Universidad de Alcalá Escuela Politécnica Superior Departamento de Teoría de la Señal y Comunicaciones	**Curso 2021-22** **Guion de las demos** **Curso:** 3º B **Asignatura:** Comunicaciones Digitales

Demos de Codificación de Canal

Práctica 3.1: demodulación dura / demodulación blanda.

PARTE 1: demodulación dura frente a demodulación blanda: cáculo de LLRs (*log-likelihood ratios*).

Considerar una modulación cuyos posibles símbolos $\mathbf{s}_i$ vienen definidos por M bits, por lo que habrá $i = 1, \cdots, 2^M$ posibles símbolos distintos. En el canal AWGN, una muestra recibida tendrá el valor $\mathbf{r} = \mathbf{s}_j + \mathbf{n}$, donde $\mathbf{n}$ es una muestra del ruido blanco gaussiano aditivo con potencia σ^2, y $\mathbf{s}_j$ es el símbolo enviado de entre los 2^M posibles.

Como es sabido, un demodulador basado en máxima verosimilitud hallará el símbolo candidato $\mathbf{s}_i$ más próximo a $\mathbf{r}$ y dará una salida binaria correspondiente a los M bits que definen dicho símbolo estimado. Este procedimiento se conoce como **demodulación dura**. La secuencia de bits recuperados normalmente pasan a un decodificador de canal (en el caso de haber aplicado la codificación de canal en el transmisor), con el objetivo de detectar o corregir posibles errores en los bits estimados de esta forma.

Sin embargo, en este proceso se pierde cierta información que podría servir para mejorar el rendimiento del sistema. En efecto, si el símbolo más cercano a una muestra recibida $\mathbf{r}$ está dado por un bloque de bits $10\cdots11$, y el siguiente más cercano por $11\cdots11$, de alguna forma podemos tener una certeza mayor de que el primer bit corresponde a un 1, mientras que, para el segundo bit, decidir sobre su valor en función de lo que nos ofrece el canal ($\mathbf{r}$), resulta más problemático. Una forma de medir y aprovechar esta información se realiza mediante la llamada **demodulación blanda**, que ofrece como salida para cada bloque de bits la probabilidad de que cada uno de sus elementos sea un 0 ó un 1.

Dada una modulación genérica como la descrita más arriba, y una muestra recibida $\mathbf{r}$ desde un canal AWGN, los valores de probabilidad para el demodulador blando respecto a un bit b_k, $k = 1, \cdots, M$, se calculan de acuerdo a la siguiente expresión de verosimilitud:

$$P\left(b_k = b \,\middle|\, \mathbf{r}\right) = \frac{p(\mathbf{r} \mid b_k=b)\cdot P(b_k=b)}{p(\mathbf{r})} = \frac{P(b_k=b)}{p(\mathbf{r})} \cdot \sum_{\mathbf{s}_i/b_k=b} p\left(\mathbf{r} \mid \mathbf{s}_i\right),$$

$$P\left(b_k = b \,\middle|\, \mathbf{r}\right) = \frac{P(b_k=b)}{p(\mathbf{r})} \cdot \sum_{\mathbf{s}_i/b_k=b} \frac{1}{\sqrt{2\pi\sigma^2}} \mathrm{e}^{-\frac{|\mathbf{r}-\mathbf{s}_i|^2}{2\sigma^2}},$$

donde b puede tomar los valores 0 ó 1, y $P\left(b_k = b\right)$ es la distribución de probabilidades a priori. Observar que, si los bits son equiprobables, el término a la izquierda del sumatorio en el lado derecho de la última expresión es igual para los casos $b = 0$ y $b = 1$.

Finalmente, se suele trabajar con el cociente de probabilidad entre los casos $b = 0$ y $b = 1$, ya que son complementarios, y, más concretamente, con el logaritmo de dicho cociente, debido a la presencia de los términos exponenciales correspondientes al ruido blanco gaussiano aditivo. De tal forma, lo que se computa normalmente en un demodulador blando son los llamados valores de LLR (*log-likelihood ratio*, logaritmo del cociente de verosimilitudes), según la expresión:

$$\lambda_k = \log\left(\frac{P(b_k = 1 \mid \mathbf{r})}{P\left(b_k = 0 \,\middle|\, \mathbf{r}\right)}\right) = \log\left(\frac{\sum_{\mathbf{s}_i/b_k=1} \mathrm{e}^{-\frac{|\mathbf{r}-\mathbf{s}_i|^2}{2\sigma^2}}}{\sum_{\mathbf{s}_j/b_k=0} \mathrm{e}^{-\frac{|\mathbf{r}-\mathbf{s}_j|^2}{2\sigma^2}}}\right).$$

Obsérverse cómo han desaparecido algunas de las constantes y cómo se puede calcular λ_k en el demodulador a partir de la información sobre la correspondencia bits / símbolo y la muestra recibida $\mathbf{r}$. La igualdad entre el valor de LLR y el logaritmo del cociente de probabilidades a priori es cierta sólo cuando la fuente de bits es equiprobable. Un valor de λ_k positivo indica una mayor probabilidad de estar ante un 1; un valor negativo, ante un 0. A mayor valor

del módulo de λ_k, mayor certeza. Es de notar que, a diferencia de la demodulación y decisión dura, necesitamos conocer la potencia de ruido en el canal σ^2, o, de forma equivalente, la E_b/N_0 en el receptor.

Usando el código de prueba adjunto (Tema3_1_lab.m), donde se realiza una operación de modulación digital (¿a qué tipo de modulación corresponde?) y se simula el paso por un canal AWGN en su equivalente paso bajo, se trata de comparar la demodulación y decisión dura en el receptor, así como la demodulación blanda.

Representar gráficamente la secuencia de valores LLR junto con la secuencia de bits decididos según el esquema de demodulación y decisión duras y comparar resultados. Observar cómo la certeza sobre si estamos ante un 0 ó un 1 resulta variable (como está dicho, el signo de la LLR nos dice cuál de las dos alternativas es la más probable, su módulo la certeza o fiabilidad de la decisión). Buscar los errores producidos en la demodulación y decisión duras y comparar con los resultados de los valores de LLR correspondientes. Cambiar los valores del parámetro `noisePower` y ver qué sucede.

Se habrá podido observar que los valores de LLR por sí solos comparados con el criterio ML de demodulación dura por vecino más próximo no añaden nada al proceso: si usamos las LLR para decidir, tendremos el mismo resultado (comprobarlo). En la siguiente parte de la práctica veremos cómo se puede aprovechar esta información, sin embargo, para obtener cierta ganancia cuando interviene la codificación y decodificación de canal.

PARTE 2: ejemplo comparativo con SIMULINK©.

En la ventana de comando de MATLAB©, teclear lo siguiente:

```
>> modelName = 'commLLRvsHD';
>> open_system(modelName);
```

Se abrirá un modelo de SIMULINK© en que se compara, en términos de BER (*bit error rate*) tres posibilidades de demodulación con una misma codificación y decodificación de canal. Obsérvese que hay un módulo Convolutional Encoder y otro Viterbi Decoder, correspondientes a esta última. Hacer doble click sobre el módulo Convolutional Encoder y observar cómo está definido el código convolucional en términos de los polinomios generadores 171 y 133 (en octal). La memoria es 6 (MATLAB usa como parámetro $\nu+1$). Se trata de un código NSC, de tasa $R = 1/2$. La función `poly2trellis` construye el trellis a partir de la descripción polinómica. Observar en los módulos Viterbi Decoder cómo aparece la misma función con los mismos parámetros: el trellis es todo lo que se necesita tanto para codificar como para decodificar, aunque es más fácil definir el CC en términos de sus polinomios generadores y así es como se encuentra en la bibliografía.

Ejecutar una simulación (ir a `Simulation` → `Start`) y observar los valores de BER ofrecidos. ¿Hay alguna diferencia apreciable? ¿Qué diferencia hay entre la segunda y el tercera cadena decodificadora?

Hacer doble click sobre el módulo `Noise variance`. Ejecutar simulaciones para valores fijos de la varianza del ruido desde 0,60 hasta 1,60 en pasos de 0,10. Anotar los valores de SNR (dB) correspondientes y los valores de BER para cada uno de los tres decodificadores. Representar en una gráfica las tres curvas de BER vs. SNR (dB), en escala logarítmica. ¿Qué diferencias se observan en SNR (dB) entre los distintos casos para una misma BER?

Práctica 3.2: códigos bloque lineales.

PARTE 1: codificación de LBC.

Considerar un código Hamming con $m = 3$. En esta práctica se realizará la codificación con ayuda de la matriz generadora **G**. Utilizando el código de prueba que se proporciona en Tema3_2_lab.m[1], se trata de organizar la secuencia de información en bloques consecutivos de k bits (de ahí el nombre de "códigos bloque"), a los cuales se hace corresponder la palabra del código correspondiente según **G**. Los bloques de n bits resultantes se organizan para reconstituir una secuencia continua.

El código proporciona en su primera parte las funciones de generación de las secuencias y de organización en bloques, pero falta proporcionar la matriz generadora. Conociendo las propiedades del código Hamming, lo más fácil será crear una matriz **H** sistemática (que se pide más adelante) y, de ahí, la matriz **G**. Observar las secuencias de bits antes y después de codificar: si se mantiene **G** en forma sistemática, veremos reaparecer los bits de información acompañados de $n - k$ bits de paridad. ¿Por qué usamos el comando `mod(...)` en las operaciones binarias de este código?

PARTE 2: decodificación de LBC (detección y corrección).

En la segunda parte del código, se proporcionan funciones que realizan la obtención del síndrome correspondiente a cada bloque de n bits de la secuencia codificada. Incluir una matriz **H** para el código Hamming del apartado anterior. Observar cómo, en ausencia de errores, el síndrome es siempre **0** (para ello, simplemente igualar `receivedBits` con `codedBits`).

Usando ahora la función de MATLAB© `bsc(...)`[2], añadir errores aleatorios a la secuencia codificada: considerar un valor para p de 0,01. Observar cómo el síndrome ahora no siempre es **0**. Completar las líneas para que el código avise de que se ha producido un error (detección): falta un término en la comparación.

Si queremos realizar, además, corrección, hemos de tener en cuenta que los códigos Hamming corrigen errores de peso 1 ($d_{min} = 3$). Hay 7 síndromes distintos de **0** para $m = 3$. La estrategia de corrección consistirá en asociar cada síndrome posible al patrón e correspondiente, y, una vez identificado, usarlo para corregir. Téngase en cuenta que la relación $\mathbf{s} \leftrightarrow \mathbf{e}$ depende de la matriz **H** escogida, por lo que será necesario crear la tabla en función de ello. Observar la forma en que el código dado realiza dicha operación. ¿Qué función concreta realizan los comandos `de2bi` y `bi2de` en esta parte?

Obsérvese cómo se compara la secuencia enviada con la secuencia decodificada sin corregir y se cuentan los errores, y cómo se hace lo mismo con la secuencia corregida (completar los términos de comparación que faltan). ¿Mejora la situación? ¿En qué casos la corrección no tiene éxito? Probar con valores distintos para la probabilidad p del BSC.

¿Qué relación se observa entre los tres valores de error que se muestran? ¿En función del número de palabras erróneas y del número de bits erróneos antes de corregir podríamos decir algo sobre lo que nos encontraremos después de corregir?

Opcionalmente, se sugiere ampliar el código para observar cuántas palabras erróneas no se detectan.

[1] El término `COMPLETE` aparece al lado de las líneas del código donde faltan elementos que hay que completar.

[2] Ejecutar >> `help bsc` y observar cómo funciona y cuáles son sus parámetros.

Práctica 3.3: códigos convolucionales.

Esta práctica permite hacerse una buena idea de las técnicas más habituales para evaluación y caracterización de los sistemas de comunicación digital, empleando la simulación y el apoyo matemático de las cotas de error. En este caso, nos limitamos a combinar los subsistemas estudiados hasta el momento de modulación y de codificación de canal (dentro de la capa física), aunque dichas técnicas se pueden aplicar al esquema de un sistema de comunicación digital completo, pudiendo comprender incluso las capas sobrepuestas a la capa física.

PARTE 1: BER experimental con codificación convolucional: simulación de Monte Carlo.

En esta parte, con ayuda del código de prueba Tema3_3a_lab.m, se tratará de examinar la mejora en la tasa de error de bit en función del parámetro E_b/N_0 al emplear un esquema de modulación conocido (8-PSK con codificación gray, y demodulación y decisión duras) y un código convolucional, para un canal AWGN. El método que se plantea es el llamado de simulación de *Monte Carlo*. Su base consiste en caracterizar un modelo matemático del transmisor, del canal y del receptor, y ponerlo a funcionar según una dinámica aproximada a lo que se tendría en la realidad (dentro de los límites de dicho modelo): se generan bits de información, se procesan en el transmisor, se modifican de acuerdo al tipo de canal y se procesan en el receptor para obtener una estimación de los bits enviados. Como estadística resultado, normalmente se calcula el número de bits de información erróneos, cantidad que se divide entre el número del total de bits de información enviados. Así se obtiene la denominada tasa de error binaria (BER, *bit error rate*), que constituye la aproximación experimental (simulada) a la probabilidad de error binaria del sistema caracterizado.

$$\text{BER} = \frac{\text{n}^o \text{ de bits erróneos}}{\text{n}^o \text{ total de bits}}$$

Para que la aproximación de Monte Carlo tenga validez estadística, es preciso que ocurra un número significativo de errores de bit (recordar la *ley de los grandes números* que relaciona teoría de la probabilidad y estadística). Observar que, en el código propuesto, se da un valor de 1000, aunque, por regla general, 100 suele ser suficiente. Se puede demostrar que, en esas condiciones, la estimación estadística es bastante fiable.

Aumentar el número objetivo de bits erróneos antes de proceder a calcular la BER definitiva aumenta la fiabilidad de la estimación, pero hay un compromiso fundamental. Si minErr es el número mínimo de bits erróneos que se quieren obtener antes de calcular la BER, y sabemos que dicha tasa está en el entorno de un cierto valor BER_0, es fácil darse cuenta de que al menos habrá que ejecutar la simulación hasta haber enviado un número de bits de información del orden de

$$\text{n}^o \text{ de bits información}_{min} = \frac{\text{minErr}}{\text{BER}_0}$$

Si minErr es 100 y estamos en el entorno de BER de 10^{-8}, eso significa que, como mínimo, nuestro simulador debe procesar 10^{10} bits de información, lo que en algunos casos puede resultar impracticable para nuestros recursos computacionales habituales (sobre todo, si tenemos restricciones de tiempo). Esto da especial sentido a los cálculos que realizaremos en la segunda parte de la práctica, en que se sortea este tipo de dificultad mediante el cálculo de cotas.

Examinar con atención el código y familiarizarse con cada una de sus partes. En los comentarios que lo acompañan, se puede encontrar una descripción de sus partes más significativas y su función. Asimismo, rellenar las partes que faltan junto a las etiquetas `COMPLETE`. Si todo funciona bien, al ejecutarlo, se obtendrá una gráfica comparativa entre el caso 8-PSK sin codificación de canal y 8-PSK + codificación convolucional.

¿Qué se puede decir de la tendencia de las curvas? ¿Qué ocurre en la región de baja relación señal-ruido?

PARTE 2: análisis de códigos convolucionales; propiedades y cotas.

En esta segunda parte, se trata de observar cómo el cálculo de cotas, basándose en las propiedades de los sistemas involucrados, puede ayudar a resolver el problema planteado anteriormente: ¿cómo obtener una estimación de BER útil cuando estamos en la zona de baja tasa de error? En este sentido, el carácter asintótico de las cotas será nuestro caballo de batalla, lo cual ya plantea algunos límites: ¿qué validez tendrán las cotas en la región de baja relación señal-ruido?

Al igual que en el caso precedente, se van a obtener resultados estimados para 8-PSK sin codificación de canal y para 8-PSK con el mismo codificador convolucional. De nuevo, tendremos codificación gray en la modulación, y

demodulación y decisión duras. El archivo Tema3_3b_lab.m contiene un código incompleto que se debe analizar en detalle y completar para poder obtener un resultado congruente.

Dado que se realiza demodulación y decisión duras, y tenemos PSK con codificación gray, podemos considerar que, entre la entrada al modulador y la salida del demodulador, tenemos un canal BSC equivalente cuya probabilidad de transición de bit p es la probabilidad de error de bit de la modulación, que, en el código, se calcula con ayuda de la función de MATLAB© `berawgn`. Observar cómo funciona, sus parámetros y el resultado que devuelve. Como consecuencia, la cota de error para el sistema con codificación convolucional se puede calcular usando el BSC(p) equivalente y la fórmula (Lin & Costello-2004, p.531)

$$P_b \approx \sum_{d=d_{free}}^{\infty} B_d \cdot \left(2\sqrt{p\,(1-p)}\right)^d$$

donde B_d es el coeficiente enumerador de los sucesos de error de peso d para el codificador convolucional. Los valores de dicho coeficiente no son fáciles de obtener sin cálculos algebraicos complejos, pero los podemos aproximar con ayuda de la función de MATLAB© `distspec`, que da el valor de dfree del código (definido por su trellis) y el número de bits de información erróneos asociados a los sucesos de error para cada patrón de error de peso d (`distspec.weight`).

Completar el código, ejecutarlo y comprobar que los resultados son congruentes con los obtenidos en el apartado anterior. Observar que la tendencia es que coincidan para valores altos de la relación señal-ruido. ¿Qué sucede para valores bajos de dicha relación?

Práctica 3.4: turbocódigos.

Descomprimir el archivo .zip `Tema3_4_lab.zip` y, desde Matlab©, hacer doble click sobre `Tema3_4_lab.mdl`. Observar la estructura del modelo. Haciendo doble click sobre el turbocodificador y el turbodecodificador se verán los bloques de los que se componen. Abrir el diálogo de los codificadores convolucionales del turbocodificador y observar su definición en forma de trellis (`poly2trellis(...)`). ¿De qué tipo de codificador convolucional se trata? ¿Qué memoria tiene? ¿Cuáles serían sus valores de n y k? Observar cómo funciona el bloque `Bit Reordering`; a tenor de ello, ¿cuál sería la tasa del turbocódigo?

En la parte del decodificador, los bloques `APP Decoder` son los que hemos llamado SISO, y actualizan la propabilidad a posteriori de los bits de información a partir de la información *a priori* obtenida iterativamente y de las estimaciones ruidosas de los símbolos que llegan del canal. Observar que siempre hay una salida que no se usa, ya que en ella se computan las estimaciones probabilísticas, las LLR, de los bits codificados, y esta información no se necesita en este tipo de concatenación.

Finalmente, si pulsamos sobre `Model Parameters`, se abre un diálogo en el que podremos configurar la longitud del permutador (no todos los valores son permitidos), la relación señal-ruido y el número total de iteraciones.

PARTE 1: localización de las zonas de BER, efecto de la longitud de bloque.

En esta parte, se trata de realizar simulaciones para distintos valores de la relación señal-ruido en el entorno de 1 dB (en pasos de 0,1 dB). Se tabularán los valores obtenidos y se representarán en una gráfica log-log para caracterizar las tres zonas de BER y sus límites: la zona de baja SNR y alta BER, la zona de caída brusca (*waterfall region*), y la zona del pedestal de error (*error floor*). ¿Para qué E_b/N_0 se produce la caída brusca? ¿Qué valor estimado tiene el pedestal de error?

Cambiar el valor del tamaño de bloque por un número superior. Observar que, salvo que se tenga suerte, se producirá un error donde se detallan las longitudes de bloque admisibles. Elegir una superior a 2048. Repetir el experimento del párrafo anterior. ¿Hay alguna diferencia apreciable?

PARTE 2: efecto de los códigos constituyentes.

En esta segunda parte, se trata de examinar cómo influyen los códigos constituyentes en el comportamiento del turbocódigo. Para ello, volvemos a poner 2048 como la longitud de bloque, y cambiamos la definición del código convolucional: ello deberá hacerse tanto en los dos codificadores convolucionales como en los bloques APP Decoder para que todo sea congruente. La modificación consistirá en quitar los caracteres `",13"` que aparecen al final de los parámetros de `poly2trellis(...)`. Realizar una simulación para un valor de E_b/N_0 para el que se tengan resultados en el apartado anterior. Repetir la simulación con la segunda longitud de bloque. Comparar los resultados. ¿Qué diferencia hay entre el código convolucional original y el modificado?

Ir a la línea de comando de MATLAB© y, usando `distspec(...)`, comparar el espectro de distancias entre el código convolucional original (con `",13"` incluido) y el código convolucional de la práctica anterior (definido por los vectores en octal 171 133, sin realimentación). ¿Cuál sería más potente utilizado aisladamente? Sin embargo, si se sustituye el código convolucional del sistema concatenado por el supuestamente más potente, no está garantizado que los resultados sean mejores. Realizar la prueba, teniendo cuidado de definir adecuadamente el nuevo codificador convolucional en los bloques correspondientes del turbocodificador y del turbodecodificador. Lanzar alguna simulación para alguna E_b/N_0 ya caracterizada en los casos precedentes y comparar los resultados. ¿Hay alguna mejora? ¿Y empeoramiento?

Práctica 3.5: códigos *Low Density Parity Check* (LDPC).

En la ventana de comando de MATLAB©, teclear lo siguiente:

```
>> modelName = 'comm/DVBS2LinkIncludingLDPCCodingSimulinkExample';
>> openExample(modelName);
```

Observar el esquema que aparece y sus bloques constitutivos. Se trata de parte de la capa física de un estándar de televisión digital. Los bloques `BCH Encoder` y `BCH Decoder` realizan operaciones de protección frente a errores de tipo ráfaga, y complementan al código LDPC, que es el que nos ocupa aquí.

PARTE 1: la matriz H.

En esta parte, se examinará la definición del código LDPC empleado. Si se abre el diálogo del bloque `LDPC Encoder`, se observará que el código está definido a través de la variable `dvb.LDPCParityCheckMatrix`, que nos da una cierta matriz **H**. Ir a la línea de comandos de MATLAB©, y ejecutar `spy(...)` introduciendo como parámetro dicha variable: `spy(...)` representa gráficamente una matriz dispersa, para las que en MATLAB© no se almacenan los "0", sólo las posiciones donde hay valores no nulos. Una vez obtenida la representación gráfica de la matriz, con puntos en las posiciones donde hay valores no nulos, intentar determinar las dimensiones de **H**. Usar el zoom para observar el carácter disperso de la misma.

PARTE 2: codificación y decodificación.

Abrir el diálogo de `Model Parameters`. En este caso, podemos cambiar el tipo de modulación empleado (elegir una según preferencias, o mantener la que viene por defecto: en todo caso, anotarlo), la E_s/N_0 y el número de iteraciones en el algoritmo de decodificación del código LDPC (dejar el valor por defecto). Observar que los valores de salida del decodificador son decisiones "duras", en forma de valores binarios; sin embargo, el algoritmo de decodificación es de tipo probabilístico, y acepta los valores estimados por parte del bloque `Soft-Decision Demodulator`. El algoritmo de decodificación está basado en el de paso de mensajes que mencionamos en clase. Si buscamos en la ventana de ayuda `fec.ldpcdec`, obtendremos mucha información sobre cómo funciona internamente este decodificador.

En esta parte de la práctica, se trata de partir de una E_s/N_0 de 0 dB, que se irá incrementando de 0,1 dB en 0,1 dB. Para cada valor, se lanzará una simulación y se anotará el valor de BER final del código LDPC (ignorar los resultados del código BCH). Mientras está en marcha la simulación, se puede pulsar sobre el módulo `RX Constellation` para visualizar la constelación recibida: observar el efecto del ruido sobre la misma.

Como en la práctica sobre turbocódigos, representar los valores de BER frente a E_s/N_0 en una gráfica log-log y localizar la zona de caída brusca, propia también de los códigos LDPC. Observar, según progresan las simulaciones, si se llega asimismo a una zona de *error floor* o no.