función de la capacidad correctora o detectora.
- En función de la forma en la que se introduce la información.

Capacidad detectora

- ARQ (Automatic Repeat Request). El decodificador es capaz de detectar información errónea y solicitar al emisor su reenvío.
- FEC (Forward Error Control). El decodificador es capaz de corregir por sí mismo los errores introducidos por el canal.

Inserción de redundancia

- Códigos bloque. Dividen los datos en bloques de igual longitud y les añaden una redundancia al final del mismo.

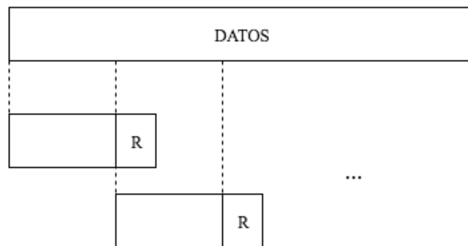


Figura 12: Diagrama de generación de código bloque

- Códigos continuos o convolucionales. La redundancia se introduce de de forma continua a medida que entran los datos.

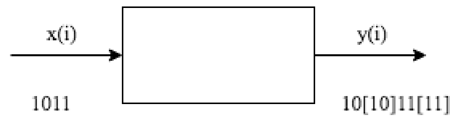


Figura 13: Diagrama de bloque de código continuo.

3.2. Códigos convolucionales

La información redundante se introduce de manera continua a medida que llega el mensaje al codificador. Se utilizan registros de desplazamiento en los que se almacena el valor de entradas pasadas. El valor conjunto de los registros define el estado del codificador.

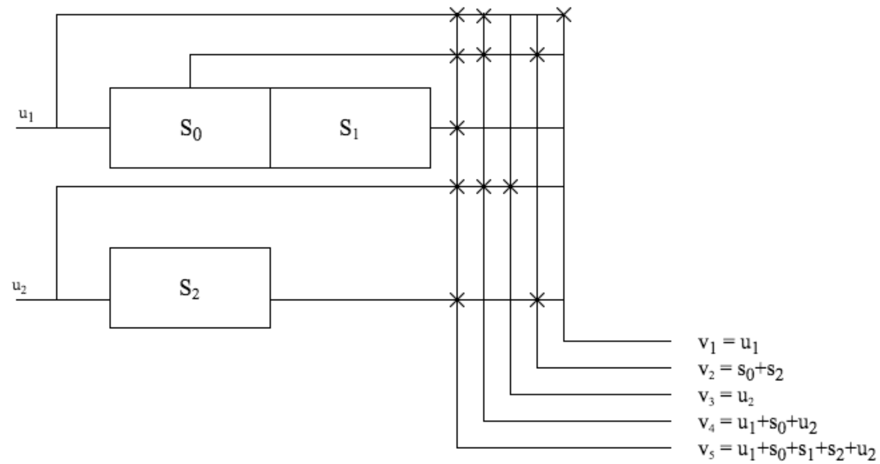


Figura 14: Diagrama de generación de código continuo.

La salida depende del estado de la entrada, de la misma manera que lo haría una máquina de estados.

Se puede caracterizar con los siguientes parámetros:

M: memoria de almacenamiento total.
L_{max}: longitud de influencia o memoria máxima.
k: longitud de información.
n: longitud de código.
k/n: tasa del codificador.
$s = q^M$: número de estados.

A continuación se analiza un ejemplo de codificación continua con longitud de información $k = 1$, longitud de código $n = 3$ y 4 estados presentado en la siguiente figura.

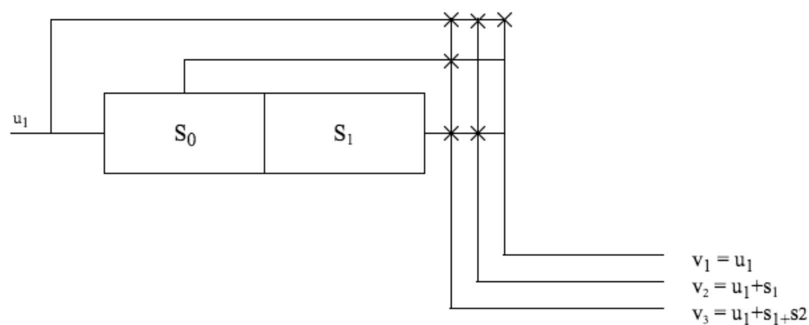


Figura 15: Ejemplo diagrama de codificador continuo

Con estos datos podemos representar el diagrama de estados del codificador continuo juntamente con la tabla de posibles combinaciones.

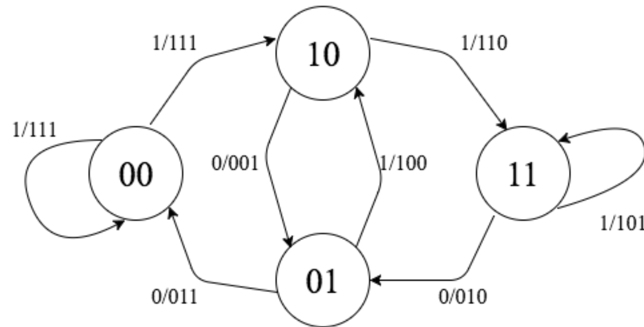


Figura 16: Diagrama de estados continuo k/n=1/3 (formato: entrada/salida)

Estado/Entrada	0	1
00	000/00	111/10
01	011/00	100/10
10	001/01	110/11
11	010/01	101/11

Cuadro 2: Tabla de estados y salidas de un código continuo (formato: salida/estado)

A continuación vamos a codificar la secuencia $\{1, 0, 0\}$ partiendo del estado 00. Veamos que ocurre paso a paso:

1. Partiendo del estado 00, llega un 1 a la entrada, por lo tanto la salida será 111 y el nuevo estado será el 10.

2. Partiendo ahora de 10, llega un 0 al codificador, por lo que la salida será 001 y el pasará al estado 01.

3. Finalmente llega un 0 estando en el estado 01. El codificador volverá al estado inicial 00 y la salida será 011.

4. Por lo tanto la secuencia final codificada será 111 001 011.

Entrada	Salida Codificada
100	111001011

Podemos representar el proceso de codificación utilizando el conocido diagrama de Trellis.

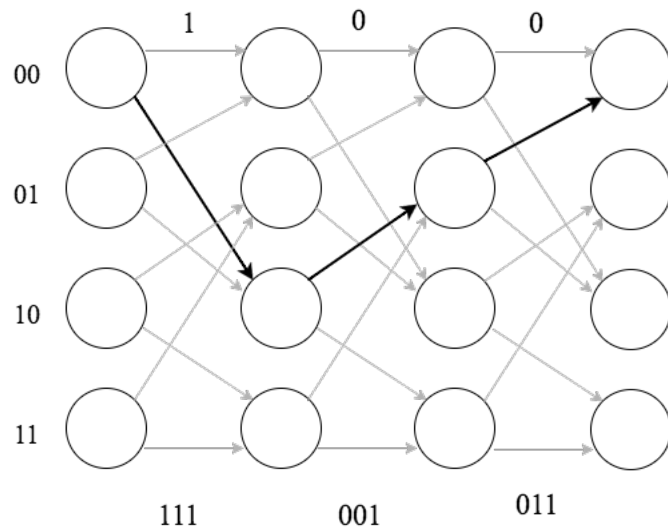


Figura 17: Diagrama Trellis del codificador.

Una manera de medir la calidad de un código es determinando su distancia libre. La distancia libre de un código es el peso (número de unos) de la codificación más cercana a la secuencia todo ceros. Cuanto mayor sea la distancia libre, mejor es el codificador.

En el ejemplo anterior:

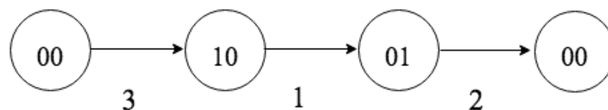


Figura 18: Diagrama distancia libre de código.

Por lo tanto,

$$d_{libre} = 6$$

3.2.1. Demodulación firme o Hard-decision Decode

Trabaja con secuencias de bits, por lo que la métrica empleada es la distancia de Hamming. Se trata de hallar la secuencia generable por el codificador más próximo a la secuencia recibida. Por lo tanto, previamente hace uso de un cuantificador, redondeando el valor recibido Y_i a 0 ó 1 según cual esté más cerca en el espacio de símbolos de modulación.

La distancia de Hamming se mida contando el número de bits diferentes que tienen dos secuencias.

Consideremos un ejemplo con k=1, n=2 y las siguientes métricas de salida:

$$
\begin{aligned}
v_1 &= u_1 \\
v_2 &= u_1 + s_0
\end{aligned}
$$

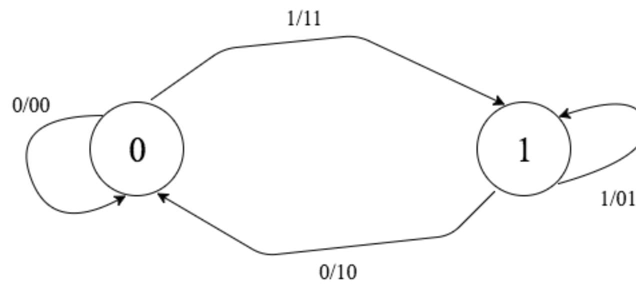


Figura 19: Diagrama estados código convolucional r=1/2

Siendo la secuencia recibida:

$$11 \quad 11 \quad 01$$

Efectuamos una evaluación exhaustiva calculando todas las entradas posibles buscando la que tiene la mínima distancia de Hamming.

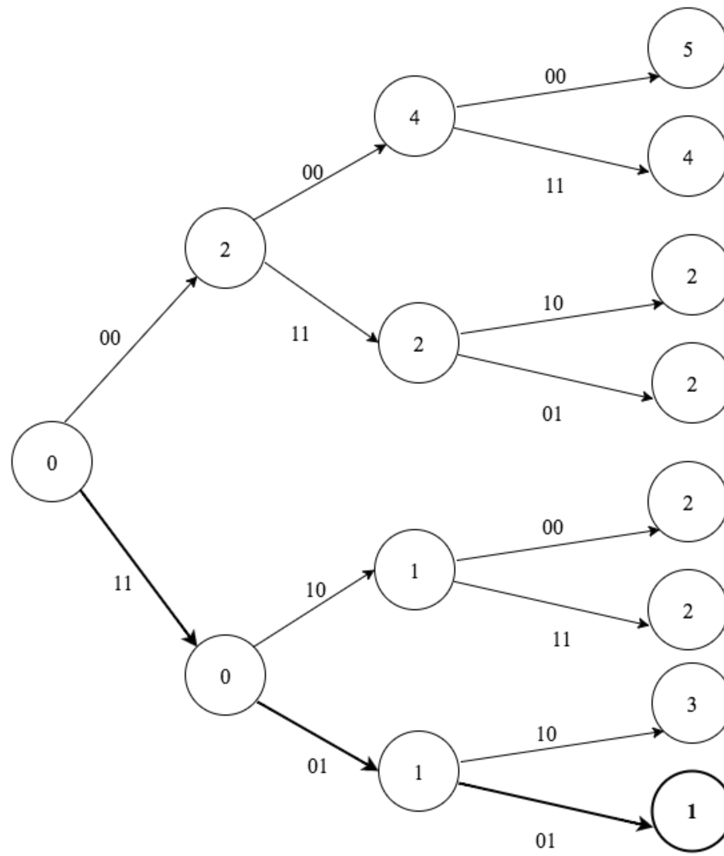


Figura 20: Evaluación con estado inicial 0.

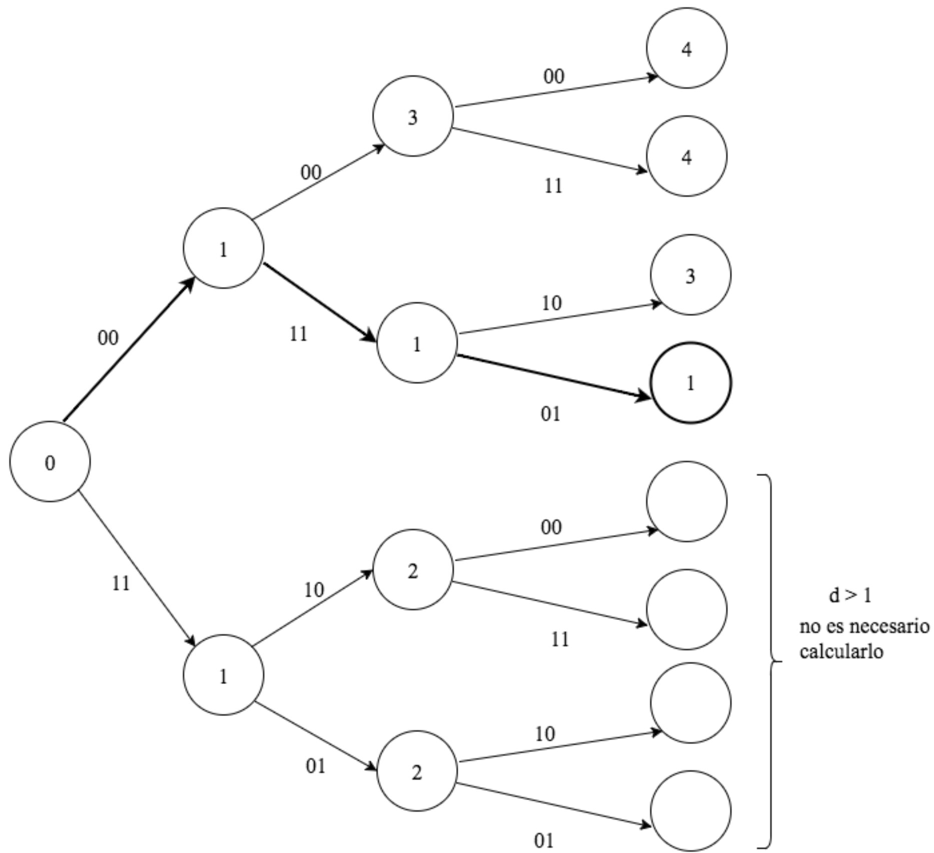


Figura 21: Evaluación con estado inicial 1.

Encontramos que existen dos secuencias generables a la misma distancia de Hamming. Cada una de ellas partiendo de un estado inicial distinto.

Estado inicial	seq_1	seq_2	seq_3	Entrada posible X
0	11	$\underline{\mathbf{0}}$1	01	111
1	1$\underline{\mathbf{0}}$	11	01	011

3.2.2. Algoritmo de Viterbi

El algoritmo de Viterbi permite hallar la secuencia de símbolos decodificados que está más cerca de la secuencia recibida. Especialmente en el contexto de fuentes de información de Markov.

Se aplica generalmente en los códigos convolucionales usados en telefonía móvil como GSM y CDMA, módems, satélites, redes Wi-Fi 802.11, ...

Consideremos la variable $\delta_t(i)$ que se define como:

$$\delta_t(i) = \max_{q_1,q_2,\ldots,q_{t-1}} P(q_1, q_2, \ldots, q_t = i, o_1, o_2, \ldots, o_t|\mu)$$

$\delta_t(i)$ es la probabilidad del mejor camino hasta el estado i habiendo visto las t primeras observaciones. Esta función se calcula para todos los estados e instantes de tiempo.

$$\delta_{t+1}(j) = \left[\max_{1\leq i\leq N} \delta_t(i)(a_{ij})\right] b_j(o_{t+1})$$

Puesto que el objetivo es obtener las secuencia de estados más probable, será necesario almacenar el argumento que hace máxima la ecuación anterior en cada instante de tiempo t y para cada estado j y para ello utilizamos la variable $\varphi_t(j)$.

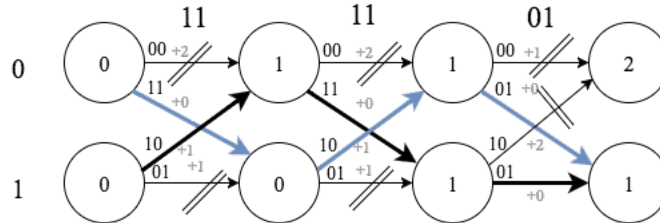


Figura 22: Algoritmo de Viterbi

Nuevamente encontramos dos caminos posibles para el ejemplo anterior. Buscando el camino más corto y descartando rápidamente los caminos con grandes distancias. En este caso, sin disponer de criterio acerca del estado inicial, el decodificador puede escoger cualquiera de las dos soluciones.

3.2.3. Demodulación indecisa o Soft-decision Decode

Este tipo de decodificación no utiliza un cuantificador como sistema de decisión. Esta vez las muestras se pasan directamente al decodificador de canal. Se trabaja con distancias euclídeas en el algoritmo de Viterbi. Aumenta la precisión en la demodulación debido a que dispone de mayor resolución que en el caso de utilizar un cuantificador, que conllevaba una pérdida de información.

Seguidamente se presenta un ejemplo para el caso de una modulación PAM-2, con los valores de señal -1 y 1 correspondientes a los bits '0' y '1' respectivamente.

La palabra recibida es:

$$z = \hat{v_1}\hat{v_2}\hat{v_3} = 0,8\ -0,1\ 0,5$$

Los valores de las muestras para la palabra código son:

$$
\begin{array}{lcrrrl}
000 & \rightarrow & -1 & -1 & -1 & = v_1^{000} v_2^{000} v_3^{000} \\
111 & \rightarrow & 1 & 1 & 1 & = v_1^{111} v_2^{111} v_3^{111}
\end{array}
$$

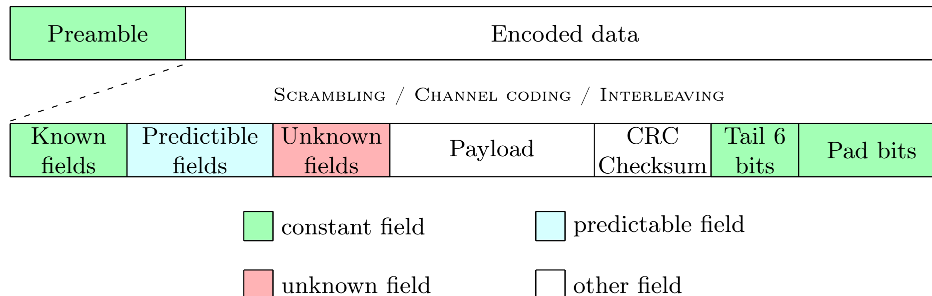
Medimos la distancia para cada uno de los casos:

$$
\begin{aligned}
distancia(z,'000') &= (v_1^{000} - \hat{v}_1)^2 + (v_2^{000} - \hat{v}_2)^2 + (v_3^{000} - \hat{v}_3)^2 = 6,3 \\
distancia(z,'111') &= (v_1^{111} - \hat{v}_1)^2 + (v_2^{111} - \hat{v}_2)^2 + (v_3^{111} - \hat{v}_3)^2 = 1,5
\end{aligned}
$$

Por lo tanto, la secuencia 111 sería mucho más verosímil.

4. Estructura Genérica de Paquete en Capa Física

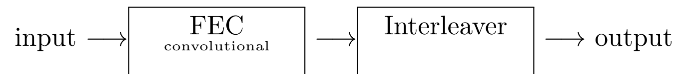
En el nivel más bajo del conocido modelo OSI se encuentra la denominada capa física, también denominada capa PHY. Se ocupa de traducir las comunicaciones lógicas de la capa de enlace inmediatamente superior, conocida como capa MAC, en operaciones concretas de hardware para su posterior transmisión a través del canal como señales físicas. Éstas pueden ser de distinta naturaleza, como señales electromagnéticas, ópticas, etc. Antes de proceder a la transmisión del paquete, esta capa soporta un conjunto de operaciones de procesamiento de los datos a tener en cuenta. Estas operaciones pueden ser la agregación de diversos paquetes MAC, scrambling, entrelazado o codificación de canal . En cuanto a la encapsulación del paquete, acostumbra a introducir una cabecera PHY mientras que el payload o carga útil es el paquete MAC entero (incluyendo sus propias cabeceras). Seguidamente suele añadir un campo de CRC o checksum para verificar la integridad del paquete para ayudar al receptor en el proceso de decodificación. Finalmente, bits de 'tail' y 'padding' son también introducidos. El proceso de Tail-biting se utiliza para asegurar que el estado final del decodificador es el mismo que el inicial. El bit-padding simplemente añade algunos bits con la finalidad de asegurar que el paquete que se transmite tenga un tamaño de paquete estándar. Tras llevar a cabo la codificación de canal, un preámbulo sin codificar suele ser añadido al inicio del paquete. Esto ayudará al receptor a sincronizar la secuencia de bits, identificando perfectamente los bits que conforman el paquete a decodificar. Esta técnica es utilizada en mucho protocolos bien conocidos como WLAN[16] o DVI[17].



El siguiente paso será estudiar el paquete para identificar los bits candidatos a ser "bits piloto". Debido a las condiciones que impone el estándar y la configuración que tenga el sistema, es posible localizar una serie de bits constantes en la cabecera MAC. La configuración del sistema determina ciertos parámetros del modo en que la comunicación tendrá lugar, como si la comunicación se lleva a cabo punto a punto, uplink, downlink, multicast, etc. Esos bits serán siempre constantes para sucesivos paquetes, por lo que son una fuente de redundancia de la que el decodificador puede sacar provecho. Los campos constantes serán denotados como **k**. Los campos predecibles **p** consisten en cabeceras que han sido

decodificadas con éxito previamente por capas superiores. A efectos prácticos, tanto **k** como **p** pueden ser considerados como campos conocidos por el receptor y ser utilizados como “bits piloto” por el decodificador de canal. Los campos desconocidos **u**, son los que deben ser completamente estimados por la capa física. Aún así, debido a cierta redundancia, los posibles valores de u se puede restringir a algún subconjunto de posibles valores Ω_u. El campo denominado **o** (por otros), engloba los bits de los cuales no tenemos conocimiento explícito en esta capa, éstos deben ser procesados por capas superiores. Finalmente, un campo de comprobación **c** suele estar incluido para verificar la integridad del paquete en el receptor, haciendo uso de técnicas como CRC, checksum o bits de paridad.

Por motivos de simplicidad en cuanto a la notación, en la siguiente demostración asumiremos que no hay scrambling de datos y el codificador de canal es un código Forward Error Correcting code (FEC). De todas maneras, el método propuesto es fácilmente extensible al caso de un scrambler sin memoria. Tras llevar a cabo la codificación de canal se aplicará un entrelazador para combatir posibles errores de ráfaga que pueda introducir el canal de comunicación en el paquete.



5. Algoritmo BCJR

5.1. Descripción

Su nombre se debe al nombre de sus inventores: Bahl, Cocke, Jelinek y Raviv. El algoritmo BCJR [20] es capaz de ofrecer decodificación de máxima verosimilitud a posteriori para códigos correctores de errores definibles por Trellises , típicamente códigos convolucionales.

Se considera el problema general de estimar las probabilidades de los estados y transiciones de una fuente de Markov observada desde un canal discreto sin memoria (DMC). Entonces el caso concreto de decodificación de códigos convolucionales para minimizar la probabilidad de error de símbolo es estudiada.

Se hará una adaptación del algoritmo BCJR a nuestro caso, de manera parecida a como se hizo con la decodificación de códigos convolucionales en presencia de símbolos piloto [14]. Los bits que son conocidos con certeza mediante redundancia introducida por el protocolo jugarán ese rol de bits piloto.

5.2. El problema general

El problema general sería la estimación de las probabilidades a posteriori (APP) de los estados y transiciones de una fuente de Markov observada a través de una canal discreto ruidoso sin memoria DMC.

Figura 23: Diagrama esquemático de un sistema de transmisión

Asumiremos que la fuente se trata de una fuente de Markov de tiempo discretizado y número de estados finito con M estados posibles indexados por el rango m = 0, 1, ..., M-1. El estado de la fuente en tiempo será denotado como S_t, y su símbolo de X_t. Una secuencia de estado de tiempo t a t' será referida como $S_t^{t'}$. La misma notación será utilizada para la secuencias de salida.

Las transiciones de estados de la fuente de Markov son gobernadas por la probabilidades de transición y las probabilidades de las salidas, respectivamente:

$$\begin{aligned} p_t(m|m') &= Pr\{S_t = m \,|\, S_{t-1} = m'\} \\ q_t(X \,|\, m, m') &= Pr\{X_t = X \,|\, S_{t-1} = m'; S_t = m\} \end{aligned}$$

Donde X pertenece a algún alfabeto finito y discreto. La fuente de Markov comienza en el estado inicial $S_0 = 0$, y produce una secuencia de salida X_1^τ acabando en el estado terminal $S_\tau = 0$. La secuencia X_1^τ es la entrada a un DMC ruidoso siendo Y_1^τ la secuencia de salida resultante. Las probabilidades de transición del DMC quedan definida por la matriz de canal de probabilidades condicionales $R(\cdot|\cdot)$ para todo $1 \leq t \leq \tau$.

$$Pr\left\{Y_1^t|X_1^t\right\} = \prod_{j=1}^{t} R(Y_j|X_j)$$

En el ejemplo de la Fig. 24, se considera una fuente de Markov de 3 estados $S\epsilon\{0,1,2\}$ y 4 símbolos de salida posibles $X\epsilon\{00,01,10,11\}$ para cada bit de entrada y consecuente cambio de estado. Los estados inicial y final serán 0, como se ha comentado anteriormente, la transición de estados más simple posible pasando por los 3 estados $0 \rightarrow 1 \rightarrow 2 \rightarrow 0$ sería generada por la entrada $\{1,1,0\}$ produciendo la secuencia de símbolos de salida $X_1^3 = \{11,10,11\}$.

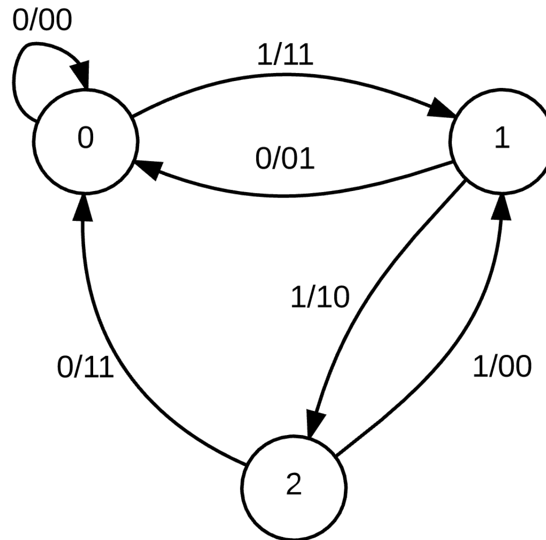


Figura 24: Diagrama de transición de estados y salidas de una fuente de Markov de 3 estados.

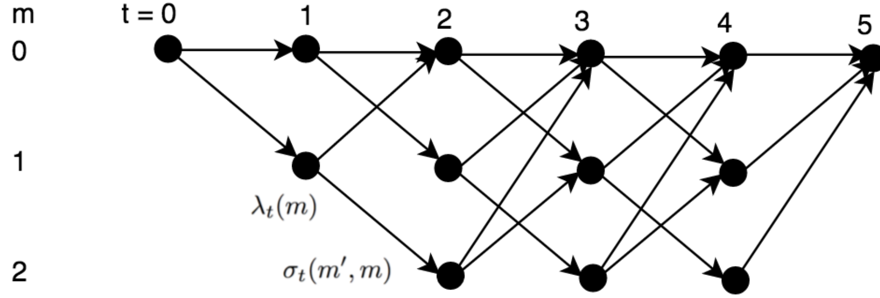


Figura 25: Diagrama de Trellis de la Fig. 24.

El objetivo del decodificador es examinar Y_1^τ y estimar las APP de los estados y transiciones de la fuente de Markov, respectivamente:

$$Pr\{S_t = m|Y_1^\tau\} = \frac{Pr\{S_t = m; Y_1^\tau\}}{Pr\{Y_1^\tau\}} \quad (1)$$

$$Pr\{S_{t-1} = m´; S_t = m|Y_1^\tau\} = \frac{Pr\{S_{t-1} = m´; S_t = m; Y_1^\tau\}}{Pr\{Y_1^\tau\}} \quad (2)$$

Para una secuencia Y_1^τ dada, $\Pr\{Y_1^\tau\}$ es una constante, por lo que podemos definir las probabilidades conjuntas:

$$\begin{aligned} \lambda_t(m) &= Pr\{S_t = m; Y_1^\tau\} \\ \sigma_t(m, m') &= Pr\{S_{t-1} = m´; S_t = m; Y_1^\tau\} \end{aligned}$$

podemos obtenerlas del decodificador como $\lambda_\tau(0)$ o simplemente normalizar $\lambda_t(m)$ y $\sigma_t(m, m')$. El decodificador tiene que computar esas dos últimas probabilidades; las podrá obtener como:

$$\begin{aligned} \lambda_t(m) &= \alpha_t(m)\cdot\beta_t(m) \\ \sigma_t(m', m) &= \alpha_{t-1}(m´).\gamma_t(m´, m).\beta_t(m) \end{aligned}$$

Definiendo las funciones de probabilidad:

$$\begin{aligned} \alpha_t(m) &= Pr\left\{S_t = m; Y_1^t\right\} \\ \beta_t(m) &= Pr\left\{Y_{t+1}^\tau|S_t = m\right\} \\ \gamma_t(m´, m) &= Pr\{S_t = m; Y_t|S_{t-1} = m´\} \end{aligned}$$

Debido a que estamos trabajando con una fuente de Markov y un canal sin memoria, si S_t es conocido, evento después de tiempo t no dependen de eventos pasados Y_1^t.

$$\lambda_t(m) = Pr\left\{S_t = m; Y_1^{\mathrm{t}}\right\} \cdot Pr\left\{Y_{t+1}^{\tau} | S_t = m; Y_1^{\mathrm{t}}\right\}$$
$$= \alpha_t(m) \cdot Pr\left\{Y_{t+1}^{\tau} | S_t = m\right\}$$
$$= \alpha_t(m) \cdot \beta_t(m) \quad (3)$$

De manera similar,

$$\sigma_t(m', m) = Pr\left\{S_{t-1} = m\acute{}; Y_1^{\mathrm{t\text{-}1}}\right\} \cdot Pr\{S_t = m; Y_t | S_{t-1} = m'\} \cdot Pr\{Y_{t+1}^{\tau} | S_t = m\}$$
$$= \alpha_{t-1}(m\acute{}).\gamma_t(m\acute{}, m).\beta_t(m) \quad (4)$$

para t = 1,2,$\cdots$,τ

$$\alpha_t(m) = \sum_{m'=0}^{M-1} Pr\{S_{t-1} = m\acute{}; S_t = m; Y_1^{\mathrm{t}}\}$$
$$= \sum_{m'} Pr\{S_{t-1} = m\acute{}; Y_1^{\mathrm{t}-1}\} \cdot Pr\{S_t = m; Y_1^{\mathrm{t}} | S_{t-1} = m\acute{}\}$$
$$= \alpha_{t-1}(m\acute{}).\gamma_t(m\acute{}, m) \quad (5)$$

Asumiendo nuevamente que los eventos que ocurren después de $t-1$ no son influenciados por $Y_1^{\mathrm{t}-1}$ si S_{t-1} es conocido.

Para el tiempo inicial $t = 0$, encontramos las siguientes condiciones de contorno:

$$\alpha_0(0)\text{=}1 \text{ and } \alpha_0(\mathrm{m})\text{=}0 \text{ for } \mathrm{m}\neq 0 \quad (6)$$

De manera similar para t = 1,2,$\cdots$,τ-1

$$\beta_t(m) = \sum_{m'=0}^{M-1} Pr\left\{S_{t+1} = m'; Y_{t+1}^{\tau} | S_t = m\right\}$$
$$= \sum_{m'} Pr\left\{S_{t+1} = m'; Y_{t+1} | S_t = m\right\} \cdot Pr\{Y_{t+2}^{\tau} | S_{t+1} = m'\}$$
$$= \sum \beta_{t+1}(m') \cdot \gamma_{t+1}(m, m') \quad (7)$$

Con condiciones de contorno:

$$\beta_\tau(0)\text{=}1 \text{ and } \beta_\tau(\mathrm{m})\text{=}0 \text{ for } m \neq 0 \quad (8)$$

Dadas las condiciones de contorno (6) y (8) se observa en las expresiones (5) y (7) que $\alpha_t(m)$ y $\beta_t(m)$ se pueden obtener de manera recursiva una vez que γ_t(m,m') ha sido determinado.

Finalmente,

$$\begin{aligned}
\gamma_t(m,m') &= Pr\{S_t = m; Y_t | S_{t-1} = m\'\} \\
&= \sum_X Pr\{S_t = m | S_{t-1} = m'\} \cdot Pr\{X_t = X | S_{t-1} = m', S_t = m\} \cdot Pr\{Y_t | X\} \\
&= \sum_X p_t(m|m') \cdot q_t(X|m', m) \cdot R(Y_t, X) \qquad (9)
\end{aligned}$$

donde el sumatorio en (9) se efectúa sobre todos los símbolos X posibles en la transición dada $m' \to m$.

Con el algoritmo definido, podemos proceder a elaborar la lista de instrucciones que el decodificador debe seguir para calcular $\lambda_t(m)$ y $\sigma_t(m', m)$, y consecuentemente, estimar las probabilidades a posteriori APP:

1. $\alpha_0(m)$ y $\beta_\tau(m)$ para $m\epsilon\{0, 1, \cdots, M-1\}$ son inicializados según (6) and (8).

2. Cada vez que un Y_t es recibido, el decodificador calcula $\gamma_t(m, m')$ usando (9) y $\alpha_t(m)$ usando (5). Los valores de $\alpha_t(m)$ obtenidos son guardados para todo tiempo t y estado m.

3. Una vez que la secuencia Y_1^τ completa ha sido recibida, el decodificador calcula $\beta_t(m)$ recursivamente utilizando (7). Una vez obtenido, lo multiplicará por el $\alpha_t(m)$ y $\gamma_t(m, m')$ apropiados para obtener $\lambda_t(m)$ y $\sigma_t(m', m)$ usando (3) y (4).

5.3. Aplicación a códigos convolucionales

Dado un codificador convolucional con tasa binaria k_o/n_o y longitud de código $k_o\nu$. La entrada del codificador en tiempo t es la secuencia $I_t = (i_t^{(1)}, i_t^{(2)}, \cdots, i_t^{(k_o)})$, generando la salida resultante $X_t = (x_t^{(1)}, \cdots, x_t^{(n_o)})$. También serán necesarios k_o registros de desplazamiento de longitud ν, incrementado la memoria del codificador o longitud de código descrita previamente. Lo que llamaremos el estado del del codificador, será el contenido de estos registros. Concretamente los últimos ν bloques de entrada recibidos. Para toda la tupla $k_o\nu$:

$$S_t = (s_t^{(1)}, \cdots, s_t^{(k_o\nu)}) = (I_t, I_{t-1}, \cdots, I_{t-\nu+1}) \qquad (10)$$

Por convenio, para establecer unos límites, el codificador siempre debe comenzar y acabar en el estado $S_o = 0$. Entonces una secuencia información I_1^T alimenta el codificador, seguida por ν bloques de ceros $I_{T+1}^\tau = (0, 0, \cdots, 0)$ para forzar el estado final S_τ . Siendo τ la longitud de toda la secuencia de entrada $\tau = T + \nu$.

Considerando un codificador de tasa $1/2$, $\nu = 2$ y longitud de secuencia $\tau = 6$, obtenemos el diagrama de trellis [21] de la figura 26.

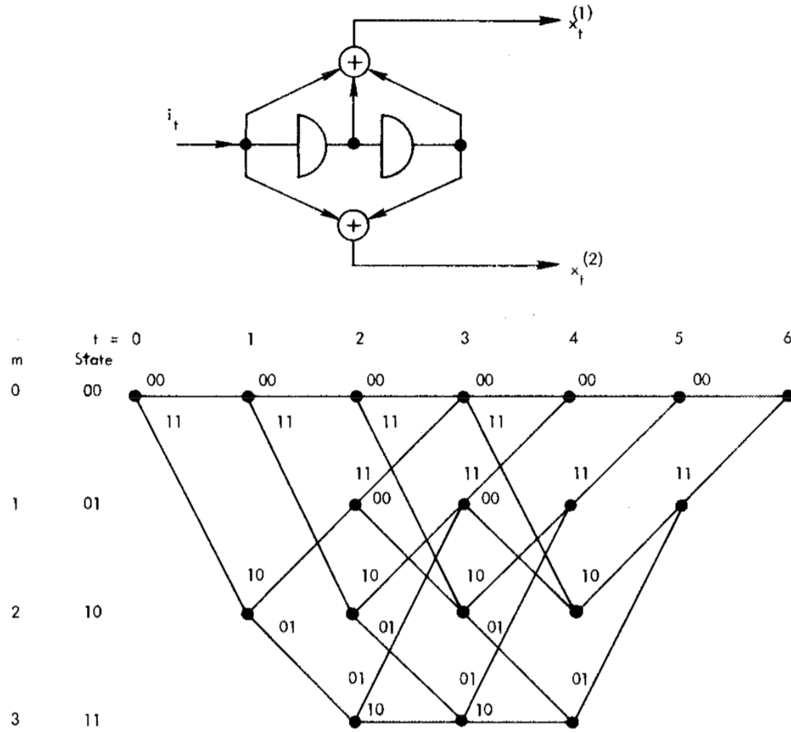


Figura 26: Codificador de tasa 1/2 y diagrama de Trellis

Las probabilidades de transición $p_t(m|m')$ dependen de la secuencia de entrada. Habitualmente todas las entradas posibles son consideradas equiprobables para $t \leq T$. Dado que para cada estado tendremos 2^{k_0} transiciones posibles, podemos decir que $p_t(m|m') = 1/2^{k_0}$ durante ese periodo de tiempo. Para $t > T$, ya que la entrada son siempre 0, sólo hay una transición posible, teniendo probabilidad 1. La salida es una función determinista que depende de las transiciones, por lo que tendrá una probabilidad de distribución binaria $q_t(X|m, m')$ sobre el alfabeto de n-tuplas binarias. Para códigos invariantes $q_t(\cdot|\cdot)$ es independiente de t.

Sobre el canal entre el codificador y el decodificador, consideraremos que la secuencia de entrada es enviada a través de un DMC con probabilidades de símbolo $r(\cdot|\cdot)$, las probabilidades de transición de bloque son:

$$R(Y_t|X_t) \quad = \quad \prod_{j=1}^{n_0} r(y^{(j)}|x_t^{(j)})$$

Siendo $Y_t = (y_t^{(1)}, \cdots, y_t^{(n_0)})$ el bloque de datos a la entrada del receptor. Un ejemplo sencillo sería el caso de una modulación BSC con probabilidad cruzada p_c y distancia de Hamming d entre Y_t y X_t:

$$R(Y_t|X_t) = (p_c)^d(1-p_c)^{n-d}$$

La finalidad es minimizar la probabilidad de error de símbolo. Para ello, debemos encontrar los dígitos de entrada $i_t^{(j)}$ más verosímiles según la secuencia recibida Y_1^τ . Asumiendo que hemos computado $\lambda_t(m)$ como se ha descrito en la sección 5.2. Sea $A_t^{(j)}$ el conjunto de estados S_t tal que $s_t^{(j)} = 0$. Nótese que $A_t^{(j)}$ no depende de t. Entonces usando (10) obtenemos:

$$s_t^{(j)} = i_t^{(j)}, \quad j = 1, 2, \cdots, k_0$$

obteniendo la probabilidad como suma de las lambdas:

$$Pr\{i_t^{(j)} = 0; Y_1^\tau\} = \sum_{S_t \in A_t^{(j)}} \lambda_t(m)$$

Normalizamos por $Pr\{Y_1^\tau\} = \lambda_\tau(0)$ quedando:

$$Pr\{i_t^{(j)} = 0|Y_1^\tau\} = \frac{1}{\lambda_\tau(0)} \cdot \sum_{S_t \in A_t^{(j)}} \lambda_t(m)$$

Finalmente podemos decodificar $i_t^{(j)}$ según:

$$Pr\{i_t^{(j)} = 0|Y_1^\tau\} \begin{cases} > 0{,}5 & i_t^{(j)} = 0 \\ < 0{,}5 & i_t^{(j)} = 1 \end{cases}$$

Dependiendo de la aplicación, podríamos tener necesidad de encontrar las APP de los dígitos de salida del codificador . Por ejemplo en el caso de usar decodificación híbrida bootstrap [22]. Esa sería una función dependiente de las transiciones. Podemos, entonces, seguir un razonamiento similar al anterior. Sea $B_t^{(j)}$ el conjunto de transiciones $S_{t-1} = m' \rightarrow S_t = m$ donde el j-ésimo dígito de salida $x_t^{(j)}$ en esa transición es 0. Siendo $B_t^{(j)}$ independiente de t para códigos invariantes. Entonces:

$$Pr\{x_t^{(j)} = 0; Y_1^\tau\} \quad = \sum_{(m',m) \in B_t^{(j)}} \sigma_t(m', m)$$

6. Protocol-Assisted Channel Decoding

Considere la capa física del transmisor. El enésimo paquete inalámbrico z^s incluye un preámbulo z^p y un paquete z PHY con codificación de canal. El paquete es transmitido sobre un canal (aire) que consideraremos sin memoria y con probabilidad de transmisión $p(y|z)$.

6.1. Estimador Óptimo

El objetivo de un decodificador de canal asistido por protocolo óptimo sería efectuar una estimación del contenido $x = (k, p, u, o, c)$ del paquete físico decodificado de la salida del canal y, aprovechando el hecho de que tanto k como p son perfectamente conocidos, que $u \in \Omega_u$, y que

$$c = f(k, p, u, o) \tag{11}$$

donde f es alguna función de codificación conocida y determinista.

Ya que k y p son conocidos, sólamente u y o deben ser estimados:

$$(\hat{u},\hat{o})_{MAP} = arg \max_{u \in \Omega_u, o, c=f(k,p,u,o)} p(u, o|y, k, p) \tag{12}$$

Obsérvese que realmente el único campo que haría falta estimar es u, y esa perfectamente precisa soft información se transmitiría a capas superiores para un posterior procesamiento como se propuso en [18], que permitiría un procesamiento conjunto completamente eficiente llevado a cabo en todas las capas. En este caso asumiremos que se efectúa una decisión firme (hard) tras el procesamiento en la capa física, que además tiene la ventaja de no introducir un retraso adicional en comparación con la situación clásica.

El estimador óptimo (12) tiene una gran complejidad de implementación, ya que sólo las parejas (u, o) que satisfacen (11) deben ser tenidas en cuenta. De hecho, la decodificación óptima implicaría un trellis global sobre el código convolucional y el código de comprobación [19]. En ese caso sólo los caminos pertenecientes a $u \in \Omega_u$ deben ser tenidos en cuenta.

6.2. Estimador Subóptimo

Dada la gran complejidad del estimador óptimo comentada en la sección anterior, se limitaran una serie de condiciones para encontrar un estimador subóptimo más implementable. Por un lado, dejaremos de tener en cuenta la relación (11) y la condición $u \in \Omega_u$ será traducida a probabilidades a priori $p(u_i = 0), i = 1, ..., long(u)$ para cada bit de u. Podemos decir que k y p tienen el mismo rol en cuanto a la estimación, por lo que englobaremos k dentro de p. Bajo estas

condiciones podemos definir un estimador subóptimo bit a bit para la entrada x,

$$\hat{x_i} \quad = \quad arg \max_{x \in \{0,1\}} p(x|y,k) \tag{13}$$

con probabilidades a priori para las entradas de x como:

$$p(x_i = 0) \quad = \quad \left\{ \begin{array}{cc} p(k_{\tau(i)} = 0) & si\ x_i\ es\ \tau - ésimo\ bit\ de\ k, \\ p(u_{\tau(i)} = 0) & si\ x_i\ es\ \tau - ésimo\ bit\ de\ u, \\ 0,5 & else \end{array} \right\} \tag{14}$$

donde $\tau(\cdot)$ es alguna función de traducción de índices.

Seguidamente, utilizando una ligera adaptación del algoritmo BCJR [20] podemos obtener la estimación (13). Algo parecido al caso de decodificación convolucional haciendo uso de símbolos piloto [14]. Asumiremos que la tasa de codificador convolucional es $1/\nu$. El estado del codificador es $S_n, n = 0, ..., lon(x)$, con valores en S. El siguiente estado $s' = t(s, x)$ y la salida $v = \sigma(s, x)$ se obtienen cuando el estado inicial del codificador es s y el bit de entrada es x.

7. Aplicación al estándar 802.11a

El rendimiento del decodificador de canal asistido por protocolo propuesto será evaluado sobre el popular protocolo WiFi 802.11a [15].

Como se ha mencionado anteriormente, esta técnica requiere la recepción previa de cabeceras de la capa MAC visibles para la capa física. En este caso sólo la capa MAC será visible, pero se podría extender a capas superiores. El estimador MAP es implementado para interpretar las cabeceras y carga útil con CRC en la capa MAC sobre la modulación OFDM (Orthogonal Frequency Division Multiplexing) para la banda de 5 Ghz definida en el estándar.

Procederemos pues, a estudiar las capas PHY y MAC del protocolo 802.11a para identificar los campos descritos en la sección 4.

7.1. Encapsulación y codificación

La encapsulación en el caso del protocolo 802.11a es muy similar el caso genérico visto en la sección 4.

El MPDU (MAC Protocol Data Unit) llega a la capa PHY con una cabecera y un tail de bits. Entonces un scrambling sin memoria y codificación convolucional es aplicada a parte del paquete (principalmente MPDU) para obtener PSDU (Physical layer Service Data Unit). Se añaden unos símbolos de entrenamiento (preámbulo) antes de efectuar el entrelazado y mapeo del paquete resultante a símbolos OFDM a ser transmitidos utilizando una modulación BPSK, QPSK, 16-QAM o 64-QAM.

7.2. Descripción de la capa MAC

Consideremos una transmisión downlink no encriptada de tramas MAC ordenados con la retransmisión desactivada y en modo ahorro de energía con el medio de transmisión previamente reservado por un diálogo RTS-CTS satisfactorio entre el AP (Access Point) y el receptor. Con esta configuración, todos los bits del segundo byte del campo de *Frame Control* excepto el flag *More Frag* son conocidos.

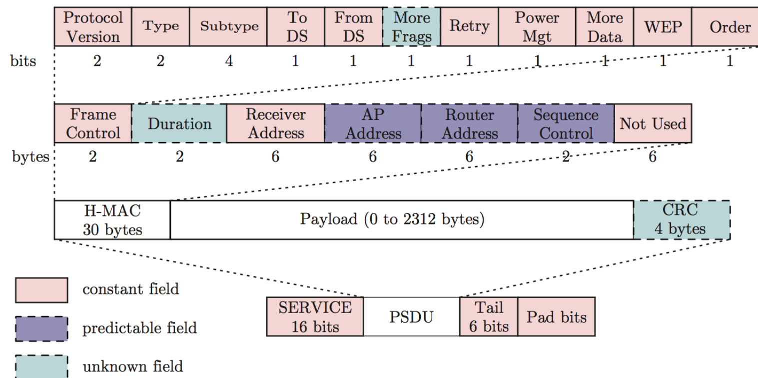


Figura 27: Tramas PHY y MAC para 802.11a

El campo *AP Address* contiene la dirección MAC del AP. Durante el proceso de reserva del medio, la dirección es transmitida el receptor, pudiendo ser utilizada por el mismo.

El campo *Router Address* corresponde a la dirección MAC del rúter. Asumiendo que el AP tan sólo está conectado a un sólo rúter, la dirección ha sido recibida previamente y es conocida por el receptor.

El campo *Sequence Control* está compuesto de dos parámetros: un número de secuencia y un número de fragmento. El primero indica el valor del contador de trama IP actual. Mientras que el segundo indica el valor del contador de trama MAC. En este caso, los tramas son enviados ordenadamente, por lo que ambos valores se pueden determinar fácilmente. El número de secuencia se incrementa en uno por cada RTS-CTS, y el número de fragmento se incrementará en uno por cada trama MAC.

El último campo de la cabecera MAC se reserva para redes inalámbricas locales se asumirá que está compuesto por 6 bytes de ceros. Un código CRC de 32 bits protege la cabecera MAC y la carga útil.

El campo *Signal* consiste de 24 bits que definen la tasa de datos y longitud de trama.

La versión de 802.11a para OFDM utiliza una combinación de BPSK, QPSK y QAM dependiendo de la tasa de datos escogida.

El campo *Length* identifica el número de bytes en la trama.

El preámbulo PLCP y campo *Signal* codificados convolucionalmente y enviados a 6Mbps utilizando BPSK siempre. Por lo que la tasa de codificación dependerá de la tasa de datos escogida.

El flag *More Frag* informa de si el la trama MAC actual es la última de una secuencia de fragmentos de una trama IP.

El campo *Duration* indica el numero de microsegundos necesarios para transmitir la siguiente trama MAC y algunos parámetros de control. Su valor depende de la modulación y el tamaño de la próxima trama MAC.

Por lo tanto, todos ellos son conocidos por el receptor.

Finalmente, *Payload* sería la carga útil y contendría los datos a ser transmitidos. Su tamaño varía entre 0 y 2312 bytes.

7.3. Descripción de la capa PHY

La capa física OFDM contiene la subcapa PLCP (Physical Layer Convergence Procedure). La trama de este último contiene la cabecera OFDM PLCP, la PSDU (que sería la MPDU a nivel MAC) y bits de tail y padding, como se puede ver en la figura 28.

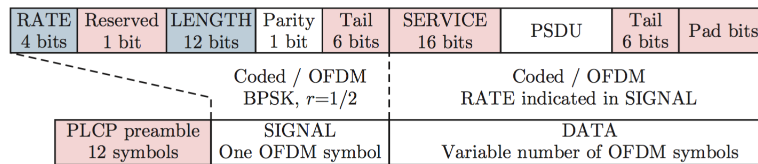


Figura 28: Trama PHY para 802.11a

La cabecera PLCP contiene los campos: *Length, Rate, Reserved bit, Parity bit* y *Service*. A efectos de modulación, estos campos conforman un símbolo OFDM separado. denotado como *Signal*.

El campo *Service* de la cabecera PLCP, el PSDU, además de los bits de tail y padding son agrupados como DATA. Son transmitidos a la tasa de datos determinada por el campo *Rate* y pueden contener diversos símbolos OFDM codificados con un código convolucional no recursivo no sistemático.

La cabecera PLCP es transmitida con la combinación más robusta de modulación BPSK y tasa de código de R=1/2. Los bits de tail en el símbolo *Signal* permiten la decodificación de los campos *Length* y *Rate* inmediatamente tras la recepción de los bits de tail. Asumiremos que los campos *Length* y *Rate* son recuperados sin errores.

La subcapa OFDM PMD acepta primitivas de la subcapa PLCP y le añade el preámbulo OFDM PLCP. Esta capa, la más baja, aporta finalmente los medios por los cuales se transmite y reciben los datos por el medio físico.

7.4. Resultados de la Simulación

La simulación ha sido diseñada con código C++ haciendo uso de la librería IT++ y extendiendo su funcionalidad añadiendo el decodificador de canal asistido por protocolo propuesto. El código puede ser consultado en el apéndice A.

Seguidamente se establecen los parámetros en los cuales se ha llevado cabo la simulación.

El canal, presente tras el entrelazador del emisor y antes del desentrelazador del emisor, ha sido modelado por un canal de ruido gaussiano blanco aditivo

AWGN con una calidad de señal a ruido SNR conocida. En consecuencia, juntamente con las hipótesis de la sección 7.2, los campos que quedan por estimar son: la carga útil MAC, *Duration, More Frags* y el CRC de 32 bits.

El programa se encarga de generar paquetes acordes con el estándar 802.11a. La carga útil MAC ocupa 30 bytes. Una modulación BPSK con tasa de codificación R=1/2 para código convolucional se aplica en todas las partes del paquete PHY.

Los test se llevan a cabo para distintos decodificadores para evaluar las mejoras e influencia de los distintos parámetros. Además se hace un barrido para distintas SNR para observar su comportamiento en distintas condiciones.

El decodificador indeciso no informado se trata de un BCJR clásico para código convolucional como el descrito en la sección 4. No tiene ningún conocimiento del contenido de los paquetes.

El decodificador indeciso informado es igual que el anterior, pero además hace uso del conocimiento de los campos conocidos y predecibles tras haber hecho la decodificación. En otras palabras, se hace una corrección a posteriori de los bits de los cuales se conocen con certeza.

El decodificador robusto es una implementación del estimador subóptimo propuesto en la sección 6.

Para medir el rendimiento de cada uno de los decodificadores, en primer lugar se obtienen los resultados en términos de HER (Header Error Rate), calculando la proporción de cabeceras MAC decodificadas erróneamente como una función de la SNR. Se observa una ganancia de más de 3dB en comparación a los otros dos decodificadores, que tienen un comportamiento similar.

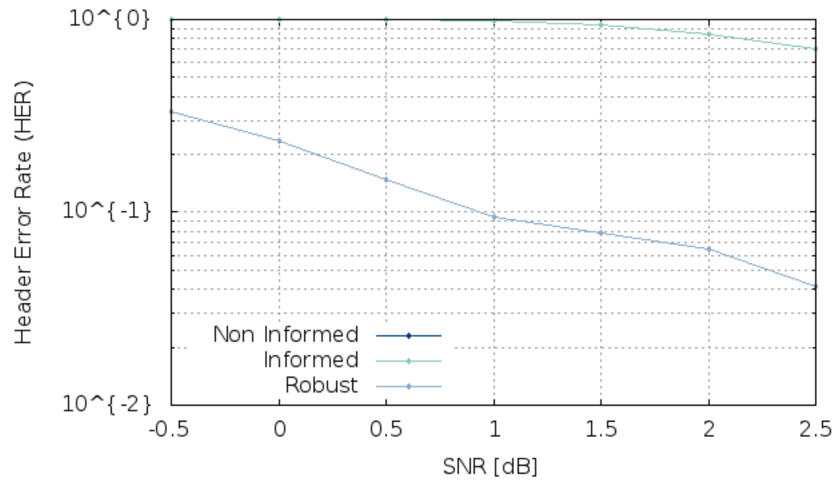


Figura 29: Header Error Rate simulación

También se hace un análisis del rendimiento en términos de FER (MAC Frame Error Rate). En este caso el decodificador robusto presenta una gran ganancia respecto a los otros dos estimadores, ya que estos apenas consiguen evitar errores Sí además se añade un entrelazador, como es el caso de la simulación, el contenido del paquete antes de llevar a cabo la codificación convolucional, la ganancia aumenta más de 1dB. Esto se debe a que los bits que son utilizados como pilotos, se reparten mejor en el paquete ayudando a solucionar el Trellis y decodificar los bits a estimar más eficientemente.

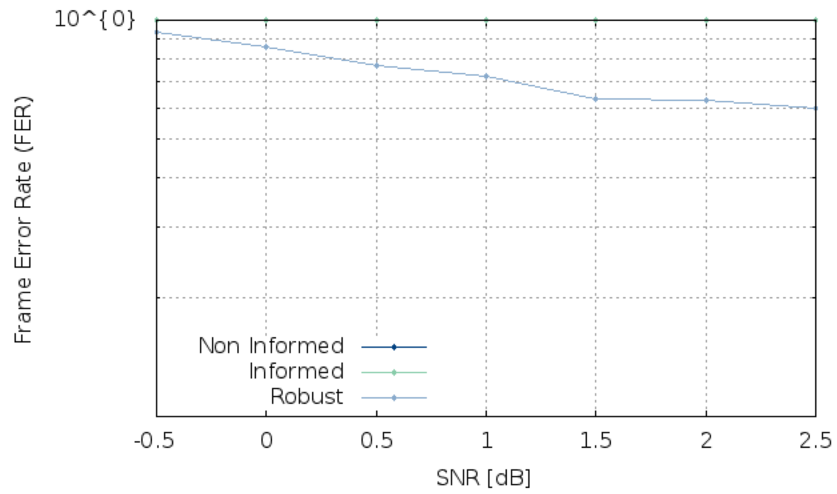


Figura 30: Frame Error Rate simulación

Complementariamente se adjuntan en el apéndice A los resultados de la simulación llevada a cabo con sucesivas iteraciones hasta que el decodificador robusto contabiliza 1000 errores.

8. Conclusiones

En este documento se ha podido demostrar como se puede sacar provecho de la redundancia presente en la pila de protocolo utilizando esa información conocida como bits piloto. Una técnica que recuerda a la decodificación de canal con símbolos piloto. Pero esta vez se obtienen los bits piloto mediante una exhaustiva inspección del protocolo.

Las simulaciones llevadas a cabo demuestran una mejora tanto en HER como en FER, demostrando las posibilidades de la solución. Concretamente, respecto al decodificador indeciso clásico se han medido unas mejoras de más de 3 dB para el HER y mayor para el FER, aunque debido al pobre comportamiento de los decodificadores soft. Si además se hace uso de entrelazado antes y después de la transmisión se consigue aumentar el FER en 1 dB más (incluido en la simulación).

Ciertamente estos resultados han sido obtenidos en un contexto simplificado. Cabría tener en cuenta un caso más cercano a la práctica, situación con múltiples usuarios o APs que introduzcan interferencias, una carga útil mayor, la inclusión del establecimiento de RTS-CTS. En este caso el decodificador debería perder eficiencia. Aún así, se podría extender la técnica al conocimiento de redundancia presente en capas superiores y hacer uso del campo de CRC. En este caso, sería posible llevar a cabo una decodificación recursiva entre la codificación de canal y el CRC. Esta solución recuerda en parte al comportamiento de los populares turbo códigos, que también trabajan de forma iterativa. Sería interesante llevar acabo esta una comparación en el futuro para determinar si podrían competir con los turbo códigos en ciertas condiciones.

A. Apéndice

A.1. Código desarrollado

convcode_mod.h

```
/*
 *  convcode_mod.h
 *  Definitive
 *
 *  Created by Dani on 18/11/10.
 *
 */

#ifndef CONVCODE_MOD_H
#define CONVCODE_MOD_H

#include <itpp/comm/convcode.h>
#include <itpp/comm/llr.h>

    namespace itpp {

    //enum CONVOLUTIONAL_CODE_MOD_TYPE {MFD, ODS};
    //enum CONVOLUTIONAL_CODE_MOD_METHOD {Trunc, Tail, \
        Tailbite};

        class Convolutional_Code_Mod : public \
            Convolutional_Code
        {
        public:
            double Lc;
            void set_awgn_channel_parameters(double Ec, \
                double N0);
            void output(const int state, const bin input, \
                bvec &output);
//          void calc_output_table(bvec output_table[]);
            double branch_prob(int t, int m, int input, vec \
                P0, vec P1);
            void protocol_assisted_channel_decoding(const \
                vec P0, const vec P1, const int TL, const \
                vec maskBits, bvec &output);
            void map_decode(const vec input, bvec &output, \
                mat &trans_prob, bool in_terminated);
            void log_decode(const QLLRvec &input, bvec &\
                output, QLLRmat &trans_prob, bool \
                in_terminated);


        private:
            mat gamma, alpha, beta, lambda;
```

```
            vec denom;
            QLLRmat gamma_q, alpha_q, beta_q, lambda_q;
            QLLRvec denom_q;
            LLR_calc_unit llrcalc;
        };
    }

#endif
```

convcode_mod.cpp

```
/*
 *  convcode_mod.cpp
 *  RSCC_test
 *
 *  Created by Daniel on 18/11/10.
 *  Copyright 2010 SupÃ©lec. All rights reserved.
 *
 */

//Assuming BPSK Modulation

#include "convcode_mod.h"
#include <iostream>

using std::cout;
using std::endl;

namespace itpp{

    void Convolutional_Code_Mod::\
        set_awgn_channel_parameters(double Ec, double N0)
    {
        Lc = 4.0 * std::sqrt(Ec) / N0;
    }


    void Convolutional_Code_Mod::output(const int state, \
        const bin input, bvec &output){
        int prev_state = 0;

        prev_state = encoder_state;
        encoder_state = state;
        encode_bit(input, output);
        encoder_state = prev_state;
    }

    //void Convolutional_Code_Mod::calc_output_table(bvec \
        output_table[]){
//      for(int i = 0; i < pow2i(K); i++){
//          output_table[i][0] = dec2bin(K, i);
//          set_start_state(i & (pow2i(m)-1));
//          init_encoder();
//          encode_bit((i >> m), output_table[i][1]);
//      }
//  }
    //bvec output_table[pow2i(K)][2];
```

```
double Convolutional_Code_Mod::branch_prob(int t, int m,\
     int input, vec P0, vec P1){

    bvec codeword;
    output(m, input, codeword);

    //first row of "P_coded_bits" is Prob. that coded \
        bits equal to 0
    //second row of "P_coded_bits" is Prob. that coded \
        bits equal to 1
    mat P_coded_bits;

    P_coded_bits.set_size(2, P0.size(), false);
    P_coded_bits.set_row(0, P0);
    P_coded_bits.set_row(1, P1);

    return (P_coded_bits(codeword(0), 2*t) * \
        P_coded_bits(codeword(1), 2*t+1));
}

void Convolutional_Code_Mod::\
    protocol_assisted_channel_decoding(const vec P0, \
    const vec P1, const int TL, const vec maskBits, bvec\
     &output){


    int t, k, i, in, trans, s, s0, s1, s_prim, s_prim0, \
        s_prim1, input_symbols = pow2i(K - m);
    double b_pr[no_states][TL][input_symbols];
    double temp0, temp1, nom, den;
    mat app0, app1; //APP of the encoder output digits
    vec app_input;
    //bvec output_table[pow2i(K)][2];
    bvec **output_table = new bvec* [pow2i(K)];
    for (int i = 0; i < pow2i(K); i++){
        output_table[i] = new bvec[2];
    }

    for(t = 0; t < TL; t++){
        for(s = 0; s < no_states; s++){
            for(in = 0; in < input_symbols; in++){
                b_pr[s][t][in] = 0;
            }
        }
    }

    //Calculation of the branch probability for every \
        transition in the trellis
    for(t = 0; t < TL; t++){
        for(s = 0; s < no_states; s++){
```

```
        switch (int(maskBits(t))) {
            case 0:
                b_pr[s][t][0] = branch_prob(t, s, 0,\
                     P0, P1);
                break;
            case 1:
                b_pr[s][t][1] = branch_prob(t, s, 1,\
                     P0, P1);
                break;
            case 2:
                for(in = 0; in < input_symbols; in\
                    ++){
                     b_pr[s][t][in] = branch_prob(t, \
                         s, in, P0, P1);
                }
                break;
            default:
                cout << "Invalid␣value␣in␣maskBits";
                exit(-1);
        }
    }
}



gamma.set_size(2*no_states, TL + 1, false);
gamma.zeros();

for(t = 0; t < TL; t++){
    for(s = 0; s < no_states; s++){
        for(in = 0; in < input_symbols; in++){
            gamma(s | (in << m), t+1) = b_pr[s][t][\
                in];
        }

    }
}/*
cout << "branch_prob: ";
          for(t = 0; t < TL; t++){
             for(s = 0; s < no_states; s++){
                 for(in = 0; in < 2; in++){
                     cout << b_pr[s][t][in] << " ";
                 }
                 cout << endl;
             }
}

cout << "gamma:" << endl << gamma << endl;
```

```
        //cout << P0.size() << endl;
//      cout << P1.size() << endl;*/
//


        //cout << maskBits;
        //cout << gamma;

        //Alpha calculation
        alpha.set_size(no_states, TL + 1, false);
        alpha.set_col(0, zeros(no_states));
        alpha(0, 0) = 1.0;
        denom.set_size(TL + 1, false);
        output.set_size(TL, false);

        //Calculate alpha and denom going forward through \
            the trellis
        for(t = 1; t < TL +1; t++){
            for (s = 0; s < no_states; s++) {
                s_prim0 = ((s&(pow2i(m-1)-1)) << 1);     //\
                    s_prim0 and s_prim1 are the 2 possible \
                    past states (m') that can evolve to s (m\
                    )
                s_prim1 = ((s&(pow2i(m-1)-1)) << 1) | 1;
                temp0 = alpha(s_prim0, t - 1) * gamma((s << \
                    1) | 0, t);
                temp1 = alpha(s_prim1, t - 1) * gamma((s << \
                    1) | 1, t);
                alpha(s, t) = temp0 + temp1;
                denom(t)  += temp0 + temp1;
            }
            alpha.set_col(t, alpha.get_col(t) / denom(t));
        }

        //Initiate beta
        //if (terminated) {
//          beta.set_col(TL, zeros(no_states));
//          beta(0, TL) = 1.0;
//      }
//      else {
//          beta.set_col(TL, alpha.get_col(TL));
//      }
        beta.set_size(no_states, TL + 1, false);
        beta.set_col(TL, zeros(no_states));
        beta(0, TL) = 1.0;
        denom.zeros();

        //Calculate beta going backward in the trellis
        for (t = TL; t >= 2; t--) {
            for (s_prim = 0; s_prim < no_states; s_prim++) {
```

```
            s0 = (s_prim >> 1); //s0 is the next state \
                if the input is 0
            s1 = (s_prim >> 1) | (1 << (m-1)); //s1 is \
                the next state if the input is 1
            beta(s_prim, t - 1) = (beta(s0, t) * \
                gamma((s0 << 1) | s_prim, t)) + (beta(s1\
                , t) * gamma((s1 << 1) | s_prim, t));
            denom(t)  += beta(s_prim, t - 1);

        }
        if(denom(t) == 0) denom(t) = 1;
        beta.set_col(t - 1, beta.get_col(t - 1) / \
            denom(t));
    }

    //cout << "alpha: " << endl <<  alpha << endl;
    //cout << "beta: " << endl <<  beta << endl;

    app_input.set_size(TL + 1, false);
    app0.set_size(n, TL + 1, false);
    app0.zeros();
    app1.set_size(n, TL + 1, false);
    app1.zeros();
    lambda.set_size(no_states, TL + 1, false);


    //compute lambda
    for (k = 0; k <= TL; k++){
        for(s = 0; s < no_states; s++){
            lambda(s, k) = alpha(s, k) * beta(s, k);
        }
    }

    for(int i = 0; i < pow2i(K); i++){
        output_table[i][0] = dec2bin(K, i);
        set_start_state(i & (pow2i(m)-1));
        init_encoder();
        encode_bit((i >> m), output_table[i][1]);
    }

    //APP probabilities
    for (k = 1; k <= TL; k++){
        for(i = 0; i < n; i++){
            for(trans = 0; trans < 2*no_states; trans++)\
                { //all possible transitions
                 if(output_table[trans][1](i) == bin(0)) \
                    //transition such that the ith \
                    output digit is 0
                         app0(i, k) += alpha(trans & \
                            (pow2i(m)-1), k-1) * \
```

```
                         gamma(trans, k) * beta(trans\
                          >> 1, k);
                 if(output_table[trans][1](i) == bin(1)) \
                     //transition such that the ith \
                     output digit is 1
                         app1(i, k) += alpha(trans & \
                             (pow2i(m)-1), k-1) * \
                             gamma(trans, k) * beta(trans\
                              >> 1, k);
             }
             app0(i, k) = trunc_log(app0(i, k) / app1(i, \
                 k));
         }
     }

     for (k = 1; k <= TL; k++){
         nom = 0;
         den = 0;
         for(s_prim = 0; s_prim < no_states; s_prim++){ \
             //all possible transitions
             s0 = s_prim >> 1;
             s1 = (s_prim | (1 << m)) >> 1;
                     nom += alpha(s_prim, k-1) * \
                         gamma(s_prim, k) * beta(s0, k);
                     den += alpha(s_prim, k-1) * \
                         gamma(s_prim | (1 << m), k) * \
                         beta(s1, k);
         }
         app_input(k) = trunc_log(nom / den);
     }

     //Decisor
     for (k = 1; k <= TL; k++){
         if(app_input(k) > 0) output(k-1) = itpp::bin(0);
         else output(k-1) = itpp::bin(1);
     }
     //cout << "app_input: " << app_input << endl;

     for (int i = 0; i < pow2i(K); i++){
         delete [] output_table[i];
     }
     delete [] output_table;

}

void Convolutional_Code_Mod::map_decode(const vec input,\
     bvec &output, mat &trans_prob, bool in_terminated)
{
     // input : Output of Gaussian channel (BPSK- \
        modulated)
```

```
int i;
mat prob_trans;
//bvec* output_table = new bvec[pow2i(K)][2];
bvec temp;
bvec **output_table = new bvec* [pow2i(K)];
for (int i = 0; i < pow2i(K); i++)
    output_table[i] = new bvec[2];

//std::vector<bvec> output_table(pow2i(K))[];
//bvec output_table[pow2i(K)][2], temp;

bool terminated = false;

double temp0, temp1, nom, den;
int s0, s1, k, kk, s, s_prim, s_prim0, s_prim1, \
    trans, block_length = input.length()/n; //number\
     of symbols recieved
ivec p0, p1;
mat app0, app1; //APP of the encoder output digits
vec app_input;

app_input.set_size(block_length + 1, false);
app0.set_size(n, block_length + 1, false);
app0.zeros();
app1.set_size(n, block_length + 1, false);
app1.zeros();
temp.set_size(n, false);

alpha.set_size(no_states, block_length + 1, false);
beta.set_size(no_states, block_length + 1, false);
gamma.set_size(2*no_states, block_length + 1, false)\
    ;
lambda.set_size(no_states, block_length + 1, false);
denom.set_size(block_length + 1, false);
output.set_size(block_length, false);

if (in_terminated) { terminated = true; }

//Output table
for(int i = 0; i < pow2i(K); i++){
    output_table[i][0] = dec2bin(K, i);
    set_start_state(i & (pow2i(m)-1));
    init_encoder();
    encode_bit((i >> m), output_table[i][1]);
}


//Calculate gamma
for (k = 1; k <= block_length; k++) {
```

```
    kk = k - 1;
    for(i = 0; i < n; i++){
        if(input(kk*n+i) > 0) temp(i) = 0;
        else temp(i) = 1;
    }
    for (trans = 0; trans < 2*no_states; trans++) { \
        //trans is the extended state (input<<2|\
        state) all possible transitions
        gamma(trans, k) = 0.5 * \
            trans_prob(bin2dec(output_table[trans][1\
            ], true), bin2dec(temp, true));  //\
            partial hard decision??
    }
}

//cout << "gamma: " << gamma << endl;

//Initiate alpha
alpha.set_col(0, zeros(no_states));
alpha(0, 0) = 1.0;

//Calculate alpha and denom going forward through \
    the trellis
for (k = 1; k <= block_length; k++) {
    for (s = 0; s < no_states; s++) {
        s_prim0 = ((s&(pow2i(m-1)-1)) << 1);     //\
            s_prim0 and s_prim1 are the 2 possible \
            past states (m') that can evolve to s (m\
            )
        s_prim1 = ((s&(pow2i(m-1)-1)) << 1) | 1;
        temp0 = alpha(s_prim0, k - 1) * gamma((s << \
            1) | 0, k);
        temp1 = alpha(s_prim1, k - 1) * gamma((s << \
            1) | 1, k);
        alpha(s, k) = temp0 + temp1;
        denom(k)  += temp0 + temp1;
    }
    alpha.set_col(k, alpha.get_col(k) / denom(k));
}

//Initiate beta
if (terminated) {
    beta.set_col(block_length, zeros(no_states));
    beta(0, block_length) = 1.0;
}
else {
    beta.set_col(block_length, \
        alpha.get_col(block_length));
}
```

```
//Calculate beta going backward in the trellis
for (k = block_length; k >= 2; k--) {
    for (s_prim = 0; s_prim < no_states; s_prim++) {
        s0 = (s_prim >> 1);
        s1 = (s_prim >> 1) | (1 << (m-1));
        beta(s_prim, k - 1) = (beta(s0, k) * \
            gamma((s0 << 1) | s_prim, k)) + (beta(s1\
            , k) * gamma((s1 << 1) | s_prim, k));
    }
    beta.set_col(k - 1, beta.get_col(k - 1) / \
        denom(k));
}

//compute lambda
for (k = 0; k <= block_length; k++){
    for(s = 0; s < no_states; s++){
        lambda(s, k) = alpha(s, k) * beta(s, k);
    }
}

//APP probabilities
for (k = 1; k <= block_length; k++){
    for(i = 0; i < n; i++){
        for(trans = 0; trans < 2*no_states; trans++)\
            { //all possible transitions
            if(output_table[trans][1](i) == bin(0)) \
                //transition such that the ith \
                output digit is 0
                    app0(i, k) += alpha(trans & \
                        (pow2i(m)-1), k-1) * \
                        gamma(trans, k) * beta(trans\
                         >> 1, k);
            if(output_table[trans][1](i) == bin(1)) \
                //transition such that the ith \
                output digit is 1
                    app1(i, k) += alpha(trans & \
                        (pow2i(m)-1), k-1) * \
                        gamma(trans, k) * beta(trans\
                         >> 1, k);
        }
        app0(i, k) = trunc_log(app0(i, k) / app1(i, \
            k));
    }
}

for (k = 1; k <= block_length; k++){
    nom = 0;
    den = 0;
    for(s_prim = 0; s_prim < no_states; s_prim++){ \
        //all possible transitions
```

```
            s0 = s_prim >> 1;
            s1 = (s_prim | (1 << m)) >> 1;
                    nom += alpha(s_prim, k-1) * \
                        gamma(s_prim, k) * beta(s0, k);
                    den += alpha(s_prim, k-1) * \
                        gamma(s_prim | (1 << m), k) * \
                        beta(s1, k);
        }
        app_input(k) = trunc_log(nom / den);
    }

    //Decisor
    for (k = 1; k <= block_length; k++){
        if(app_input(k) > 0) output(k-1) = itpp::bin(0);
        else output(k-1) = itpp::bin(1);
    }

    for (int i = 0; i < pow2i(K); i++){
        delete [] output_table[i];
    }
    delete [] output_table;
}


//  === Below new decoder function using QLLR \
    arithmetics ========================================\
    =====
//  ==================================================\
    ================================================


void Convolutional_Code_Mod::log_decode(const QLLRvec &\
    input, bvec &output, QLLRmat &trans_prob, bool \
    in_terminated){
    // input : Output of Gaussian channel (BPSK- \
        modulated)

    int i;
    mat prob_trans;
    //bvec output_table[pow2i(K)][2], temp;
    bvec temp;
    bvec **output_table = new bvec* [pow2i(K)];
    for (int i = 0; i < pow2i(K); i++){
        output_table[i] = new bvec[2];
    }
    bool terminated = false;

    QLLR nom, den, temp0, temp1;
    int s0, s1, k, kk, s, s_prim, s_prim0, s_prim1, \
        trans, block_length = input.length()/n; //number\
```

```
     of symbols recieved
ivec p0, p1;
QLLRmat app0, app1; //APP of the encoder output \
    digits
QLLRvec app_input;


app_input.set_size(block_length + 1, false);
app0.set_size(n, block_length + 1, false);
app0.zeros();
app1.set_size(n, block_length + 1, false);
app1.zeros();
temp.set_size(n, false);

alpha_q.set_size(no_states, block_length + 1, false)\
    ;
beta_q.set_size(no_states, block_length + 1, false);
gamma_q.set_size(2*no_states, block_length + 1, \
    false);
lambda_q.set_size(no_states, block_length + 1, false\
    );
denom_q.set_size(block_length + 1, false);
output.set_size(block_length, false);

for (k = 0; k <= block_length; k++) { denom_q(k) = -\
    QLLR_MAX; }


if (in_terminated) { terminated = true; }

Lc = 1.0;
//Check that Lc = 1.0
it_assert(Lc == 1.0,
          "Rec_Syst_Conv_Code::log_decode:␣This␣\
              function␣assumes␣that␣Lc␣=␣1.0.␣Please\
              ␣use␣proper␣scaling␣of␣the␣input␣data"\
              );

//Output table
for(i = 0; i < pow2i(K); i++){
    output_table[i][0] = dec2bin(K, i);
    set_start_state(i & (pow2i(m)-1));
    init_encoder();
    encode_bit((i >> m), output_table[i][1]);
}

//Calculate gamma_q
for (k = 1; k <= block_length; k++) {
    kk = k - 1;
    for(i = 0; i < n; i++){
```

```
            if(input(kk*n+i) > 0) temp(i) = 0;
            else temp(i) = 1;
        }
        for (trans = 0; trans < 2*no_states; trans++) { \
            //trans is the extended state (input<<2|\
            state) all possible transitions
            gamma_q(trans, k) = itpp::trunc_log(0.5) + \
                trans_prob(bin2dec(output_table[trans][1\
                ], true), bin2dec(temp, true));
        }
    }

    //Initiate alpha
    for (i = 1; i < no_states; i++) { alpha_q(i, 0) = -\
        QLLR_MAX; }
    alpha_q(0, 0) = 0;

    //Calculate alpha and denom going forward through \
        the trellis
    for (k = 1; k <= block_length; k++) {
        for (s = 0; s < no_states; s++) {
            s_prim0 = ((s&(pow2i(m-1)-1)) << 1);      //\
                s_prim0 and s_prim1 are the 2 possible \
                past states (m') that can evolve to s (m\
                )
            s_prim1 = ((s&(pow2i(m-1)-1)) << 1) | 1;
            temp0 = alpha_q(s_prim0, k - 1) + gamma_q((s\
                 << 1) | 0, k);
            temp1 = alpha_q(s_prim1, k - 1) + gamma_q((s\
                 << 1) | 1, k);
            alpha_q(s, k) = llrcalc.jaclog(temp0, temp1)\
                ;
            denom_q(k)   = llrcalc.jaclog(alpha_q(s, k),\
                 denom_q(k));

        }
        //Normalization of alpha_q
        for (s = 0; s < no_states; s++) { alpha_q(s, k) \
            -= denom_q(k); }
    }

    //Initiate beta
    if (terminated) {
        for (i = 1; i < no_states; i++) { beta_q(i, \
            block_length) = -QLLR_MAX; }
        beta_q(0, block_length) = 0;
    }
    else {
        beta_q.set_col(block_length, \
            alpha_q.get_col(block_length));
```

```
}

//Calculate beta going backward in the trellis
for (k = block_length; k >= 2; k--) {
    for (s_prim = 0; s_prim < no_states; s_prim++) {
        s0 = (s_prim >> 1);
        s1 = (s_prim >> 1) | (1 << (m-1));
        beta_q(s_prim, k - 1) = \
            llrcalc.jaclog(beta_q(s0, k) + \
            gamma_q((s0 << 1) | s_prim, k), \
            beta_q(s1, k) + gamma_q((s1 << 1) | \
            s_prim, k));
    }
    //Normalization of beta_q
    for (s = 0; s < no_states; s++) { beta_q(s, k - \
        1) -= denom_q(k); }
}

//compute lambda
for (k = 0; k <= block_length; k++){
    for(s = 0; s < no_states; s++){
        lambda_q(s, k) = alpha_q(s, k) + beta_q(s, k\
            );
    }
}

//APP probabilities
for (k = 1; k <= block_length; k++){
    for(i = 0; i < n; i++){
        for(trans = 0; trans < 2*no_states; trans++)\
            { //all possible transitions
            if(output_table[trans][1](i) == bin(0)) \
                //transition such that the ith \
                output digit is 0
                app0(i, k) = llrcalc.jaclog(app0(i, \
                    k), alpha_q(trans & (pow2i(m)-1)\
                    , k-1) + gamma_q(trans, k) + \
                    beta_q(trans >> 1, k));
            if(output_table[trans][1](i) == bin(1)) \
                //transition such that the ith \
                output digit is 1
                app1(i, k) = llrcalc.jaclog(app1(i, \
                    k), alpha_q(trans & (pow2i(m)-1)\
                    , k-1) + gamma_q(trans, k) + \
                    beta_q(trans >> 1, k));
        }
        app0(i, k) = app0(i, k) - app1(i, k);
    }
}
```

```
        for (k = 1; k <= block_length; k++){
            nom = -QLLR_MAX;
            den = -QLLR_MAX;
            for(s_prim = 0; s_prim < no_states; s_prim++){ \
                //all possible transitions
                s0 = s_prim >> 1;
                s1 = (s_prim | (1 << m)) >> 1;
                nom = llrcalc.jaclog(nom, alpha_q(s_prim, k-\
                    1) + gamma_q(s_prim, k) + beta_q(s0, k))\
                    ;
                den = llrcalc.jaclog(den, alpha_q(s_prim, k-\
                    1) + gamma_q(s_prim | (1 << m), k) + \
                    beta_q(s1, k));
            }
            app_input(k) = nom - den;
        }

        for (k = 1; k <= block_length; k++){
            if(app_input(k) > 0) output(k-1) = itpp::bin(0);
            else output(k-1) = itpp::bin(1);
        }
        for (int i = 0; i < pow2i(K); i++){
            delete [] output_table[i];
        }
        delete [] output_table;
    }
}
```

main.cpp

```
/*
 *  convcode_mod.h
 *  Definitive
 *
 *  Created by Dani on 18/11/10.
 *
 */

#include <itpp/itcomm.h>
#include "convcode_mod.h"
#include <iostream>
#include <fstream>

using namespace itpp;
using namespace std;
using std::cout;
using std::endl;

//Variables declaration
int constraint_length, m, Nobits, n, nb_errors, total_bits,i\
    ;
double EbN0, N0, Ec,Eb, error_rate, coderate;
ivec generators;
vec trans_symbols, rec_symbols, app, EbN0dB;
bvec uncoded_bits, coded_bits, decoded_bits, temp, rnd;
mat trans_prob;
Convolutional_Code_Mod conv_code;
BPSK bpsk;
BERC berc;
AWGN_Channel channel;
QLLRmat trans_prob_q;
QLLRvec rec_symbols_q, decoded_bits_q;
LLR_calc_unit llrcalc;

vec Pro_Version("0␣0"), type("1␣0"), subtype("0␣0␣1␣1"), \
    Fr_To_DS("0␣0"), More_frag = ("2"), Retry = ("0");
vec Pwr_mgt = ("0"), More_data = ("1"), WEP = ("1"), Order =\
     ("1"), Fr_ctrl_a, Fr_ctrl_b, Fr_ctrl;
vec Duration, T1_adr, AP_adr, R1_adr, Seq_control, Not_used,\
     H_MAC;
vec Payload, MAC_fr_without_CRC, CRC, MAC_fr_total;
int lenght_PSDU, len_fld, N_dbps, N_symb, N_data, N_pad, \
    restLens, TL;
vec tail, service, maskBits, maskBits1, maskBits2;
vec data_conv, data_conv1, data_conv2, P0, P1;
vec maskBits_test, bits_test;

double code_Rate;
```

```
Sequence_Interleaver<double> seq_interleaver(528);
CRC_Code crc_code("CRC-32");

void bit_prob(){

    double a,b;

    P0.set_size(rec_symbols.size(), false);
    P1.set_size(rec_symbols.size(), false);

    for(i = 0; i < rec_symbols.size(); i++){
        P0(i) = itpp::trunc_exp(- double(pow((rec_symbols(i)\
             - 1), 2.0) / N0));
        P1(i) = itpp::trunc_exp(- double(pow((rec_symbols(i)\
             + 1), 2.0) / N0));
        a = P0(i) / (P0(i)+P1(i));
        b = P1(i) / (P0(i)+P1(i));
        P0(i) = a;
        P1(i) = b;
    }
}

void channel_effect(mat &probs, int n, BPSK bpsk, double Ec,\
     double N0){
    int x, y, i;
    vec x_coded, y_coded, denom;
    double temp;
    double Lc = 4.0 * std::sqrt(Ec) / N0;

    denom.set_size(pow2i(n), false);
    denom.clear();

    for(y = 0; y < pow2i(n); y++){  //should be pow2i(n)\
        ////////////////////////////////////////
        bpsk.modulate_bits(dec2bin(n, y), y_coded);
        for(x = 0; x < pow2i(n); x++){
            temp = 0;
            bpsk.modulate_bits(dec2bin(n, x), x_coded);
            for(i = 0; i < n; i++){
                temp += std::pow(double(y_coded(i) - \
                    x_coded(i)), 2.0);
            }
            probs(x, y) = itpp::trunc_exp(-0.25 * Lc * temp)\
                ;
            denom(y) += probs(x, y);
        }
        probs.set_col(y, probs.get_col(y) / denom(y));
    }
}
```

```
void channel_effect_q(QLLRmat &probs, int K, int n, BPSK \
    bpsk, double Ec, double N0){ //supose Lc = 1
    int x, y, i;
    vec x_coded, y_coded;
    double temp;
    QLLRvec denom;
    double Lc = 4.0 * std::sqrt(Ec) / N0;
    LLR_calc_unit llrcalc;
    mat probas;

    probas.set_size(pow2i(K), pow2i(K), false);
    probas.zeros();
    denom.set_size(pow2i(K), false);
    denom.clear();

    for(y = 0; y < pow2i(K); y++){
        bpsk.modulate_bits(dec2bin(n, y), y_coded);
        for(x = 0; x < pow2i(K); x++){
            temp = 0;
            bpsk.modulate_bits(dec2bin(n, x), x_coded);
            for(i = 0; i < n; i++){
                temp += std::pow((y_coded(i) - x_coded(i)), \
                    2.0);
            }
            probas(x, y) = -0.25 * Lc * temp;
            probs(x, y) = llrcalc.to_qllr(probas(x, y));
            denom(y) = llrcalc.jaclog(probs(x, y), denom(y))\
                ;
        }
        //normalization of probs
        for(x = 0; x < pow2i(K); x++){ probs(x, y) -= \
            denom(y); }
    }
}

void encapsulation(){ //Preparation of the different fields \
    of the convolutional coded parts in MAC layer

    //MAC Layer
    //Frame Control
    Fr_ctrl_a = concat(Pro_Version, type, subtype, Fr_To_DS,\
         More_frag);
    Fr_ctrl_b = concat(Retry, Pwr_mgt, More_data, WEP, Order\
        );
    Fr_ctrl = concat(Fr_ctrl_a, Fr_ctrl_b);

    //H-MAC
    Duration = 2*ones(16);
    T1_adr = concat(ones(16), zeros(8), ones(16), zeros(8));
    AP_adr = concat(ones(15), zeros(9), ones(16), zeros(8));
```

```
    R1_adr = concat(ones(15), zeros(9), ones(14), zeros(10))\
        ;
    Seq_control = "1␣1␣0␣1␣0␣1␣0␣1␣0␣1␣1␣1␣1␣0␣1␣1";
    Not_used = zeros(48);
    Fr_ctrl_a = concat(Fr_ctrl, Duration, T1_adr, AP_adr, \
        R1_adr);
    H_MAC = concat(Fr_ctrl_a, Seq_control, Not_used);

    //Unknown Payload at MAC layer
    Payload = 2*ones(240); //Random data (30 bytes)

    //Concatenation for the MAC frame without CRC
    MAC_fr_without_CRC = concat(H_MAC, Payload);

    //Unknown CRC-32 at MAC layer
    CRC = 2*ones(32);

    //Concatenation for the MAC frame with CRC
    MAC_fr_total = concat(MAC_fr_without_CRC, CRC);
    lenght_PSDU = MAC_fr_total.size()/8;

    //PHY Layer
    tail = zeros(6);
    service = zeros(16);

    //Pading bits
    //we assume we know the length field = length_PSDU \
        (bytes)
    //and we have BPSK modulation andcode rate = 1/2
    code_Rate = 0.5;
    N_dbps = 48*code_Rate;
    len_fld = lenght_PSDU;
    N_symb = ceil(double(16 + 8*len_fld + 6)/double(N_dbps))\
        ;
    N_data = N_symb * N_dbps;
    N_pad = N_data - (16 + MAC_fr_total.size() + 6);

    //Concatenation for the convolutional coded parts
    maskBits1 = concat(service, MAC_fr_total);
    seq_interleaver.randomize_interleaver_sequence();
    maskBits2 = seq_interleaver.interleave(maskBits1);
    maskBits = concat(maskBits2, tail, zeros(N_pad));
    restLens = tail.size() + N_pad;
}

void set_trellis(){

    coderate = 1/2;
    n = 2;
    generators.set_size(n, false);
```

```
    generators(0) = 0171;
    generators(1) = 0133;
    constraint_length = 7;

    //Test
    //generators(0) = 2;
    //generators(1) = 5;
    //constraint_length = 3;

    m = constraint_length-1;

    trans_prob.set_size(pow2i(n), pow2i(n), false);
    trans_prob.zeros();

    //trans_prob_q.set_size(pow2i(constraint_length), \
       pow2i(constraint_length), false);
    //trans_prob_q.zeros();
    conv_code.set_generator_polynomials(generators, \
       constraint_length);
}

void test(){ //code-decode test short bitstream

        int EbN0dB_test;

        EbN0dB_test = 6;
        Ec = 1;
        Eb = Ec / conv_code.get_rate();
        EbN0 = inv_dB(EbN0dB_test);
        N0 = Eb * pow(EbN0, -1);
        conv_code.set_awgn_channel_parameters(Ec, N0);
        Nobits = 1000;
        total_bits += Nobits;
        //trans_prob.set_size(pow2i(n), pow2i(n), false);
        //channel_effect(trans_prob, n, bpsk, Ec, N0);

        //Random fields are filled
        RNG_randomize();
        H_MAC(10) = randb();
        rnd = randb(16);
        for(i = 0; i < 16; i++) H_MAC(16+i) = rnd(i);
        Payload = to_vec(randb(Payload.size()));

        //Paquet is build up and interleaved
        data_conv1 = concat(service, \
            to_vec(crc_code.encode(to_bvec(concat(H_MAC, \
            Payload)))));
        data_conv2 = seq_interleaver.interleave(data_conv1);
        data_conv = concat(data_conv2, tail, zeros(N_pad)); \
            //Data interleaved
```

```
        //Convolutional encoder
        //coded_bits = \
            conv_code.encode_trunc(to_bvec(data_conv)); //\
            Encode data
        //TEST/////////
        bits_test =     "1␣0␣1␣0␣0";
        maskBits_test = "2␣2␣2␣2␣2";
        coded_bits = \
            conv_code.encode_trunc(to_bvec(bits_test)); //\
            Encode data
        TL = bits_test.size();
        //TL = data_conv.size();


        //Channel
        channel.set_noise(N0 / 2.0);

        //Error counter
        berc.clear();

        bpsk.modulate_bits(coded_bits, trans_symbols);
        rec_symbols = channel(trans_symbols);
        bit_prob(); //Calculate P0 and P1 of rec_symbols

        //cout << "bits: " << bits_test << endl;
//      cout << "maskbits : " << maskBits_test << endl;
//      cout << "coded bits: " << coded_bits << endl;
//      cout << "rec_symb: " << rec_symbols << endl;

        //conv_code.protocol_assisted_channel_decoding(P0, \
            P1, TL, maskBits_test, decoded_bits);
        channel_effect(trans_prob, n, bpsk, Ec, N0);
        conv_code.map_decode(rec_symbols, decoded_bits, \
            trans_prob, true);

        cout << "decoded␣bits:␣" << decoded_bits << endl;
        //cout << "data: " << data_conv.size() << data_conv \
            << endl;
        //cout << "coded bits: " << coded_bits.size() << \
            endl << coded_bits;
}


    int main () {

        int p, test_points = 7, NumErrH_robustDecoder, \
            NumErrFr_robustDecoder;
        int Total_Fr, NumErrH_Informed_mapDecoder, \
            NumErrFr_Informed_mapDecoder, \
```

```
    NumErrH_nonInformed_mapDecoder, \
    NumErrFr_nonInformed_mapDecoder;
vec robustHER, robustFER, Informed_mapHER, \
    Informed_mapFER, nonInformed_mapHER, \
    nonInformed_mapFER;

ofstream results_robust, results_Informed, \
    results_nonInformed;

robustHER.set_size(test_points, false);
robustFER.set_size(test_points, false);
Informed_mapHER.set_size(test_points, false);
Informed_mapFER.set_size(test_points, false);
nonInformed_mapHER.set_size(test_points, false);
nonInformed_mapFER.set_size(test_points, false);

encapsulation();
set_trellis();

results_robust.open("Robust␣Decode␣Results.txt");
results_Informed.open("Informed␣Map␣Decode␣\
    Results.txt");
results_nonInformed.open("Non␣Informed␣Map␣Decode␣\
    Results.txt");

Ec = 1;
Eb = Ec / conv_code.get_rate();
//EbN0dB = 0;
//EbN0dB = linspace(10, 13, test_points);
EbN0dB = linspace(-0.5, 2.5, test_points);

results_robust << endl << "EbN0dB\t\tHER\t\tFER" << \
    endl;
results_nonInformed << endl << "EbN0dB\t\tHER\t\tFER\
    " << endl;
results_Informed << endl << "EbN0dB\t\tHER\t\tFER" \
    << endl;

cout << "EbN0\tBER" << endl;

//while(EbN0dB <= 3){
for (p = 0; p < EbN0dB.length(); p++) {

    //Processed frame counter
    Total_Fr = 0;

    //MAC-Header error counter
    NumErrH_Informed_mapDecoder = 0;
    NumErrH_nonInformed_mapDecoder = 0;
    NumErrH_robustDecoder = 0;
```

```
//Paquet error counter
NumErrFr_Informed_mapDecoder = 0;
NumErrFr_nonInformed_mapDecoder = 0;
NumErrFr_robustDecoder = 0;

nb_errors = 0;
total_bits = 0;

do{
    Total_Fr++;
    total_bits += data_conv.size();

    //Channel computation
    EbN0 = inv_dB(EbN0dB(p));
    N0 = Eb * pow(EbN0, -1);
    conv_code.set_awgn_channel_parameters(Ec, N0\
        );

    //Random fields are filled
    RNG_randomize();
    H_MAC(10) = randb();
    rnd = randb(16);
    for(i = 0; i < 16; i++) H_MAC(16+i) = rnd(i)\
        ;
    Payload = to_vec(randb(Payload.size()));

    //Packet is build up and interleaved
    data_conv1 = concat(service, \
        to_vec(crc_code.encode(to_bvec(concat(H_MAC\
        , Payload)))));
    data_conv2 = \
        seq_interleaver.interleave(data_conv1);
    data_conv = concat(data_conv2, tail, \
        zeros(N_pad)); //Data interleaved

    //Convolutional encoder
    coded_bits = \
        conv_code.encode_trunc(to_bvec(data_conv\
        )); //Encode data
    TL = data_conv.size();

    //Channel
    channel.set_noise(N0 / 2.0);

    //Error counter
    berc.clear();

    bpsk.modulate_bits(coded_bits, trans_symbols\
```

```
    );
rec_symbols = channel(trans_symbols);
bit_prob(); //Calculate P0 and P1 of \
    rec_symbols

//cout << "bits: " << data_conv << endl;
//cout << "maskbits : " << maskBits << endl;
//cout << "coded bits: " << coded_bits << \
    endl;
//cout << "rec_symb: " << rec_symbols << \
    endl;

//maskBits = 2*ones(maskBits.size());

//Non informed map decode
bvec nonInformed_mapUncoded_1, \
    nonInformed_mapUncoded_2;
vec nonInformed_mapUncoded_3, \
    nonInformed_mapUncoded;
channel_effect(trans_prob, n, bpsk, Ec, N0);\
     //A priori probabilities
conv_code.map_decode(rec_symbols, \
    nonInformed_mapUncoded_1, trans_prob, \
    true);
nonInformed_mapUncoded_2 = \
    nonInformed_mapUncoded_1.left(nonInformed_mapUncoded_1.size(\
    ) - restLens);
nonInformed_mapUncoded_3 = \
    seq_interleaver.deinterleave(to_vec(nonInformed_mapUncoded_2\
    ));
nonInformed_mapUncoded = \
    nonInformed_mapUncoded_3;
//nonInformed_mapUncoded = \
    concat(nonInformed_mapUncoded_3, tail, \
    zeros(N_pad));


//Informed map decode
vec Informed_mapUncoded_2, \
    Informed_mapUncoded_3, \
    Informed_mapUncoded;
Informed_mapUncoded_2 = data_conv2;
for(i = 0; i < TL-restLens; i++){
    if(maskBits2(i) == 2){
        Informed_mapUncoded_2(i) = \
            nonInformed_mapUncoded_2(i);
    }
}
```

```
Informed_mapUncoded_3 = \
    seq_interleaver.deinterleave(Informed_mapUncoded_2\
    );
Informed_mapUncoded = Informed_mapUncoded_3;
//Informed_mapUncoded = \
    concat(Informed_mapUncoded_3, tail, \
    zeros(N_pad));

berc.count(to_bvec(data_conv1), \
    to_bvec(Informed_mapUncoded));
nb_errors += berc.get_errors();


//Robust decode
bvec robustUncoded_1, robustUncoded_2;
vec robustUncoded, robustUncoded_3;
conv_code.protocol_assisted_channel_decoding(P0\
    , P1, TL, maskBits, robustUncoded_1);

robustUncoded_2 = \
    robustUncoded_1.left(robustUncoded_1.size(\
    ) - restLens);
robustUncoded_3 = \
    seq_interleaver.deinterleave(to_vec(robustUncoded_2\
    ));
robustUncoded = robustUncoded_3;
//17-7-2016 removed tail and pads for size \
    difference with data_conv1
//robustUncoded = concat(robustUncoded_3, \
    tail, zeros(N_pad));

//Compare the H-MAC
vec Data_H_MAC;
//reference
Data_H_MAC = concat(data_conv1(26), \
    data_conv1.get(32, 47));
if(any(to_bvec(concat(robustUncoded(26), \
    robustUncoded.get(32, 47))) - \
    to_bvec(Data_H_MAC))) \
    NumErrH_robustDecoder++;
if(any(to_bvec(concat(Informed_mapUncoded(26\
    ), Informed_mapUncoded.get(32, 47))) - \
    to_bvec(Data_H_MAC))) \
    NumErrH_Informed_mapDecoder++;
if(any(to_bvec(concat(nonInformed_mapUncoded(26\
    ), nonInformed_mapUncoded.get(32, 47))) \
    - to_bvec(Data_H_MAC))) \
    NumErrH_nonInformed_mapDecoder++;
```

```
    if(any(to_bvec(robustUncoded) - \
        to_bvec(data_conv1))) \
        NumErrFr_robustDecoder++;
    if(any(to_bvec(Informed_mapUncoded) - \
        to_bvec(data_conv1)))  \
        NumErrFr_Informed_mapDecoder++;
    if(any(to_bvec(nonInformed_mapUncoded) - \
        to_bvec(data_conv1)))  \
        NumErrFr_nonInformed_mapDecoder++;

    //cout << "decoded bits: " << decoded_bits \
        << endl;
    //cout << "data: " << data_conv.size() << \
        data_conv << endl;
    //cout << "coded bits: " << coded_bits.size(\
        ) << endl << coded_bits;

    //berc.count(to_bvec(data_conv), \
        decoded_bits);
    //nb_errors += berc.get_errors();
    //cout << "errors: " << nb_errors << endl;
    //cout << "HER" << NumErrH << endl;
    //cout << "FER" << NumErrFr << endl;
}while(NumErrFr_robustDecoder < 1000);

error_rate = double(nb_errors) / \
    double(total_bits);
cout << EbN0dB(p) << "      " << error_rate << \
    endl;


robustHER(p) = double(NumErrH_robustDecoder) / \
    double(Total_Fr);
robustFER(p) = double(NumErrFr_robustDecoder) / \
    double(Total_Fr);
Informed_mapHER(p) = \
    double(NumErrH_Informed_mapDecoder) / \
    double(Total_Fr);
Informed_mapFER(p) = \
    double(NumErrFr_Informed_mapDecoder) / \
    double(Total_Fr);
nonInformed_mapHER(p) = \
    double(NumErrH_nonInformed_mapDecoder) / \
    double(Total_Fr);
nonInformed_mapFER(p) = \
    double(NumErrFr_nonInformed_mapDecoder) / \
    double(Total_Fr);
```

```
        results_robust << EbN0dB(p) << "\t\t" << \
            robustHER(p) << "\t\t" << robustFER(p) << \
            endl;
        results_Informed << EbN0dB(p) << "\t\t" << \
            Informed_mapHER(p) << "\t\t" << \
            Informed_mapFER(p) << endl;
        results_nonInformed << EbN0dB(p) << "\t\t" << \
            nonInformed_mapHER(p) << "\t\t" << \
            nonInformed_mapFER(p) << endl;
        //error_rate = double(nb_errors) / \
            double(total_bits);

    }

    /*//HER and FER Results
    cout << endl << "EbN0dB\t\tHER\t\tFER" << endl;
    for (p = 0; p < EbN0dB.length(); p++) {
        cout << EbN0dB(p) << "\t\t" << robustHER(p) << "\
            \t\t" << robustFER(p) << endl;
    }*/

    results_robust.close();
    results_nonInformed.close();
    results_Informed.close();
    return 0;

}
```

A.2. Resultados simulación

Non Informed Map Decode Results

```
EbN0dB HER FER
-0.5 1 1
0 1 1
0.5 0.993098 1
1 0.982721 1
1.5 0.937738 1
2 0.844025 1
2.5 0.708458 0.9976
```

Informed Map Decode Results

```
EbN0dB HER FER
-0.5 1 1
0 1 1
0.5 0.993098 1
1 0.982721 1
1.5 0.937738 1
2 0.844025 1
2.5 0.708458 0.997001
```

Robust Map Decode Results

```
EbN0dB HER FER
-0.5 0.33209 0.932836
0 0.233476 0.858369
0.5 0.147239 0.766871
1 0.0950324 0.719942
1.5 0.0781449 0.635324
2 0.0647799 0.628931
2.5 0.0413917 0.59988
```

BER Robust Results

```
EbN0  BER
-0.5    0.203083
0   0.182506
0.5   0.153073
1   0.121665
1.5   0.086251
2   0.0561674
2.5   0.0329249
```

Referencias

[1] J. F. Kurose and K. W. Ross, Computer Networking: A Top-Down Approach Featuring the Internet, 3rd ed. Boston: Addison Wesley, 2005.

[2] C. Bormann, C. Burmeister, M. Degermark, H. Fukushima, H. Hannu, L.-E. Jonsson, R. Hakenberg, T. Koren, K. Le, Z. Liu, A. Martensson, A. Miyazaki, K. Svanbro, T. Wiebke, T. Yoshimura, and H. Zheng, "Robust header compression (ROHC): Framework and four profiles," Tech. Rep. RFC 3095, 2001.

[3] D. G. Sachs, S. Adve, and D. L. Jones, "Cross-layer adaptive video coding to reduce energy on general-purpose processors," in Proc. Int. Conf. Image Process., Barcelona, Spain, 2003, pp. III–109–III–112.

[4] M. Van der Schaar and S. Shankar, "Cross-layer wireless multimedia transmission: Challenges, principles, and new paradigms," IEEE Wire- less Communications Magazine, vol. 12, no. 4, pp. 50–58, 2005.

[5] H.Jenkac,T.Stockhammer,andW.Xu,"Cross-Layerassistedreliability design for wireless multimedia broadcast," EURASIP Signal Processing Journal, Special Issue on Advances in Signal Processing-assisted Cross- layer Designs, vol. 86, no. 8, pp. 1933–1949, 2006.

[6] F. Fu and M. van der Schaar, "A new systematic framework for autonomous cross-layer optimization," IEEE trans. on Vehicular Tech- nology, vol. 58, no. 4, pp. 1887–1903, May 2009.

[7] P. Duhamel and M. Kieffer, Joint source-channel decoding: A cross-layer perspective with applications in video broadcasting. Academic Press, 2009, to appear.

[8] C. Marin, Y. Leprovost, M. Kieffer, and P. Duhamel, "Robust mac-lite and soft header recovery for packetized multimedia transmission," IEEE Trans. on Communications, 2010, to appear.

[9] J.L.Massey,"Optimum frame synchronization,"IEEETrans.onComm., vol. 20, no. 4, pp. 115–119, 1972.

[10] P. T. Nielsen, "Some optimum and suboptimum frame synchronizers for binary data in Gaussian noise," IEEE Transactions on Communications, vol. 21, no. 6, pp. 770–772, 1973.

[11] M. Chiani and M. G. Martini, "On sequential frame synchronization in AWGN channels," IEEE Trans. Comm., vol. 54, pp. 339 – 348, 2006.

[12] M. G. Martini and M. Chiani, “Optimum metric for frame synchroniza- tion with Gaussian noise and unequally distributed data symbols,” in Proc. IEEE SPAWC, Perugia, Italy, 21-24 June 2009.

[13] U. Ali, M. Kieffer, and P. Duhamel, “Joint protocol-channel decoding for robust aggregated packet recovery at WiMAX MAC layer.” in Proc. IEEE SPAWC, Perugia, Italy, 21-24 June 2009, pp. 672 – 676.

[14] M. C. Valenti and B. D. Woerner, “Iterative channel estimation and decoding of pilot symbol assisted turbo codes over flat-fading channels,” IEEE Journal on Selected Areas in Communications, vol. 19, no. 9, pp. 1697–1705, 2001.

[15] ANSI/IEEE, “802.11, part 11 : Wireless LAN medium access control (MAC) and physical layer (PHY) specifications,” Tech. Rep., 1999.

[16] Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, IEEE Computer Society Std., June 2007.

[17] “Digital video broadcasting (DVB); framing structure, channel coding and modulation for satellite services to handheld devices (SH) below 3 GHz,” ETSI, Tech. Rep. EN 302 583, 2008.

[18] G. Panza, E. Balatti, G. Vavassori, C. Lamy-Bergot, and F. Sidoti, “Supporting network transparency in 4G networks.” in Proc. IST Mobile and Wireless Communication Summit, 2005.

[19] J. K. Wolf, “Efficient maximum-likelihood decoding of linear block codes using a trellis,” IEEE Trans. Inform. Theory, vol. 24, no. 1, pp. 76–80, 1978.

[20] L. R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, “Optimal decoding of linear codes for minimizing symbol error rate,” IEEE Trans. Info. Theory, vol. 20, pp. 284–287, 1974.

[21] G. D. Forney Jr., Review of random tree codes, 1967.

[22] F. Jelinek and J. Cocke, "Bootstrap hybrid decoding for symmetrical binary input channels", Inform. Contr., vol. 18, pp. 261-298, 1971.

[23] http://itpp.sourceforge.net/