Práctica2

Técnicas Avanzadas de codificación

Procesamiento de señal en sistemas de comunicaciones y audiovisuales

Máster Universitario en Ingeniería de Telecomunicación

ETSIT-UPV

Índice

1. Introducción y Objetivos

Los canales de comunicaciones no son ideales ya que su funcionamiento está determinado por diferentes factores que ocasionan que los datos lleguen con errores al receptor, es decir, la información recibida en recepción puede diferir en uno o más bits con respecto a la información transmitida. En cualquier sistema de comunicaciones es necesario recibir de forma fiable y libre de errores la información transmitida desde la fuente. Para ello existen dos estrategias posibles:

- ARQ (Automatic Repeat Request), que se basa en la detección de errores, pero sin la posibilidad de corrección, solicitando al transmisor la repetición del mensaje en caso de error.
- FEC (Forward Error Correction), que se basa en la detección y corrección en el extremo receptor de los posibles errores.

En ambos casos, es necesario añadir cierta redundancia controlada al mensaje a transmitir, de tal forma que los errores producidos en el canal de comunicaciones puedan ser detectados y/o corregidos, al proceso de añadir esta redundancia es a lo que se denomina codificación de canal. En la época actual los dos tipos de códigos más utilizados en los sistemas de comunicaciones modernos son los Turbo Códigos (TC) y los códigos de chequeo de paridad de baja densidad (LDPC). Estos códigos se caracterizan por utilizar métodos iterativos en la decodificación de la información.
El objetivo de esta práctica es familiarizar al alumno con las técnicas de codificación de canal utilizadas en los principales estándares de comunicaciones actuales como son WiMax, WiFi o LTE. Para ello nos plantearemos como objetivo analizar la probabilidad de error de bit tras realizar la decodificación de canal en recepción, empleando distintos códigos y utilizando como herramienta de simulación Matlab. En esta práctica utilizaremos técnicas de codificación mas avanzadas que las utilizadas en la práctica 1, en concreto implementaremos tres tipos de códigos de los estudiados en teoría: los códigos LDPC, los códigos convolucionales y los Turbo Códigos, siendo estos los que se implementan en los sistemas de comunicaciones actuales. Para cada uno de los códigos utilizados se implementará el sistema representado en la figura 8 obteniéndose las curvas de probabilidad de error de bits en función de la Eb/No, y comparando los resultados obtenidos con los resultados sin el empleo en el sistema de codificación de canal, y evaluando la ganancia que proporciona el uso de cada uno de los códigos.

2. Diseño de los codificadores

2.1. Codificación y decodificación LDPC en Matlab

Los LDPC son códigos bloques lineales sistemáticos, los cuales basan su funcionamiento en una matriz de comprobación de paridad utilizada en el proceso de codificación y decodificación. Dicha matriz define las relaciones entre los distintos símbolos de codificación (símbolos fuente y símbolos de paridad). La matriz está formada por elementos con valores 0 y 1, y es dispersa, ya que la mayoría de elementos son nulos. A través de dicha matriz el codificador crea los símbolos de paridad a partir de los símbolos fuente (y otros símbolos de paridad ya creados). Asimismo, en recepción, la matriz se emplea para reconstruir de forma iterativa los bits que fueron enviados. Una manera conveniente de describir los códigos LDPC es mediante el grafo de Tanner, introduciendo una representación gráfica. El grafo de Tanner es un gráfico bipartito, lo que significa que los nodos están separados en dos conjuntos. Por un lado tenemos los llamados nodos variables, y en el otro lado los llamados nodos de comprobación. Los dos tipos de nodos están conectados si el elemento de **H** correspondiente es igual a 1. En la figura 1 se muestra un ejemplo de grafo de Tanner, el cual está asociado con la siguiente matriz de comprobación de paridad.

$$\mathbf{H} = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \end{pmatrix} \tag{1}$$

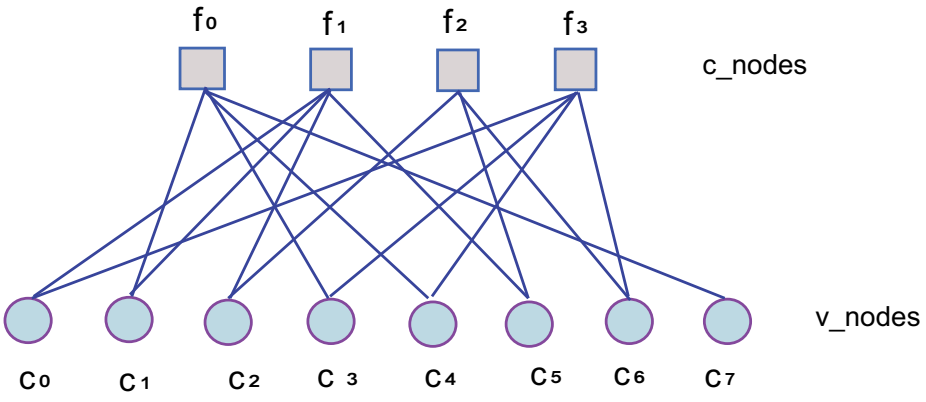

Figura 1: Grafo de Tanner asociado a la matriz H

La forma de codificar y decodificar mediante el empleo de códigos LDPC y

haciendo uso de la herramienta de simulación Matlab es la siguiente:

- Mediante las funciones de Matlab correspondientes construiremos los objetos de asociados al codificador y decodicador LDPC correspondientes a la matriz anterior. En primer lugar contruiremos la matriz de comprobación de paridad asociada al Grafo de la Figura 1, y posteriormente los objetos codificador y decodificador.

```
>>i = [2 4 1 2 2 3 1 4 1 4 2 3 3 4 1 ];
>>j = [1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 ];
>>H = sparse ( i, j, ones ( length ( i ) , 1 ) );
>>hEnc = comm.LDPCEncoder ( H );
>>hDec = comm.LDPCDecoder ( H );
```

- Con estos objetos codificamos una secuencia de bit de información y posteriormente decodificamos la palabra código

```
>> NumInfoBits = size ( hEnc.ParityCheckMatrix, 1 );
>> InfoBits = randi ( [0 1],  NumInfoBits, 1);
>>codeword = step ( hEnc, InfoBits );
>>mDec = step ( hDec, cast(not(codeword),'double'));
```

puedes observar mediante este sencillo caso como usando la misma matriz en codificación y decodificación podemos obtener el mensaje codificado en transmisión, comprueba como los bits de información son los mismos que los bits decodificados. Para la obtención de las curvas de probabilidad de error de bit del último apartado de la práctica la matriz de comprobación de paridad a utilizar será la proporcionada en el archivo `matrixH.mat`(al cargar este archivo se cargará en el directorio de trabajo la matriz H).
Nota: Cuando crees el sistema de comunicaciones completo no debes emplear las instrucciones cast(not(),'double') a la entrada del decodificador puesto que el demodulador ya entregará los valores con formato y signo adecuados.

2.2. Codificación convolucional

Los códigos convolucionales son códigos lineales, donde la suma de dos palabras código también es una palabra código. Se diferencian de los códigos

bloque en su forma estructural y las propiedades que tienen para corregir los errores. Este tipo de codificación tiene memoria, es decir, la codificación actual depende del dato que se envía ahora y del que se envió en el pasado.

Un código convolucional se especifica por medio de tres parámetros (n, k, m):

- n es el número de bits de la palabra codificada
- k es el número de bits de la palabra de información.
- m es la memoria del código o longitud restringida.

La codificación convolucional se realiza básicamente mediante el uso de un registro de desplazamiento y una lógica combinacional encargada de la realización de la suma en módulo 2. En la siguiente figura podemos observar un ejemplo de codificador convolucional, como ejemplo se ha tomado un código con valores $k = 1$, $n = 2$, $m = 2$, es decir la tasa de codificación en este caso es $R = k/n = 1/2$.

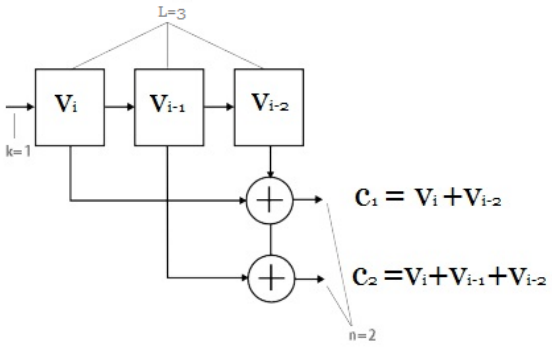

Figura 2: Ejemplo de codificador convolucional.

El número de registros utilizados es $L = 3$ y este el número de entradas que afectan a las salidas (se hace referencia al bit actual y a los dos anteriores), por tanto la memoria m en este decodificador es 2.

Para representar en Matlab el enrejado del codificador anterior se utiliza la función `"poly2trellis"`.Para el caso de la figura 2 los parámetros de entrada a la función son

```
>>m = 2;
>>Enrejado = poly2trellis ( m+1, [ 5 7] );
```

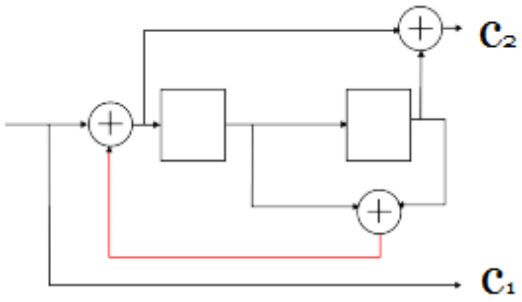

Figura 3: Ejemplo de codificador convolucional.

También existen codificadores recursivos, estos codificadores son aquellos a los que a la entrada se le añade alguna de las sumas anteriores, como es el caso del ejemplo mostrado en la Figura 3. Además, este codificador es sistemático, lo que significa que la entrada está incluida en la salida, generalmente los codificadores recursivos son sistemáticos.

En este caso mediante la siguiente instrucción crearemos el enrejado correspondiente:

```
>>m = 2;
>>Enrejado = poly2trellis ( m+1, [ 7 5], 7 );
```

Creado el enrejado que va a definir nuestro codificador pasaremos a crear el objeto codificador y posteriormente emplearemos la rutina `"step"`para codificar los bits de información:

```
>>NumInfoBits = 10;
>>InfoBits = randi( [0 1], NumInfoBits , 1);
>>hEnc = comm.ConvolutionalEncoder( Enrejado, 'TerminationMethod', ...
        'Terminated');
>>codeword = step(hEnc, InfoBits);
```

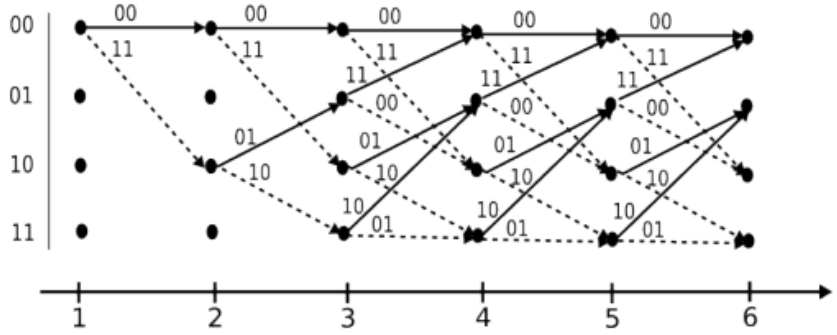

Figura 4: Diagrama de Trellis del codificador convolucional de la figura 3.

La decodificación consiste en encontrar la palabra código más probable, aquella en la que la distancia Hamming con la secuencia recibida sea menor. Esta decodificación es costosa cuando tenemos grandes cantidades de información. Como solución, una de las posibles opciones de decodificación es el empleo del algoritmo de Viterbi y el cual basa su explicación en el diagrama de Trellis. Para el caso del codificador no recursivo anterior el diagrama de Trellis es el mostrado en la figura 4.

Un decodificador basado en este método almacena cada secuencia de bits recibida y calcula la secuencia de estados más probable, siguiendo los caminos en el diagrama de Trellis con menor distancia acumulada y descartando tanto los caminos no viables como los menos probables. Una vez encontrado este camino se obtiene la cadena de bits asociada a esas transiciones.

En Matlab usaremos el objeto codificador correspondiente al decodificador de viterbi, con los parámetros de entrada correspondientes y equivalentes al codificador creado anteriormente. Una vez creado el decodificador, utilizaremos la rutina `step` para decodificar.

```
>>hDec=comm.ViterbiDecoder(Enrejado,'InputFormat','hard',...
'TracebackDepth', m+1,'TerminationMethod','Terminated');
>>deco = step( hDec,codeword);
>>mDec=deco(1:end-(m));
```

El codificador/decodificador han sido creados empleando el método `Terminated`, con este método siempre se comienza y se acaba con la secuencia todo ceros, es decir, con todos los valores de los registros a cero. Es por ello, que el decodificador nos devuelve además de los bits decodificados correspondientes a

los bits de información, los correspondientes a los bits de cierre añadidos por el codificador (en número igual a la memoria del codificador, m), los cuales deben ser extraídos a la salida del decodificador puesto que no son bits de información.

2.3. Turbo Códigos

Los TC hacen uso de varios códigos combinados y consiguen alcanzar la capacidad correctora de códigos convolucionales con memorias mayores sin que su decodificación suponga un complejidad equivalente a la de los convolucionales de gran memoria. Un caso simple es aquel en el que la misma secuencia de información es codificada dos veces en paralelo, utilizando la secuencia normal para un turbo código y la entrelazada para el otro.

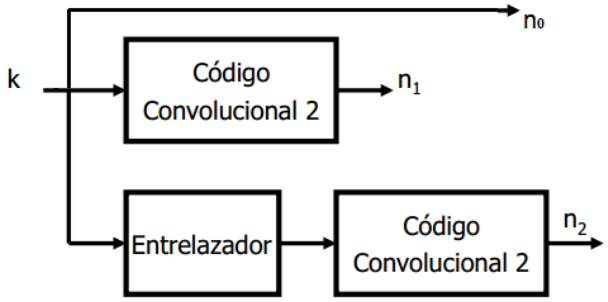

Figura 5: Diagrama de bloques de un TC.

En la práctica haremos uso de la función `turbo_codif` la cual implementa un TC de tasa 1/2 como el mostrado en la figura anterior, en este codificador perforando los bits de paridad a la salida de los decodificadores se obtiene la tasa de codificación adecuada.

De este modo creamos dos objetos codificadores para pasar a la función que realiza la turbo codificación.

```
>>Enrejado = poly2trellis ( 4, [ 13 15 ], 13 );
>>hCEnc1 = comm.ConvolutionalEncoder('TrellisStructure', Enrejado, ...
            'TerminationMethod', 'Terminated' );
>>hCEnc2 = comm.ConvolutionalEncoder('TrellisStructure', Enrejado, ...
           'TerminationMethod', 'Terminated' );
```

Creados los objetos, codificamos con el turbo codificador una serie de bits de información

```
>>InfoBits = randi ([0 1], NumInfoBits,1);
>>codeword=turbo_codif(hCEnc1,hCEnc2,InfoBits);
```

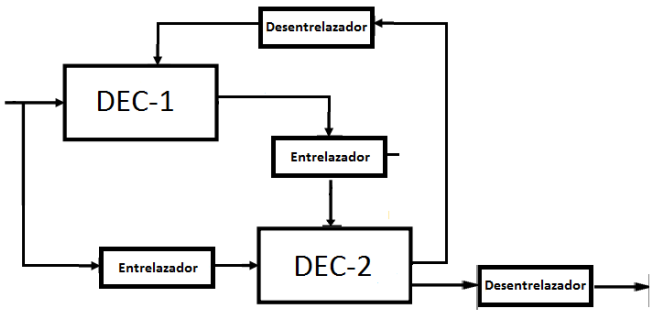

Figura 6: Diagrama de bloques de un Turbo Decodificador.

La decodificación turbo se realiza de forma iterativa. El primer decodificador recibe los datos correspondientes a su entrada junto a la información extrínseca obtenida por el segundo decodificador. Éste recibe a su vez los datos correspondientes entrelazados y la información extrínseca del primer decodificador.
La función `turbo_dec`, proporcionada en la práctica, realiza esta decodificación, siendo los parámetros de entrada: el número de iteraciones del turbo decodificador, el vector correspondiente a la palabra codificada y los objetos correspondientes a los decodificadores.

```
>>hDec1 = comm.APPDecoder('TrellisStructure', Enrejado, ...
         'TerminationMethod', 'Terminated', 'Algorithm', 'True APP');
>>hDec2 = comm.APPDecoder('TrellisStructure',Enrejado, ...
         'TerminationMethod','Terminated', 'Algorithm', 'True APP');
>>numIter=6;
>>mDec = turbo_dec(codeword, NumInfoBits,numIter,hDec1,hDec2);
```

Podemos comprobar como se decodifican los bits de Información correspondientes.

3. Curvas de probabilidades de error de bit

Para cada uno de los códigos anteriores obtendremos las curvas de probabilidad de error de bit en función de $Eb/No(dB)$, rectificando la energía

debido a la ratio de la codificación de cada código, junto a ellas representaremos la curva obtenida sin la utilización de codificación. El diagrama de bloques del modelo que vamos a implementar es el representado en la siguiente figura

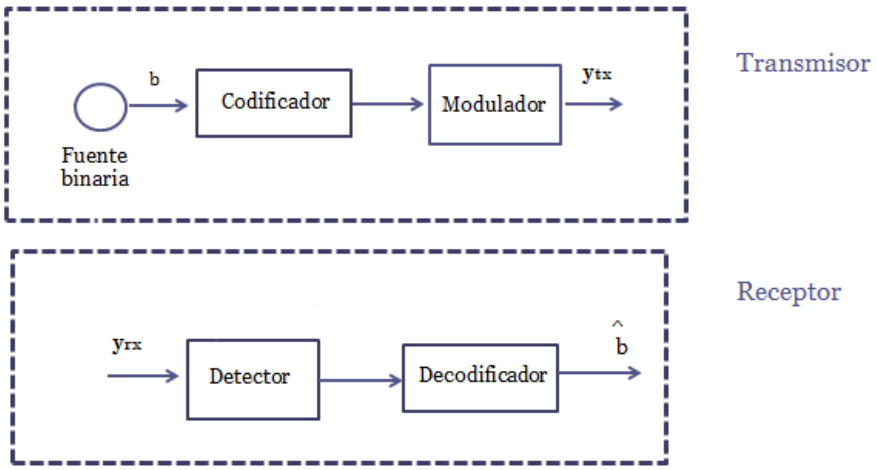

Figura 7: Diagrama de bloques del modelo de comunicaciones.

De tal modo que se generará una secuencia de bits de información, la cual se codificará y modulará para ser enviada a través de un canal. Una vez la señal enviada ha pasado por el canal, en recepción se harán los pasos opuestos al transmisor.

3.1. Modulación PSK

Para la etapa de modulación emplearemos un modulador/demodulador PSK con codificación Gray, empleando para ello objetos de Matlab. En primer lugar se diseña el objeto modulador y demodulador, y a continuación se modula la señal en transmisión y se demodula en recepción. En nuestro caso utilizaremos un modulador y demodulador PSK con codificación Gray empleando las funciones

```
>>hMod=comm.PSKModulator(M,0,'BitInput',true,'SymbolMapping','Gray');
>>hDemod=comm.PSKDemodulator(M,0,'SymbolMapping','Gray', ...
        'BitOutput',true, 'DecisionMethod','log-likelihood ratio');
```

donde M representa el tamaño de la constelación a emplear, en nuestro caso emplearemos una BPSK (M=2).

Una vez diseñado el objeto modulador y demodulador se realiza la modulación/demodulación empleando la función `step` que realiza la modulación/demodulación de la señal digital `x` empleando el objeto modulador/demodulador hMod/hDemod. El objeto modulador por defecto está preparado para recibir valores enteros.

```
>>x_mod=step(hMod,x);
>>x_dem=step(hDemod,x);
```

Sin embargo el demodulador proporcionará a su salida valores soft o hard dependiendo del valor que le demos al parámetro `DecisionMethod` (*log-likelihood ratio* o *Hard decision*). Para cada uno de los codificadores se deberá inicializar al valor que corresponda (los decodificadores TC y LDPC utilizarán valores de entrada soft. Viterbi y el caso sin codificación utilizarán valores hard). En el caso de utilizar valores soft (*log likelihood ratio*) el signo de estos valores representa si el valor representa un uno o un cero, de modo que a la salida del objeto y habiendo seleccionado el parámetro `DecisionMethod` como *log-likelihood ratio*, un valor positivo representará un uno y un valor negativo representará un cero, el valor absoluto representa la probabilidad con que ese valor es el que nos indica el signo.

3.2. Canal AWGN

El modelo de canal utilizado en nuestras simulaciones será un canal AWGN. Para diseñar un canal con ruido aditivo, blanco y gaussiano, emplearemos el objeto `comm.AWGNChannel`. Compruebe las características del objeto mediante la ayuda de Matlab . Una de las propiedades más importantes es `'NoiseMethod'` que indica el método empleado para especificar el nivel de ruido añadido. Por ejemplo, empleando el código siguiente:

```
>>hcanal=comm.AWGNChannel('NoiseMethod', ...
        'Signal to noise ratio (SNR)', 'SNR',20);
```

empleamos como método para especificar el ruido la relación señal a ruido (SNR), en este caso de valor $20dB$. Si en lugar de emplear el dato SNR empleamos la relación energía de bit a ruido (Eb/No) de valor $20dB$, debemos introducir

```
>>hcanal=comm.AWGNChannel('NoiseMethod', ...
        'Signal to noise ratio (Eb/No)', 'EbNo',20,'BitsPerSymbol',k);
```

Este método está configurado por defecto en la función, y requiere el dato del número de bits por símbolo, indicando previamente `BitsPerSymbol`.
Una vez tenemos definido el canal, para transmitir a través de él una señal modulada, es decir, para añadir el ruido gaussiano blanco diseñado a esta señal, emplearemos la función `step`:

```
>>yrx=step(hcanal,x_mod);
```

Para realizar la práctica definiremos el canal empleando la relación energía de bit a ruido.

3.3. Resultados

Trácense las curvas de probabilidad barriendo SNR desde -7 hasta 7 dB con saltos de 1dB. Recuerde corregir la Eb/No consecuentemente al ratio de codificación R para cada código:

```
>>SNR = -7 : 1 : 7;
>>EbNo = SNR-10*log10(R*log2(M));
```

Recuerde que para los códigos LDPC el ratio de codificación viene dado por las dimensiones de la matriz de comprobación de paridad, y para los TC y los códigos convolucionales viene dado por los parámetros correspondientes al objeto `Enrejado` (`Enrejado.numInputSymbols` y `Enrejado.numOutputSymbols`).

Para calcular el número de bits erróneos en la detección de la secuencia binaria transmitida definiremos un objeto del tipo `comm.ErrorRate`. Este objeto nos proporciona además, una estimación de la probabilidad de error de bit o *Bit Error Rate* BER del sistema mediante el cociente entre el número total de bits erróneos y el número total de bits transmitidos durante todas las simulaciones.

```
>>hCalculoError=comm.ErrorRate;
>>VectorErrores=step(hCalculoError, secuencia tx, secuencia detectada);
>>bit_error_rate=VectorErrores(1);
>>numero_errores=VectorErrores(2);
```

Este objeto de Matlab no resetea sus valores si no se vuelve a definir. Es decir, mientras no defina de nuevo el objeto, cada vez que ejecutamos la función `step`, se calcula la media del número de errores y BER de la ejecución actual considerando las anteriores medidas.

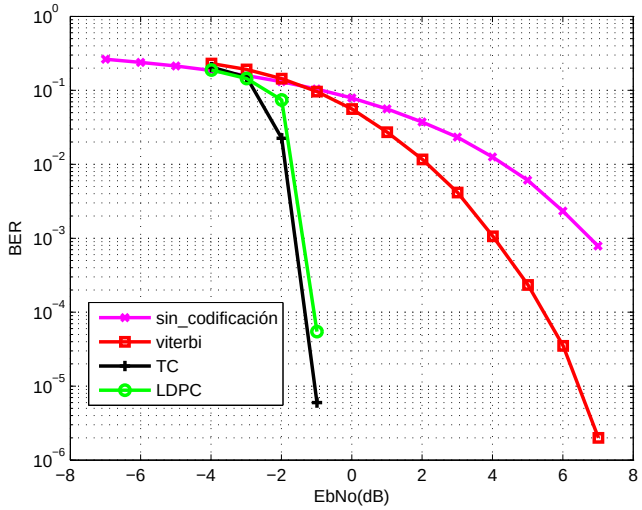

Figura 8: Probabilidad de error de bit con y sin codificación de canal.

El resultado a entregar será: una gráfica similar a la que se adjunta, así como los scripts que se han utilizado a tal efecto. Para ello se deben tener en cuenta las siguientes consideraciones en las simulaciones:

- Para calcular el BER dado un cierto EbNo, es decir, para calcular cada punto de la gráfica, se debe computar el BER promedio de Nmensajes enviados.
- Para comprobar el correcto funcionamiento de las funciones programadas utilice solamente Nmensajes=100, pues el coste computacional para algunos códigos es elevado.
- Para el caso sin codificación se debe omitir la codificación y decodificación de la señal.
- Utilizar el enrejado `poly2trellis(4,[ 13 15 ], 13)` tanto en el codificador de Viterbi como en los TC.
- El número de bits de información será inicializado a 1152.
- En el caso de los LDPC, los bits decodificados pertenecen al tipo *booleano*, se deben convertir a tipo double antes de hacer el `step` para ejecutar el objeto hCalculoError.
- Para obtener el resultado final reflejado en la gráfica de la Figura 8 se han simulado 1000 transmisiones, es decir, en este caso Nmensajes=1000.