Casa abierta al tiempo

UNIVERSIDAD AUTÓNOMA METROPOLITANA

Maestría en Ciencias y Tecnologías de la Información

# Programación y evaluación en un DSP de un esquema de corrección de errores unidireccional (Forward Error Correction, FEC)

Idónea Comunicación de Resultados
para obtener el grado de
MAESTRO EN CIENCIAS
EN TECNOLOGÍAS DE LA INFORMACIÓN
Presentada por

**Iván Fabián Luna**

Asesores:

Dr. Gerardo Abel Laguna Sánchez
Dr. Ricardo Marcelín Jiménez

Dr. Ricardo Barrón Fernández

Dr. Víctor Manuel Ramos Ramos

Dr. Gerardo Abel Laguna Sánchez

Defendida públicamente en la UAM Iztapalapa

Presidente: Dr. Ricardo Barrón Fernández, CIC-IPN
Vocal: Dr. Víctor Manuel Ramos Ramos, UAM – Iztapalapa
Secretario: Dr. Gerardo Abel Laguna Sánchez, UAM – Iztapalapa

México D. F. a 22 de Febrero 2013

# Agradecimientos

Los primeros agradecimientos son para la Universidad Autónoma Metropolitana, unidad Iztapalapa, por haberme aceptado en la licenciatura en Ingeniería Electrónica y posteriormente en el Posgrado en Ciencias y Tecnologías de la Información (PCyTI), la cual considero como mi segunda casa.

Al Consejo Nacional de Ciencia y Tecnología (CONACYT) por el apoyo económico otorgado durante dos años, para la realización de este trabajo.

A mi madre *Emelia Luna Juárez* por soportarme durante tantos años, por brindarme su apoyo incondicional y comprensión, en los momentos buenos y malos.

A mis asesores Gerardo Abel Laguna Sánchez y Ricardo Marcelín Jiménez.

A mis compañeros que conocí en esta maestría y de los que aprendí demasiadas cosas.

*"Nacemos para vivir, por eso el capital más importante que tenemos es el tiempo, es tan corto nuestro paso por este planeta que es una pésima idea no gozar cada paso y cada instante, con el favor de una mente que no tiene límites y un corazón que puede amar mucho más de lo que suponemos"*

*Facundo Cabral*

# Resumen

En las últimas décadas, la necesidad de mandar información de un lugar a otro, ha sido un tema de gran interés para los investigadores en el área de las comunicaciones. Lo que se busca es desarrollar técnicas de codificación y decodificación capaces de contrarrestar los efectos introducidos por los medios de comunicación que provocan la degradación de las señales. En la actualidad, para detección y corrección de errores de la información, se ocupan técnicas como: *Códigos convolucionales, Turbo códigos, Códigos de paridad de baja densidad (LDPC: Low Density Parity Check), Reed-Solomon y Códigos de redundancia cíclica (CRC: Cyclic Redundancy Check).*

En los sistemas de comunicación digital, para que la información pueda ser enviada, se usa una representación binaría (1 y 0). Para transmitir la información se usan señales continuas y discretas. Si son continuas, son muestreadas, cuantificadas, codificadas y moduladas. Para señales discretas, sólo es necesario codificarlas y modularlas. Al recibir la información, se puede calcular el desempeño de la técnica de codificación empleada. Esto se hace al medir la *tasa de error de bit (BER: Bit Error Rate)* para diferentes relaciones de *energía de bit por densidad espectral de ruido (Eb/No: Energy per bit to Noise Power Spectral Density Ratio)*. En la actualidad se especifica un valor $1 \times 10^{-5}$ para la BER, que indica un desempeño aceptable.

La detección y corrección de errores se hace en la parte receptora. Una alternativa, cuando se presentan errores en la información, es pedirle al transmisor que vuelva a retransmitirla. Este comportamiento puede llegar a ser repetitivo, por ejemplo si se presentan muchos errores, el receptor no dejaría de solicitar que la información sea reenvíada y, como consecuencia de ello, aumenta el procesamiento en ambas partes, además del tiempo. En sistemas donde se requieren aplicaciones en tiempo real esto no es factible y lo que se busca es evitar el reenvío de la información, es decir, que los errores que se presenten sean corregidos en el receptor, a este proceso se le conoce como el método *corrección de errores FEC (Forward Error Correction).*

Entre los métodos FEC de alto desempeño tenemos a los turbo códigos y códigos de paridad de baja densidad LDPC. Los turbo códigos basan su funcionamiento en los *códigos convolucionales recursivos (RSC: Recursive Systematic Convolutional)*, los que son una variante de los códigos convolucionales y proporcionan comunicaciones fiables muy cerca del límite de *Shannon*. Estas técnicas, por arriba de un nivel de *Eb/No = - 1.5917 dB*, pueden

llegar a tener un BER=0.

Las técnicas anteriormente descritas son utilizadas en dispositivos, FPGA (Field Programmable Gate Array), *Digital Signal Processor (DSP)* y en otros de procesamiento de datos, con aplicaciones en teléfonos celulares, módems y en aplicaciones de videojuegos, que son muy comúmes en nuestra vida diaria.

Los turbo códigos, por ser una técnica muy eficiente en la corrección de errores, se eligieron para hacer una realización práctica en el DSP TMS320C6416 de Texas instruments. Esto con la finalidad de observar su desempeño y compararlo con el desempeño simulado en una PC. Para la realización de lo anterior se analizó la arquitectura del DSP y el rango de las variables de programación. El problema principal en la realización fue el de migración de los códigos de la PC al DSP y garantizar su buen funcionamiento debido a las limitaciones de representación númerica del DSP.

Como resultado de la realización práctica se obtuvieron desempeños similares entre la PC y el DSP. Para lograr a los turbo códigos fue necesario programar códigos convolucionales, códigos RSC y códigos RSC con algoritmo *MAP (Maximum A Posteriori)*. El algoritmo MAP es una parte esencial de la turbo decodificación. En la decodificación se emplean tantos decodificadores como los codificadores RSC empleados. La información recibida pasa por el primer decodificador MAP y los parámetros obtenidos son enviados al siguiente decodificador MAP y, así, sucesivamente hasta llegar al último decodificador MAP, que retroalimenta información adicional al primero. El proceso se repite varias veces y en cada repeticion se obtiene una mejor estimación del mensaje que fue enviado. En el proceso descrito, el algoritmo MAP utiliza el concepto de probabilidades conjuntas condicionales, esto es por que el proceso de codificación se modela como un proceso de *Markov*. Este proceso puede ser representado gráficamente por un diagrama de *Trellis* y un diagrama de estados. Para poder determinar lo que se envió, es necesario calcular parámetros como: las métricas $\gamma$ de las ramas en el diagrama de Trellis, los coeficientes de recursión hacia adelante $\alpha$ y hacia atrás $\beta$. Para determinar las métricas $\gamma$ en el diagrama de Trellis es necesario multiplicar $\alpha$ y $\beta$ anteriores, lo que provoca que crezcan y rebasen el rango de representación numérica del DSP. Si esto sucede en el DSP se obtienen resultados incorrectos. Para evitar esto se utilizarón métodos como: normalización y mapeo de los valores utilizando la función trigonométrica *arctan()*.

# Índice general

# Índice de figuras

# Índice de tablas

# Capítulo 1
# Introducción

Para la realización de este trabajo se eligieron a los turbo códigos, esto por que es una técnica muy eficiente en la corrección de errores. Desde su presentación en 1993, han sido ampliamente utilizados en sistemas donde no está permitida la retransmisión.

La realización práctica de los turbo códigos requiere de dispositivos capaces de efectuar un gran número de operaciones aritméticas a muy alta velocidad. El DSP es un dispositivo de propósito específico que cumple con estas características, su funcionamiento se fundamenta en procesadores o microprocesadores que poseen un juego de instrucciones, hardware y software, optimizados para realizar este tipo de aplicaciones. Para aumentar la velocidad de las aplicaciones, dispone de unidades computacionales específicas que trabajan en paralelo, como las unidades multiplicadoras y acumuladoras MAC, que permiten que el CPU realice múltiples operaciones en una sola instrucción [3].

## 1.1. Motivación

En este trabajo se muestra la realización práctica y programación de la técnica de codificación de canal conocida como turbo códigos. Se trata de dar una idea más clara de la programación en el DSP TMS320C6416 de la familia de Texas Instruments. Se eligió a éste por su rapidez, la que se debe en gran parte a su arquitectura de punto fijo. Este resulta ser más rápido que el que cuenta con una arquitectura de punto flotante, es decir, ejecuta un menor número de instrucciones para realizar aritmética si se usa punto fijo y es muy útil en el procesamiento y representación de señales analógicas en tiempo real.

## 1.2. Objetivos

Lo que se propone es la realización práctica de la técnica de corrección de errores turbo códigos, la cual corrige la gran mayoría de los errores que son provocados por los efectos del canal a partir de un nivel de $Eb/No = 0.7\ dB$.

- **Objetivo genral**

  - Programar en un DSP una técnica de turbo códigos y obtener la gráfica de desempeño (BER vs Eb/No).

- **Objetivos particulares**

  - Programar en un *DSP* códigos convolucionales requeridos.
  - Programar en un *DSP* códigos RSC requeridos.
  - Programar en un *DSP* códigos RSC con el algoritmo MAP que se va a utilizar.
  - Programar en un *DSP* una variante de turbo códigos.

## 1.3. Metodología

La metodología propuesta en este trabajo consiste en programar las distintas técnicas de codificación y decodificación en la PC y migrar estos códigos al DSP, esto con la finalidad de obtener las gráficas de BER vs Eb/No, para comparar desempeños. Para ello se plantea programar las siguientes técnicas.

- Códigos convolucionales de tasa $R = 1/2$
- Códigos convolucionales de tasa $R = 1/3$
- Código RSC de tasa $R = 1/2$
- Código RSC con algoritmo MAP de tasa $R = 1/2$
- Turbo código Paralell Concatened Convolutional Code (PCCC) $R = 1/3$

## 1.4. Contribución

Turbo códigos en un *DSP* de arquitectura de punto fijo y proponer alternativas para resolver los problemas que se presenten.

Este trabajo tiene como aporte la publicación del artículo *Realización práctica de los Turbo códigos de alto desempeño con un TMS320C6416* en la *XIV Reunión de Otoño de Potencia Electrónica y Computación, ROPEC 2012* celebrada en la Universidad de Colima, los días del 7-9 de Noviembre del 2012 [4].

## 1.5. Estructura del documento

El resto del trabajo se estructura de la siguiente manera, en el Capítulo 2 se aborda una breve reseña de la historia de la teoría de la información la cual sienta las bases de la capacidad de canal que un sistema de comunicación puede tener. En nuestro caso es muy importante, ya que los turbo códigos se acercan al límite de esta capacidad. En este mismo capítulo se resume la teoría que describe las diferentes técnicas de codificación de canal, empezando con los códigos convolucionales, códigos convolucionales recursivos RSC, RSC con algoritmo MAP y por último los turbo códigos.

En el Capítulo 3, se explica cómo se llevó a cabo la realización práctica de las técnicas de codificación de canal propuestas, de los criterios que se tomaron en cuenta para no rebasar el rango de representación numérica del *DSP* y de los métodos que se propusieron para solucionar esto, particularmente para los turbo códigos.

En el Capítulo 4, se muestran los resultados del desempeño obtenido con el *DSP* para las técnicas de codificación de canal y su comparación con el obtenido por la simulación en la *PC*.

Finalmente se presentan las conclusiones, el trabajo a futuro y los apéndices donde se encuentra el código en lenguaje C de los módulos más importantes.

# Capítulo 2
# Marco teórico

## 2.1. Antecedentes históricos

*Claude Shannon* en 1948, estableció las bases de la teoría de la información y postuló el teorema llamado *límite de Shannon* [5], en donde afirmó que un canal de comunicación puede ser caracterizado por su ancho de banda y su relación señal a ruido, demostró la velocidad máxima a la cual la información puede ser transmitida.

Los datos, para poder ser transmitidos a través del canal de comunicación, se blindan para minimizar los efectos adversos que el canal introduce. Desde la década de los 40 los investigadores han desarrollado técnicas de codificación y decodificación, capaces de contrarrestar estos efectos y determinar la velocidad máxima de transmisión. Las técnicas más importantes y que más se usan en la codificación y decodificación de canal son los: códigos convolucionales, LDPC y turbo códigos.

Los códigos convolucionales fueron introducidos por *Peter Elias* [6], son ampliamente usados en la detección y corrección de errores. Se han empleado en aplicaciones tales como: comunicaciones inalámbricas, comunicaciones digitales vía satélite y sistemas de radiodifusión. En la parte de la decodificación usan el algoritmo de *Viterbi*, que consiste en encontrar la ruta más probable en el diagrama de Trellis (árbol binario), esto se hace al comparar la información recibida con la correspondiente a todas las ramas del árbol y se elige aquella la que se considera que fue transmitida con más probabilidad. Recientes aplicaciones en los turbo códigos, han mostrado que los códigos convolucionales pueden ser combinados en esquemas de concatenado o entrelazado, acercando su desempeño al límite de Shannon.

Los LDPC fueron introducidos por primera vez en 1962 por *R. G. Gallager* [7] y rápidamente olvidados, pero en 1996 fueron retomados por *David JC Mackay* y *Radford M. Neal* en su artículo *Near Shannon Limit Performance of Low Density Parity Check Codes* [8] llegándose a convertir en el esquema de decodificación más usado.

Los turbo códigos fueron introducidos en 1993 por *Claude Berrou*, *Alain Glavieux* y *Punya Thitimajshima*, en su artículo *Near Shannon Limit Error Correcting Coding and Decoding Turbo-Codes* [9], en el que presentan la concatenación de varios códigos convolucionales y desmuestran que con esta nueva estructura es posible obtener un desempeño cercano al límite de Shannon. Al principio no fueron muy bien recibidos por la comunidad científica, precisamente porque presentaban un desempeño cerca del límite de Shannon para canales con *ruido blanco gaussiano aditivo (AWGN: Additive White Gaussian Noise)*. Esto motivo a que muchos investigadores en el área de los sistemas digitales se concentraran en estudiar su funcionamiento a detalle. El objetivo principal al utilizar turbo códigos es realizar transmisiones de datos usando potencias bajas, a la mayor velocidad posible y con valores de BER muy bajos.

En la actualidad los turbo códigos son utilizados en sistemas de comunicación espacial, sistemas de telefonía de tercera generación, redes Wi-Fi y en dispositivos portátiles con aplicaciones de datos de multimedia (video y audio). Países como Japón han convertido a los turbo códigos en un estándar de la telefonía móvil de tercera generacion *3G*. En los laboratorios del *NTT Communications* están siendo usados en transmisiones de datos, video y correo, en septiembre del 2003. La agencia espacial europea *(ESA)*, lanzó el SMART-1, la primera prueba para ir al espacio con transmisión de datos utilizando turbo códigos, de igual manera, la *NASA* los está usando en misiones a *Marte* la *Mission Cassini* y la *Mercury Messenger Mission* donde se requieren comunicaciones confiables [10].

## 2.2. Definición de codificación y decodificación de canal

La codificación y decodificación de canal, son parte fundamental en los sistemas de comunicación. Con ello, se procura que la comunicación sea lo más *confiable* posible, es decir, que esté libre de errores.

La codificación de canal tiene como próposito transformar y añadir redundancia a los datos originales, para procurar que al ser enviados a través del canal de comunicación Figura 2.1, estos puedan ser recuperados a pesar de los efectos adversos del canal.

En la decodificación de canal, los datos recibidos se procesan para estimar o decidir cuál es la secuencia de datos originales enviada. Por la forma en que las técnicas de decodificación procesan la información se pueden clasificar en:

- ***Dura (Hard)*** La información recibida pasa directamente del detector al decodificador en donde, mediante procesos de comparación y distancia *Hamming*, se decide cuál es la información que fue enviada.
- ***Suave (Soft)*** A diferencia de las técnicas Hard, no se cuenta con un detector. La información recibida es directamente procesada por el decodificador de canal. Para determinar cuál fue la información original enviada, se compara la información recibida con un conjunto posible de valores. El resultado de la comparación *(distancias Euclidianas)* se usa para encontrar la probabilidad y con esta estimar los datos enviados con mayor probabilidad.

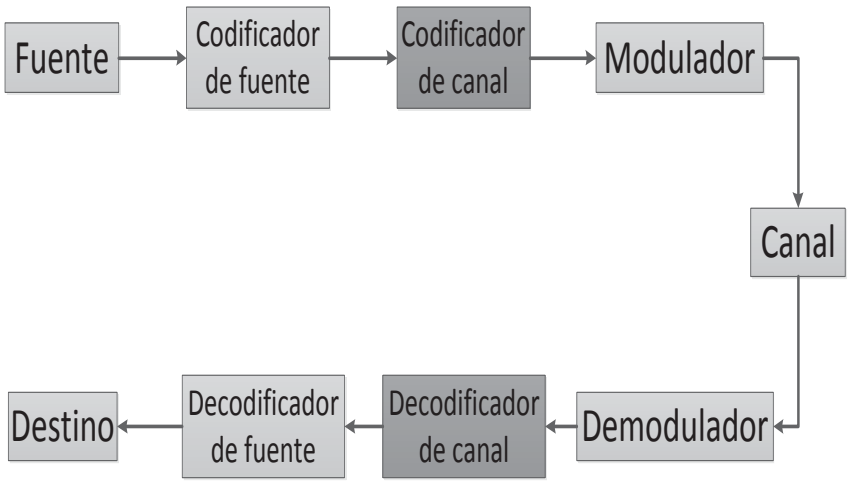



Figura 2.1: Esquema de comunicación

En los esquemas de comunicación se encargan de montar/desmontar los símbolos codificados en señales. Para lograr lo anterior, en el modulador se agrupan bits en *símbolos*, que se montan sobre señales y son enviados a través del canal, donde sufren los efectos del mismo. Al llegar a su destino, se trata de recuperarlos y para ello, se usa un detector que toma una decisión y determina qué símbolo corresponde a la señal que se está recibiendo.

Algunas de las ventajas del proceso de modulación/demodulación son las siguientes: se aprovecha mejor el canal, se puede transmitir más información en forma simultánea, se reduce la interferencia entre símbolos, sólo por mencionar algunas.

## 2.3. Teoría de la información

Antes de abordar los conceptos teóricos de las técnicas de codificación y decodificación, es necesario tener el conocimiento de conceptos como el SNR, de su relación con Eb/No y del origen del límite de Shannon. En las siguientes secciones se da una breve explicación de estos conceptos.

La teoría de la información estudia las leyes que rigen la transmisión y el procesamiento de la información. Tiene como objetivo principal medir la información, determinar su representación y determinar la capacidad de transmisión de un sistema de comunicación, así como de su procesamiento.

La teoría de la información fue desarrollada inicialmente, en 1948 por el ingeniero electrónico y matemático estadounidense Claude E. Shannon, en su artículo *A Mathematical Theory of Communication (Teoría matemática de la comunicación)* [5]. La teoría de la información surgió por la necesidad de un marco teórico para la tecnología de la comunicación. Fue aumentando su complejidad y aplicación a diversos medios de comunicación, tales como el teléfono, las redes teletipo y los sistemas de comunicación por radio. En general, abarca todas las formas de transmisión y almacenamientos de información, incluyendo la televisión, computadoras y la grabación óptica de datos e imágenes.

Antes de continuar con teoría de la información, es importante tener en cuenta los siguientes conceptos, ya que comúnmente suelen confundirse [11].

- ***Información:*** El término información se refiere a los mensajes a tramsmitir, los cuales pueden ser: voz, imágenes, documentos, video, entre otros.
- ***Ancho de banda de la señal:*** Se define como el rango de frecuencias en el que se concentra la mayor potencia de la señal.
- ***Ancho de banda del canal:*** Es el resultado de la diferencia entre *frecuencia máxima* - *frecuencia mínima* que puede pasar por cierto canal. El canal debe ser lo suficientemente

grande para poder dejar pasar todas las frecuencias importantes con señales de datos.

- ***Capacidad de información:*** Es una medida de la cantidad de información que se puede transferir a través de un sistema de comunicaciones en determinado tiempo.

En el año de 1948 R. Hartley, de *Bell Telephone Laboratories*, desarrolló una relación útil entre el ancho de banda, el tiempo de transmisión y la capacidad de información.

$$I = Bt \tag{2.1}$$

donde $I$ es la capacidad de información (bits/s), $B$ es el ancho de banda (Hertz) y $t$ es el tiempo de transmisión (s).

## 2.4. Límite de Shannon

En 1948, Claude Shannon, retomando los estudios de Hartley encontró la capacidad máxima a la cual se puede transmitir información con un nivel dado de errores tolerados.

$$I = B \log_2(1 + \frac{S}{N}) \tag{2.2}$$

donde $\frac{S}{N}$ es la relación de potencia de señal a ruido.

El ruido es uno de los problemas presentes en los canales de comunicación y que es difícil de eliminar. La intención de usar técnicas de codificación y decodificación es procurar que la información sea lo menos afectada por el ruido. El ruido en estos sistemas existe como una energía eléctrica indeseable.

Por su naturaleza, el ruido puede ser clasificado como correlacionado y no correlacionado. El ruido es correlacionado cuando entre la señal y éste existe una relación, éste solo se presenta si la señal existe. El ruido no correlacionado, como su nombre lo indica, no tiene ninguna relación con la señal y existe siempre.

El ruido térmico es no correlacionado y está asociado con el movimiento rápido y aleatorio de los electrones dentro de un conductor, producido por la agitación térmica. Este movimiento fue observado por primera vez por Robert Brown.

El movimiento aleatorio de los electrones fue reconocido en 1927 por primera vez, por J. B. Johnson de los Bell Telephone Laboratories [11]. Johnson encontró una relación para la potencia del ruido térmico:

$$N = KTB \tag{2.3}$$

de donde $N$ es la potencia del ruido (watts), $K$ la Constante de Boltzmann $1{,}38 \times 10^{-23}$ y $T$ la temperatura.

El ruido térmico es aleatorio y continuo, tiene componentes de todas las frecuencias, por lo que se le suele llamar ruido blanco, por ser análogo a la luz blanca que contiene todas las frecuencias de la luz visible.

En la Ecuación 2.2 existe una relación entre la potencia de la señal y la potencia del ruido. Dicho lo anterior, lo siguiente es encontrar el límite teórico de Shannon, además de explicar la relación que existe entre SNR y $\frac{Eb}{No}$.

La razón $\frac{Eb}{No}$ es una medida que se utiliza en los sistemas de comunicación digital, esta se mide en el receptor y sirve para indicar la intensidad de la señal con respecto al ruido. Esta relación, junto con técnicas de codificación y modulación, ayudan a determinar o caracterizar el desempeño de un sistema de comunicación digital, la razón $\frac{Eb}{No}$ puede expresarse como:

$$E_b = CT_b \tag{2.4}$$

Donde

- $E_b$ = Energía de un solo bit (J/s)
- $C$ = Potencia promedio de la portadora (watts)
- $T_b$ = Tiempo de un solo bit (segundos)

Por otro lado,

$$N_o = N/B \tag{2.5}$$

Donde

- $N_o$ = Densidad de potencia del ruido (watts/hertz)
- $N$ = Potencia promedio del rudio térmico (watts)
- $B$ = Ancho de banda en (hertz)

Ahora dividiendo $E_b$ entre $N_o$ se tiene:

$$\frac{E_b}{N_o} = \frac{CB}{Nf_b} \tag{2.6}$$

En la Ecuación 2.6 aparece el término $f_b$ que es la frecuencia de bits por segundos, es decir:

$$f_b = \frac{1}{t_b} \tag{2.7}$$

¿Cómo se relaciona la razón SNR con $\frac{E_b}{N_o}$ de la Ecuación 2.6?. Se observa que el término $\frac{C}{N}$ es la SNR y el segundo término es una simple normalización, la cual depende del ancho de banda $B$ y el tiempo de bit $f_b$. Si el $B = f_b$ se tiene:

$$\frac{E_b}{N_o} = SNR \tag{2.8}$$

Después de haber explicado la relación que existe entre ambos, se procede a encontrar el límite de Shannon teórico.

Partiendo de 2.6 y haciendo $I = f_b$ para simplificar, se tienen las siguientes relaciones:

$$S = E_b I \tag{2.9}$$

$$N = N_o B \tag{2.10}$$

Sustituyendo las Ecuaciones 2.9 y 2.10 en 2.2

$$\frac{I}{B} = \log_2(1 + \frac{E_b I}{N_o B}) \tag{2.11}$$

Aplicando inverso de $\log_2(a)$ a 2.11

$$(2^{\frac{I}{B}} - 1)(\frac{I}{B})^{-1} = \frac{E_b}{N_o} \tag{2.12}$$

Recordando que

$$\frac{\partial a^x}{\partial x} = a^x \ln(a) \tag{2.13}$$

Haciendo un cambio de variable

$$n = \frac{I}{B} \tag{2.14}$$

Se tiene

$$\frac{E_b}{N_o} = \frac{2^n - 1}{n} \tag{2.15}$$

Aplicando la ley de *L'Hopital* [12] a la ecuación 2.15 para el caso que nos interesa cuando $n \to 0$

$$\frac{E_b}{N_o} = \ln(2) \lim_{n \to 0} 2^n \tag{2.16}$$

Lo que da como resultado el límite de Shannon Figura 2.2:

$$\frac{E_b}{N_o} = 0{,}6931 \tag{2.17}$$

$$\frac{E_b}{N_o} dB = -1{,}5917 dB \tag{2.18}$$

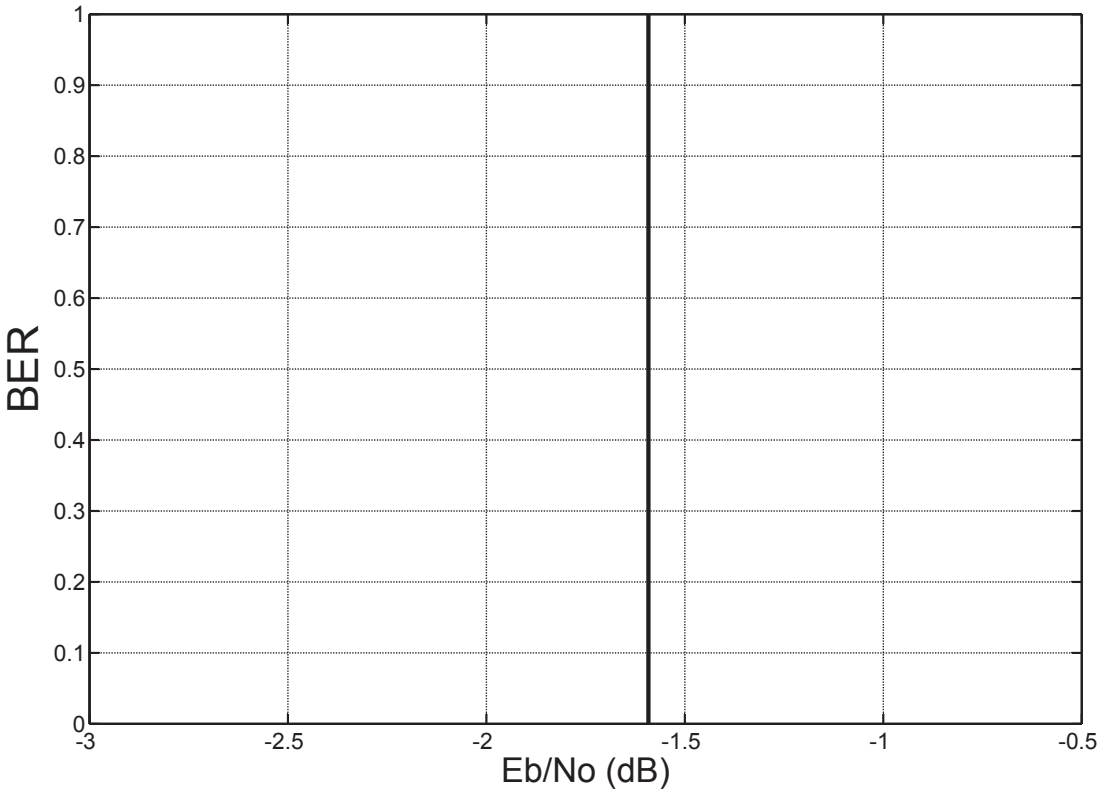



Figura 2.2: Límite teórico de Shannon.

El límite de Shannon indica que por arriba de este nivel de Eb/No, se puede transmitir información libre de errores y que por debajo de éste la información que se transmite contiene un gran número de errores.

## 2.5. Códigos convolucionales

Los códigos convolucionales, son códigos correctores de error que transforman la información binaria original, continuamente y en bloques, de tal manera que no son sistemáticos. Se dice que son de bloque ya que por cada k bits que entran al codificador se tienen a la salida n bits, es decir, que son de tasa $R = k/n$. Son no sistemáticos, ya que en la *palabra codificada* $c$ el mensaje no es explícito. Un código convolucional puede ser descrito de la siguiente manera:

$$c = (n, k, M) \tag{2.19}$$

- c es la palabra codificada
- n es la tupla de bits de salida
- k es la tupla de bits de entrada
- M es el tamaño de memoria (# de registros de k-tuplas)

Los códigos convolucionales han sido muy usados en aplicaciones como telefonía, televisión, en comunicaciones satelitales y en transmisiones de voz.

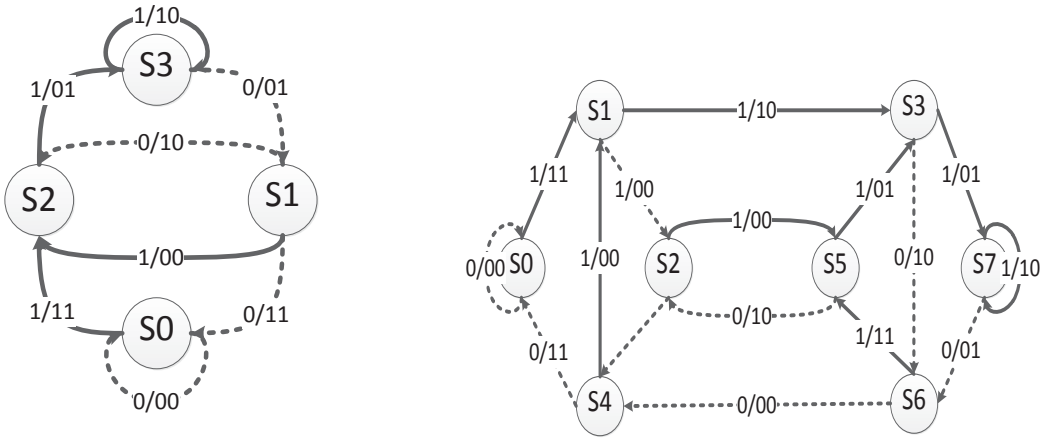
### 2.5.1. Codificación convolucional

La codificación convolucional se puede realizar de distintas maneras: mediante vectores generadores, diagramas de estados y registros de corrimiento.

En la implementación con vectores generadores, la codificación se efectúa al multiplicar el vector por una matriz que contiene el mensaje a codificar y resulta ser más compleja de implementar tanto en software como en hardware, ya que se requiere más memoria en software y más dispositivos en hardware, para almacenar los resultados.

En la implementación mediante diagrama de estados, es necesario determinar el ciclo que sigue el proceso de la codificación. Para describir este ciclo, se tienen que encontrar todos los estados por los que se pasa en el proceso de codificación, las entradas que provocan las transiciones de los estados y las salidas que se obtienen como respuesta. Los estados, entradas y salidas, son colocados en una tabla y se ocupan en la codificación de cada uno de los elementos de la secuencia a codificar. El elemento a codificar se compara con cada una de las entradas de la tabla, así como el estado actual, si estos coinciden, se toma como estado actual el estado siguiente y la salida correspondiente, esto se hace hasta terminar con el último elemento de la secuencia. Lo antes descrito hace que este método sea difícil de implementar, porque se requieren $2^{\mathrm{M}}$ comparaciones, que corresponden a las transiones de estados. El número de comparaciones depende del tamaño de la memoria y se incrementa exponencialmente. Este número a su vez, es igual al doble del total de los $2^{M-1}$ estados que

conforman el ciclo del proceso de codificación. En la obtención del diagrama de estados, el incremento del número de estados dificultad su elaboración como se muestra en las Figuras 2.3(a) y 2.3(b):




(a) Diagrama de estados (2,1,3)

(b) Diagrama de estados (2,1,4)

Figura 2.3: Ejemplo de diagrama de estados para tamaños de memoria M $= 3, 4$

La implementación mediante registros de corrimiento es más sencilla que las anteriores, esto porque no requiere de una tabla, ni de matrices. En este caso sólo se usa un registro de corrimiento de tamaño M, los k bits de entrada son introducidos en el registro que previamente ha sido recorrido k posiciones y enseguida se efectúan las sumas en módulo 2 para obtener las salidas vea la Figura 2.4. Para efectuar las operaciones en módulo 2, es necesario adecuar el vector generador a binario para indicar qué posiciones del registro participarán en la suma.

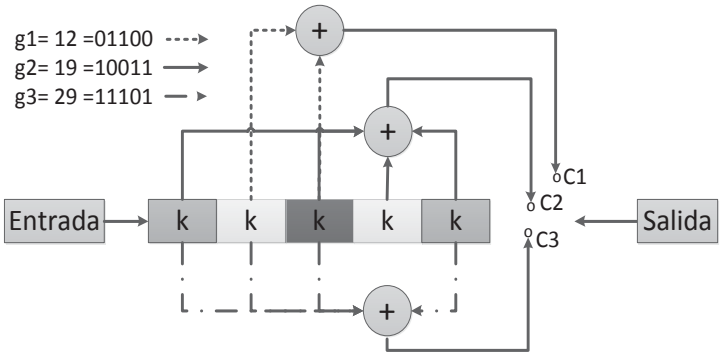



Figura 2.4: Ejemplo de codificador convolucionador.

En la figura anterior el codificador convolucionador está compuesto por los vectores generadores $g_1$, $g_2$ y $g_3$ y es de tamaño M = 5.

Como ejemplo de la codificación convolucional, se tiene un convolucionador como el de la Figura 2.5 de características (2,1,3) y vectores generadores $g_1 = 7$ y $g_2 = 5$.

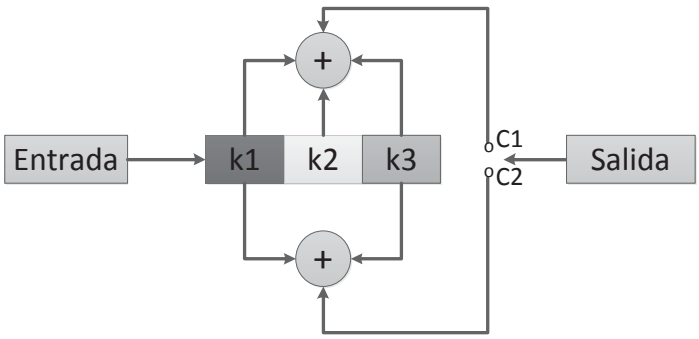



Figura 2.5: Codificador convolucionador (2,1,3).

Como primer paso es determinar el número de estados ne utilizando la ecuación siguiente:

$$ne = 2^{M-1} \tag{2.20}$$

Sustituyendo en 2.20 M = 3 se tiene que el ne = 4 que son *S0=00, S1=01, S2=10 y S3=11.*

A continuación, se muestra la Tabla 2.1 con los resultados de la codificación del mensaje m = 11011 usando el codificador mostrado en la Figura 2.5.

Tabla 2.1: Resultado de la codificación.

| Entrada | Contenido del registro | Estado actual | Estado anterior | c1 | c2 |
|---|---|---|---|---|---|
| - | 000 | 00 | 00 | - | - |
| 1 | 100 | 10 | 00 | 1 | 1 |
| 1 | 110 | 11 | 10 | 0 | 1 |
| 0 | 011 | 01 | 11 | 0 | 1 |
| 1 | 101 | 10 | 01 | 0 | 1 |
| 1 | 110 | 11 | 10 | 0 | 1 |
| 0 | 011 | 01 | 11 | 0 | 1 |
| 0 | 001 | 00 | 01 | 1 | 1 |

La tabla anterior, muestra las columnas de estado actual y estado siguiente, que se obtienen al tomar las localidades del registro $k1$, $k2$ para el estado siguiente y $k2$, $k3$ para el estado actual.

Para obtener la palabra codificada se lee de arriba hacia abajo comenzando con $c1$ y, enseguida, con $c2$, al hacer esto se obtiene la palabara codificada.

$$c = 1100101010111$$

La construcción del diagrama de estados de este ejemplo, se hace tomando en cuenta las columnas del estado siguiente, estado actual, entrada, $c1$ y $c2$, el cual se muestra en la Figura 2.6.

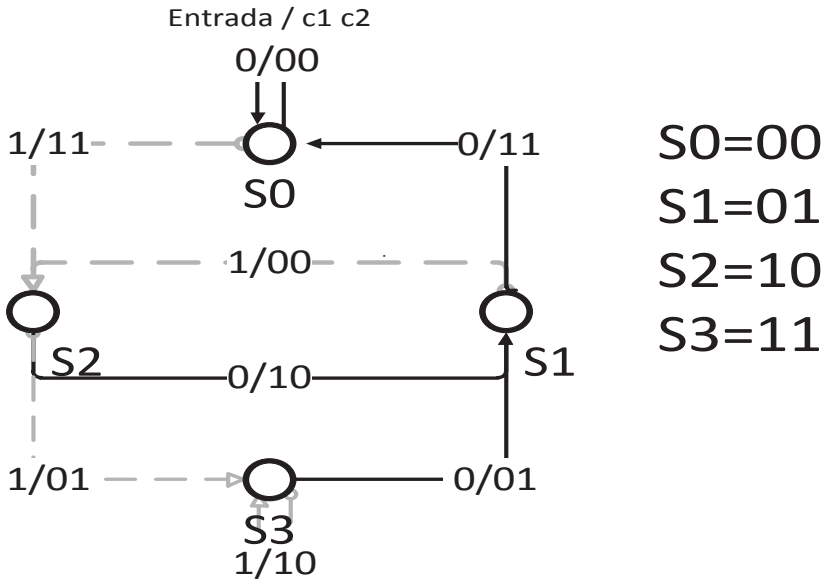



Figura 2.6: Diagrama de estados (2,1,3).

### 2.5.2. Decodificación convolucional dura

En la decodificación los bits entregados por el demodulador son introducidos en un diagrama de Trellis [13], para determinar cuál es la ruta más probable que pudo recorrer la información al ser codificada. Para la elaboración del diagrama de Trellis, es necesario conocer las transiciones del estado y la información de entrada que la genera, así como de los resultados de la codificación.

¿En qué consiste la decodificación usando el diagrama de Trellis árbol binario? Para empezar, es necesario conocer el diagrama de estados ó máquina de estados. Se comienza en el estado S0=00, que sólo puede hacer transiciones a dos estados siguientes. En la etapa siguiente estos dos estados pueden hacer transiciones a otros dos estados, hasta llegar a una etapa donde todos los estados conforman un ciclo (máquina de estados). En la Figura 2.7, se muestra el diagrama de Trellis del convolucionador $R = 1/2$ utilizado en la sección de codificación convolucional ver la Figura 2.6.

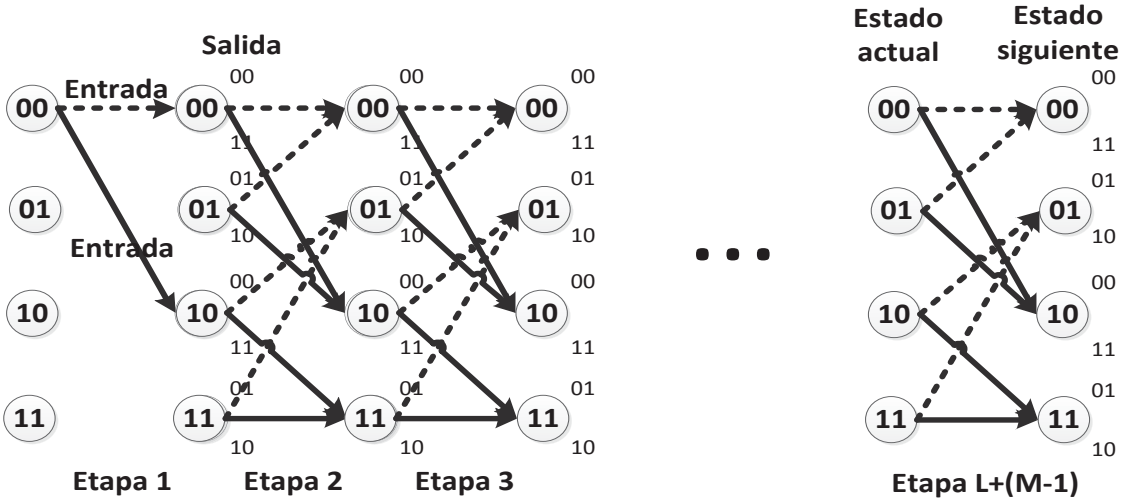



Figura 2.7: Diagrama de Trellis (2,1,3).

En la figura anterior, en la etapa 3 se puede observar que todos los estados generan un patrón, que se repetirá también en todas las etapas siguientes. Las líneas punteadas indican que la transición fue provocada por 0 y las líneas sólidas por un 1 a la entrada.

En la decodificación, se van introduciendo los símbolos a cada una de las etapas del diagrama, se comparan las salidas y se contabilizan los bits de código en que difieren. Se guarda la diferencia, así como el dato que provoca la transición. Al final de cada etapa, se descartan aquellas rutas con la mayor distancia de Hamming y se elige a la de menor distancia, la cual contiene el mensaje que se codificó con la mayor probabilidad [2].

Observando lo anterior *Andrew Viterbi* [14] propusó un algoritmo para la decodificación en su artículo *Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm* y que lleva su nombre en su honor. En éste no se comienza en el estado S0=00 como en el diagrama de Trellis ver la Figura 2.7, se empieza analizar todas las ramas del diagrama ver la Figura 2.8 y no hasta llegar a la etapa donde los estados repiten un patrón,

observó que los resultados que se obtienen con este método son similares a los obtenidos comenzando en el estado S0=00.

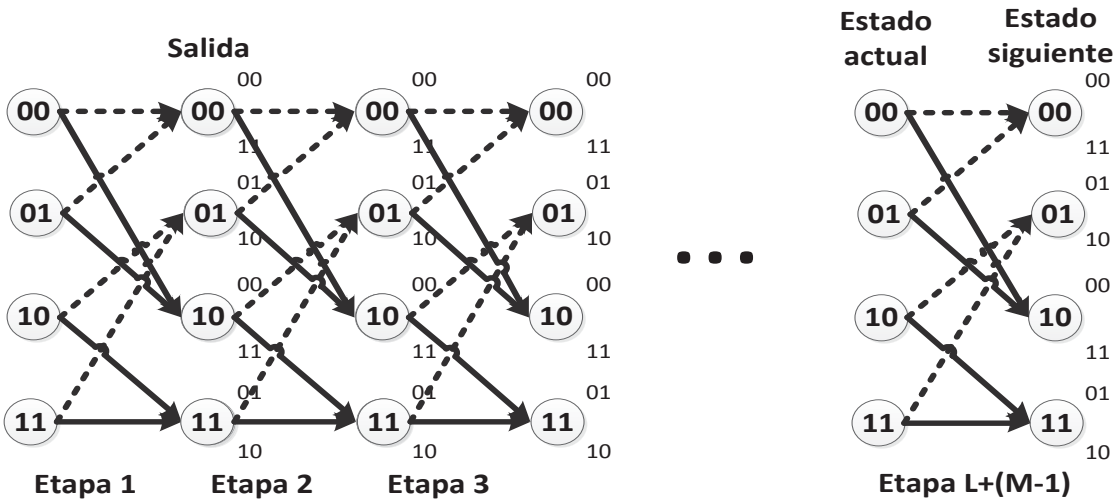



Figura 2.8: Diagrama de Trellis modificación Viterbi.

Al introducir al diagrama de Trellis la palabra codificada se puede recuperar el mensaje original m=1101100 al registrar las entradas que generan la secuencia del código.

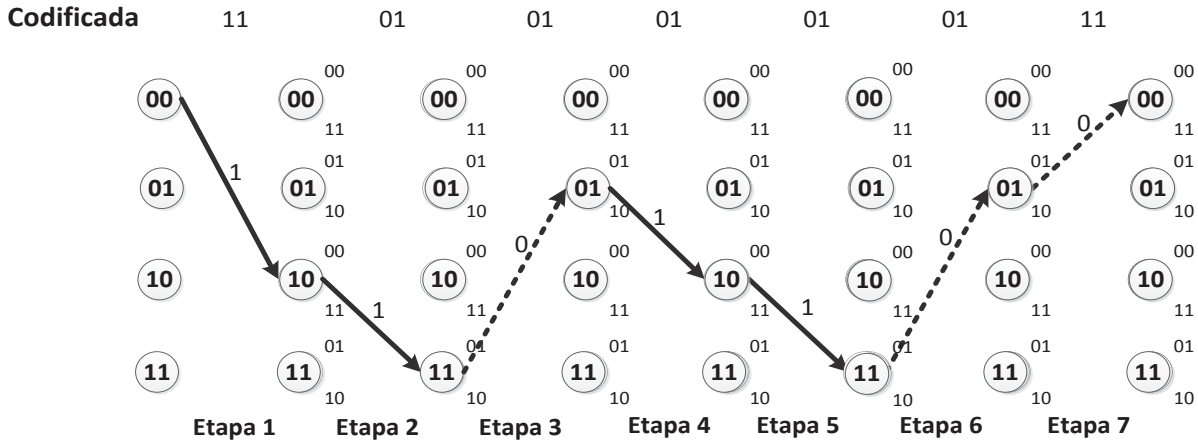



Figura 2.9: Ruta que contiene el mensaje original.

### 2.5.3. Códigos convolucionales recursivos sistemáticos

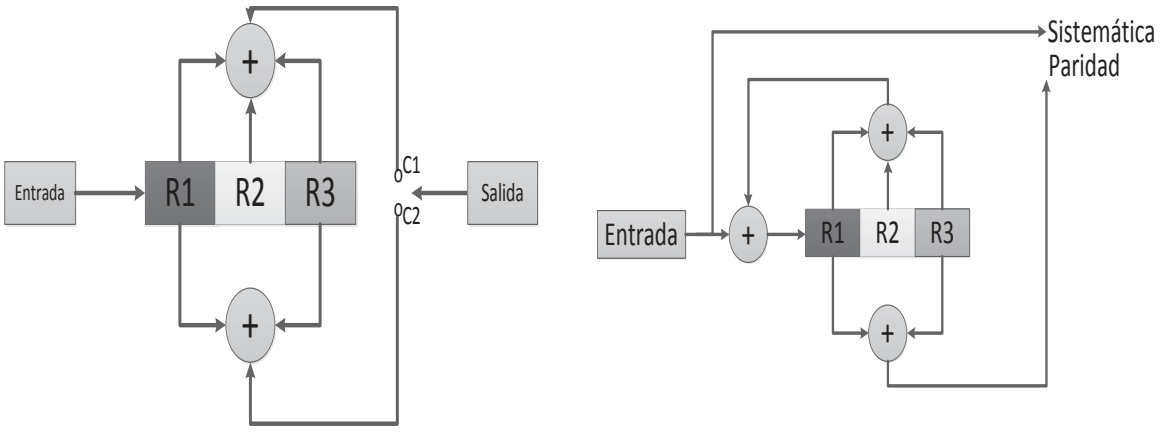
Hasta ahora se ha hablado de los códigos convolucionales en su forma no sistématica. Los códigos convolucionales sistemáticos recursivos RSC, son códigos convolucionales donde la salida contiene los bits de entrada y el registro incluye retroalimentación de un polinomio. Presentan un mejor desempeño que los no recursivos debido a la baja correlación que existe entre las entradas y salidas[15].

En el proceso de la codificación convolucional no sistemática, se comienza en el estado inicial S0=00 y se termina en este mismo, para esto se agregan ceros suficientes a los datos originales para limpiar el registro de corrimiento y llegar al final al estado inicial. En la codificación RSC esto no sucede, porque el codificador, debido a la retroalimentación, se comporta como un filtro de respuesta al impulso infinita (IIR). En este caso se calcula el número de bits de código y no se espera regresar al estado inicial. En la decodificación se siguen las mismas reglas de los códigos convolucionales. Se representan de la siguiente manera:

$$G = (1, \frac{g_1}{g_0}) \tag{2.21}$$

donde $G$ es la matriz generadora, $g_1$ el vector generador de retroalimentación y $g_0$ el vector generador de alimentación hacia adelante.

En la Figura 2.10(a) se muestra el codificador convolucional de tasa $R = 1/2$ y un RSC de tasa $R = 1/3$, como se puede apreciar la diferencia que existe entre estos codificadores, es de que para el caso del RSC, este se retroalimenta con una de sus salidas ver la Figura 2.10(b).




(a) Convolucionador (2,1,3). (b) RSC (2,1,3).

Figura 2.10: Codificador convolucional y RSC.

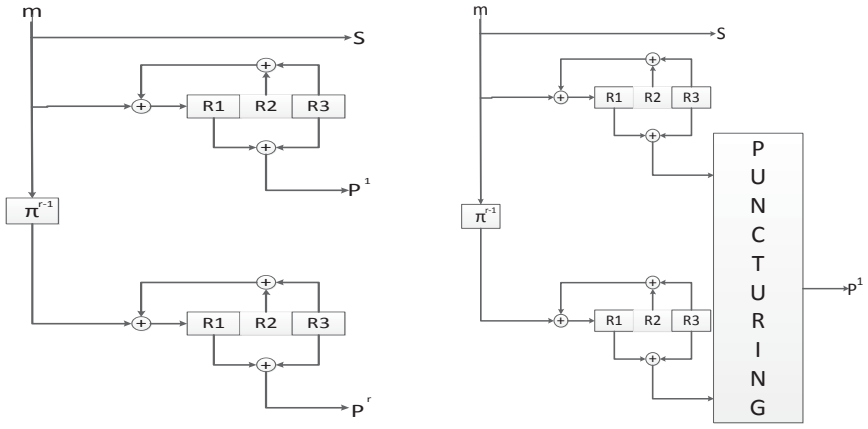
## 2.6. Turbo códigos

Los turbo códigos fueron presentados en el año de 1993 en la IEEE *Internaional Conference on Communications* en *Ginebra, Suiza*, por *Claude Berrou* y *Alain Glavieux*.

Con el tiempo, los turbo códigos han sido ampliamamente estudiados y adoptados por varios sistemas de comunicación, permitiendo tasas de transmisión cercanas a la capacidad de canal y con probabilidades de error muy bajas. Estos son ocupados ampliamente en dispositivos portables, con aplicaciones multimedia que requieren tasas de transmisión muy elevadas, del orden de $1 \times 10^4$ bits, con una BER de $1 \times 10^{-5}$ a menos de 1 dB de SNR.

Los turbo códigos se componen generalmente de dos o más codificadores RSC como se muestra en la Figura 2.11. Cuentan con un entrelazador pseudo-aleatorio al que se denota con $\pi$ el cual tiene como objetivo reacomodar la información que entra al turbo codificador. Esto con la finalidad de que la información que entra a cada RSC esté correlacionada lo menos posible con la información de entrada.

Los turbo codificadores pueden contar además con mecanismos de eliminación de bits o *puncturing* (ponchado), cuya finalidad es reducir el ancho de banda. El puncturing selecciona ciertas salidas de los RSC que serán enviadas en la palabara codificada. Como ejemplo de lo anterior, tenemos un turbo código que cuenta con dos RSC de tasa R=1/3. Al aplicarle el mecanismo de puncturing que consiste el seleccionar una de las salidas de los RSC, da como resultado un turbo código de tasa R=1/2.




(a) Turbo codificador sin puncturing. (b) Turbo codificador con puncturing.

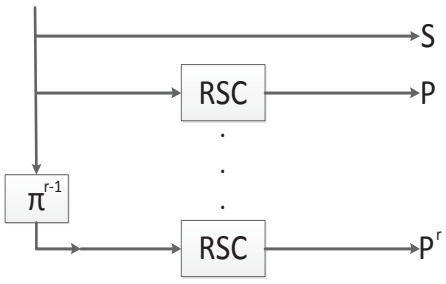
Figura 2.11: Turbo codificadores

En la figura anterior, el exponente $r$ indica el número de codificadores RSC que se utilizan en la turbo codificación.

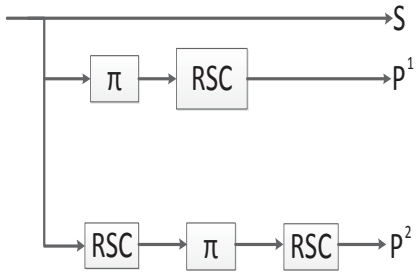
Los turbo codificadores se pueden clasificar según su arquitectura, es decir, según la forma en que están acomodados los entrelazadores y codificadores[16]. Tomando en cuenta lo anterior, tenemos:

- **Paralell Concatened Convolutional Code (PCCC).** Este tipo de codificador es el más usado y recibe su nombre, debido a que el mensaje es codificado una, dos o más veces y se transmiten todos los bits de código. Para ello, se hace uso de la secuencia normal del mensaje y la secuencia del mensaje entrelazada ver la Figura 2.12(a).
- **Serial Concatened Convolutional Code (SCCC).** Este codificador, está compuesto por un codificador interno, un entrelazador y un codificador externo. El mensaje a codificar primero es codificado por el coficicador externo su salida es entrelazada y, por último, es pasada al codificador interno, el cual arroja la palabra codificada ver la Figura 2.12(c).
- **Hybrid Concatened Convolutional Code (HCCC).** Este codificador es la combinación de los codificadores PCCC y SCCC ver la Figura 2.12(b).

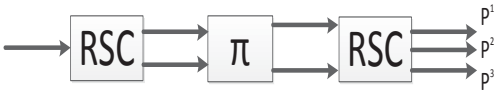
En las Figuras2.12(a), 2.12(b) y 2.12(c) se muestran las diferentes arquitecturas de los turbo codificadores.




(a) Turbo codificador PCCC.




(b) Turbo codificador HCCC.




(c) Turbo codificador SCCC.

Figura 2.12: Turbo codificadores.

Una característica importante que diferencia a las distintas arquitecturas, es que a niveles altos de SNR resulta mejor utilizar el codificador PCCC y para niveles bajos de SNR los codificadores SCCC. Por otra parte el tamaño, del entrelazador define el punto de SNR en que el desempeño de un codificador PCCC deja de ser mejor que un codificador SCCC [2].

### 2.6.1. Turbo codificación

En el proceso de la turbo codificación se emplean uno o más RSC, en la Figura 2.13 se muestra un turbo codificador PCCC, en el cual los RSC están acomodados en forman paralela, son códigos de bloque porque por cada $k$ bits que entren a la salida se tendrán $n$ bits [17]. Son sistemáticos porque, a la salida, el mensaje $m$ se encuentra explicíto en la palabra codificada $c$.

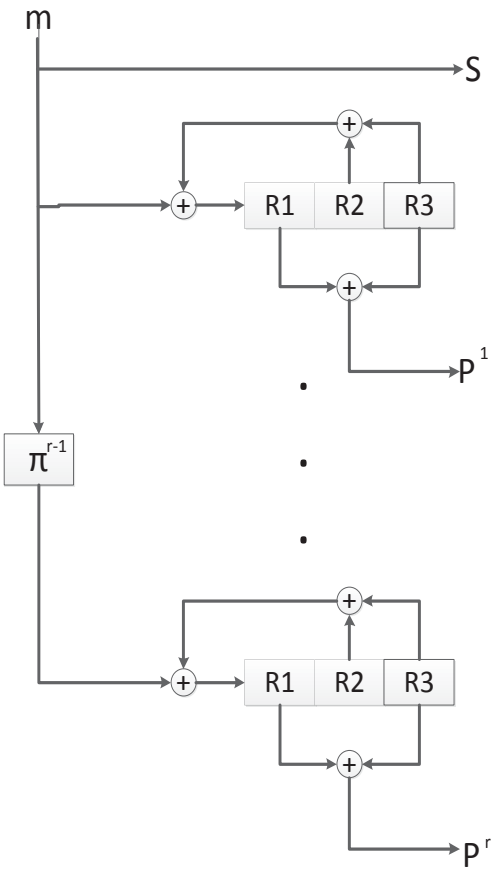



Figura 2.13: Turbo codificador PCCC [1].

En la figura anterior, además de los RSC, se cuenta con un entrelazador $\pi$, encargado de

reacomodar la información de entrada, esto con la finalidad de que las distintas secuencias que entran a los codificadores RSC estén correlacionadas lo menos posible. Existen diferentes tipos de entrelazadores pero los más usados son:

**1) Renglón - Columna**

Este consiste en escribir renglón por renglón a lo largo de una matriz el mensaje y leerlo en columnas. Para ejemplificar este procedimiento se tiene el mensaje siguiente:

$$m = [m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_6, m_9]$$

Escribiendo en la matriz se obtiene la Tabla 2.2.

Tabla 2.2: Entrelazador Renglón-Columna

| | | |
|---|---|---|
| $m_1$ | $m_2$ | $m_3$ |
| $m_4$ | $m_5$ | $m_6$ |
| $m_7$ | $m_8$ | $m_9$ |

Ahora, leyendo la Tabla 2.2 por columnas de izquierda a derecha, se obtiene el nuevo mensaje entrelazado a codificar:

$$m' = [m_1, m_4, m_7, m_2, m_5, m_8, m_3, m_6, m_9]$$

**2) Entrelazado Espiral**

El procedimiento es el mismo que el de renglón-columna, sólo que en este caso la información es leída en forma diagonal.

$$m' = [m_1, m_4, m_2, m_7, m_5, m_3, m_8, m_6, m_9]$$

**3) Entrelazado pseudo-aleatorio**

En éste la secuencia del mensaje se escribe en la matriz en una forma pseudo-aleatoria, enseguida es leída por columnas. La forma en que se genera la secuencia, es mediante un generador de números en el rango de 1 a $L$, que es el tamaño del mensaje a codificar.

De la Figura 2.13, como resultado de la codificación, se obtienen los vectores siguientes:

$$S = [S_0, S_1, \ldots, S_{L-1}] \quad (2.22)$$
$$P^0 = [{P_0}^0, {P_1}^0, \ldots, {P_{L-1}}^0] \quad (2.23)$$
$$\vdots = \vdots \quad (2.24)$$
$$P^r = [{P_0}^r, {P_1}^r, \ldots, {P_{L-1}}^r] \quad (2.25)$$

donde $S$ es el vector que contiene el mensaje m original, $P^r$ el vector de paridad que contiene los resultados de la codificación de cada RSC, $L$ indica el tamaño del mensaje y $r$ es el número total de codificador RSC.

Para obtener la palabra codificada, se multiplexan los vectores anteriores para dar como resultado:

$$c = [(S_0, {P_0}^0, \ldots, {P_0}^r), \ldots, (S_{L-1}, {P_{L-1}}^0, \ldots, {P_{L-1}}^r)]$$

La palabra codificada tiene un tamaño de n$L$, donde n es el número de bits que conforman la tupla de salida del turbo codificador y $L$ es el tamaño del mensaje.

### 2.6.2. Turbo decodificación

La turbo decodificación es la parte más importante de este trabajo. En esta etapa se usa el algoritmo MAP, introducido en 1974 por *Bahl* [18]. El objetivo es reparar los daños que los datos hayan sufrido por los efectos del canal para obtener una estimación de los datos transmitidos.

La turbo decodificación cuenta con varias etapas de decodificación, las que interactúan entre sí con la finalidad de que cada etapa haga una mejor estimación que la anterior. El turbo decodificador tiene una estructura complementaria al codificador. En este proceso, un primer decodificador utiliza los bits del mensaje codificado, los bits de paridad correspondientes y una estimación de la decodificación hecha por otro decodificador (información a priori). La información a priori es adicional a las otras entradas del decodificador. Con estos datos el decodificador crea una secuencia de salida y otra información extrínseca que pasa a otro decodificador. El siguiente decodificador utilizará la nueva información extrínseca como información a priori y la información extrínseca resultante se retroalimenta al decodificador anterior. Es gracias a este proceso de retroalimentación que se les denomina turbo códigos ver la Figura 2.14. En cada iteración, el decodificador se va acercando más al mensaje original transmitido, conforme más iteraciones se realizan, la decodificación mejora gradualmente. La decodificación se hace de una manera suave, es decir, procesa los bits recibidos sin pasarlos por un proceso de detección dura.

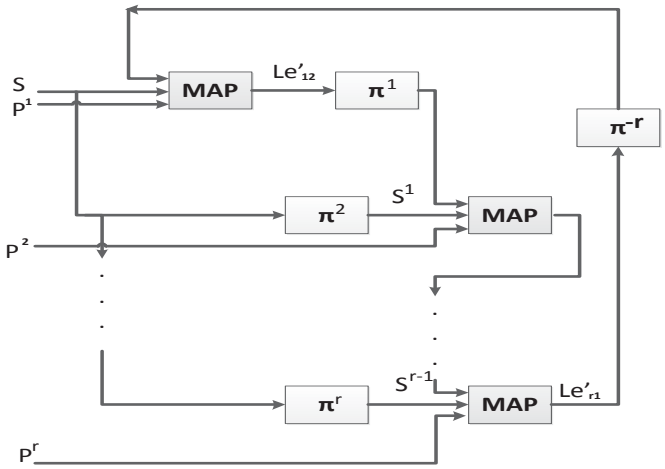



Figura 2.14: Turbo decodificador iterativo PCCC.

Todo este proceso de decodificación iterativa hace uso del algoritmo conocido como Maximum A Posteriori (MAP) el que, a su vez, hace uso del enrejado o diagrama de Trellis. Ya se mencionó que entre mayor sea el tamaño de la memoria M, es más complicado realizar el proceso de decodificación; la derivación de este algoritmo se omite aquí debido a su alta complejidad. Podemos comentar que su deducción involucra temas relacionados con conceptos de probabilidad y cadenas de Markov. Aquí sólo se muestran los parámetros más importantes considerados en el algoritmo.

Este algoritmo calcula las métricas $\gamma$ de cada rama en el diagrama de Trellis, así como de los coeficientes de recursión hacia adelante $\alpha$ y hacia atrás $\beta$. A continuación se describe cómo se obtienen todos estos parámetros y cómo son usados por el algoritmo MAP.

### Cálculo de métricas $\gamma$

Se toma un diagrama de Trellis de tamaño $t = 1, \ldots, L$ como el de la Figura 2.15. Para cada bit de mensaje original se deben calcular todas las métricas $\gamma$ correspondientes a las posibles rutas. En términos, generales se obtendrá como resultado del cálculo de las métricas $\gamma$ una matriz de $2^M \times t$, las métricas $\gamma$ de cada rama se calculan con la fórmula siguiente:

$$\gamma_t{}^i(l', l) = p_t(i) \exp\left(\frac{(-\sum_{j=0}^{N-1}(r_{t,j}{}^i - x_{t,j}{}^i(l))^2)}{2\sigma^2}\right) \tag{2.26}$$

En la Ecuación 2.26 los términos $l$ y $l$ indican la transición del estado anterior al estado siguiente, $i$ toma valores entre 0 y 1 que son los bits del mensaje original que generan las transiciones. En la suma, $N$ indica el tamaño de la tupla que será procesada por el decodificador.

El término ${r_{t,j}}^i$ toma los valores recibidos de cada elemento de la tupla, la cual está comformada por la información sistemática y sus respectivas paridades, ${x_{t,j}}^i(l)$ toma los valores ideales de cada elemento de la tupla de salida según el RSC. La suma de la ecuación 2.26 indica que se trabaja con distancias Euclidianas y no con distancias Hamming dada la naturaleza suave de la decodificación MAP.

En la Ecuación 2.26 aparece la probabilidad a priori de recibir un 0 o un 1. Esta probabilidad en la primera iteración es $p_t(i)$=0.5 para el decodificador 1. Para el decodificador 2, ésta se calcula tomando en cuenta la verosimiltud extrínseca proveniente del decodificador 1. La verosimilitud extrínseca encontrada por el decodificador 2 es retroalimentada al decodificador siguiente, y así sucesivamente, hasta llegue al último decodificador, el cual la retroalimentará al primer decodificador. El valor de la verosimilitud extrínseca sólo depende de los valores de la paridad, esto se explica más adelante. El cálculo de las probabilidades se hace considerando un canal binario simétrico, es decir, lo que se envía sólo puede tomar dos posibles valores y se calcula con las siguientes ecuaciones:

$$p_t(+1) = \frac{e^{Le_{df,dd}(c_t)}}{1 + e^{Le_{df,dd}(c_t)}} \tag{2.27}$$

$$p_t(-1) = \frac{1}{1 + e^{Le_{df,dd}(c_t)}} \tag{2.28}$$

En las Ecuaciones 2.27 y 2.28 aparece el término $Le_{df,dd}$, el cual indica el decodificador *df* del que proviene la verosimilitud y al decodificador *dd* que se dirige.

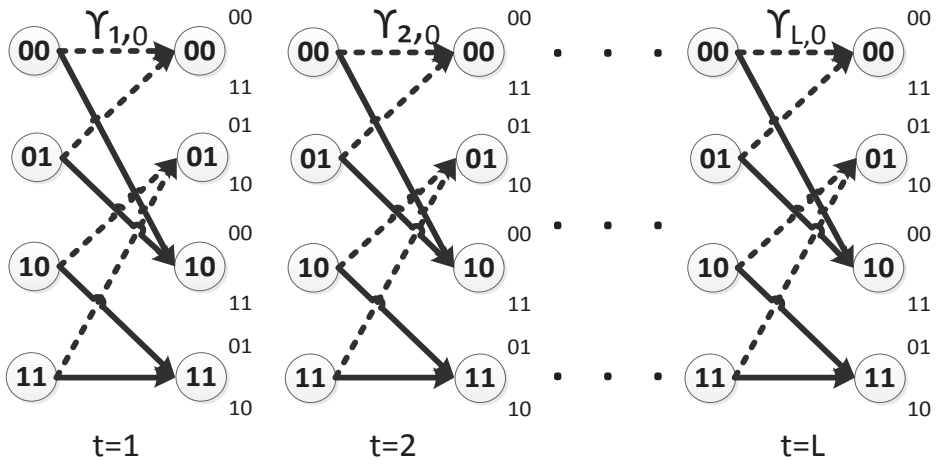



Figura 2.15: Diagrama de Trellis cálculo de las métricas.

### Cálculo de los coeficientes $\alpha$

El cálculo de los coeficientes hacia adelante $\alpha$ se obtiene al multiplicar la métrica $\gamma$ de la rama por la cual se se llega a ese estado por la alfa del estado proveniente ver la Figura 2.16, comenzando en $t = 1$.

$$\alpha_{t,z}(l) = \sum_{l'=0}^{ne-1} \sum_{i\in(0,1)} \alpha_{t-1}(l')\gamma_t{}^i(l', l) \tag{2.29}$$

En la Ecuación 2.29 $t$ indica la etapa actual, $z$ indica el número de la alfa y va de 0 a $ne - 1$, $l$ indica el estado actual y $l$ indica el estado anterior.

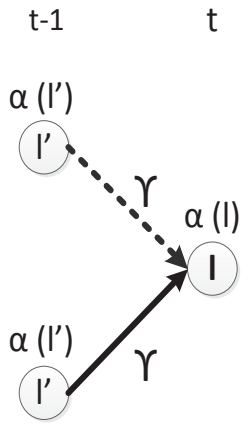



Figura 2.16: Cálculo de los coeficientes de recursión hacia adelante.

Para cada bit de entrada en el diagrama de Trellis se tendrá ne=$2^{M-1}$ de alfas. En la etapa $t = 0$ las alfas son inicializadas como sigue:

$\alpha_{0,0}(0) = 1$ y $\alpha_{0,z}(l) \neq 0$ para $l \neq 0$ y $z \neq 0$.

### Cálculo de los coeficientes $\beta$

El cálculo de los coeficientes hacia atrás $\beta$, similarmente a las $\alpha$, se obtiene al multiplicar la métrica $\gamma$ de la rama por la que se llega a ese estado por $\beta$ del estado orígen ver la Figura 2.17, comenzando en $t = L$ y regresando hacia $t = 1$.

$$\beta_{t,z}(l) = \sum_{l'=0}^{ne-1} \sum_{i\in(0,1)} \beta_{t-1}(l')\gamma_t{}^i(l,l') \tag{2.30}$$

En la Ecuación $t$ indica la etapa actual, $z$ indica el número de la alfa y va de 0 a $ne-1$, $l$ indica el estado actual y $l$ indica el estado anterior.

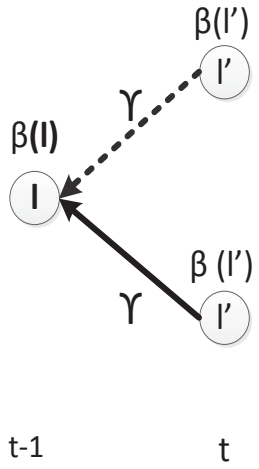



Figura 2.17: Cálculo de los coeficientes de recursión hacia atrás.

Para cada bit en el diagrama de Trellis se tendrán un número ne=$2^{M-1}$ de $\beta$. Si en el proceso de codificación se termina en el estado S0=00 las $\beta$ son inicializadas de la siguiente manera:

$$\beta_{L,0}(0) = 1 \text{ y } \beta_L(l,z) \neq 0 \text{ para } l \neq 0 \text{ y } z \neq 0.$$

Por el contrario, cuando el proceso de codificación termina en un estado distinto al S0=00, las betas $\beta$ son inicializadas de la siguiente manera:

$$\beta_{L,z}(0) = 1/ne \text{ para } z = 0, 1, \ldots, ne-1$$

Después de haber encontrado los parámetros $\alpha$, $\beta$ y $\gamma$, lo que sigue es estimar, para cada bit el *LLR (Log Likelihood Ratio)*. Esto se hace calculando lo siguiente:

$$L(c_t) = \log(\frac{\sum_{l=0}^{ne-1} \alpha_{t-1}(l')\gamma_t{}^1(l',l)\beta_t(l)}{\sum_{l=0}^{ne-1})\alpha_{t-1}(l')\gamma_t{}^0(l',l)\beta_t(l)} \tag{2.31}$$

Sustituyendo la Ecuación 2.26 en la Ecuación 2.31 y desarrollando, se tiene que la verosimilitud extrínseca.

$$Le_{df,dd}(c_t) = \log \frac{\sum_{l=0}^{ne-1} \alpha_{t-1}(l') e^{\frac{(-\sum_{j=1}^{N-1}(r_{t,j}-x_{t,j}{}^{1}(l))^2)}{2\sigma^2}} \beta_t(l)}{\sum_{l=0}^{ne-1} \alpha_{t-1}(l') e^{\frac{(-\sum_{j=1}^{N-1}(r_{t,j}-x_{t,j}{}^{1}(l))^2)}{2\sigma^2}} \beta_t(l)} \tag{2.32}$$

Resumiendo lo anterior se tiene el algoritmo siguiente:

**Algoritmo 2.1** El algortimo MAP

| | | |
|---|---|---|
| 1: | Calcular probabilidad $c_t = +1$ | usando la ecuación 2.27 |
| 2: | Calcular probabilidad $c_t = -1$ | usando la ecuación 2.27 |
| 3: | Calcular métricas $\gamma$ | usando la ecuación 2.26 |
| 4: | Calcular alfas $\alpha$ | usando la ecuación 2.29 |
| 5: | Calcular betas $\beta$ | usando la ecuación 2.6.2 |
| 6: | Calcular la verosimilitud $Le_{df,dd}$ | usando la ecuación 2.32 |

Como se mencionó anteriormente, en la decodificación turbo se emplean módulos de decodificación con base en el algoritmo MAP, a los que se alimenta el mensaje, sin entrelazar o entrelazado según sea el caso, con los bits de paridad de cada codificador RSC. Aunque la Ecuación 2.31 nos da una estimación de la información que se transmitió, lo que se necesita en cada decodificador es la verosimilitud extrínseca para poder calcular la $p_t(i)$ (Ecuaciones 2.28 y 2.27). La verosimilitud se calcula con la Ecuación 2.32 o de la siguiente manera:

$$Le_{df,dd}(c_t) = L(c_t) - \frac{2}{\sigma^2} r_{t,0} - Le_{df,dd}(c_t) \tag{2.33}$$

El proceso de la decodificación turbo, es un proceso iterativo en el que las verosimilitudes extrínsecas son pasadas de un codificador a otro. En cada iteración, los valores de las verosimilitudes cambian hasta llegar a un punto donde permanecen en un estado prácticamente constante, lo que quiere decir que por más iteraciones que se hagan los resultados de la estimación no cambiarán. Al proceso anterior se le conoce como algoritmo iterativo MAP.

Para el buen desempeño de los turbo códigos se toman en cuenta las siguientes consideraciones [17].

1. Para obtener una BER de máximo $1 \times 10^{-5}$ se deben utilizar codificadores de corta duración por lo general de M = 4 o menos.

2. Los codificadores RSC, no necesariamente deben de ser iguales para un obtener un buen desempeño.

3. Los códigos convolucionales recursivos tienen un mejor desempeño que los códigos convolucionales.

4. Adicionando codificadores se obtienen tasas de codificación menores. Esto es, se agrega más redundancia a la información.

5. El mejor entrelazador es el pseudo-aleatorio. Un entrelazado de renglón-columna tiene un buen desempeño para mensaje cortos.

Como ejemplo del desempeño de los turbo códigos vea la Figura 2.18. En ella se observa el comportamiento de un código con una tasa de R=1/2 que logra una BER de $1 \times 10^{-5}$ para una $\frac{\text{Eb}}{\text{No}}$=0.7 dB, con un tamaño de bloque de K=$2^{16}$ bits, después de 18 iteraciones, comparado con código convolucional de caracteristicas (2, 1 ,6) de la NASA, que requiere para alcanzar la misma BER un $\frac{\text{Eb}}{\text{No}}$ =4.2 dB.

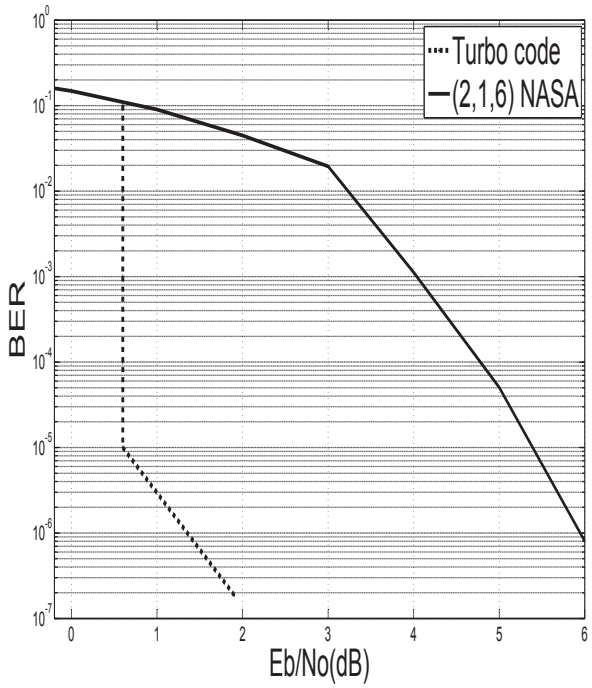
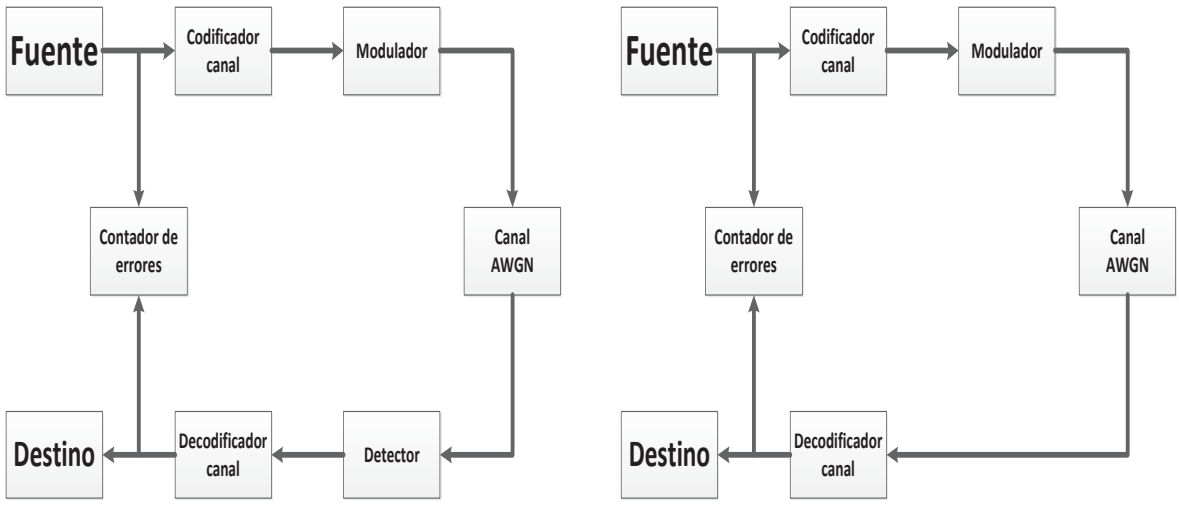



Figura 2.18: Comparación de un Turbo Código $R = 1/2$ y un convolucional (2, 1 ,6) [2].

# Capítulo 3
# Realización práctica

En este capítulo, se describen los criterios tomados en cuenta en la realización práctica de las siguientes técnicas de codificación y decodificación de canal usando un DSP: códigos convolucionales, RSC, RSC con algoritmo MAP y turbo códigos.

La realización práctica de las técnicas de codificación y decodificación de canal se dividió en dos partes. La primera parte consistió en programar las técnicas hard los códigos convolucionales y los códigos RSC, para ello se tomó como referencia el esquema de comunicación de la Figura 3.1(a). En la segunda parte, se programaron las técnicas soft para el algoritmo MAP y los turbo códigos. A diferencia de las técnicas hard, las soft no requieren del módulo demodulador, esto porque la información se procesa tal y como es recibida ver la Figura 3.1(b).




(a) Esquema de comunicación con demodulador. (b) Esquema de comunicación sin demodulador.

Figura 3.1: Esquemas de comunicación utilizados en la realización práctica de las técnicas duras y suaves en el DSP.

Los esquemas de comunicación mostrados en las figuras anteriores, tienen en común los módulos, fuente, modulador, contador de errores, canal y destino, por lo que la realización en el DSP es la misma. Los módulos codificador y decodificador del canal son los de mayor relevancia en este trabajo, ya que es aquí donde se implementa cada una de las diferentes técnicas estudiadas.

A continuación, se describe el funcionamiento de los módulos que conforman a los sistemas de comunicación mencionados.

- ***Fuente.*** Este módulo se encarga de generar una secuencia de números binarios aleatorios, los cuales son guardados en un arreglo de determinado tamaño, que es enviado al módulo codificador de canal.

- ***Modulador.*** Recibe del codificador de canal la secuencia que contiene los números binarios aleatorios codificados. Éste usa una señalización antipodal, es decir, al dígito binario con valor de 0 le asigna un $-1$ y al 1 el valor $+1$.

- ***Canal AWGN.*** Este módulo se encarga de generar ruido blanco gaussiano con caracteristicas ($\mu = 0$, $\sigma^2$), el cual se añade a la secuencia proveniente del modulador.

Para generar los valores, es necesario proporcionar la varianza que está en función de la relación *Eb/No* y se calcula de la siguiente manera:

$$\sigma^2 = \frac{p_t}{10^{\frac{\frac{Eb}{No}}{10}}} \tag{3.1}$$

donde $p_t$ es la potencia de la señal.

- ***Destino.*** Recibe el arreglo que proviene del decodificador de canal, que contiene los números aleatorios decodificados.

- ***Contador de errores.*** Cuenta los bits erróneos, para esto compara bit a bit cada elemento de la secuencia original (fuente) con el elemento correspondiente de la secuencia que contiene los números aleatorios decodificados (destino).

- ***Codificador de canal.*** Se encarga de transformar la información en una secuencia redundante, que es generada por la técnica de codificación de canal.

- ***Detector.*** Este módulo está presente sólo para las técnicas hard, y tiene como función determinar el símbolo que se transmitió. Para lograr esto, compara el nivel de las señales recibidas con un nivel de referencia. En este trabajo, por tratarse de señalización antipodal, se toma como nivel de referencia al 0, si el valor de la señal es mayor que cero se le asigna un valor de $+1$ y en caso contrario el valor de $-1$.

- ***Decodificador de canal.*** Procesa la secuencia de información proveniente del detector si se trata de decodificación dura ó directamente del canal si se trata de decodificación suave, para recuperar la secuencia de datos codificados.

En las secciones siguientes se describe a detalle el funcionamiento de los módulos codificador y decodificador de canal, para las técnicas hard y las técnicas soft, y de los criterios que se toman en cuenta para lograr la realización práctica en el DSP.

## 3.1. Técnicas de decodificación dura

En la realización práctica de las técnicas de códigos convolucionales y RSC fue necesario analizar el funcionamiento de cada una por separado. Como resultado de este análisis, se tiene que las técnicas siguen procedimientos similares en la parte de la codificación y decodificación.

### 3.1.1. Códigos convolucionales y RSC

En la parte de la codificación, para obtener un resultado correcto, es necesario limpiar los registros que se utilizan y para ello se agregra a los datos una secuencia de 0 de longitud M. En nuestro caso, se observó que al agregar sólo una secuencia de 0 de tamaño $M-1$ se obtiene una codificación y decodificación correcta, por lo que el tamaño de la secuencia codificada, correspondiente a cada una de las ramas del codificador convolucional, es $L+M-1$.

Los codificadores utilizados en esta realización práctica son de tasa $R = 1/2$ y $R = 1/3$, es decir, dos ramas y tres ramas respectivamente. Con lo anterior, se tiene que para diferentes tamaños de memoria M y mensajes a codificar de tamaño $L = 8$ bits, se obtienen secuencias codificadas de diferentes tamaños como se muestra en la Tabla 3.1

Tabla 3.1: Tamaños de secuencia codificada para $R = 1/2$ y $R = 1/3$

| Tamaño de memoria | $R = 1/2$ tamaño en bits | $R = 1/3$ tamaño en bits |
|---|---|---|
| 3 | 20 | 30 |
| 4 | 22 | 33 |
| 5 | 24 | 36 |
| 6 | 26 | 39 |
| 7 | 28 | 42 |

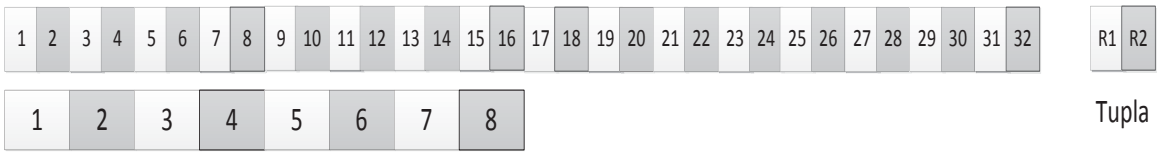
Para determinar el tamaño de la secuencia resultado de la codificación, se calcula como el número de ramas*(L+ (M-1)). En el proceso de codificación si las secuencias que se obtienen

son mayores a los 32 bits de la palabra del DSP, se obtienen resultados no deseados lo que provoca una realización práctica incorrecta.

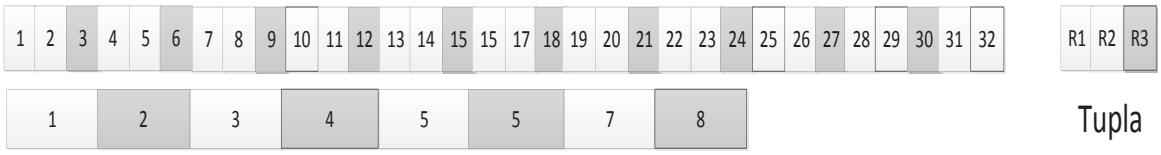
En la parte de codificación de canal y tomando en cuenta la Tabla 3.1 para los distintos codificadores convolucionales de $R = 1/2$, no es necesario validar el tamaño de la palabra y esto es porque la secuencia resultado de la codificación no es mayor a 32 bits. En la implementación del módulo de ruido AWGN, los valores que este entrega pueden ser representados sin ningún problema por el rango de representación númerica del DSP.

En la realización práctica de los codificadores convolucionales de $R = 1/3$, como se aprecia en la Tabla 3.1, a partir de M = 4 se rebasa el rango de la palabra del DSP. En resumen se puede decir que:

Los bits a codificar de la secuencia original, se encuentran en las primeras L tuplas de bits resultado de la codificación, las tuplas están compuestas por el número de ramas del codificador convolucional. En este caso, el tamaño de mensaje que se está codificando es L $= 8$ bits y codificadores convolucionales de 2 y 3 ramas, se tienen secuencias codificadas para $R = 1/2$ de 16 bits y $R = 1/3$ de 24 bits ver la Figura 3.2, donde se encuentra la información útil. Tomando en cuenta lo anterior los 8 bits restantes de los 32 bits para el caso $R = 1/3$ contienen información no útil, es decir, corresponde a los 0 que se agregan para limpiar los registros.




(a) Secuencia codificada, codificador convolucional $R = \frac{1}{2}$.




(b) Secuencia codificada, codificador convolucional $R = \frac{1}{3}$.

Figura 3.2: Posiciones de los 32 bits disponibles que contienen información útil.

En la simulación en la PC de los codificadores convolucionales $R = 1/2$ . Debido a que su secuencia codificada no rebasa los 32 bits, no es necesario validar el tamaño de la palabra. Para los codificadores convolucionales de $R = 1/3$, se tiene que para valores M $> 4$ la secuencia codificada rebasa el rango de 32 bits, la PC permite representar palabras de mayor tamaño, lo que quiere decir que los 0 que son agregados para limpiar los registros están presentes en la secuencia codificada resultante. Para el caso del DSP se agregan dos 0 más para limpiar el registro y los dos bits restantes de los 32 son colocados en 0.

Tomando en cuenta lo anterior y sabiendo que los codificadores RSC son una variante de los códigos convolucionales, lo siguiente es su realización práctica en el DSP. Con base en los resultados obtenidos para los códigos convolucionales, se eligió al RSC de características (2,1,3) por ser el más rápido en su realización y además de garantizar que el rango de la palabra no sea rebasado.

En el proceso de decodificación la representación númerica de las secuencias recibidas no es necesario validarlas, ya que como se mencionó los valores que ruido que se agregan a cada uno de los elementos de la palabra codificada son perfectamente representables.

## 3.2. Técnicas de decodificación suave

Las técnicas soft que se realizaron en el DSP fueron los códigos RSC con el algoritmo MAP en la parte de decodificación y los turbo códigos. El proceso de decodificación de estas técnicas, tiene un mayor grado de dificultad que al de las técnias hard, ya que al procesar los datos sin pasar por el detector, se emplean conceptos de probabilidad que involucran operaciones exponenciales y logarítmicas, el resultado que se obtiene rebasa en ocasiones el rango de la arquitectura de punto fijo del DSP.

A continuación se mencionan los criterios que se tomaron en cuenta en la realización práctica del algoritmo MAP y los turbo códigos en el DSP.

### 3.2.1. RSC con algoritmo MAP

El algoritmo MAP fue desarrollado por los investigadores *Bahl* en el año de 1974 [18], quienes observaron que el proceso de codificación, puede ser visto como proceso de tiempo discreto y que es posible aplicar el concepto de cadenas de Markov [2].

Los códigos RSC con algoritmo MAP, utilizan en la parte de codificación un codificador RSC que siguen el mismo procedimiento que los códigos convolucionales, en la decodificación utiliza un módulo con el algoritmo MAP.

En este trabajo se usa un codificador RSC de tasa $R = 1/2$ como el mostrado en la Figura 3.3 y tamaño de memoria $M = 3$. Se eligió a este codificador por presentar el menor tiempo de procesamiento ver el Apéndice A.1. El tiempo de procesamiento depende del tamaño de la memoria, mientras mayor sea, el tiempo de procesamiento también es mayor. Esto se debe a que se efectúa un mayor número de operaciones. Como se mencionó, el tamaño de la secuencia codificada resultante tendrá un tamaño de 20 bits garantizando así que la palabra del DSP no se rebase.

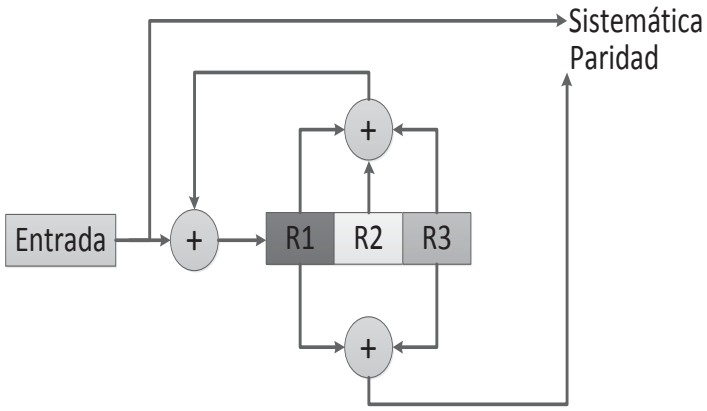



Figura 3.3: Codificador RSC (2,1,3)

En la parte de la decodificación, el algoritmo MAP, requiere de los parámetros denominados como: las métricas ($\gamma$), alfas ($\alpha$) y betas ($\beta$), necesarias para predecir la secuencia que con mayor probabilidad se envió.

- ***Métrica*** $\gamma$. Es una medida de la distancia Euclidianas entre el símbolo recibido y cada uno de los símbolos utilizados por la técnica de modulación empleada. Es una estimación de la probabilidad de que el dato recibido pertenece a una secuencia válida de acuerdo al diagrama de Trellis.
- ***Alfa*** $\alpha$. Coeficiente de recursión hacia adelante, calcula la probabilidad de la relación del estado actual con el estado anterior.
- ***Beta*** $\beta$. Coeficiente de recursión hacia atrás, calcula la probabilidad de la relación del estado actual con estados futuros.

El algoritmo MAP es una técnica suave, pero para calcular los parámetros anteriores, al igual que las técnicas hard se utiliza el diagrama de Trellis. Los valores iniciales que se obtienen para los parámetros $\alpha$, $\beta$, $\gamma$ pueden representarse con los 32 bits de palabra del DSP mediante notación de punto fijo, por lo que no es necesario validar el tamaño de la misma.

### 3.2.2. Turbo códigos

La realización práctica de los turbo códigos en el DSP, es uno de los méritos de este trabajo debido a la dificultad implícita. Aquí es donde se deben de aplicar los conceptos y criterios desarrollados, en las realizaciones prácticas de las técnicas duras, suaves y el algoritmo MAP, tanto para la parte de codificación y decodificación. En la actualidad existen dispositivos como los FPGA de Lattice [19], que ya traen esta técnica implementada y el usuario solo debe de habilitarla. La desventaja que se tiene es de que los parámetros no pueden ser modificados. En este trabajo los parámetros pueden ser modificados de acuerdo a las consideraciones que se dan.

#### Codificación turbo

En la turbo codificación se utilizó una arquitectura PCCC: Paralell Concatened Convolutional Code de tasa $R = 1/3$ de características (3, 1, 3), como el de la Figura 3.4, el cual se compone de dos codificadores RSC de tasa $R = 1/2$, tamaño de memoria M = 3 y vectores generadores $g_1 = 111$ y $g_2 = 101$.

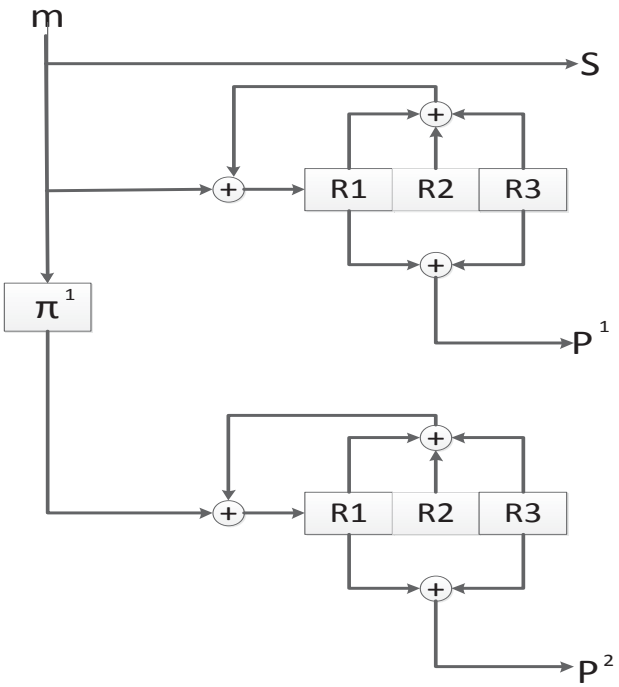



Figura 3.4: Turbo codificador $R = 1/3$

En la figura anterior los codificadores RSC están conectados por medio de un entrelazador

llamado $\pi$, encargado de reordenar la secuencia original, esto para procurar que las entradas a cada uno de los codificadores estén lo menos correlacionadas y con ello obtener un desempeño óptimo. Existen diferentes tipos de entrelazadores, pero en nuestro caso se utiliza un entrelazador renglón columna, que consiste en tomar la secuencia de datos y llenar con ella, renglón por renglón, una matriz bidimensional de cuatro columnas. La secuencia entrelazada se obtiene al leer la matriz columna por columna.

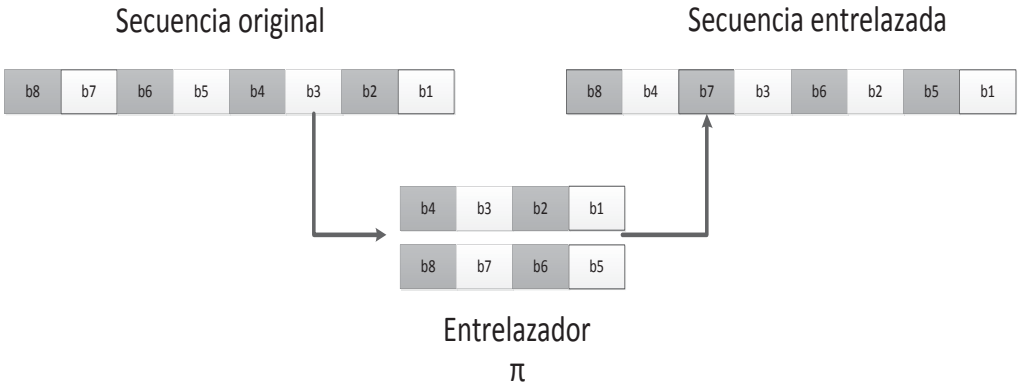



Figura 3.5: Proceso de entrelazado.

El turbo codificador recibe a la entrada secuencias de 128 bytes aleatorios y se codifica cada byte de la secuencia. Como el tamaño de la secuencia es de 8 bits, se obtiene como resultado una secuencia codificada (palabra código) de tamaño de 24 bits, este se obtiene al multiplicar el número de ramas del turbo codificador por el tamanño del byte, con lo que se garantiza que el tamaño de la palabra del *DSP* no se rebase.

Para llevar a cabo lo anterior fue necesario programar los módulos *llena(), entrelaza(), codifica(), XOR() y agregaruido()*.

El módulo ***void llena(int a, int b, int c)*** se encarga de generar la secuencia de 128 bytes aleatorios y los almacena en un arreglo para después ser codificados. La manera en que este módulo genera los bytes aleatorios es la siguiente:

**Algoritmo 3.1** Módulo llena, genera un arreglo de de 128 bytes aleatorios.

**para** $i = 0$ hasta $MAX$ **hacer**
  $a \leftarrow \frac{a \times b}{c}$
  $arreglo[i] \leftarrow a$
**fin para**

El módulo ***int entrelaza(int número)*** tiene como función reordenar el número que recibe y regresar su versión entrelazada; utiliza para ello un entrelazado renglón columna descrito en esta sección ver la Figura 3.5.

**Algoritmo 3.2** Módulo que entrelaza el byte y genera su versión entrelazada.

```
1: uno ← 1, dos ← 16
2: para i = 0 hasta i < 4 hacer
3:    primerbit ← número AND uno
4:    primerbit ← número AND dos
5:    primerbit ← primerbit << i+1
6:    segundobit ← segundobit >> (4-i)
7:    primerbit ← primerbit OR segundobit
8:    entrelazada ← entrelazada OR primerbit
9:    uno ← uno << 1
10:   dos ← dos << 1
11: fin para
12: regresa(entrelazada)
```

Módulo ***void codifica(int índice, int g1, int g2, int memoria)***, se encarga de codificar el *byte* en la posición *índice* del arreglo original, este *byte* se proporciona al módulo ***int entrelaza (número)*** para obtener su versión entrelazada. Los generadores *g1 y g2* corresponden a las ramas del codificador RSC, que se proporcionan al módulo ***int XOR(int g1, int registro, int memoria)*** que es el encargado de efectuar las operaciones en módulo 2 y regresar el resultado.

Como resultado de la turbo codificación, se obtiene la secuencia codificada siguiente:


Figura 3.6: Secuencia de bits resultado de la codificación.

Donde $Si$ es la secuencia original, $P1,i$ y $P2,i$ las paridades de los correspondientes a los codificadores RSC, donde el índice $i = 1, 2, \ldots, 8$ que es tamaño del byte. A continuación se muestra el algoritmo que realiza la turbo codificación.

**Algoritmo 3.3** Codifica uno a uno el byte original y su versión entrelazada.

```
1: numero ← arreglo[índice]
2: numero1 ← entrelaza(número)
3: Los nombres de que al final tienen el '1', indican que se trabaja con el RSC 2
4: para i = 0 hasta i < 8 hacer
5:    retroalimentacion ← XOR(int g1, int registro, int memoria)
6:    retroalimentacion1 ← XOR(int g1, int registro1, int memoria)
7:    primerbit ← número AND 1
8:    primerbit1 ← número1 AND 1
9:    sitemática ← primerbit
10:   sitemática1 ← primerbit1
11:   primerbit ← primerbit1 XOR sistemática
12:   primerbit1 ← primerbit1 XOR sistemática1
13:   primerbit ← primerbit<<2
14:   primerbit1 ← primerbit1<<2
15:   registro ← registro OR primerbit
16:   registro1 ← registro1 OR primerbit1
17:   rama ← XOR(g2, registro, memoria)
18:   rama1 ← XOR(g2, registro1, memoria)
19:   sistemática ← sistemática << (3*i)+2
20:   rama ← rama <<(3*i)+1
21:   rama1 ← rama1 <<(3*i)
22:   sistemática ← sistemática OR rama OR rama1
23:   codificada ← codificada OR sistemática
24:   registro ← registro>>1
25:   registro1 ← registro1>>1
26:   número ← número>> 1
27:   número1 ← número1 >> 1
28: fin para
29: arrcodificada[índice] ← codificada
```

El módulo ***int XOR(int g1, int registro, int memoria)*** efectúa la operación módulo 2 correspondiente a la rama del codificador RSC, para ello es necesario proporcionarle el registro donde están almacenados los bits a codificar. Este módulo lo que hace es sacar del registro y del generador un bit, si el bit del generador es 1 se almacena en una variable auxiliar el bit del registro y se efectúa la XOR con el valor previo, esto se repite hasta que todos los bits del registro hayan sido procesados.

**Algoritmo 3.4** Módulo encargado de la suma XOR.

```
1: para i = 0 hasta i < memoria hacer
2:    aux ← g1 AND 1
3:    si aux = 1 entonces
4:       bit ← registro AND 1
5:       XORS ← XORS XOR bit
6:    fin si
7:    registro ← registro >> 1
8:    g1 ← g1 >> 1
9: fin para
10: regresa (XORS)
```

Una vez que se realizó la turbo codificación, lo siguiente es agregar el ruido AWGN generado por el módulo ***float RN($\mu$, $\sigma^2$)*** y que tiene una distribucción gaussiana Figura 3.7. Los valores que este módulo genera no es necesario validarlos y esto es por que son perfectamente representables por el rango númerico del *DSP*.

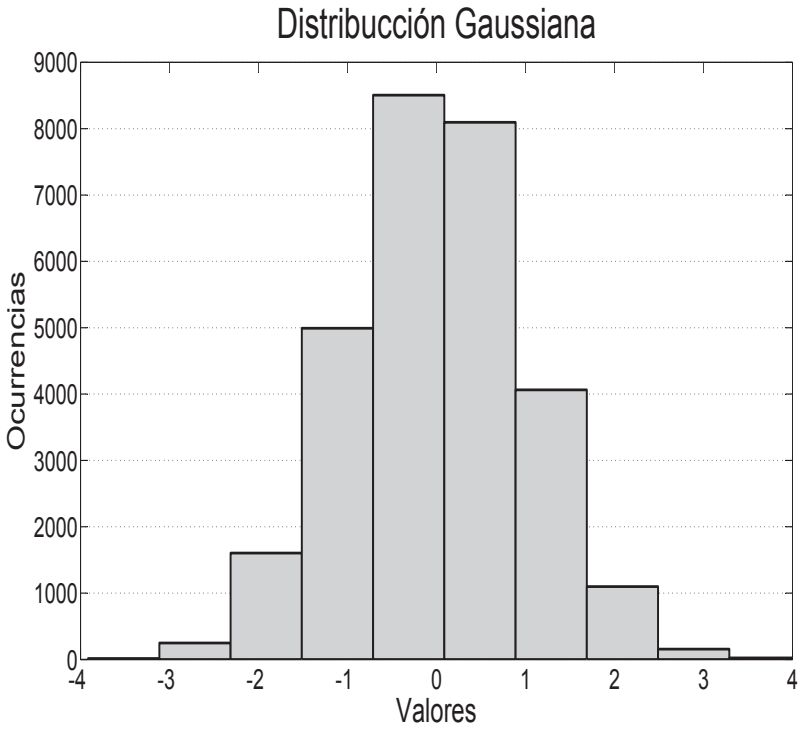



Figura 3.7: Histograma de la función que genera el módulo float RN($\mu$, $\sigma^2$).

Para agregar el ruido, es necesario tomar del arreglo la palabra codificada y enseguida agregar a cada uno de los bits que la conforman. Para lograr esto, el módulo ***void agregaruido(int índice, float gauss)*** recibe como parámetros el índice y la potencia del ruido *gauss*, el algoritmo que realiza esto es el siguiente:

**Algoritmo 3.5** Módulo encargado de agregar el ruido AWGN

```
1: codificada ← arreglocodificada[índice]
2: para i ← 0 hasta i < 24 hacer
3:    bit ← codificada AND 1
4:    si bit = 0 entonces
5:       codificadaconruido[índice][i] ← -1+RN(0,gauss)
6:    si no
7:       codificadaconruido[índice][i] ← 1+RN(0,gauss)
8:    fin si
9:    codificada ← codificada >> 1
10: fin para
```

El módulo anterior, al agregar el ruido tambien efectúa una señalización antipodal, es decir, al 0 le asigna un valor de $-1$ y al 1 el valor de $+1$.

### Decodificación turbo

Para el proceso de turbo decodificación, una vez resuelto el RSC con el algoritmo MAP, lo siguiente fue implementar el proceso iterativo de turbo decodificación usando la siguiente arquitectura:

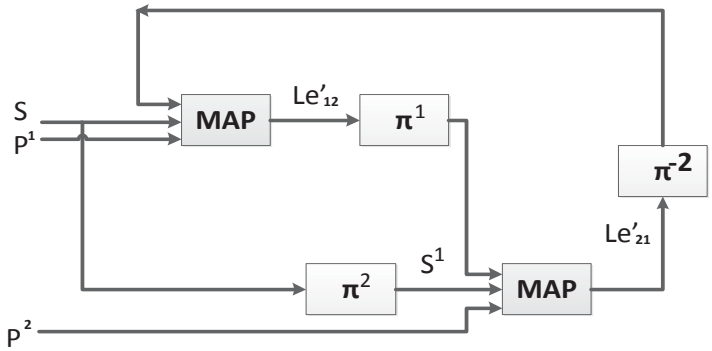



Figura 3.8: Turbo decodificador.

Si el algoritmo MAP se invoca una sola vez no presenta problema con la representación de punto fijo, sin embargo, al realizar la implementación de éste en el proceso iterativo, mediante el Algoritmo 3.6, que se muestra en el Apéndice A.1, los resultados que se obtienen pueden ser no satisfactorios cuando el rango representable se rebasa.

**Algoritmo 3.6** Algoritmo iterativo de turbo decodificación

1: Verosimilitud extrínseca $Le'_{da,ds} = 0$
2: **para** $i = 0$ hasta $k$ **hacer**
3: **Módulo 1 Decodificador MAP 1 calcular:**
4: Probabilidad de $c_t \leftarrow +1$
5: Probabilidad de $c_t \leftarrow -1$
6: Métricas $\gamma$
7: Alfas $\alpha$
8: Betas $\beta$
9: Verosimilitud $Le_{da,ds}$
10: Verosimilitud extrínseca $Le'_{da,ds}$
11: **Módulo 2 Decodificador MAP 2 calcular:**
12: Probabilidad de $c_t \leftarrow +1$
13: Probabilidad de $c_t \leftarrow -1$
14: Métricas $\gamma$
15: Alfas $\alpha$
16: Betas $\beta$
17: Verosimilitud $Le_{da,ds}$
18: **si** $i = k$ **entonces**
19: Hacer detección dura
20: **fin si**
21: Verosimilitud extrínseca $Le'_{da,ds}$
22: **fin para**
23: Termina

Sin embargo, el algoritmo[20] anterior puede no funcionar si no se tomaron en cuenta los factores siguientes:

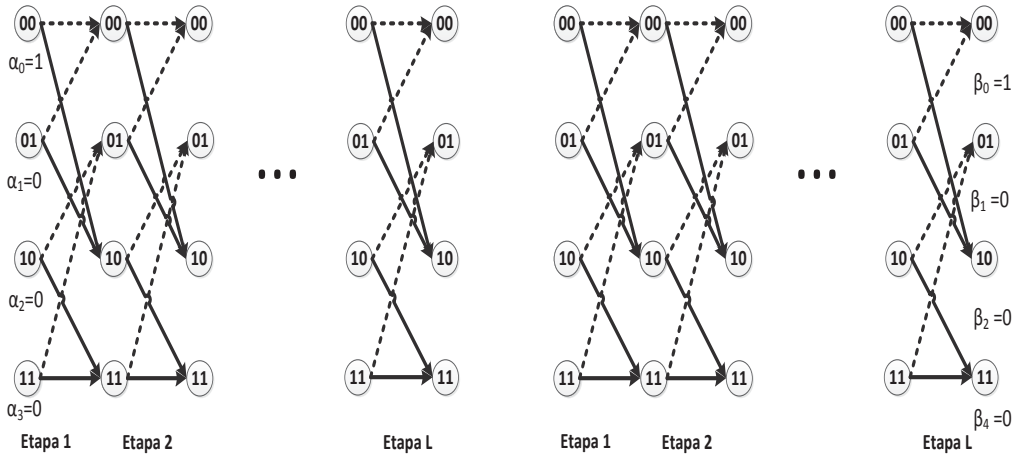
- Salida infinita de la turbo codificación.

  1. En las técnicas convolucionales sin retroalimentación el estado final donde termina la codificación se conoce.
  2. Los códigos RSC al tener una retroalimentación no se sabe el estado final donde termina la codificación, y es por que nunca termina.
  3. Se fuerza a que la turbo codificación termine en el estado S0=00, y agregar a la secuencia de datos originales los valores correspondientes.

- Desborde en el cálculo de las métricas $\gamma$ de las transiciones de estado. Al calcular la exponencial de la distancia Euclidianas, el resultado que se obtiene puede llegar a superar la representación númerica disponible.
- Desborde en la obtención de los coeficientes de recursión hacia adelante $\alpha$. En la primera iteración, los valores no rebasan el rango, es a partir de la segunda iteración que puede rebasar el rango.
- Desborde en la obtención de los coeficientes de recursión hacia atrás $\beta$. De igual forma que en el cálculo de las $\alpha$, a partir de la segunda iteración el rango puede rebasarse.
- Desborde en el cálculo de la verosimilitud. Es necesario validar el logaritmo de un valor negativo. La función logarítmica no está definida para valores negativos.

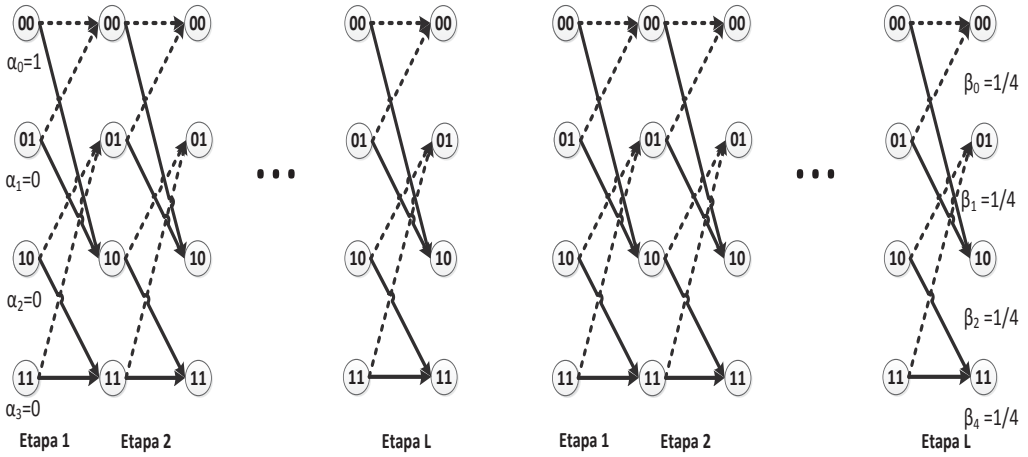
Para solucionar los problemas anteriores, se propusieron dos métodos: el mapeo mediante una función de compresión de las distancias Euclidianas y la Normalización de los coeficientes de recursión.

En el cálculo de los coeficientes de recursión $\alpha$ y $\beta$ utilizando los métodos propuestos, es necesario, inicilizar para cada módulo como se muestra en las Figuras 3.9 y 3.10.




(a) Valores iniciales $\alpha$ (b) Valores iniciales $\beta$

Figura 3.9: Valores iniciales de los coeficientes de recursión $\alpha$ y $\beta$ para el método compresión y normalización, módulo 1.




(a) Valores iniciales $\alpha$ (b) Valores iniciales $\beta$

Figura 3.10: Valores iniciales de los coeficientes de recursión $\alpha$ y $\beta$ para el método compresión y normalización, módulo 2.

### *Mapeo con función de compresión*

Este método trata de evitar que el valor que se obtiene al calcular la exponencial de las distancias Euclidianas en (2.26), rebase el rango de la palabra. La función exponencial Figura 3.11 tiene como característica principal que crece o disminuye de manera rápida. Los valores que toman las métricas $\gamma$, afectan de igual manera a los coeficientes de recursión, es decir, aumentan o disminuyen de manera rápida.

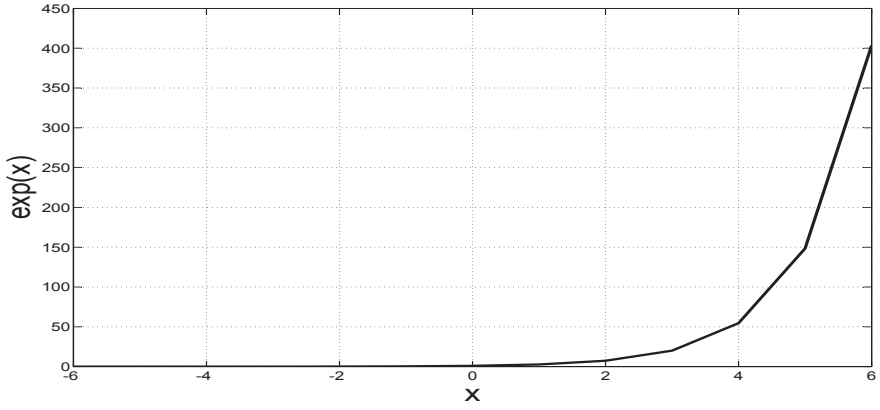



Figura 3.11: Comportamiento de la función *exponencial()*.

Una manera de evitar el comportamiento anterior, es el mapear el valor de las distancias *Euclidianas*, utilizando para ello la función *arctan()*.

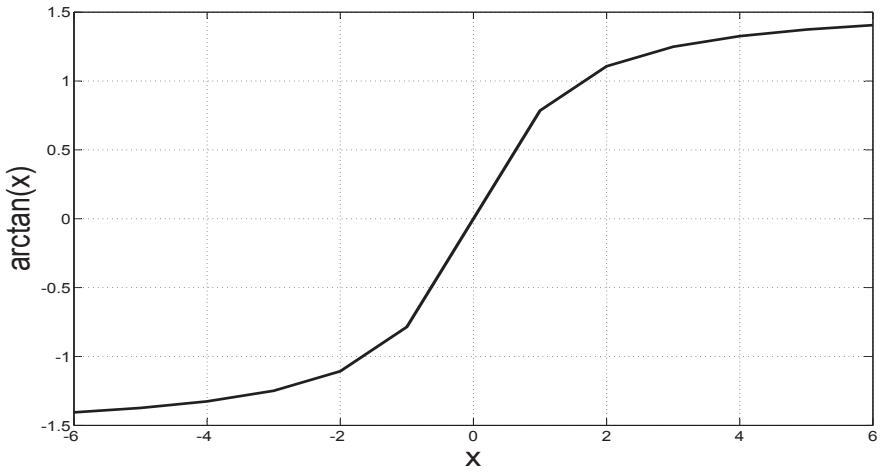



Figura 3.12: Comportamiento de la función *arctan()*.

El efecto que la función *arctan()*, provoca es el siguiente:

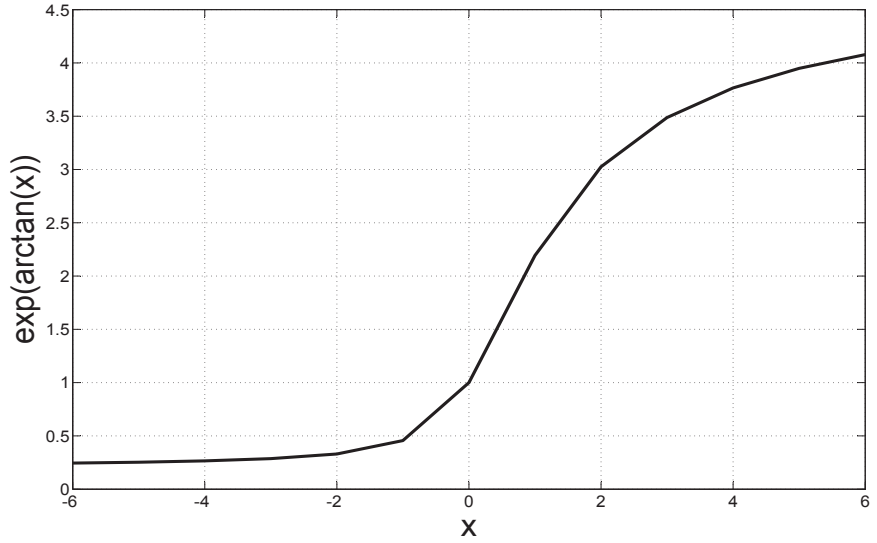



Figura 3.13: Comportamiento de la exponencial de *arctan()*.

Tabla 3.2: Ejemplo de $\gamma$ sin compresión.

| Etapa 0 | Etapa 1 | Etapa 2 | Etapa 3 | Etapa 4 | Etapa 5 | Etapa 6 | Etapa 7 |
|---|---|---|---|---|---|---|---|
| 0.33626 | 4.87762 | 3.77743 | 1.20221 | 0.04591 | 8.71938 | 18.95743 | 67.52497 |
| 0.94915 | 7.36126 | 0.38486 | 5.90096 | 0.41667 | 1.43698 | 2.19346 | 5.59077 |
| 0.33626 | 4.87762 | 3.77743 | 1.20221 | 0.04591 | 8.71938 | 18.95743 | 67.52497 |
| 0.94915 | 7.36126 | 0.38486 | 5.90096 | 0.41667 | 1.43698 | 2.19346 | 5.59077 |
| 2.97389 | 0.20501 | 0.26473 | 0.83179 | 21.77760 | 0.11468 | 0.52749 | 0.01480 |
| 1.05356 | 0.13573 | 2.59828 | 0.16946 | 2.39984 | 0.69590 | 0.45589 | 0.17886 |
| 2.97389 | 0.20501 | 0.26473 | 0.83179 | 21.77760 | 0.11468 | 0.52749 | 0.01480 |
| 1.05356 | 0.13573 | 2.59828 | 0.16946 | 2.39984 | 0.69590 | 0.45589 | 0.17886 |

Tabla 3.3: Ejemplo de $\gamma$ con compresión.

| Etapa 0 | Etapa 1 | Etapa 2 | Etapa 3 | Etapa 4 | Etapa 5 | Etapa 6 | Etapa 7 |
|---|---|---|---|---|---|---|---|
| 0.34272 | 2.35218 | 3.76936 | 0.97984 | 0.11019 | 6.06782 | 8.64267 | 12.07801 |
| 1.0000 | 3.55272 | 0.38404 | 4.80948 | 0.99999 | 1.00000 | 0.99999 | 1.00000 |
| 0.34272 | 2.35218 | 3.76936 | 0.97984 | 0.11019 | 6.06782 | 8.64267 | 12.07801 |
| 1.0000 | 3.55272 | 0.38404 | 4.80948 | 0.99999 | 1.00000 | 0.99999 | 1.00000 |
| 2.82268 | 0.42513 | 0.26529 | 1.02057 | 9.07458 | 0.16480 | 0.11570 | 0.08279 |
| 1.00000 | 0.28147 | 2.60384 | 0.20792 | 0.99999 | 1.00000 | 0.99999 | 1.00000 |
| 2.82268 | 0.42513 | 0.26529 | 1.02057 | 9.07458 | 0.16480 | 0.11570 | 0.08279 |
| 1.00000 | 0.28147 | 2.60384 | 0.20792 | 0.99999 | 1.00000 | 0.99999 | 1.00000 |

Como ejemplo las Tablas 3.2 y 3.3 corresponden a las métricas $\gamma$ de una secuencia codificada de un número aleatorio, para un nivel de *Eb/No= -2 dB*. Como se puede apreciar, en la etapa 7 los valores que se obtienen utilizando la función de compresión son cinco veces menores a los obtenidos de manera tradicional.

Tabla 3.4: Ejemplo de $\alpha$ sin compresión.

| Inicial | Etapa 0 | Etapa 1 | Etapa 2 | Etapa 3 | Etapa 4 | Etapa 5 | Etapa 6 | Etapa 7 |
|---|---|---|---|---|---|---|---|---|
| 1 | 0.336 | 1.640 | 6.302 | 14.739 | 7326.81 | 63890.60 | 1211218.6 | 81788168 |
| 0 | 2.973 | 0.068 | 1.959 | 15.594 | 336.441 | 1243.684 | 9372.464 | 133748.53 |
| 0 | 0 | 0.403 | 8.611 | 336.407 | 46.264 | 316.617 | 1715.075 | 18262.043 |
| 0 | 0 | 21.909 | 56.956 | 21.211 | 57.402 | 523.408 | 2966.605 | 52929.980 |

Tabla 3.5: Ejemplo de $\alpha$ con compresión.

| Inicial | Etapa 0 | Etapa 1 | Etapa 2 | Etapa 3 | Etapa 4 | Etapa 5 | Etapa 6 | Etapa 7 |
|---|---|---|---|---|---|---|---|---|
| 1 | 0.35427 | 0.83331 | 3.35184 | 7.61506 | 1149 | 6977 | 60312 | 728500 |
| 0 | 2.82268 | 0.15061 | 3.21588 | 7.57874 | 83 | 362 | 1771 | 10715 |
| 0 | 0.00000 | 0.79451 | 4.24348 | 126.53194 | 28.4867 | 111.5338 | 473 | 2244 |
| 0 | 0.00000 | 10.02822 | 26.16983 | 20.90803 | 28.4867 | 111.5338 | 473 | 2244 |

Tabla 3.6: Ejemplo de $\beta$ sin compresión.

| Etapa 0 | Etapa 1 | Etapa 2 | Etapa 3 | Etapa 4 | Etapa 5 | Etapa 6 | Etapa 7 | Inicial |
|---|---|---|---|---|---|---|---|---|
| 81788144 | 185416 | 14711 | 977.259 | 567.5324 | 11161.768 | 1280.1101 | 67.5254 | 1 |
| 4433683 | 27481048 | 554389 | 41626 | 354.6098 | 2.52544 | 0.0067 | 0.0000 | 0 |
| 9792162 | 2707119 | 157500 | 898.389 | 243076 | 146.8709 | 3.5619 | 0.0148 | 0 |
| 4325002 | 1666267 | 3727324 | 1434398 | 73.5379 | 5.14108 | 0.03248 | 0.0000 | 0 |

Tabla 3.7: Ejemplo de $\beta$ con compresión.

| Etapa 0 | Etapa 1 | Etapa 2 | Etapa 3 | Etapa 4 | Etapa 5 | Etapa 6 | Etapa 7 | Inicial |
|---|---|---|---|---|---|---|---|---|
| 728500 | 6329 | 722 | 101 | 83 | 633 | 104 | 12 | 1 |
| 63531 | 257393 | 10888 | 1287 | 19 | 1 | 0.080 | 0 | 0 |
| 109017 | 25919 | 4879 | 103 | 5748 | 17 | 1.39 | 0.08 | 0 |
| 63531 | 37611 | 72034 | 27649 | 19 | 1 | 0.08 | 0 | 0 |

Los efectos de la compresión, son más notorios en el cálculo de los coeficientes de recursión hacia adelante $\alpha$ y hacia atrás $\beta$. En las Tablas 3.4 y 3.5 los coeficientes de recursión hacia adelante son alrededor de cinco veces menores a los que se obtienen sin usar el método de compresión, pero aún es notorio el comportamiento exponencial. Esto es porque en el cálculo de las $\alpha$ de la etapa actual, requiere de los valores de las $\alpha$ de la etapa anterior. En el cálculo de las $\beta$, el comportamieto es similar al que se presentó en el cálculo de las $\alpha$ como se observa en las Tablas 3.6 y 3.7.

***Normalización***

Este método consiste en normalizar los coeficientes de recursión hacia adelante $\alpha$ y hacia atrás $\beta$. Las métricas $\gamma$ no sufren cambio alguno por que no se trabaja con ellas y es por que la modificación es sólo sobre los coeficientes de recursión. El proceso de normalización es el siguiente:

- Se suman los coefientes de recursión de la etapa anterior.
- Se calcula cada uno de los coeficientes de recursión de la etapa actual y se divide entre la suma de los coeficientes de la etapa anterior.
- Se suman los coeficientes de la etapa actual.
- Se divide cada uno de los coeficientes de la etapa actual entre su suma.

Como ejemplo en la Tabla 3.8 se muestran las métricas $\gamma$ de una determinada secuencia que se recibió para ser decodificada, y son las mismas para el cálculo tradicional y de normalización.

Tabla 3.8: Ejemplo de $\gamma$.

| Etapa 0 | Etapa 1 | Etapa 2 | Etapa 3 | Etapa 4 | Etapa 5 | Etapa 6 | Etapa 7 |
|---|---|---|---|---|---|---|---|
| 0.31916 | 4.86314 | 10.74232 | 0.96009 | 0.01091 | 12.52962 | 41.58253 | 377.51996 |
| 0.99999 | 7.38905 | 0.13533 | 7.38905 | 1.00000 | 1.00000 | 1.00000 | 1.000000 |
| 0.31916 | 4.86314 | 10.74232 | 0.96009 | 0.10191 | 12.52962 | 41.58253 | 377.51996 |
| 0.99999 | 7.38905 | 0.13533 | 7.38905 | 1.00000 | 1.00000 | 1.00000 | 1.000000 |
| 3.13320 | 0.20562 | 0.09308 | 1.04156 | 52.26288 | 0.07981 | 0.02404 | 0.00264 |
| 0.99999 | 0.13533 | 7.38905 | 0.13533 | 1.00000 | 1.00000 | 1.00000 | 1.00000 |
| 3.13320 | 0.20562 | 0.09308 | 1.04156 | 52.26288 | 0.07981 | 0.02404 | 0.00264 |
| 0.99999 | 0.13533 | 7.38905 | 0.13533 | 1.00000 | 1.00000 | 1.00000 | 1.00000 |

Tabla 3.9: Ejemplo de $\alpha$ sin normalización.

| Inicial | Etapa 0 | Etapa 1 | Etapa 2 | Etapa 3 | Etapa 4 | Etapa 5 | Etapa 6 | Etapa 7 |
|---|---|---|---|---|---|---|---|---|
| 1 | 0.31916 | 1.55213 | 16.71296 | 19.8245 | 66098 | 828198 | 34438604 | 1.300e+10 |
| 0 | 3.13320 | 0.06562 | 4.69959 | 20.8813 | 1059 | 6262 | 67259 | 2885152 |
| 0 | 0.00000 | 0.42403 | 3.61814 | 1264 | 78.7595 | 1138 | 7400 | 74660 |
| 0 | 0.00000 | 23.15144 | 171.07619 | 57.8781 | 78.7595 | 1138 | 7400 | 74660 |

Tabla 3.10: Ejemplo de $\alpha$ con normalización.

| Inicial | Etapa 0 | Etapa 1 | Etapa 2 | Etapa 3 | Etapa 4 | Etapa 5 | Etapa 6 | Etapa 7 |
|---|---|---|---|---|---|---|---|---|
| 1 | 0.09244 | 0.16160 | 0.08522 | 0.01453 | 0.98191 | 0.98979 | 0.99762 | 0.99976 |
| 0 | 0.09075 | 0.00260 | 0.02396 | 0.01531 | 0.01574 | 0.00748 | 0.00194 | 0.00022 |
| 0 | 0.00000 | 0.01683 | 0.01844 | 0.92769 | 0.00116 | 0.00136 | 0.00021 | 0.000005 |
| 0 | 0.00000 | 0.91895 | 0.87236 | 0.04245 | 0.00116 | 0.00136 | 0.00021 | 0.000005 |

Tabla 3.11: Ejemplo de $\beta$ sin normalización.

| Etapa 0 | Etapa 1 | Etapa 2 | Etapa 3 | Etapa 4 | Etapa 5 | Etapa 6 | Etapa 7 | Inicial |
|---|---|---|---|---|---|---|---|---|
| 1.30e+10 | 3037635 | 188206 | 5383 | 4283 | 196693 | 15698 | 377 | 1 |
| 2.37e+08 | 4.14e+09 | 10321339 | 1400537 | 1262 | 9 | 0.00026 | 0 | 0 |
| 1.33e+09 | 50232900 | 15045527 | 5625 | 10279746 | 1252 | 9 | 0.002 | 0 |
| 2.27e+08 | 1.87e+08 | 5.61e+08 | 75957784 | 1262 | 9 | 0.00026 | 0 | 0 |

Tabla 3.12: Ejemplo de $\beta$ con normalización.

| Etapa 0 | Etapa 1 | Etapa 2 | Etapa 3 | Etapa 4 | Etapa 5 | Etapa 6 | Etapa 7 | Inicial |
|---|---|---|---|---|---|---|---|---|
| 0.87788 | 0.00069 | 0.00032 | 0.000006 | 0.00041 | 0.99357 | 0.99942 | 0.99999 | 1 |
| 0.01602 | 0.94523 | 0.01758 | 0.01810 | 0.00012 | 0.0000045 | 0.0000001 | 0.00000 | 0 |
| 0.09006 | 0.01144 | 0.02563 | 0.000007 | 0.99934 | 0.00632 | 0.00057 | 0.0000007 | 0 |
| 0.01602 | 0.04263 | 0.95645 | 0.98175 | 0.00012 | 0.0000045 | 0.0000001 | 0 | 0 |

En las tablas anteriores, donde se muestran los resultados de los coeficientes de recursión hacia adelante y hacia atrás, se puede observar que al no aplicar la normalización se alcanzan valores del orden de $1 \times 10^{10}$. Si la longitud de la secuencia a codificar es mayor que la que se toma como referencia en este trabajo, es decir, $L>8$ los valores comienzan o crecer de manera rápida llegando a ser del orden de $1 \times 10^{40}$ y $1 \times 10^{-40}$ y ya no pueden ser representados por la notación de punto fijo.

Para lograr la turbo decodificación y la implementación de los métodos propuestos fue necesario programar los módulos ***void métricas()***, ***void alfas()***, ***void betas*** y ***void verosimilitud***.

El módulo ***void métricas(int indice, float sigma, int bandera)***, se encarga de calcular las métricas $\gamma$ de los módulos 1 y 2. Para ello utiliza un diagrama de Trellis de 8 etapas, que corresponde a cada uno de los bits del byte enviado, se le debe proporciona el índice del elemento del arreglo con el que se va a trabajar. El parámetro *bandera* indica el número del

modulo y las secuencias de datos con las que se trabajará; es decir, si la bandera es 1 los datos con los que se trabajará serán la información sistemática sin entrelazar y la paridad 1, si el valor de la bandera es 2 los datos corresponden a la información sistemática entrelazada y la paridad 2, resumiendo lo anterior se tiene:

**Algoritmo 3.7** Algoritmo que calcula las métricas $\gamma$

```
1: //El índice i indica el número de etapa, es decir, la columna de la matriz métrica.
2: para i = 1 a 8 hacer
3:    si bandera = 1 entonces
4:       probabilidad ← extrinsecamodulo2desen[indice][i]
5:       probabilidad 1 ← probabilidad1
6:       probabilidad 0 ← probabilidad0
7:       sistemática ← codificadaconruido[indice][2+(3*i)] //secuencia 2,5,8,11,...,23
8:       paridad ← codificadaconruido[indice][1+(3*i)] //1,4,7,10,...,22
9:    si no
10:      probabilidad ← extrinsecamodulo1entre[indice][i]
11:      probabilidad 1 ← probabilidad1
12:      probabilidad 0 ← probabilidad0
13:      sistemática ← codificadaconruidoentre[indice][i]
14:      paridad ← codificadaconruido[indice][3*i] //0,3,6,9,12,...,21
15:   fin si
16:   //El índice j indica la rama, es decir, el renglón de la matriz métrica de la etapa i.
17:   //Si se trabaja con el método de compresión, es necesario, efectuar en esta parte el
      mapeo de las distancia Euclidianas empleando la función trigonométrica atan().
18:   para j = 0 a 7  hacer
19:      si j <4 entonces
20:         γ ← γ
21:         metricas[j][i] ← γ
22:      si no
23:         γ ← γ
24:         metricas[j][i] ← γ
25:      fin si
26:   fin para
27: fin para
```

En el algoritmo anterior, el índice i indica el número de etapa y j el número de ramas de cada etapa, lo que da como resultado la matriz métrica de tamaño $i \times j$.

Uno de los métodos propuestos es la compresión de las distancias Euclidianas. La compresión se efectua al momento de calcular la $\gamma$, esto se ve más a detalle en el código que se encuentra en el Apéndice A.1.

El siguiente método propuesto en la realización práctica es la *normalización* de los coeficientes de recursión $\alpha$ y $\beta$. Los algoritmos que se utilizan para encontrar estos coeficientes siguen la misma metodología, es decir, tienen en común los mismos ciclos *for*, por lo que la explicación es la misma.

El módulo ***void alfas( )***, se encarga de calcular los coeficientes de recursión hacia adelante $\alpha$ para el módulo 1 y 2. Trabaja con las alfas de etapa 0 previamente inicializadas y la matriz que contiene las métricas calculadas con el módulo ***void métricas(int indice, float sigma, int bandera)***. La manera en que se calculan los coeficientes $\alpha$ es la siguiente:

**Algoritmo 3.8** Algoritmo que calcula los coeficientes $\alpha$

```
1: alfas[0][0]=1
2: //El índice i recorre la matriz de las alfas por columnas, es decir, la etapa.
3: para i = 1 hasta 8 hacer
4:    //Suma de las alfas de la etapa anterior.
5:    //El índice j recorre la matriz las transiciones de los estados.
6:    para j = 0 hasta 3 hacer
7:       //El índice k recorre la matriz de las métricas.
8:       para k = 0 hasta 7 hacer
9:          anterior ← estados[k][0]
10:         siguiente ← estados[k][1]
11:         si siguiente = j entonces
12:            α ← calculando α ecuación (2.29) + α
13:         fin si
14:      fin para
15:      //Si se trabaja con normalización se divide a α entre la suma de la etapa anterior.
16:      alfas[i][j] ← α;
17:   fin para
18:   //Si se trabaja con normalización se suman los coeficientes α de la etapa actual y se
      divide a cada uno entre su suma.
19: fin para
```

En la obtención de los coeficentes de recursión se obtienen dos matrices de tamaño $8 \times 4$ una para las $\alpha$ y otra para las $\beta$. Lo que se hace en la normalización es sumar los coeficientes de la etapa anterior comenzando en la etapa 0 para el caso de las $\alpha$ y para las $\beta$ en la etapa 8. La suma de los coeficientes se hace antes de entrar al ciclo *for* con el índice $j$, una vez que se encontró el coeficiente de recursión de la etapa actual, se divide entre la suma los coeficientes de la etapa anterior, ya que se encontraron todos los coeficientes, lo que se hace enseguida es sumarlos y dividir a cada uno de éstos por el resultado de su suma, esto se hace después de terminar el ciclo *for* con el indice $j$.

El módulo ***void betas(int bandera)*** calcula los coeficientes de recursión $\beta$ y sigue la misma metodología que para los coeficientes $\alpha$, recibe el parámetro bandera el cual indica con que valores se deben de inicializar las $\beta$ de la última etapa. Si el valor de la bandera es 1 quiere decir que se está trabajando con el módulo 1, si el valor de la bandera es distinto de 1 el módulo con el que se está trabajando es el 2.

**Algoritmo 3.9** Algoritmo que calcula los coeficientes $\beta$

```
1: si bandera =1 entonces
2:    betas[8][0]=1
3: si no
4:    para i = 0 hasta 3 hacer
5:       betas[8][i]=1/4
6:    fin para
7: fin si
8: //El índice i recorre la matriz de las betas por columnas, es decir, la etapa.
9: para i = 1 hasta 8 hacer
10:   //Suma de las betas de la etapa anterior.
11:   //El índice j recorre la matriz las transiciones de los estados.
12:   para j = 0 hasta 3 hacer
13:      //El índice k recorre la matriz de las métricas.
14:      para k = 0 hasta 7 hacer
15:         anterior ← estados[k][0]
16:         siguiente ← estados[k][1]
17:         si anterior = j entonces
18:            β ← calculando β ecuación (2.6.2) + β
19:         fin si
20:      fin para
21:      //Si se trabaja con normalización se divide a β entre la suma de la etapa anterior.
22:      betas[i][j] ← β;
23:   fin para
24:   //Si se trabaja con normalización se suman los coeficientes β de la etapa actual y se
      divide a cada uno entre su suma.
25: fin para
```

Una vez calculado las $\gamma$, $\alpha$ y $\beta$, lo siguiente es obtener el valor de la *logverosimilitud*, esto se hace con el módulo ***void verosimilitud(int índice, float sigma, int bandera)***, se le debe proporcionar el índice del arreglo que contiene la secuencia recibida, la potencia del ruido con la cual se calcula la verosimilitud debida al canal y la bandera la que indica con que secuencia de datos se trabajará.

**Algoritmo 3.10** Algoritmo que calcula la verosimilitud $L$

```
1: para i = 0 hasta 8 hacer
2:   para k = 0 hasta 8 hacer
3:     si k<4 entonces
4:       numerador ← numerador + numerador con ecuación 2.31
5:     si no
6:       denominador ← denominador + denominador con ecuación 2.31
7:     fin si
8:   fin para
9:   Lc = Verosimilitud debida al canal.
10:  si bandera = 1 entonces
11:    verosimilitudmódulo1[índice][i] ← verosimilitud con ecuación 2.32 - Lc - verosimilitudmódulo1
12:  si no
13:    //Resultado de la decodificación.
14:    si verosimilitud<0 entonces
15:      decodificada[indice][i] ← 0
16:    si no
17:      decodificada[indice][i] ← 1
18:    fin si
19:    verosimilitudmódulo2[índice][i] ← verosimilitud con ecuación 2.32 - Lc - verosimilitudmódulo2
20:  fin si
21: fin para
```

En el algoritmo anterior se efectúa la decodificación, ya que es aquí donde se estima el valor del bit que se envio, esto se hace al comparar el valor de la verosimilitud con el valor del umbral que para nuestro caso es 0 ya que como se mencionó, se está trabajando con una señalización antipodal. Los algoritmos anteriores son colocados en forma secuencial dentro de un ciclo *for*, para implementar el proceso iterativo vea el Algoritmo 3.6.

En general, al utilizar estos métodos, se obtienen buenos resultados como se puede apreciar en el siguiente capítulo.

# Capítulo 4
# Resultados

En este capítulo, se muestran los resultados de las realizaciones prácticas con el DSP de las diferentes técnicas de codificación de canal estudiadas. Estos resultados son comparados con los que se obtienen mediante la simulación en la PC, con la finalidad de observar si el comportamiento de los turbo códigos es similar o si es afectado por el tipo de representación númerica en el *DSP*. Para la obtención de estos resultados se utilizaron los siguientes dispositivos:

Tabla 4.1: Características del TMS320C6416 y la PC.

| | DSP | PC |
|---|---|---|
| Reloj | 720 Mhz | 2.30 Ghz |
| Representación aritmética | Fija | Flotante |
| Memoria RAM | 1056 KB | 4 GB |

Y se siguió la siguiente metodología:

- Simulación Monte Carlo para:
  1. Códigos convolucionales y RSC con decodificación dura de -2 a 9 dB.
  2. Código convolucional RSC con decodificación suave con algoritmo MAP de -2 a 9 dB.
  3. Turbo código de -2 a 9 dB.
- Datos de entrada 61440 bits, 60 arreglos de 128 bytes.
- Señalización antipodal.
- Canal AWGN.

## 4.1. Técnicas con decodificación dura

En esta sección sólo se muestran dos gráficas de desempeño que corresponden a los codificadores convolucionales de tasa $R = 1/2$ y $R = 1/3$ de tamaño de memoria M = 3. Se omiten los demás codificadores propuestos, porque son muy similares sus gráficas y no son el objetivo principal de este trabajo, en el Apéndice B se encuentran más detalles de la simulación y la realización práctica de estas técnicas.

### 4.1.1. Códigos convolucionales

A continuación, se muestran las gráficas de desempeño obtenidas para los codificadores convolucionales del tipo (2,1,3) y (3,1,3).

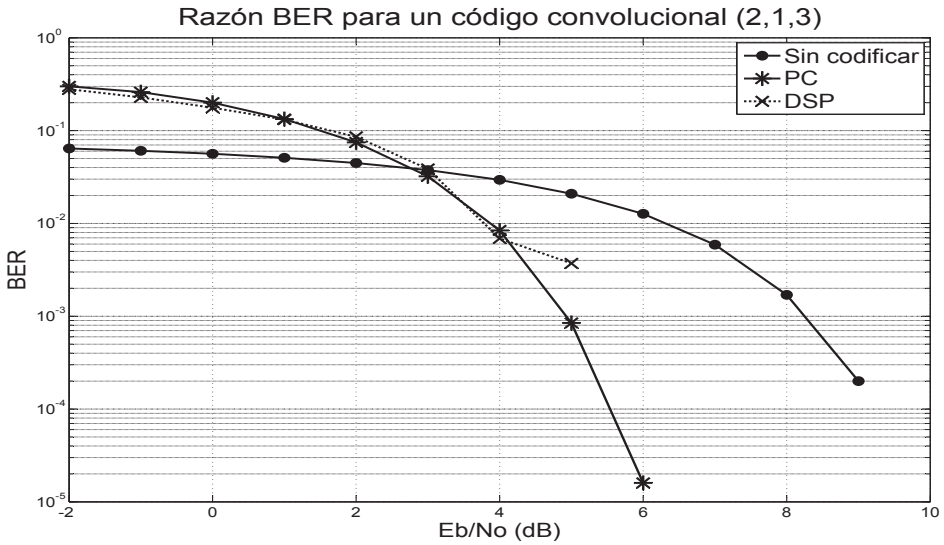



Figura 4.1: Gráfica de desempeño del codificador convolucional (2,1,3).

En la figura anterior, se observa que efectivamente las gráficas de desempeño siguen el comportamiento descrito teóricamente para esta técnica. El desempeño de la versión en el DSP es muy similar a la versión en la PC, incluso se podría decir igual, si no fuera por el cambio que se presentan en Eb/No= 4 dB, en esta misma se aprecia que la simulación resulta ser mejor para valores de Eb/No <5 dB, por encima de este sucede lo contrario, el desempeño del DSP es mejor y es por que alcanza una BER=0.

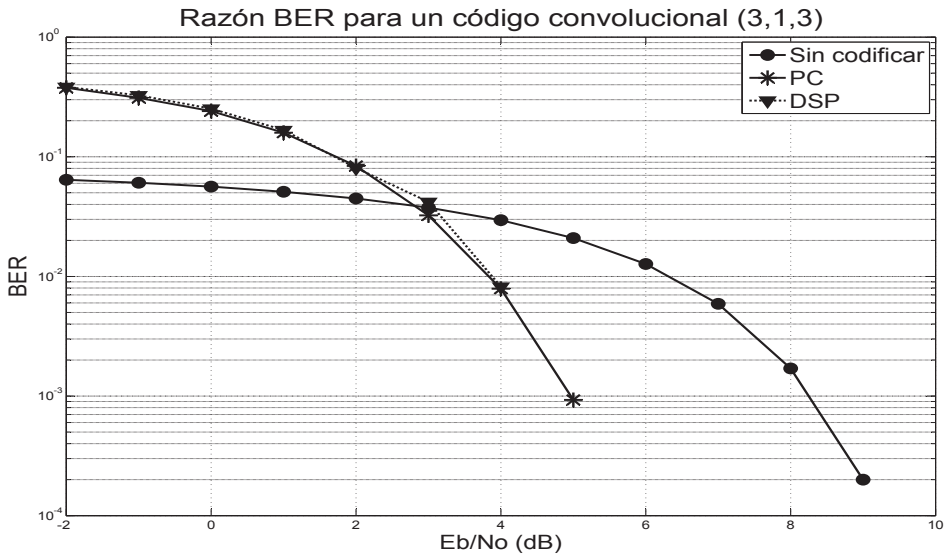



Figura 4.2: Gráfica de desempeño del codificador convolucional (3,1,3).

En la Figura 4.2 se muestra que el desempeño que se obtiene para el DSP y PC son muy similares hasta Eb/No = 4 dB. El DSP es mejor ya que alcanza una BER = 0 antes que la simulación en la PC y es que para este caso se siguen teniendo errores hasta Eb/No = 5 dB.

### 4.1.2. RSC

A continuación, se muestra el desempeño que se obtuvo para el *RSC* (2,1,3).

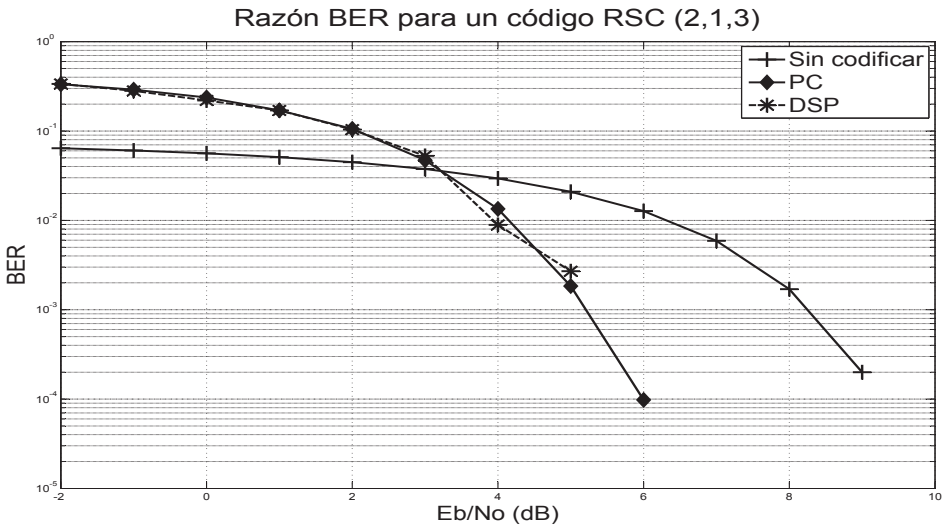



Figura 4.3: Gráfica de desempeño codificador *RSC* (2,1,3).

Al igual que con los códigos convolucionales, la gráfica de desempeño obtenida sigue el mismo comportamiento al descrito teóricamente. El resultado de la simulacion y el desempeño del DSP son muy similares hasta Eb/No= 4 dB, a partir de donde el DSP obtiene una BER = 0.

Las técnicas RSC y códigos convolucionales, a pesar de ser técnicas que generan de manera diferente la secuencia codificada, la esencia es la misma y los desempeños obtenidos en el *DSP* son muy similares. Tienen en común que para un nivel mayor *Eb/No= 3 dB*, es conveniente codificar la información, ya que para niveles por debajo de éste se obtiene un mayor número de errores que un enlace no codificado, lo que indica que es más conveniente enviar al información sin codificar.

## 4.2. Técnicas de decodificación suave

Como se ha venido mencionando, este tipo de técnicas no necesita de un detector, ya que procesan de manera suave la secuencia recibida, lo que hace que la predicción sea más precisa.

### 4.2.1. RSC con el algoritmo MAP

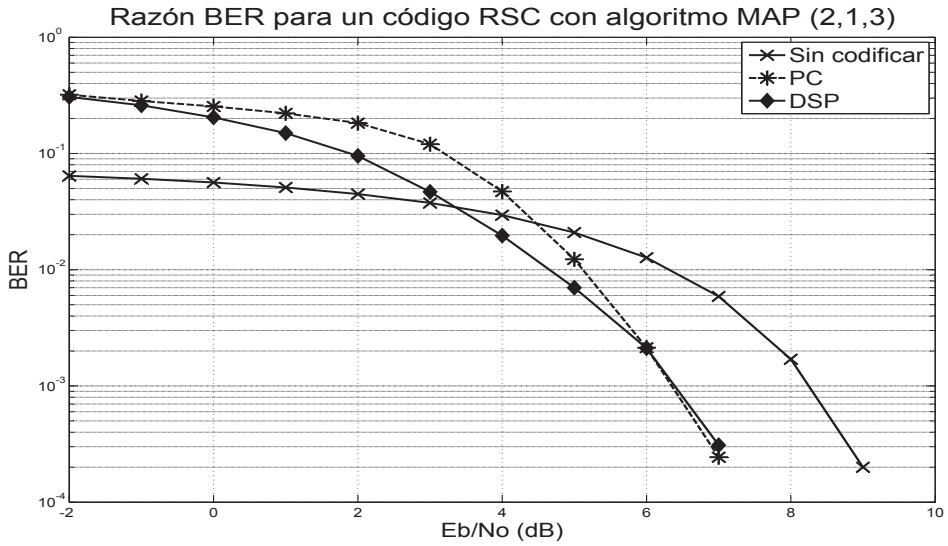



Figura 4.4: Gráfica de la BER del RSC con algoritmo MAP (2,1,3).

La gráfica de la Figura 4.4 muestra que el desempeño de la versión en el DSP está por debajo del desempeño que se obtiene mediante la simulación con la PC, aunque coinciden a partir de un nivel de Eb/No = 6 dB, hasta llegar a un nivel donde la BER se vuelve cero. La diferencia entre los resultados se debe, en parte, a la representación de los datos en DSP y a la manera en que se realizan la operaciones en cada caso.

Es claro que sólo conviene usar esta técnica para niveles mayores Eb/No = 3.5 dB. Por intuición, se esperaría que el desempeño fuera mejor al obtenido con las técnicas hard, y es porque se hace uso de probabilidades para predecir con más exactitud lo que fue enviado. Los resultados demuestran lo contrario, ya que el nivel de Eb/No que indica a partir de donde es conveniente usar la codificación, es muy parecido al encontrado en la técnica códigos convolucionales de tasa $R = 1/2$ y $R = 1/3$ que es de Eb/No = 3 dB.

### 4.2.2. Turbo códigos

La realización práctica de esta técnica es la culminación y la parte más importante de este trabajo. Para llegar a esto, fue necesario completar la realización práctica de cada una de las técnicas previas en el *DSP*. Los turbo códigos aplican todas las técnicas anteriores y tienen como característica principal, que aprovechan la estimación que hace el decodificador en cada intento para el siguiente (verosimilitud extrínseca). Se asume el empleo de varios decodificadores concatenados, los que repiten el mismo proceso y obtiene una mejor estimación de la verosimilitud en cada intento, esto se repite hasta que se cumple un número dado de iteraciones, el que se determina midiendo a partir de qué iteración ya no hay mejora sustancial en la estimación.

La problemática principal para hacer esta realización fue también la representación numérica de los datos. Para solucionarlo, se usaron los métodos propuestos de compresión y la normalización. A continuación se muestran las gráficas de desempeño correspondientes a cada uno de estos métodos.

**Resultados con función de compresión.**

El desempeño de la realización práctica en el DSP y simulación en la PC, del turbo código PCCC (3,1,3), utilizando el método de compresión se muestra en la gráfica de la Figura 4.5:

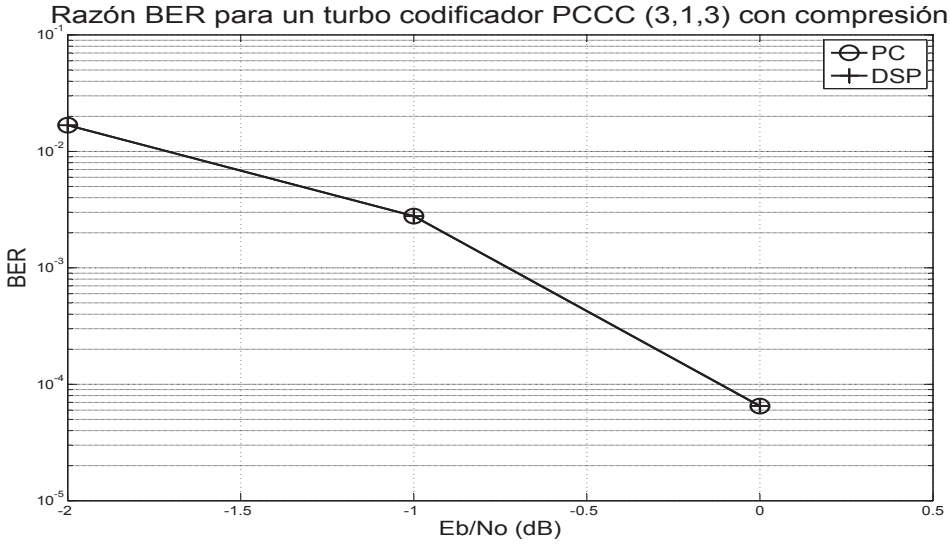



Figura 4.5: BER del turbo codificador *PCCC* (3,1,3) con compresión.

Los resultados obtenidos al utilizar el método de compresión en la realización práctica de los turbo códigos, tanto en la PC como en el DSP, son muy similares, ya que en la gráfica de la Figura 4.5 no se aprecia cuál gráfica corresponde al DSP y cuál a la simulación en la PC.

### Resultados con normalización

El desempeño de la realización práctica en el DSP y simulación en la PC del turbo código PCCC (3,1,3), utilizando el método de normalización se muestra en la siguiente gráfica de la Figura 4.6:

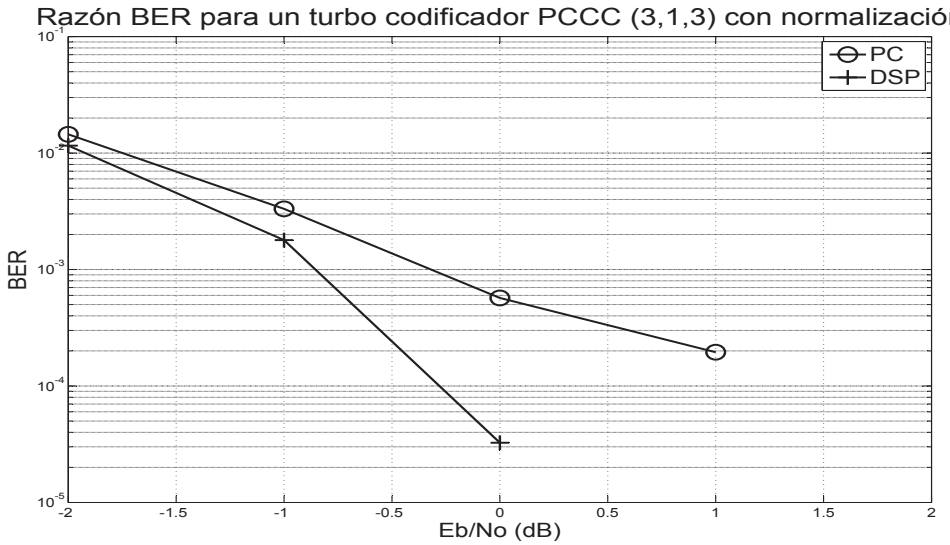



Figura 4.6: Gráfica de la BER del turbo codificador *PCCC* (3,1,3) método normalización.

En la gráfica anterior, como se puede apreciar, el desempeño que se obtiene con la PC no es similar al que se obtiene con el DSP pero sigue la misma tendencia. La diferencia que se presenta hace que la BER del DSP esté por debajo al de la PC, lo que quiere decir que es mejor, además de que alcanza un BER = 0 para un nivel de Eb/No = 0 dB y no en Eb/No = 1 dB como sucede en la PC.

Tabla 4.2: Ganancia utilizando el método de normalización.

| Eb/No (dB) | Razón BER PC | Razón BER DSP | Ganancia (dB) |
|---|---|---|---|
| -2, | 0.0116 | 0.0145 | 0.9692 |
| -1 | 0.0018 | 0.0033 | 2.6820 |
| 0 | 3.2552e-05 | 5.7000e-04 | 12.4330 |

En la Tabla 4.2, se muestra que se obtiene una ganancia de *12 dB* para un nivel de *Eb/No = 0 dB* en el *DSP*.

### Comparación de las técnicas de compresión y normalización

A continuación se muestra la comparación del desempeño del turbo codificador (3,1,3) en el *DSP*, utilizando los métodos propuestos para evitar el desborde del rango de la palabra.

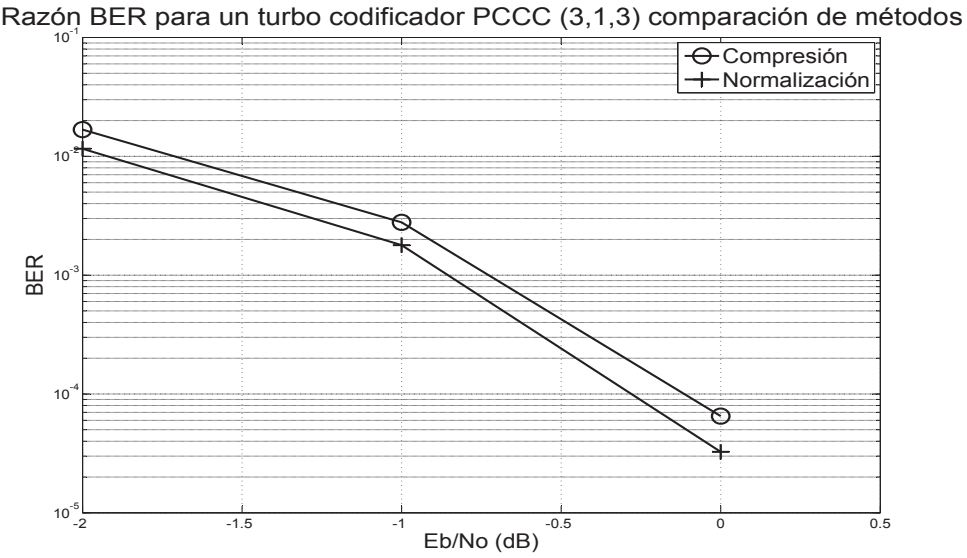



Figura 4.7: Gráfica comparativa del BER de los métodos utilizados.

El desempeño que se obtiene usando el método de normalización es mejor al que se obtiene usando el método compresión, ya que se tiene una ganancia de hasta de 3 dB utilizando el método de normalización para un nivel de Eb/No = 0 dB.

Tabla 4.3: Ganancia utilizando los métodos compresión y normalización.

| Eb/No (dB) | Razón BER Compresión DSP | Razón BER Normalización DSP | Ganancia (dB) |
|---|---|---|---|
| -2 | 0.0168 | 0.0116 | 1.6078 |
| -1 | 0.0028 | 0.0018 | 1.9160 |
| 0 | 6.5104e-05 | 3.2552e-05 | 3.0103 |

Las gráficas donde se comparan el desempeño tienen en común que a partir de un nivel por encima de *Eb/No=0* corrigen por completo los errores. La diferencia que se presenta se debe a que el método de mapeo con compresión de los valores, asigna un valor menor de proporción no lineal, a diferencia de la normalización donde el valor asignado es un valor entre [-1,1], en una proporción lineal.

Las gráficas mostradas en este capítulo, demuestran que efectivamente es posible hacer la realización práctica de los turbo códigos en el DSP TMS230C6416 de arquitectura de punto fijo, además de llegar a obtener el desempeño que los caracteriza.

# Capítulo 5
# Conclusiones y trabajo a futuro

## 5.1. Conclusiones

En este trabajo se presentó la realización práctica en el DSP de las técnicas de codificación de canal, códigos convolucionales, códigos RSC, códigos RSC con algoritmo MAP y turbo códigos.

Para realizar la programación de los algoritmos de codificación de canal en el DSP, se utilizaron las herramientas que acompañan al kit de desarrollo *DSK6416* ofrecido por *Texas Instruments*, las que permiten además de la programación, ejecutar y depurar los programas en tiempo real. La realización práctica de los códigos convolucionales fue la primera aproximación a los turbo códigos, se programó en el DSP los códigos convolucionales de tasa $R = \frac{1}{2}$ y $R = \frac{1}{3}$. La dificultad para lograr su realización en el DSP fue, principalmente, a tratar de determinar la ruta que siguió la secuencia original, en el diagrama de Trellis, ya que las probabilidades crecen de manera exponencial. Los desempeños que se obtienen son los esperados, ya que siguen el mismo comportamiento que se describe en la teoría [13]. En la prueba de estas técnicas se tomó el tiempo de procesamiento, con la finalidad de escoger aquel que tomara el menor tiempo. Se observó que los de menor tamaño de memoria son más rápidos.

La realización práctica en el DSP del código RSC, como variante de los códigos convolucionales, fue más rápida ya que se partió del codificador convolucional previamente desarrollado. El desempeño que se obtuvo fue similar al de los códigos convolucionales.

La realización práctica de los códigos RSC con algoritmo MAP en la parte de la decodificación, procesa la información tal y como se recibe del canal, con esta técnica se esperaba un desempeño mejor, porque el algoritmo MAP trabaja con probabilidades en un continuo y no con decisiones a priori como se hace en las técnicas hard. Sin embargo, los resultados muestran que el desempeño es muy similar al de los códigos convolucionales, lo que muestran que a pesar de no contar con un detector el algoritmo MAP no se obtiene ganancia significativa en los convolucionales independientes.

El próposito general de este trabajo y el más importante fue la realización práctica en el DSP del turbo código de arquitectura PCCC. Esta técnica resultó ser la más compleja de realizar, por que en su proceso iterativo de decodificación el rango de la palabra de representación numérica se puede rebasar. El algoritmo MAP en los turbo códigos, a diferencia de los códigos RSC con algoritmo MAP, además de hacer la función de un detector en cada iteración, estima la probabilidad de los datos que originalmente fueron enviados.

En general el desempeño obtenido, para la realización práctica en el DSP de la técnica turbo códigos, empleando los métodos de normalización y compresión para evitar desbordamiento de la palabra de representación numérica en el DSP, fueron muy similares, además de obtener a partir de una relación Eb/No=0 una BER=0, acercándose al límite de Shannon [5] que es de Eb/No=-1.5917dB. Para alcanzar este nivel de la BER, sólo fue necesario hacer cuatro iteraciones. Se observó que, más alla de cuatro iteraciones las verosimilitudes extrínsecas no cambian, por lo que todos los parámetros del algoritmo MAP permanecen constantes y los resultados obtenidos son los mismos.

En realización de la turbo codificación, uno de los aspectos importantes a tomar en cuenta, fue el tamaño del mensaje a codificar, ya que entre mayor sea el mensaje los parámetros $\gamma$, $\alpha$, $\beta$ del algoritmo MAP, crecen o disminuyen conforme se va recorriendo el diagrama de Trellis, hasta llegar a un punto donde ya no pueden ser representados por rebasar el rango de punto fijo, además de que, siendo un algoritmo iterativo, se puede desbordar la pila y el programa en ejecución, entre en un ciclo infinito.

La realización práctica de las diferentes técnicas de codificación y decodificación de canal, es posible hacerse sin rebasar el rango de la representación númerica del DSP. Para lograr esto, es necesario analizar el valor máximo que alcanzan las variables, así como del tamaño de las palabras con que se trabaja, además de proponer métodos que resuelvan los problemas que se presenten en la representación de los datos y comprobar que, al utilizarlos, el desempeño no se vea afectado como se hizo en este trabajo.

## 5.2. Trabajo a futuro

El próposito de este trabajo era mostrar la realización práctica de la técnica de codificación de canal llamada turbo código, lo que se logró satisfatoriamente. Sin embargo, por el tiempo de duración de la maestría no se pudo hacer la realización práctica de más configuraciones. Lo que se propone como continuación de este trabajo es lo siguiente:

- Hacer la realización práctica de los códigos *RSC* en el *DSP* para distintos tamaños de memoria y distintas tasas de codificación.
- Hacer la realización práctica de los códigos *RSC* ahora con el *algoritmo MAP* en la parte de la decodificación, para distintos tamaños de memoria y distintas tasas de codificación.
- Hacer la realización práctica de los *Turbo códigos* en el *DSP* para distintos tamaños de memoria y distintas tasas de codificación y distintas arquitecturas.
- Utilizar diferentes tipos de entrelazadores $\pi$ en la turbo codificación.

# Apéndice A

# Programa en lenguaje C Turbo código PCCC (3,1,3)

## A.1. Módulos encargados de la codificación

A continuación, se muestra el diagrama de flujo de la turbo decodificación, además del código en C de los módulos que lo conforman.

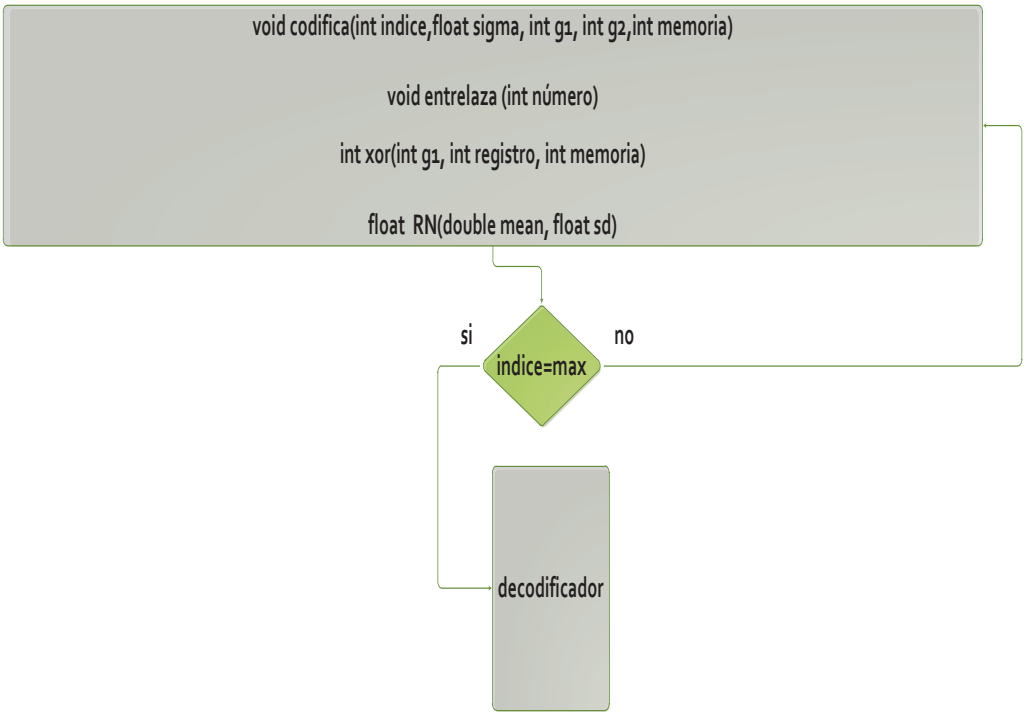



Figura A.1: Diagrama de flujo del turbo codificador PCCC (3,1,3).

```
//Se encarga de hacer la operación módulo 2 y regresa el resultado
//para ello necesita del polinomio generador el registro de
   corrimiento y el tamano de
//de la memoria
int xor(int g1, int registro,int memoria)
{
        int i=0, unbit=0, result=0,codificada=0;
        for(i=1; i<=memoria; i++)
        {
                unbit=g1&1;
                if(unbit==1)
                {
                        result=registro&1;
                        codificada=codificada^result;
                }
                if(i<memoria)
                {
                        g1=g1>>1;
                        registro=registro>>1;
                }
        }
        return(codificada);
}
```

```
//llena el arreglo con una secuencia de números aleatorios
//para ello se le proporciona dos números cualquiera a y b,
//la función rand() estándar de las librerías en c no funciona en el
   DSP
int llenar(int a,int b)
{
        int c=89, i=0, d=0;
        d=a;
        for(i=0; i<MAX; i++)
        {
                d=d*b%c;
                arreglocar[i]=d;
        }
        return(d);
}
```

```
1  //Se encarga de forzar que el estado final de la codificación
2  //sea S0=00 y de encontrar la secuencia original a codificar
3  void encuentra(int g1, int g2, int indice, int memoria)
4  {
5          int i=0, primerbit=0, segundobit=0, k=0, registroconv=0,
              registroconv1=0, j=0;
6          int palabra=0, palabra1=0, xor1=0, xor2=0, xor3=0, xor4=0,
              mascara=0;
7          float ruido=0;
8          int inicial=0, final=0, final1=0, codificada=0, codificada1
              =0;
9          int estadoinicial=0;
10         int bandera;
11         int sistematica=0, sistematica1=0;
12
13         mascara=enmascarar(memoria);
14         palabra=arreglocar[indice];
15
16         for(i=0; i<=5; i++)
17         {
18
19                 primerbit=palabra&1;/////////////////primer palabra
                      del arreglo normal
20
21                 if(primerbit==0)
22                         caracter[indice][i]=0;
23                 else
24                         caracter[indice][i]=1;
25
26                 xor1=xor(g1,registroconv,memoria);//primer
                      codificador
27                 xor1=xor1^primerbit;
28
29                 registroconv=registroconv|xor1;//ahora con la entrada
30
31                 xor2=xor(g2,registroconv,memoria);//aqui va la
                      paridad1
32
33                 if(xor2==0)
34                         paridad1[indice][i]=0;
35                 else
36                         paridad1[indice][i]=1;
37
38                 registroconv=registroconv<<1;
```

```
39                  registroconv=registroconv&mascara;
40
41                  if(i<5)
42                          palabra=palabra>>1;
43          }
44
45          k=i;
46          estadoinicial=registroconv&6;
47          estadoinicial=estadoinicial>>1;
48          bandera=0;
49          for(j=0; j<=7; j++)
50          {
51                  inicial=estado[j][1];
52                  if(estadoinicial==inicial)
53                          {
54                          if(bandera==0)
55                                  {
56                                          final=estado[j][0];
57                                          codificada=estado[j][3];
58                                          sistematica=estado[j][2];
59                                          bandera=bandera+1;
60                                  }
61                                  else
62                                  {
63                                          final1=estado[j][0];
64                                          codificada1=estado[j][3];
65                                          sistematica1=estado[j][2];
66                                  }
67
68                          }
69          }//////////////////////
70          for(j=0; j<=7; j++)
71          {
72                  inicial=estado[j][1];
73                  if(final==inicial && estado[j][0]==0)
74                  {
75                          if(codificada==-1)
76                          codificada=0;
77                          paridad1[indice][k]=codificada;
78
79                          codificada=estado[j][3];
80                          if(codificada==-1)
81                          codificada=0;
82                          paridad1[indice][k+1]=codificada;
83
```

```
                        if(sistematica==-1)
                        sistematica=0;
                        caracter[indice][k]=sistematica;

                        sistematica=estado[j][2];
                        if(sistematica==-1)
                        sistematica=0;
                        caracter[indice][k+1]=sistematica;
                }

                if(final1==inicial && estado[j][0]==0)
                {
                        if(codificada1==-1)
                        codificada1=0;
                        paridad1[indice][k]=codificada1;
                        codificada1=estado[j][3];
                        if(codificada==-1)
                        codificada1=0;
                        paridad1[indice][k+1]=codificada1;

                        if(sistematica1==-1)
                        sistematica1=0;
                        caracter[indice][k]=sistematica1;

                        sistematica1=estado[j][2];
                        if(sistematica1==-1)
                        sistematica1=0;
                        caracter[indice][k+1]=sistematica1;
                }
        }
        k=0;
}
```

```
//Se encarga de entrelazar cada una de las secuencias a codificar
//Encontradas por el módulo encuentra
void entrelazaencuentra(int indice)
{
        int i=0, j=0, k=0;
                for(j=0; j<=3; j++)
                {
                        caractere[indice][k]=caracter[indice][j];
                        caractere[indice][k+1]=caracter[indice][j+4];
                        k=k+2;
                }
```

```
}
```

```
//Aquí solo se agrega a la secuencia resultado de la codificación el ruido AWGN
void codifica(int indice,float sigma, int g1, int g2,int memoria)
{
        int i=0, j=0, bit=0, xor1=0, xor2=0, registroconv=0, mascara=0;
        float ruido=0;
        mascara=enmascarar(memoria);

        for(i=0; i<=7; i++)
        {

                ////////////////////////////////////////////////////
                xor2=caracter[indice][i];
                ruido=RN(0,sigma);
                if(xor2==0)
                {

                        c[indice][j]=-1+ruido;
                        caracterc[indice][i]=-1+ruido;
                }
                else
                {
                        c[indice][j]=1+ruido;
                        caracterc[indice][i]=1+ruido;
                }
                ////////////////////////////////////////////////////
                xor2=paridad1[indice][i];
                if(xor2==0)
                        c[indice][j+1]=-1+ruido;
                else
                        c[indice][j+1]=1+ruido;
                ////////////////////////////////////////////////////

                bit=caractere[indice][i];

                xor1=xor(g1,registroconv, memoria);
                xor1=xor1^bit;

                registroconv=registroconv|xor1;

                xor1=xor(g2,registroconv, memoria);
```

```
41
42                  if(xor1==0)
43                          c[indice][j+2]=-1+ruido;
44                  else
45                          c[indice][j+2]=1+ruido;
46
47                          j=j+3;
48
49                  registroconv=registroconv<<1;
50                  registroconv=registroconv&mascara;
51                  ////////////////////////////////////////////////////
52          }
53          xor1=0;
54  }
```

```
1   //Entrelaza el caracter a codificar utilizando
2   //para ello un entrelazador renglón-columna
3   void entrelazacaracterc(int indice)
4   {
5                   int j=0, k=0;
6                   for(j=0; j<=3; j++)
7                   {
8                           caracterce[indice][k]=caracterc[indice][j];
9                           caracterce[indice][k+1]=caracterc[indice][j
                               +4];
10                          k=k+2;
11                  }
12  }
```

```
1   //Aquí se entrelaza la verosimilitud extr\'inseca del módulo 1
2   void entrelazavero1(int indice)
3   {
4           int j=0, k=0;
5           for(j=0; j<=3; j++)
6                   {
7                           veroe[indice][k]=vero[indice][j];
8                           veroe[indice][k+1]=vero[indice][j+4];
9                           k=k+2;
10                  }
11  }
```

```
1   //
       ////////////////////////////////////////////////////////////////
```

```
2  //Aquí se deseentrelaza la verosimilitud extr\'inseca del módulo 2
3  void dentrelazavero1(int indice)
4  {
5          int j=0, k=0;
6          for(j=0; j<=3; j++)
7                  {
8                          vero1e[indice][j]=vero1[indice][k];
9                          vero1e[indice][j+4]=vero1[indice][k+1];
10                         k=k+2;
11                 }
12 }
```

## A.2. Módulos de decodificación del algoritmo MAP

En esta sección se muestra el diagrama de flujo del turbo decodificador y el código en C de los módulos que lo conforman.

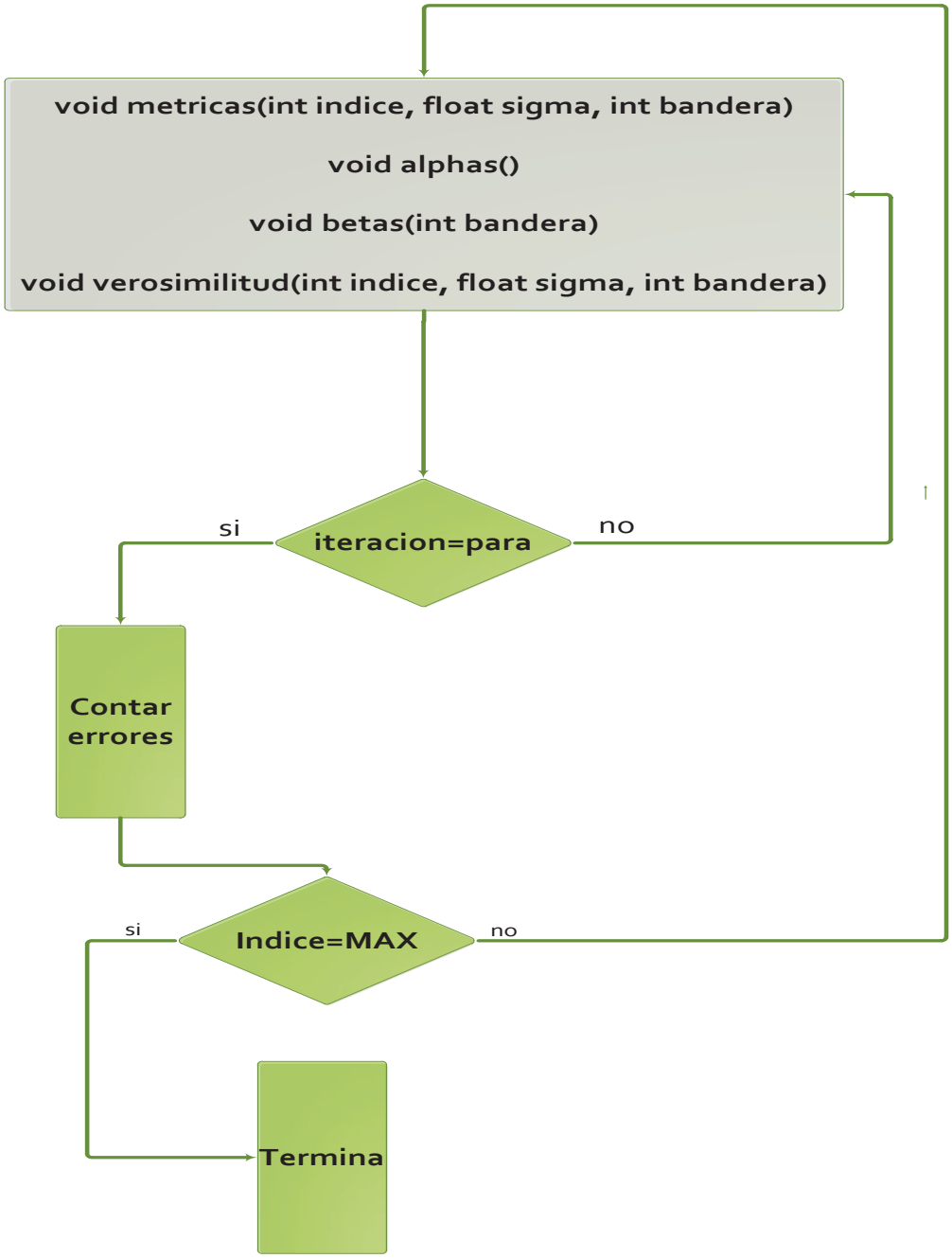



Figura A.2: Diagrama de flujo del turbo decodificador PCCC (3,1,3).

```
//Lo que hace este código es calcular las métricas gamma en las
//ramas del diagrama de Trellis, recibe el índice del arreglo
//que contiene la secuencia a decofificar, además de la varianza
//del ruido sigma, la bandera indica con que módulo se está
//trabajando si el uno o el 2 de decodificación.
void metricas(int indice, float sigma, int bandera)
{
        int i=0, j=0, k=0;//Indices de iteración
        float Lc=0;//Verosimilitud
        float probabilidad=0; //
        float primero=0, segundo=0, primero1=0, segundo1=0;

        Lc=2/(sigma*sigma);

        for(i=0; i<=7; i++)
        {
                j=i*3;
                //Si bandera es igual a cero se trabaja
                // con el módulo 1, se toma la verosimilitud
                // entrelazada proveniente del módulo 2, se
                // toma de la tupla dos elementos, el
                //correspondiente a la información sistemática
                //y al de una paridad1
                if(bandera==0)
                {
                        probabilidad=vero1e[indice][i];
                        primero1=c[indice][j];
                        segundo1=c[indice][j+1];
                }
                //Si bandera es distinta de cero se trabaja
                //con el módulo 2, se toma la verosimilitud
                //entrelazada proveniente del módulo 1, se toma
                //la secuencia sistemática entrelazada y la paridad2
                else
                {
                        probabilidad=veroe[indice][i];
                        primero1=caracterce[indice][i];
                        segundo1=c[indice][j+2];
                }
                //calcula
                //las métricas con los elementos
                //anteriormente descritos
                for(k=0; k<=7; k++)
                {
```

```
45
46                          primero=(0.5)*estado[k][2]*(probabilidad+(Lc*
                               primero1));
47                          primero=exp(atan(primero));
48
49                          segundo=(0.5)*Lc*segundo1*estado[k][3];
50
51                          //Se utiliza para el caso cuando
52                          //se trabaja con el método de
53                          //compresión
54                          segundo=exp(atan(segundo));
55
56                          //Se guarda la métrica total
57                          metri[k][i]=primero*segundo;
58                          //Se guarda la métrica extrínseca
59                          metriex[k][i]=segundo;
60                  }
61          }
62  }
```

```
1   void alphas()
2   {
3           int i=0, j=0, k=0, i1=0;
4           int indi=0, indi1=0;
5           float resultado=0, temp=0;
6
7           alfas[0][0]=1.0;
8
9           for(i=1; i<=8; i++)
10          {
11                  temp=0;
12
13                  //Es para el caso de la normalización
14                  //Aqui se calcula la suma de las alfas
15                  //De la etapa anterior y se les asigna
16                  //A temp
17                  for(j=0; j<=3; j++)
18                  temp=temp+alfas[i-1][j];
19
20                  for(j=0; j<=3; j++)
21                  {
22                          for(k=0; k<=7; k++)
23                          {
```

```
                                    indi=estado[k][0]; indi1=estado[k
                                       ][1];

                                    if(indi==j)
                                            resultado=resultado+(alfas[i
                                               -1][indi1]*metri[k][i-1]);
                            }
                            alfas[i][j]=resultado;//temp;
                            resultado=0;
                    }
            }

            //Para el caso de normalización se calcula la
            //la suma de la etapa actual de las alfas
            for(i=0; i<=8; i++)
            {
                    for(j=0; j<=3; j++)
                    resultado=resultado+alfas[i][j];
                    alfas[i][j]=resultado;
                    resultado=0;
            }
            //Se divide las alfas entre la etapa actual entre su suma
            for(i=0; i<=8; i++)
            {
                    for(j=0; j<=3; j++)
                    alfas[i][j]=alfas[i][j]/alfas[i][4];
            }
    }
```

```
    //Es necesario mandar una bandera llamada iteración la cual indica
       con que
    //módulo se est\'a trabajando
    void bbetas(int iteracion)
    {
            int i=0, j=0, k=0;
            int indi=0, indi1=0;
            float resultado=0;
            float temp=0;

            //si iteración es  igual a cero se
            //esta trabajando con el módulo 1
            if(iteracion%2==0)
                    betas[8][0]=1.0;
            //En otro caso se trabaja con el módulo 2
```

```
15          //y las betas son inicializadas a 1/4
16          else
17          {
18                  betas[8][0]=1.0/4;
19                  betas[8][1]=1.0/4;
20                  betas[8][2]=1.0/4;
21                  betas[8][3]=1.0/4;
22          }
23          for(i=0; i<=7; i++)
24          {
25                  temp=0;
26                  //Es para el caso de la normalización
27                  //Aqui se calcula la suma de las betas
28                  //De la etapa anterior y se les asigna
29                  //A temp
30                  for(j=0; j<=3; j++)
31                  temp=temp+betas[8-i][j];
32
33                  for(j=0; j<=3; j++)
34                  {
35                          for(k=0; k<=7; k++)
36                          {
37                                  indi=estado[k][0];  indi1=estado[k
                                       ][1];
38                                  if(indi1==j)
39                                          resultado=resultado+betas[8-i
                                             ][indi]*metri[k][7-i];
40                          }
41                          betas[7-i][j]=resultado;//temp;
42                          resultado=0;
43                  }
44          }
45          //Para el caso de normalización se calcula la
46          //la suma de la etapa actual de las betas
47          for(i=0; i<=7; i++)
48          {
49                  for(j=0; j<=3; j++)
50                  resultado=resultado+betas[i][j];
51                  betas[i][j]=resultado;
52                  resultado=0;
53          }
54          //Se divide las betas entre la etapa actual entre
55          //Su suma
56          for(i=0; i<=7; i++)
57          {
```

```
                for(j=0; j<=3; j++)
                betas[i][j]=betas[i][j]/betas[i][4];
        }
}
```

```
//Se hace la decodificación
//de los elementos enviados, y para ello se le manda
//el \'indice donde se encuentra la secuencia codificada
//recibida directamente del canal, para ellos también
//es necesario pasar como parámetros la varianza del ruido
//ademas de la bandera que indica con que módulo se est\'a
//trabajando
void verosimilitud(int indice, float sigma, int bandera)
{
        int i=0, j=0, k=0;
        int indi=0, indi1=0;
        float numerador=0, denominador=0, resultado=0;
        float numerador1=0, denominador1=0, resultado1=0;
        float Lc=0;

        Lc=2/(sigma*sigma);

        for(i=0; i<=7; i++)
        {
                j=3*i;
                for(k=0; k<=7; k++)
                {
                        indi=estado[k][0]; indi1=estado[k][1];
                        //Se calcula el denominador el cual
                           corresponde
                        //a las probabilidades de que se recibe '1'
                        //es un cero
                        if(k<=3)
                        {
                                denominador=denominador+(alfas[i][
                                   indi1]*metri[k][i]*betas[i+1][indi
                                   ]);
                                denominador1=denominador1+(alfas[i][
                                   indi1]*metriex[k][i]*betas[i+1][
                                   indi]);
                        }
                        //Se calcula el denomidor al el cual
                           corresponde
```

```
33                          //a las probabilidades de que lo que se
                               recibe
34                          //es un uno
35                          else
36                          {
37                                  numerador=numerador+(alfas[i][indi1]*
                                       metri[k][i]*betas[i+1][indi]);
38                                  numerador1=numerador1+(alfas[i][indi1
                                       ]*metriex[k][i]*betas[i+1][indi]);
39                          }
40                  }
41                  //se calcula la verosimilitud
42                  resultado=numerador/denominador;
43                  resultado=log(resultado);
44                  //se calcual la verosimilitud estrínseca
45                  resultado1=log(numerador1/denominador1);
46
47                  //se hace la decisión en el módulo 1
48                  //con la secuencia no entrelazada vero1e
49                  if(bandera==0)
50                  {
51                          vero[indice][i]=resultado-(vero1e[indice][i
                               ]+(Lc*c[indice][j]));
52
53                          if(resultado>=0)
54                                  final[indice][i]=1;
55                          else
56                                  final[indice][i]=0;
57                  }
58                  //se hace la decisión en el módulo 2
59                  //con la secuencia entrelazada caracterce
60                  else
61                  {
62                          vero1[indice][i]=resultado-(veroe[indice][i
                               ]+(Lc*caracterce[indice][i]));
63
64                          if(resultado>=0)
65                                  final[indice][i]=1;
66                          else
67                                  final[indice][i]=0;
68                  }
69
70                  resultado1=resultado1;
71                  numerador1=0;
72                  denominador1=0;
```

```
73                  numerador=0;
74                  denominador=0;
75          }
76          resultado1=0;
77  }
```

## A.3. Programa principal

```
1   //este el módulo principal donde se mandan a llamar
2   //todo el proceso de Turbo codificación y Turbo
3   //decodificación
4   void main()
5   {
6           int a=0, i=0, j=0, k=0, indice=0, k1=0;
7           float error=0, errorf=0;
8           float secs=0;
9           float er[16]={0};
10
11          FILE *resultados;
12          time_t comienzo, final;
13
14          diagrama(7,5,3);//Genera el diagrama de estados
15
16          comienzo=time(NULL);//para medir el tiempo de procesamiento
17          //archivo donde se guardan los resultados de la BER y el
               tiempo
18          resultados=fopen("C:\\Users\\Portatil\\Desktop\\turbov1.txt",
               "a+");
19
20          //Simulación Montcarlo de -2 a 9 dB se comienza en 0 a 15
21          //y esto es por que los valores del Eb/No estan almacenados
22          //en un arreglo
23          for(k=0; k<=15; k++)
24          {
25                  a=60;
26                  //aqui el índice indica el número de total de
                       arreglos
27                  //a codificar
28                  for(k1=0; k1<total; k1++)
29                  {
30                          //llena el arreglo
31                          a=llenar(a,45555);
32
```

```
33 ////////////////CODIFICACIÓN//////////////////
34 ////////////////////////////////////////////
35 //se codifica uno a uno los número por eso va de 0 a MAX
36                         for(indice=0; indice<MAX; indice++)
37                         {
38                                 //Se encuentra el número a codificar
39                                 encuentra(7,5,indice,3);
40                                 //Entrelaza el número encontrado
41                                 entrelazaencuentra(indice);
42                                 //Codifica el número entrelazado y no
                                       entrelazado
43                                 codifica(indice,eb[k]/2,7,5,3);
44                                 //entrelaza el carcater codificado ya
                                       con el ruido agregado
45                                 entrelazacaracterc(indice);
46                         }
47
48 //se decodifica uno a uno la secuencia correspondiente al número
49 //codificado después de haber pasado por el canal y se comienza de
50 //0 hasta MAX
51                         for(indice=0; indice<MAX; indice++)
52                         {
53 //el índice j indica el número de módulo con el que se esta
      trabajando
54                                 j=0;
55
56 ////////////////////////DECODIFICACIÓN////////////////////////////
57 ////////////////////////////////////////////////////////////////////
58 //este es el proceso de turbo decodificación
59                                 for(i=0; i<=iter; i++)
60                                 {
61                                         metricas(indice,1,0);
62                                         alphas();
63 //el índice como se menciona indica con que módulo se esta
64 //trabajando
65                                         bbetas(j);
66                                         verosimilitud(indice,1,0);
67
68                                         entrelazavero1(indice);
69                                         j=j+1;
70
71                                         metricas(indice,1,1);
72                                         alphas();
73                                         bbetas(j);
74                                         verosimilitud(indice,1,1);
```

```
75                                          dentrelazavero1(indice);
76                                          //se limpian las alfas y
                                               betas
77                                          limpiabetas();
78                                          j=j+1;
79                                  }
80  //se limpia los arreglos donde se guardan las verosimilitudes
81                                  limpiavero();
82                          }
83  //se compara el arreglo original con el arreglo que
84  //se obtiene en la turbo decodificación
85                          errorf=errorf+erroresfi();
86                  }////////////fin de los arreglos
87  //se divide el número total de errores entre el total
88  //de los bits enviados
89                  errorf=errorf/(8*MAX*total);
90                  fprintf(resultados,"%7f\n",errorf);
91                  er[k]=errorf;
92                  errorf=0;
93          }
94          final=time(NULL);
95          secs=difftime(final, comienzo);
96          secs=secs;
97          fprintf(resultados,"%7g\n",secs);
98          fclose(resultados);
99  }
```

# Apéndice B

# Resultados de la simulación en la PC y de la realización práctica DSP

## B.1. Códigos convolucionales

En esta parte se presenta los resultados del desempeño de la realización práctica en el *DSP* y la simulación *PC*, de los *Códigos convolucionales* de tasa $R = \frac{1}{2}$ para diferentes tamaños de memoria [13].

Tabla B.1: Razón BER para códigos convolucionales de tasa $R = 1/2$ en la PC.

| Eb/No | Memoria 3 | Memoria 4 | Memoria 5 | Memoria 6 | Menmoria 7 |
|---|---|---|---|---|---|
| -2 | 0.300179 | 0.300163 | 0.376921 | 0.413184 | 0.435205 |
| -1 | 0.260124 | 0.262386 | 0.321842 | 0.368343 | 0.380941 |
| 0 | 0.2002600 | 0.208268 | 0.254753 | 0.303516 | 0.304248 |
| 1 | 0.1334800 | 0.145020 | 0.167301 | 0.215332 | 0.210514 |
| 2 | 0.0747560 | 0.084310 | 0.090365 | 0.126693 | 0.117139 |
| 3 | 0.0321940 | 0.036963 | 0.032829 | 0.049967 | 0.046077 |
| 4 | 0.0084150 | 0.010579 | 0.009554 | 0.011035 | 0.010710 |
| 5 | 0.0008460 | 0.001432 | 0.000911 | 0.001270 | 0.0017740 |
| 6 | 1.600e-05 | 0.000114 | 6.50e-05 | 6.50e-05 | 3.300e-05 |
| 7 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 8 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 9 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |

Tabla B.2: Razón BER para códigos convolucionales de tasa $R = 1/2$ en el DSP.

| Eb/No | Memoria 3 | Memoria 4 | Memoria 5 | Memoria 6 | Menmoria 7 |
|---|---|---|---|---|---|
| -2 | 0.2790853 | 0.2804688 | 0.3306803 | 0.4167318 | 0.4207357 |
| -1 | 0.2283366 | 0.2347494 | 0.2861328 | 0.3682943 | 0.3828125 |
| 0 | 0.17633460 | 0.1728678 | 0.2441406 | 0.3050944 | 0.3203125 |
| 1 | 0.13162440 | 0.1278971 | 0.1689453 | 0.2226888 | 0.1801921 |
| 2 | 0.08574219 | 0.0871419 | 0.0771484 | 0.1272461 | 0.1142090 |
| 3 | 0.03868476 | 0.0364257 | 0.0292968 | 0.0563476 | 0.0449218 |
| 4 | 0.00688476 | 0.0082194 | 0.0146484 | 0.0142903 | 0.0078125 |
| 5 | 0.00371093 | 0.0013997 | 0.0009765 | 0.0033528 | 0.0 |
| 6 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 7 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 8 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 9 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |

La Tabla B.3 muestra los tiempos de procesamiento de los codificadores convolucionales.

Tabla B.3: Tiempos de procesamiento código convolucional tasa $R = 1/2$

| Memoria | PC (s) | DSP (s) |
|---|---|---|
| 3 | 3 | 25 |
| 4 | 7 | 48 |
| 5 | 16 | 121 |
| 6 | 49 | 375 |
| 7 | 160 | 1296 |

En la tabla anterior, como se puede apreciar el convolucionador de tamaño de memoria M = 3 es más rápido.

A continuación, se muestran los resultados del desempeño de la realización práctica en el DSP y la simulación PC, de los códigos convolucionales de tasa $R = 1/3$ para diferentes tamaños de memoria.

Tabla B.4: Razón BER para códigos convolucionales de tasa $R = 1/3$ en la PC.

| Eb/No | Memoria 3 | Memoria 4 | Memoria 5 | Memoria 6 | Menmoria 7 |
|---|---|---|---|---|---|
| -2 | 0.377246 | 0.268392 | 0.265771 | 0.379883 | 0.328678 |
| -1 | 0.310645 | 0.202100 | 0.211117 | 0.341162 | 0.252116 |
| 0 | 0.2409180 | 0.141683 | 0.148486 | 0.282292 | 0.177295 |
| 1 | 0.1588380 | 0.080762 | 0.090169 | 0.201253 | 0.100081 |
| 2 | 0.0838220 | 0.042480 | 0.044710 | 0.121370 | 0.050798 |
| 3 | 0.0323890 | 0.015951 | 0.019499 | 0.057650 | 0.018929 |
| 4 | 0.0078940 | 0.003988 | 0.004118 | 0.013997 | 0.004753 |
| 5 | 0.0009280 | 0.000423 | 0.000505 | 0.001969 | 0.000879 |
| 6 | 0.0 | 3.300e-05 | 9.80e-05 | 0.000130 | 0.0 |
| 7 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 8 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 9 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |

Tabla B.5: Razón BER para códigos convolucionales de tasa $R = 1/3$ en el DSP.

| Eb/No | Memoria 3 | Memoria 4 | Memoria 5 | Memoria 6 | Menmoria 7 |
|---|---|---|---|---|---|
| -2 | 0.3833822 | 0.2588542 | 0.2638509 | 0.3791504 | 0.3305664 |
| -1 | 0.3262533 | 0.2029460 | 0.2090658 | 0.3340983 | 0.2518880 |
| 0 | 0.2523926 | 0.1337565 | 0.1473470 | 0.2682617 | 0.1787761 |
| 1 | 0.1669922 | 0.0729003 | 0.0931803 | 0.2017253 | 0.0957682 |
| 2 | 0.0806803 | 0.0381510 | 0.0440918 | 0.1217611 | 0.0514648 |
| 3 | 0.0419433 | 0.0173990 | 0.0173665 | 0.0540690 | 0.0174479 |
| 4 | 0.0082031 | 0.0042968 | 0.0041341 | 0.0154296 | 0.0044596 |
| 5 | 0.0 | 0.0003580 | 0.0001953 | 0.0048177 | 0.0008789 |
| 6 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 7 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 8 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 9 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |

Tabla B.6: Tiempos de procesamiento códigos convolucionales tasa $R = 1/3$

| Memoria | PC (s) | DSP (s) |
|---|---|---|
| 3 | 4 | 40 |
| 4 | 8 | 68 |
| 4 | 19 | 152 |
| 6 | 52 | 422 |
| 7 | 173 | 1383 |

Al igual que con los códigos convolucionales de tasa $R = 1/2$, el convolucionador de tamaño de memoria $M = 3$ es mas rápido que los demás.

## B.2. Código RSC

Resultados del desempeño del código RSC en el DSP y PC.

Tabla B.7: Razón BER para RSC (2,1,3)

| Eb/No | PC | DSP |
|---|---|---|
| -2 | 0.333610 | 0.3366536 |
| -1 | 0.290413 | 0.2816895 |
| 0 | 0.237321 | 0.2197428 |
| 1 | 0.170459 | 0.1701172 |
| 2 | 0.105794 | 0.1038633 |
| 3 | 0.047168 | 0.0527181 |
| 4 | 0.013444 | 0.0089029 |
| 5 | 0.001839 | 0.0027018 |
| 6 | 9.80e-05 | 0.0 |
| 7 | 0.0 | 0.0 |
| 8 | 0.0 | 0.0 |
| 9 | 0.0 | 0.0 |

## B.3. Código RSC con algoritmo MAP

Resultados del desempeño del código RSC con algoritmo MAP en el DSP y PC.

Tabla B.8: Razón BER para un RSC con algoritmo MAP (3,1,3)

| Eb/No | PC | DSP |
|---|---|---|
| -2 | 0.320394 | 0.3067220 |
| -1 | 0.283089 | 0.2597656 |
| 0 | 0.2541500 | 0.2045085 |
| 1 | 0.2216310 | 0.1498698 |
| 2 | 0.1829750 | 0.0952962 |
| 3 | 0.1203780 | 0.0467610 |
| 4 | 0.0471680 | 0.0196614 |
| 5 | 0.0123050 | 0.0069986 |
| 6 | 0.0021320 | 0.0021158 |
| 7 | 0.0002440 | 0.0003092 |
| 8 | 0.0 | 0.0 |
| 9 | 0.0 | 0.0 |

## B.4. Turbo código

A continuación, se muestran los resultados del desempeño de la realización práctica en el *DSP* y la simulación *PC*, del *Turbo código PCCC* (3,1,3), utilizando los métodos propuestos.

Tabla B.9: Razón BER pra un Turbo código (3,1,3) compresión.

| Eb/No | PC | DSP |
|---|---|---|
| -2 | 0.016781 | 0.01678060 |
| -1 | 0.002783 | 0.00278320 |
| 0 | 6.500e-05 | 6.5104e-05 |
| 1 | 0.0 | 0.0 |
| 2 | 0.0 | 0.0 |
| 3 | 0.0 | 0.0 |
| 4 | 0.0 | 0.0 |
| 5 | 0.0 | 0.0 |

Tabla B.10: Razón BER para un Turbo código (3,1,3) normalización.

| Eb/No | PC | DSP |
|---|---|---|
| -2 | 0.014486 | 0.011588540 |
| -1 | 0.003320 | 0.001790365 |
| 0 | 0.000570 | 3.25520e-05 |
| 1 | 0.000195 | 0.0 |
| 2 | 0.0 | 0.0 |
| 3 | 0.0 | 0.0 |
| 4 | 0.0 | 0.0 |
| 5 | 0.0 | 0.0 |

# Acrónimos

**LDPC** Low Density Parity Check

**CRC** Cyclic Redundancy Check

**BER** Bit Error Rate

**Eb/No** Energy per bit to Noise Power Spectral Density Ratio

**FEC** Forward Error Correction

**RSC** Recursive Systematic Convolutional

**DSP** Digital Signal Processor

**Eb** Energía de bit

**No** Densidad de potencia de ruido

**SNR** Signal to Noise Ratio

**c** palabra codificada

**n** es la tupla de bits de salida

**k** es la tupla de bits de entrada

**M** tamaño de memoria (# de registros de k-tuplas)

**ne** número de estados

**m** mensaje

**SCCC** Serial Concatened Convolutional Code

**PCCC** Paralell Concatened Convolutional Code

**HCCC** Hybrid Concatened Convolutional Code

**L** tamaño del mensaje

**K** tamaño de bloques

**LLR** Log Likelihood Ratio

**MAP** Maximum A Posteriori

**AWGN** Additive White Gaussian Noise

# Bibliografía


[1] Barrón Fernandez R., Oropeza Rodriguez J.L., Laguna Sánchea G.A., "Low Complexity Turbo Code Specification for Power-line Communication (PLC)," *Electronics, Robotics and Automotive Mechanics Conference (CERMA), IEEE*, pp. 349–354, Noviembre 2011.

[2] Branka Vucetic, Jinhong Yuan, *Turbo Codes, Principles and Applications*. Kluwer Academic Publishers, 2004.

[3] Phil Lapsley, Jeff Bier, Amit Shoham, Edward A. Lee, "DSP Processor Fundamentals: Architecture and Features," *Press series on signal processing, IEEE*, 1997.

[4] Fabián Luna I., Laguna Sánchez G.A., Marcelín Jiménez R., "Realización práctica de los Turbo códigos de alto desempeño con un TMS320C6416," *ROPEC, IEEE*, pp. 7–12, Noviembre 2012.

[5] Shannon C., "A Mathematical Theory of Communication," *The Bell System Technical Journal*, vol. 27, pp. 379–423, October 1948.

[6] Elias P., "Coding for Noisy Channels," *IRE Convention Record*, vol. 3, pp. 37–46, May 1955.

[7] Gallager R. G., "Low-Density Parity-Check Codes," *Transactions on Information Theory, IEEE*, vol. 10, p. 172, April 1964.

[8] Mackay D.J.C, Neal, R.M., "Near Shannon limit performance of low density parity check codes ," *Electronics letters, IEEE*, vol. 1, pp. 457–458, March 1997.

[9] Berrou C., "Near Shannon limit error-correcting coding and decoding: Turbo-codes," *International Conference Communications (ICC), IEEE*, vol. 2, pp. 1064–1070, May 1993.

[10] Guizzo E., "Closing in on The Perfect Code," *IEEE Spectrum*, pp. 37–42, March 2004.

[11] Tomasi Wayne, *Sistemas de comunicaciones eletrónicas*. Prentice Hall, 2003.

[12] Stewart James, *Cálculo Trascendentes Tempranas*. Thomson Learning, 2002.

[13] Sklar B., *Digital Communications Fundamentals and Aplications*. Prentice Hall, 2001.

[14] Viterbi A.J., "Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algotithm," *Transaction on Information Theory, IEEE*, vol. 13, pp. 260–269, April 1967.

[15] Thitimajshima P., "Recursive systematic convolutional codes and application to parallel concatenation.," *Global Telecommunications Conference, GLOBECOM, IEEE*, pp. 2267–2272, 1995.

[16] S. Adrian Barbulescu, Steven S. Pietrobon, "TURBO CODES: a tutorial on a new class of powerful error correcting coding schemes - Part I: Code Structures and Interleaver Design.," 1998.

[17] Shu Lin, Daniel J. Costello Jr. , *Error Control Coding.* Prentice Hall, 2004.

[18] Bahl L., Cocke J., Jelinek F., Raviv, J., "Optimal decoding of linear codes for minimizing symbol error rate.," *Transactions on Information Theory, IEEE*, pp. 284–287, March 1974.

[19] Lattice, http://www.latticesemi.com.

[20] E. Ryan William, "A Turbo Code Tutorial.," *New Mexico State University*, pp. 4543–4550, 1997.