# Codificación de canal I: introducción y códigos de bloque

Francesc Tarrés
Margarita Cabrera

PID_00185032

Universitat Oberta de Catalunya

# Índice

# Introducción

Codificar es definir con precisión el conjunto de símbolos con los que se enviarán los mensajes de emisor a receptor. Esta definición tan genérica admite que el conjunto de símbolos pueda elegirse con el objetivo de solventar problemas muy distintos. En efecto, el diseño de códigos eficientes puede utilizarse para resolver tres problemas clásicos de los sistemas de comunicación: la compresión de los datos, la protección frente a eventuales errores de canal y la encriptación de la información para que no pueda ser interpretada por un observador externo. En muchos sistemas de comunicación, buscaremos simultáneamente la resolución de más de uno de los problemas mencionados, por lo que es posible aplicar estas técnicas en cascada. Así, por ejemplo, la transmisión de un vídeo puede suponer en primer lugar la codificación de la fuente mediante un compresor MPEG, la encriptación de los datos recibidos mediante un sistema de claves públicas y privadas para limitar el acceso a la información a quien haya pagado por ella (acceso condicional) y finalmente la codificación de los datos para su transmisión y protección frente a las características del canal.

En este módulo nos ocuparemos de este último problema, es decir, de cómo podemos establecer una comunicación robusta de tal forma que el receptor sea capaz de percatarse de que los datos recibidos han sido alterados por el canal, se han producido errores e incluso que en algunos casos sea capaz de deducir cuál era la información original que se había transmitido. Existen muchas causas que pueden originar la aparición de ruido, distorsiones y generar errores en el canal de comunicaciones. Entre estas causas debemos destacar la atenuación del canal, las interferencias producidas por otros sistemas de comunicación o fenómenos naturales, la propagación multicamino, el ruido, etc. En todos estos casos se trata de problemas originados en el canal de comunicaciones, por lo que la gravedad de los mismos y las técnicas utilizadas para solventarlos dependerán de la naturaleza del sistema de comunicaciones. Es muy diferente un sistema de comunicaciones por cable, en el que la información está fuertemente protegida de interferencias y ruidos, que un sistema de comunicaciones por satélite, en el que deben cubrirse distancias muy grandes sin utilizar repetidores y en un canal sometido a distintos fenómenos atmosféricos.

Por ello, las técnicas y estrategias utilizadas para proteger la información frente a eventuales errores se conocen con el nombre de *codificación de canal*, ya que su objetivo es acondicionar la información a las características del canal, para realizar una comunicación fiable.

Al diseñar estrategias para proteger la información frente a eventuales errores, no tenemos demasiado margen de maniobra. En efecto, la única solución po-

sible es que el receptor pueda distinguir entre un mensaje producido por el emisor y un mensaje en el que se han producido errores. Para ello, es necesario que introduzcamos elementos adicionales en la información que nos proporcionen pistas sobre la integridad de los mensajes. Esta introducción de información adicional recibe el nombre de *redundancia estructurada* y supone que el número de bits de información que finalmente se transmiten por el canal sea superior al mínimo necesario. Así pues, la inserción de códigos de protección frente a errores supone la introducción de bits adicionales en el transmisor. Estos bits representan un aumento del ancho de banda de la señal que transmitimos. El precio que debe pagarse para proteger el mensaje es, por tanto, aumentar el ancho de banda de la información.

El módulo empieza con unos conceptos elementales sobre métodos para introducir redundancia estructurada en los mensajes y una discusión genérica sobre las estrategias para la detección y la corrección de errores. Se establecen y discuten aquellos escenarios de aplicación en los que puede ser conveniente realizar una detección o una corrección de los errores. También se presentan algunas técnicas elementales de retransmisión de datos. Posteriormente, se cubren los elementos matemáticos básicos para los códigos de bloque que constituyen el núcleo de este módulo. El objetivo principal es proporcionar una idea general sobre los mecanismos de codificación y decodificación. Se presentarán algunos elementos y herramientas matemáticas básicas que no pretenden ser completas sino simplemente ilustrar parte de la tecnología implicada en el diseño de los códigos de protección de errores. Los detalles matemáticos de algunos de los códigos utilizados en la práctica son excesivamente complejos para los objetivos de este texto, por lo que muchos de los resultados se presentan sin demostración ni justificación.

Los códigos de bloque no son los únicos códigos utilizados para la corrección de errores. Existen otras variantes como los códigos convolucionales o los turbo códigos, cuyos principios de funcionamiento son algo más complejos y que se dejan para cursos de comunicaciones más avanzados.

# Objetivos

Los objetivos a los que debe llegar el estudiante al finalizar este módulo son los siguientes:

1. Comprender la necesidad de proteger la información frente a errores del canal.

2. Distinguir entre sistemas y aplicaciones que requieren la detección o la corrección de los errores.

3. Conocer las técnicas básicas de redundancia estructurada y paridad.

4. Disponer de capacidad para realizar cálculos básicos de probabilidades de detección y corrección de errores.

5. Comprender el compromiso entre ancho de banda, potencia transmitida, probabilidad de error y tasa del código.

6. Conocer los principios y elementos matemáticos básicos de los códigos de bloque.

7. Conocer las características y el proceso de generación de los códigos de Hamming.

8. Conocer las propiedades y características de los códigos BCH y de Reed-Solomon.

# 1. Redundancia estructurada: conceptos básicos

Introducir redundancia en un código es añadir un conjunto de bits que posteriormente nos permitan verificar que el grupo de bits que recibimos (al que llamaremos *palabra* o *palabra código*) concuerda con la transmitida. Se dice que la redundancia se introduce de forma estructurada porque los bits adicionales se determinan utilizando procedimientos bien definidos y conocidos tanto por el receptor como por el transmisor.

## 1.1. Ejemplo 1. Códigos de paridad simple

El ejemplo más sencillo de introducción de redundancia estructurada son los códigos de paridad simple. En la figura 1 se muestra cómo se determina la paridad simple para diferentes palabras código. En nuestro ejemplo, las palabras código tienen originalmente una longitud de 6 bits, por lo que después de añadir la redundancia tendremos palabras con una longitud de 7 bits. Para añadir el bit de paridad se utiliza el criterio de que el número total de unos de cualquier palabra sea par. Así, si una palabra ya tiene originalmente un número par de unos, el bit de paridad tomará el valor cero. En cambio, si en la palabra original el número de unos es impar, el bit de paridad tomará el valor uno. De este modo, después de introducir el bit de redundancia podemos garantizar que todas las palabras código tienen un número par de unos. El código descrito se denomina de *paridad par*. También podría definirse un código de paridad impar, en el que el número de unos de la palabra codificada siempre es impar.

La redundancia se ha introducido siguiendo un procedimiento sistemático (el número total de unos debe ser par) que confiere una propiedad única a todas las palabras código. Esta propiedad es la que aprovecha el receptor para verificar que no se ha producido ningún error en la transmisión. En efecto, si la palabra recibida tiene un número par de unos, el receptor la dará por buena, mientras que, si el número de unos es impar, el receptor podrá asegurar que se ha producido algún error (al menos uno) durante la transmisión.

Este procedimiento resulta eficiente siempre que el número de errores que se produzca sea un número impar. En efecto, si se producen dos errores, la palabra código volverá a tener un número par de unos, por lo que no podrá detectarse como palabra incorrecta. Lo mismo ocurre cuando se produce cualquier número par de errores. Todos estos casos se corresponden con situaciones de errores que no pueden ser detectadas por nuestro código. En general, para cualquier código, siempre existirán patrones de errores que

no podrán ser detectados. Por lo tanto, se dice que el código de paridad simple permite detectar cualquier número impar de errores en una palabra código.

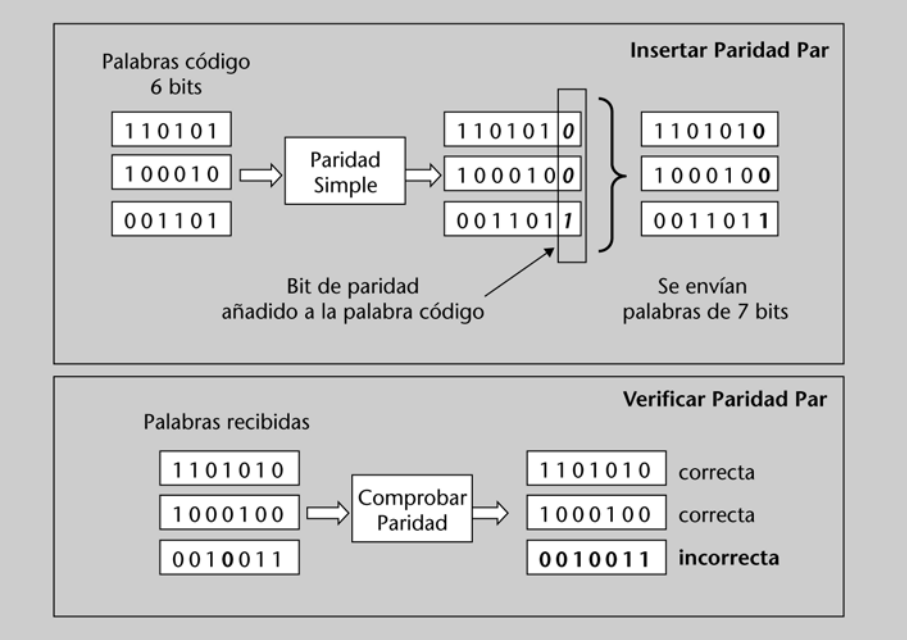


Figura 1. Códigos de paridad par: inserción y verificación

**Problema 1**

Supongamos que en un determinado sistema de comunicaciones la probabilidad, $p$, de cometer un error en un bit es $p = 10^{-4}$. Las palabras código originales son de 7 bits y se introduce un bit de paridad par adicional. Determinad la probabilidad de que en una palabra código se hayan producido errores que no puedan ser detectados.

**Solución**

La probabilidad de que se hayan producido errores que no puedan ser detectados por el código de paridad es:

$$P_{\text{error no detectado}} = P_{2\text{ errores}} + P_{4\text{ errores}} + P_{6\text{ errores}} + P_{8\text{ errores}}$$

Es decir, es la probabilidad de que se produzcan dos errores más la de que se produzcan 4, 6 u 8, ya que cualquier número impar de errores será detectado correctamente.

Para determinar la probabilidad de que se produzcan dos errores en la palabra código, debemos tener en cuenta todas las posibles situaciones en las que tendremos 2 bits erróneos en una palabra de 8 bits y ponderar cada una de estas situaciones por la probabilidad de que se produzca. Así:

$$p_{2\,errores} = \binom{8}{2} \cdot p^2 (1-p)^6$$

Donde el número combinatorio nos indica la totalidad de casos en los que se producen dos errores en una palabra de 8 bits, siendo $p^2$ la probabilidad de que se produzcan dos errores en estas posiciones y $(1-p)^6$ la probabilidad de que no se produzcan errores en el resto de las posiciones. Teniendo en cuenta la expresión anterior, la probabilidad de que se produzcan errores en una palabra y que no sean detectados por el código será:

$$p_{\text{error no detectado}} = \binom{8}{2} \cdot p^2 \cdot (1-p)^6 + \binom{8}{4} \cdot p^4 \cdot (1-p)^4 + \binom{8}{6} \cdot p^6 \cdot (1-p)^2 + \binom{8}{8} \cdot p^8$$

**Recordad que...**

... el número combinatorio $\binom{8}{2}$ representa todas las posibles combinaciones para tomar dos elementos en una palabra de 8 bits. El cálculo se realiza teniendo en cuenta la siguiente expresión:

$$\binom{n}{m} = \frac{n!}{m! \cdot (n-m)!}$$

Particularizando para el valor $p = 10^{-4}$, obtenemos:

$$p_{\text{error no detectado}} = \frac{8!}{2!6!} \cdot 10^{-8} \cdot (1-10^{-4})^6 + \frac{8!}{4!4!} \cdot 10^{-16} \cdot (1-10^{-4})^4 + \frac{8!}{6!2!} \cdot 10^{-24} \cdot (1-10^{-4})^2 + 10^{-32}$$
$$= 2{,}7983 \cdot 10^{-7}$$

Observad que únicamente el primer término de la suma es significativo.

## 1.2. Ejemplo 2. Códigos rectangulares

Los códigos rectangulares, también conocidos como *códigos de producto*, son una variante directa de los códigos de paridad. Esta variante permite visualizar, de forma muy sencilla, una estrategia para la corrección de errores. Para aplicar un código rectangular, los bits del mensaje original deben organizarse en una matriz. En la figura 2 se muestra un mensaje original de 20 bits organizado en una matriz de 5 columnas y 4 filas. Los bits de redundancia se calculan como un código de paridad simple, aplicándolo primero a las filas y después a las columnas (o viceversa). En la figura se muestran todos los bits resultantes en los que la última columna y la última fila se corresponden a los bits de paridad. Los bits pueden transmitirse en el orden que se desee, siempre que el transmisor y el receptor se pongan de acuerdo. Así, podremos transmitir todas las filas, una detrás de otra. El receptor irá situando los bits recibidos en una matriz de 6 columnas por 5 filas, por lo que los bits de información y de redundancia estarán dispuestos en el mismo orden que en el transmisor.

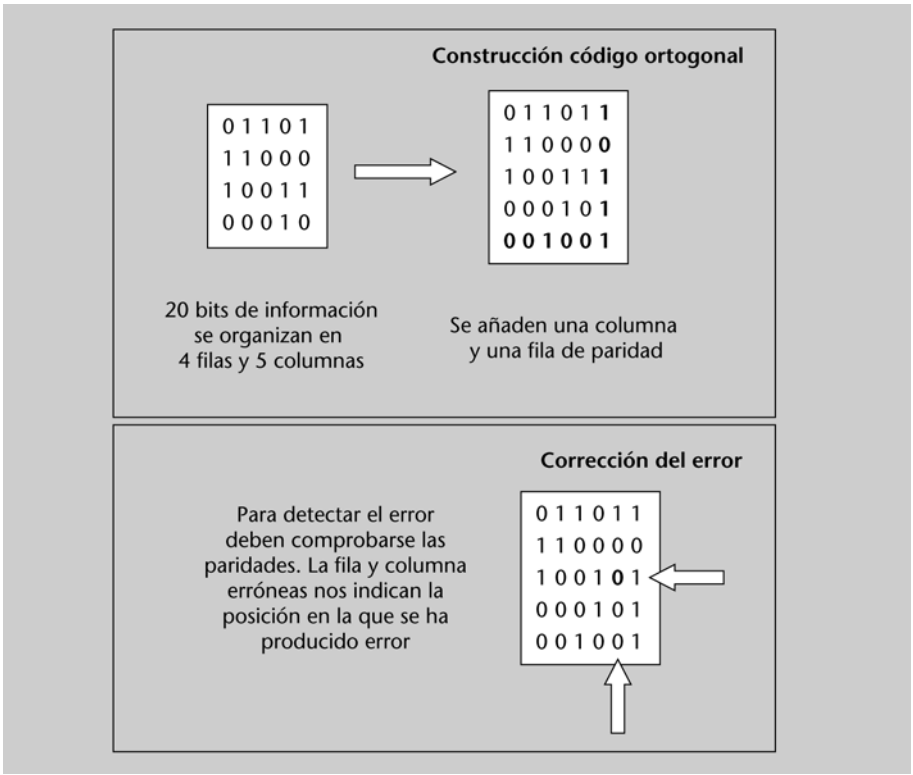


Figura 2. Construcción y corrección de errores en un código ortogonal

En esta misma figura se muestra cómo puede realizarse la corrección de un bit erróneo. El receptor realiza la comprobación de paridad de cada una de las filas, *detectando* como errónea la fila en la que se ha producido el error. Posteriormente, al realizar la comprobación de la paridad de cada una de las columnas, *detectará* la columna en la que se ha producido el mismo error. De esta forma, podemos aprovechar esta información para saber la fila y la columna en la que se ha producido el error. Es obvio que, si sabemos la posición en la que se ha producido el error, su corrección consiste simplemente en cambiar su valor.

El código propuesto sólo puede garantizar la **corrección de un error en un bit**. Es fácil pensar situaciones en las que se produce más de un error y en las que sería posible determinar las posiciones en las que se ha producido (siempre que la columna y la fila de los dos errores sean distintas). No obstante, si se producen en una misma fila o columna no será posible detectarlos. En estos casos, se dice que los errores superan las capacidades de corrección del código.

### 1.3. Tasa del código

Los códigos de paridad y los códigos ortogonales son muy simples, pero nos permiten ilustrar algunas de las definiciones sobre los códigos de protección de error que tienen validez general. Tomando como referencia la figura 3 podemos ver que, en general, a partir de un paquete de *k* bits de información (mensaje original) se añaden *r* bits de redundancia. Los bits de redundancia también reciben a menudo el nombre de *bits de paridad*. El número total de bits de cada palabra código es la suma de los bits de redundancia más los bits del mensaje original n = k + r. Un código con estas características se identifica como código (n, k). La *tasa de redundancia* de un código se define como el cociente entre los bits de redundancia y los bits totales (n – k/n) y proporciona una idea del porcentaje de bits de redundancia que contiene un código. Análogamente, la *tasa de un código* se define como el cociente entre el número de bits del mensaje y el número de bits totales de una palabra código (k/n). La tasa del código nos da una idea de la relación entre la información útil y la información total que contienen los mensajes. Así por ejemplo, un código con una tasa 2/5 nos indica que contiene dos bits de información útil por cada 5 bits que recibimos. En la figura 3 se representan de forma esquemática estos conceptos.

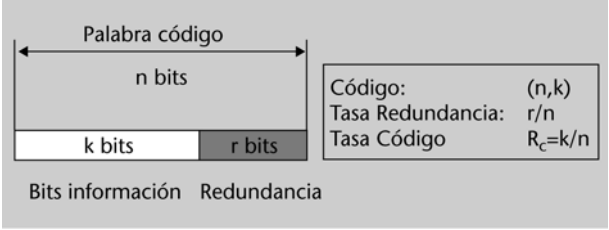


Figura 3. Definiciones básicas de redundancia y tasa de código

La tasa del código nos da una idea del coste que tiene proteger la información de los posibles errores de canal. En efecto, al añadir los bits de redundancia deberemos transmitir más bits que los estrictamente necesarios. Esto significa que, si deseamos realizar la transmisión en tiempo real, deberemos transmitir un total de n = k + r bits en el mismo tiempo que antes transmitíamos *k* bits, lo que supone que necesitaremos un mayor ancho de banda. El aumento del ancho de banda debido a la introducción de la redundancia está directamente relacionado con la tasa de bits. En efecto, si antes debíamos transmitir *k* bits en un tiempo T y ahora debemos transmitir *n*, la duración de un bit pasa de ser T/k a T/n, por lo que el ancho de banda aumenta aproximadamente en un factor n/k. Así pues, el aumento del ancho de banda coincide con el inverso de la tasa del código, que también se conoce como *factor de expansión*.

La *tasa de un código* es la relación entre el número de bits de información y el número de bits totales que contiene:

$$R_c = k / n$$

El inverso de la tasa de un código da una idea de cómo el uso del código aumenta el ancho de banda de la señal y se denomina *factor de expansión* de un código:

$$B = \frac{W_{coding}}{W_{no\ coding}} = \frac{1}{R_c} = \frac{n}{k}$$

Veremos cómo se aplican todos estos conceptos en el siguiente problema para los códigos de paridad y los códigos ortogonales.

**Problema 2**

Determinad la notación del código y las tasas de código y redundancia para los ejemplos 1 y 2 presentados en este apartado.

**Solución**

Para el código de paridad simple, hemos visto que para cada seis bits de información útil (k = 6) añadíamos un bit de paridad (r = 1). El número total de bits de una palabra código es de n = 7. Teniendo en cuenta estos valores, el código puede designarse como (7,6) con una tasa de redundancia de 1/7 y una tasa de código $R_c = 6/7$.

Para el código ortogonal, hemos visto que la información útil se organizaba en una matriz de 4 filas por 5 columnas; por tanto, cada palabra código (o bloque de información) tiene un total de 20 bits útiles (k = 20). Al calcular las paridades por filas y por columnas se añaden 10 bits de información (r = 10; 4 filas + 6 columnas). Con estos resultados obtenemos que el código debe denotarse como (30,20), su tasa de redundancia es 10/30 = 1/3 y la tasa del código es de $R_c = 20/30 = 2/3$.

## 2. Estrategias para el control de errores

La introducción de redundancia de forma estructurada en una palabra código permite que el receptor pueda detectar o corregir, según el caso, la presencia de errores. En este apartado veremos las posibles estrategias de control de error que pueden utilizarse en función de las características del canal de comunicaciones o de la aplicación, es decir, intentaremos justificar en qué tipo de situaciones resulta más conveniente utilizar técnicas de detección de errores o técnicas de corrección.

Como primera consideración, cabe aclarar que *detectar* la presencia de un error consiste en analizar la palabra código recibida y determinar que no cumple las reglas mediante las que se ha añadido la redundancia. Por tanto, sabemos que la palabra es incorrecta pero no conocemos en qué posición se ha producido el error.

En cambio, la *corrección* de errores implica no sólo determinar que la palabra código es incorrecta, sino también conocer las posiciones en las que se han producido los errores para poder proceder a corregirlos.

A partir de estos razonamientos, parece lógico que la corrección de errores será siempre más compleja que la detección, donde entendemos por *compleja* que requiere un mayor número de bits de redundancia y que aumenta la complejidad computacional para realizar la codificación y la decodificación. La necesidad de un mayor número de bits de redundancia también puede interpretarse como un aumento del ancho de banda de la señal que se va a transmitir.

Deben distinguirse, pues, dos estrategias principales para controlar los errores en el canal. La primera se basa en utilizar códigos que permitan que el receptor *detecte* los errores que se han producido en el canal solicitando la retransmisión de aquellas palabras que se han recibido de forma incorrecta. Esta técnica se conoce con el acrónimo inglés ARQ (*automatic repeat request* o también *automatic retransmission query*), que indica que el receptor pide la retransmisión del mensaje original cuando detecta que se ha producido un error.

Esta estrategia tiene como principales ventajas:

- Los códigos de detección de errores son fáciles de calcular y detectar.
- La redundancia añadida al mensaje es menor que para un código de corrección.
- El aumento del ancho de banda debido a la introducción de la redundancia también es menor.

Los principales inconvenientes son:

- Algunas aplicaciones, como en la transmisión de audio y vídeo, no se aceptan retardos en la recepción de los paquetes, ya que estos retardos podrían interrumpir el flujo continuo del reproductor.

- En muchas aplicaciones, el canal de retorno no está disponible.

Las técnicas **ARQ** tienen diversas variantes y se utilizan con profusión en la descarga de archivos por Internet (TCP/IP).

La otra alternativa se conoce con el nombre de **FEC** (*forward error correction*) y utiliza códigos de corrección de errores que permiten que el receptor pueda restaurar por sí solo la información original en el caso de que se hayan producido errores (siempre que los errores estén dentro de las capacidades correctoras del código).

La necesidad del canal de retorno impone una restricción muy importante a las estrategias de control ARQ. En efecto, pensemos por ejemplo en aplicaciones como la grabación de señales de vídeo o audio en soporte óptico o magnético. Una vez que la información ha sido grabada, si han aparecido errores en la señal registrada (generalmente debido a la degradación del soporte), es imposible pedir la regrabación de la información original. Por eso, en todos los sistemas de grabación de audio y vídeo se utilizan técnicas FEC que permiten la corrección de los errores en el propio decodificador. Los sistemas digitales de registro de señales de audio (CD-audio *compact disc audio*, DAT *digital audio tape*, minidisc, DVD-Audio *digital versatile disc-audio*, etc.) o de registro de vídeo (DVD-Vídeo) utilizan distintas variantes de códigos con estrategias FEC que se analizarán más adelante en este módulo. Otro ejemplo típico son los sistemas de difusión (televisión digital, audio digital) que tampoco permiten que el receptor resolicite la transmisión de las palabras recibidas de forma incorrecta. Así pues, la Televisión Digital Terrestre (DVB-T, *digital video broadcasting-terrestrial*) o la radio digital DAB (*digital audio broadcasting*) también utilizan códigos con estrategias FEC. Otro ejemplo típico es la difusión de contenidos audiovisuales por Internet (*streaming*). En este caso, el motivo principal para utilizar estrategias FEC es que la propia naturaleza de las señales audiovisuales requiere la reproducción continua sin que aparezcan las posibles interrupciones que puede introducir la solicitud de retransmisiones.

Elegir si en un determinado sistema de comunicaciones es más conveniente utilizar estrategias de corrección ARQ o FEC es un problema complejo que depende de muchos factores, entre los que destacamos, a modo de resumen, los siguientes:

**Complejidad del codificador/decodificador.** Algunos sistemas para la corrección de errores FEC pueden tener complejidades computacionales consi-

derables, por lo que sólo pueden ser utilizados en equipos terminales de coste elevado que puedan permitirse incorporar procesadores avanzados. En cambio, las estrategias ARQ tienen una implementación más simple que las FEC, por lo que pueden ser utilizados en equipos más económicos.

**Aumento del ancho de banda.** En general, para realizar la corrección de errores es necesario introducir más redundancia que para permitir su detección. Esto supone que el aumento de ancho de banda debido al código suele ser mayor cuando se utilizan estrategias FEC.

**Probabilidad de error del canal.** El número de errores que se producen en el sistema de comunicación (antes de su corrección) puede ser muy elevado, lo que hace que las técnicas ARQ resulten muy ineficientes (ver como ejemplo el Problema 3).

**Contenido del mensaje.** En general, cuando se transmiten archivos de datos podemos permitirnos retardos en los bloques para los que solicitamos la retransmisión. El receptor puede gestionar un *buffer* de memoria en el que se recompone el orden original de los bloques. En aplicaciones en las que intervienen señales de audio o vídeo, estos retardos no son admisibles, por lo que no suelen utilizarse técnicas de ARQ.

**Características del canal de comunicaciones.** Hemos visto que en los sistemas en los que no existe canal de retorno es necesario utilizar estrategias FEC. Los ejemplos más típicos son los sistemas de difusión audiovisual (como televisión y radio), el *streaming* de Internet y la grabación de datos en soportes ópticos o magnéticos. Cuando existe canal de retorno, pueden utilizarse distintas variantes de la estrategia ARQ.

Veremos dos ejemplos de cómo pueden incidir las características del canal en las estrategias utilizadas en las técnicas ARQ. En el primer caso, consideraremos un canal *full-duplex* y, en el segundo, un canal *half-duplex*. Recordamos estas definiciones en el recuadro de la derecha.

**Un canal es *half-duplex*...**

... cuando puede transmitir en los dos sentidos pero no de forma simultánea: primero debe realizarse en un sentido y luego en el otro. En un canal *full-duplex* se puede transmitir en ambos sentidos de forma simultánea.

### 2.1. Ejemplo 3. ARQ en canales *full-duplex*

En la figura 4 se muestra de forma esquemática una estrategia ARQ denominada *de repetición selectiva* (*selective repeat* ARQ) y que requiere un canal *full-duplex* (la comunicación puede realizarse en los dos sentidos de forma simultánea).

En este ejemplo, el transmisor va enviando continuamente los bloques del mensaje en el orden preestablecido y recibe el reconocimiento (ACK –*acknowledge*) del receptor. Cuando el ACK es negativo (NAK – *non-acknowledge*) el transmisor vuelve a repetir el bloque que se ha recibido de forma errónea. Si no aparecen errores en el canal, el retardo entre el transmisor y el receptor depende del tiempo de propagación del bloque. Si se producen errores (bloque número 4), el re-

ceptor deberá esperar a recibir este bloque de forma correcta si desea realizar la decodificación secuencial del mensaje. Observad que esta estrategia ARQ requiere que se realice una comunicación bilateral completa (*full-duplex*) entre el transmisor y el receptor. Es interesante comparar esta variante de ARQ con la que se introduce en el siguiente problema.

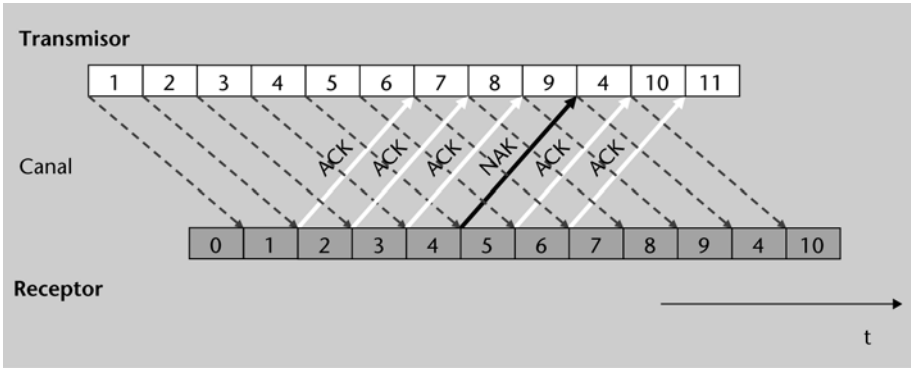


Figura 4. Ejemplo de una variante de estrategia ARQ (*selective repeat*). En este caso, se requiere disponer de un canal de comunicaciones *full-duplex*.

## 2.2. Ejemplo 4. ARQ en canales *half-duplex*

En este ejemplo se ilustra el mecanismo de control de errores de una variante de ARQ denominada Parada y Espera (*Stop&Wait*). En la figura 5 se muestra el esquema general de los paquetes enviados, su propagación por el canal y la respuesta de reconocimiento del receptor. Este mecanismo consiste en que el transmisor envía un paquete de datos y espera a recibir la confirmación de que se han recibido correctamente. Durante el tiempo de espera no se transmite más información, por lo que el canal puede ser *half-duplex*, de manera que el transmisor, una vez enviado el paquete, pasa (conmuta) a modo de recepción para esperar el ACK/NAK del receptor.

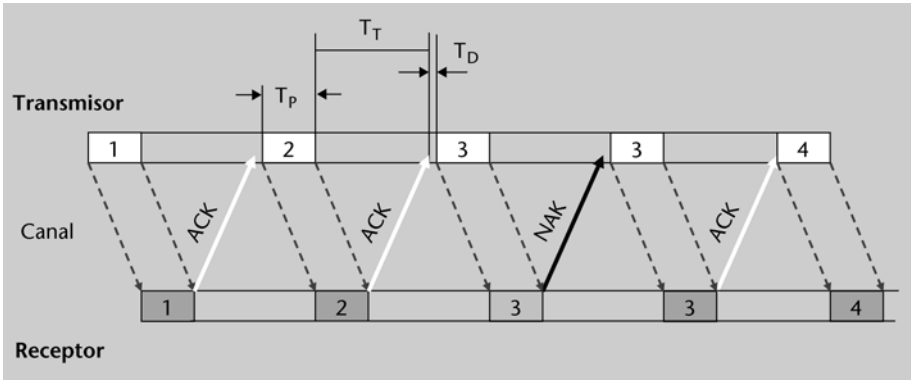


Figura 5. Esquema simplificado de la variante *Stop&Wait* para la estrategia de control de errores ARQ

En este ejemplo queremos ver los efectos que tienen en este esquema de control de errores algunos de los parámetros básicos como la distancia entre el emisor y el receptor y la probabilidad de error. Supongamos que el emisor envía paquetes de 210 bits (con los códigos de detección de error ya incluidos) a una velocidad de 100.000 bits/s, es decir, la duración de cada bit es de $10^{-5}$ s.

La distancia entre el transmisor y el receptor es de 300 km y podemos suponer, en primera aproximación, que el mensaje se propaga a la velocidad de la luz, que tomaremos como $c = 3\ 10^8$ m/s.

El paquete de ACK o NACK tiene una duración de $10^{-4}$ s, y el emisor responde con la repetición del paquete anterior o con un nuevo paquete sin retardo.

**a)** Determinaremos la tasa efectiva de transmisión de bits entre el emisor y el receptor, suponiendo que no se producen errores en los paquetes.

Para determinar la tasa efectiva de transmisión de bits entre el emisor y el receptor, debemos calcular el tiempo total que se invierte para enviar los 210 bits de cada paquete. Este tiempo debe incluir:

$T_P$ = Tiempo para transmitir los 210 bits a 100 kbits/s
$T_P = 210$ bits x $10^{-5} = 2{,}10\ 10^{-3}$ s
$T_T$ = Tiempo de ida y vuelta de un mensaje
$T_T = 2\text{x}300\text{x}10^3\text{(m)}/3\text{x}10^8$ (m/s) = $2\ 10^{-3}$ s
$T_D$ = Duración del paquete ACK + retardos de respuesta
$T_D = 10^{-4}$ s

Así, el tiempo total necesario para la transmisión de los 210 bits es $T = 4{,}2\ 10^{-3}$ s, por lo que la tasa efectiva de transmisión de bits es:

$$R = 210 \text{ bits} / 4{,}2\ 10^{-3} \text{ s} = 50.000 \text{ bits/s}$$

Para nuestro ejemplo, la tasa de bits se ha reducido a la mitad debido a la estrategia de control de errores.

**b)** Determinaremos ahora la tasa efectiva de transmisión suponiendo que la probabilidad de que se produzca algún error en un paquete sea de $10^{-2}$.

En este caso, debemos tener en cuenta que algunos paquetes de 210 bits se recibirán de forma incorrecta, por lo que volverán a transmitirse y el tiempo medio para la transmisión de un paquete aumenta.

Podemos calcular el tiempo medio para la recepción de un paquete mediante la siguiente ecuación:

$$T_m = T \cdot (1 - p) + 2 \cdot T \cdot p(1 - p) + 3 \cdot T \cdot p^2 \cdot (1 - p) + \ldots$$

Donde T representa el tiempo total calculado en el apartado anterior. Observad que el primer término de esta ecuación es el tiempo total por la probabilidad de que no se cometa error. El segundo término es el tiempo que tardaríamos en transmitir un paquete y volverlo a retransmitir por la probabilidad de que la primera vez se reciba incorrectamente y la segunda se reciba

correctamente. El tercer término corresponde a haber recibido dos paquetes con error y el tercero correctamente, y así sucesivamente. Para determinar el valor de la progresión anterior, reordenamos los términos:

$$T_m = T \cdot (1-p) \cdot \left[1 + 2 \cdot p + 3 \cdot p^2 + ...\right] = T(1-p) \cdot \sum_{k=0}^{\infty} (k+1)p^k = T(1-p) \cdot S(p)$$

Donde hemos definido S(p) como:

$$S(p) = \sum_{k=0}^{\infty} (k+1)p^k = \frac{d}{dp}\left(\sum_{k=0}^{\infty} p^{k+1}\right) = \frac{d}{dp}\left(\frac{p}{1-p}\right) = \frac{1}{(1-p)^2}$$

En esta última línea, utilizamos varias propiedades matemáticas que merece la pena comentar. En la segunda igualdad nos damos cuenta de que los términos $(k+1)p^k$ pueden expresarse como la derivada respecto a p de $p^{k+1}$. Posteriormente, utilizamos la formula de una progresión geométrica convergente para sumar todos los términos (el primer término dividido por 1 menos la razón). Finalmente, derivamos respecto a *p* el resultado.

Sustituyendo este resultado en la expresión anterior, obtenemos:

$$T_m = \frac{T}{1-p}$$

Observad que, para la probabilidad de error que tenemos ($10^{-2}$), el tiempo total no se modifica de forma muy apreciable, dando lugar a una tasa efectiva de 49.500 bits/s. No obstante, la dependencia de $T_m$ con el inverso de $(1 - p)$ nos dice que, si las tasas de error por paquete son importantes, el control de errores mediante estas técnicas puede resultar muy ineficiente.

Para finalizar, cabe comentar que el método *half-duplex* tiene una dependencia directa con la distancia entre los dos extremos de la comunicación. En efecto, el tiempo medio para recibir un paquete depende de modo directo del tiempo de transmisión, que a su vez depende de la distancia entre los terminales. Esto no era así para la estrategia ARQ del ejemplo 3, en el que la distancia sólo podía afectar al retardo global pero no a la velocidad de transmisión.

## 3. ¿Para qué sirven los códigos de protección de errores?

La respuesta a esta pregunta puede parecer un tanto inmediata aunque, si nos detenemos a reflexionar, veremos que no resulta tan evidente. En efecto, hemos visto que la estrategia para protegernos frente a los posibles errores consiste en introducir redundancia adicional en los mensajes que pretendemos transmitir. Esto significa que para protegernos debemos pagar un precio: transmitir más bits que los estrictamente necesarios. La consecuencia inmediata es que el ancho de banda del canal necesario para transmitir la información debe aumentar. En general, veremos que, si queremos proteger la información original frente a un gran número de errores, deberemos usar códigos con tasas bajas, que suponen un aumento considerable del ancho de banda. La tasa de un código es un indicativo del coste en ancho de banda que representa la introducción de este código

Si continuamos reflexionando sobre el resultado anterior, nos daremos cuenta de que al aumentar el ancho de banda se reduce el tiempo disponible para transmitir cada uno de los bits. Por lo tanto, la energía dedicada a la transmisión de un bit en el mensaje codificado es menor que la dedicada a transmitir un bit en la información sin codificar (suponemos que la potencia del transmisor se mantiene constante). Esto significa que la probabilidad de que se produzca un error en el mensaje codificado es mayor que en el mensaje sin codificar. La cuestión clave es si este aumento de la probabilidad de error queda compensado por la capacidad de corrección de errores del código.

La respuesta a la pregunta es que generalmente sí sale a cuenta, que obtenemos una ganancia neta, que la capacidad de corrección de los códigos de canal compensa la pérdida de energía en cada uno de los bits del mensaje codificado. La clave de la respuesta a esta pregunta puede encontrarse en la gráfica de la figura 6, en la que se compara la probabilidad de error que se obtiene con un mensaje sin codificar y con un mensaje codificado. Las probabilidades de error se representan en función de la relación $E_b/N_0$ que representa la relación entre la energía dedicada a un bit $E_b$ y la densidad espectral de ruido $N_0$. Es evidente que, a medida que aumentamos la energía de los bits, la probabilidad de error decrece, ya que el mensaje se ve menos afectado por el ruido. La gráfica también indica que existe una región en la que el código de protección resulta rentable, ya que se obtienen probabilidades de error menores que con el mensaje sin codificar. En cambio, existe una zona en la que no resulta rentable codificar el mensaje, ya que la probabilidad de error con el código es mayor que sin el código. En esta región se dice que los errores superan las capacidades de corrección del código. Se producen tantos errores que la corrección no sale a cuenta.

En resumen, es importante comprender que existe un delicado compromiso entre varios factores implicados y relacionados: la redundancia introducida, el aumento del ancho de banda, la tasa del código y la potencia con la que se transmiten los mensajes. Todos estos factores inciden en la probabilidad de error final del sistema de comunicaciones, y su selección adecuada exige estudios rigurosos sobre las características de los canales y los códigos que deben utilizarse. La elección de un código u otro en un sistema de comunicaciones real exige profundos estudios y simulaciones de las condiciones en las que se realizarán las comunicaciones, las características del canal y los criterios de calidad que queremos obtener.

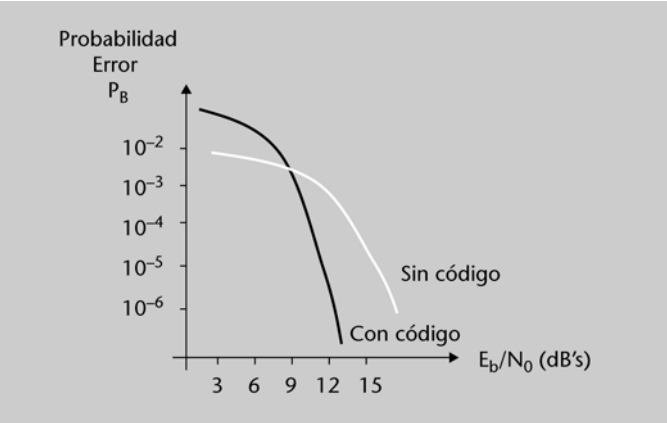


Figura 6. Ejemplo de la mejora en probabilidad de error en un sistema de comunicaciones con códigos correctores

## 4. Decisiones *soft* y decisiones *hard*

Cuando recibimos una señal por un canal con ruido, el valor que obtenemos puede considerarse una variable aleatoria. En el caso de los canales conocidos como *de ruido gaussiano*, esta variable aleatoria tiene una función densidad de probabilidad gaussiana. En la figura 7 se representa un ejemplo la función densidad de probabilidad que se recibe en un canal gaussiano cuando se transmiten los símbolos correspondientes a un bit en un determinado sistema de modulación.

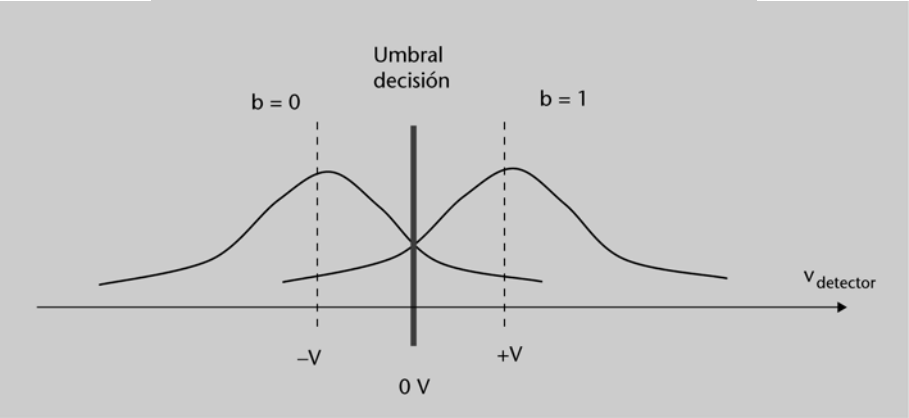


Figura 7. Resultado de la detección de un bit en un canal gaussiano

En este ejemplo concreto, estamos suponiendo que el transmisor acondiciona los bits al canal enviando tensiones positivas (de valor +V) cuando se quieren enviar un bit de valor 1 y tensiones negativas (de valor –V) cuando se quiere enviar un bit de valor 0. Supondremos, por tanto, que el detector debería detectar un nivel de tensión +V cuando el bit transmitido es un 1 y un nivel –V cuando el nivel lógico es 0.

No obstante, en la práctica, cuando se transmite un 1, la curva de la derecha nos indica la probabilidad que puede tomar la tensión recibida en el receptor. De forma análoga, la otra curva nos indica la probabilidad de los valores de tensión cuando se transmite un 0.

Decimos que realizamos una decisión "*hard*" cuando para cada bit recibido decidimos su valor, independiente de los valores que toman el resto de los bits. La intersección entre las dos curvas de densidad de probabilidad nos indica que, si el valor de tensión recibido es mayor que 0 (o, en general, mayor que el punto donde se cruzan las dos funciones densidad de probabilidad – f.d.p.), entonces la probabilidad de que se haya transmitido un 1 es mayor que la probabilidad de que se haya recibido un 0. Análogamente, si la tensión que recibimos es menor que 0 deberíamos decidir que el bit transmitido ha sido un cero lógico.

La decisión *hard* contrasta con las decisiones "*soft*", en las que no se decide el valor de cada uno de los bits recibidos hasta que se haya recibido la palabra completa. Para comprender mejor la diferencia entre las decisiones *soft* y las decisiones *hard*, consideraremos el ejemplo 5.

### 4.1. Ejemplo 5. Comparación entre decisiones *hard* y *soft*

Supongamos un código de canal en el que los únicos mensajes posibles son el {000} y el {111}. Sabemos que el valor medio de tensión que detecta el demodulador cuando transmitimos un 1 es de 1 V y que cuando transmitimos un 0 el detector obtiene –1 V.

En un canal con ruido gaussiano recibimos los 3 bits con los valores de tensión {0,8; 0,8; –3}. Veamos cuáles serían las posibles decisiones sobre el valor de la palabra transmitida:

- **Decisión *hard*:** en este caso decidimos para cada bit el valor transmitido. Si el valor de tensión transmitido es mayor que 0, decidiremos que se ha transmitido un 1, y si es menor, un 0. Por lo tanto, en nuestro ejemplo decidimos que la palabra recibida es {1 1 0}. Así pues, si calculamos la distancia de esta palabra con las dos palabras código veremos que la decisión sería que la palabra transmitida ha sido {1 1 1}

- **Decisión *soft*:** para decidir, mantenemos el valor de tensión que realmente se ha detectado en el receptor y determinamos la distancia euclídea entre la palabra recibida y las dos palabras que hubiéramos recibido en ausencia de ruido.

$$D_{111} = \sqrt{(1-0,8)^2 + (1-0,8)^2 + (1-(-3))^2} = 4,009$$
$$D_{000} = \sqrt{(-1-0,8)^2 + (-1-0,8)^2 + (-1-(-3))^2} = 3,23$$

Lo que nos indica que, si utilizamos una técnica de decisión *soft*, decidiremos que la palabra transmitida es {000}, lo contrario que si se utiliza una técnica de decisión *hard*.

Es muy importante mencionar que no existe un criterio claro sobre la conveniencia de utilizar una técnica de decisión *hard* o *soft*. En general, para sistemas con ruido gaussiano se suele considerar que la decisión *soft* produce mejores resultados, aunque suele ser computacionalmente más compleja.

Notad que en la mayoría de los casos, las dos técnicas producen el mismo resultado final. Sólo en algunas situaciones el resultado que se obtiene con el uso de uno u otro método son distintos. El ejemplo anterior quería poner énfasis sobre los casos en los que las dos técnicas producen resultados distintos para clarificar sus diferencias.

# 5. Códigos de bloque

Un **código de bloque** consiste en asignar una palabra o vector de *n* bits a cada uno de los $2^k$ posibles mensajes de *k* bits. En este texto, únicamente consideraremos los códigos de bloque binarios, en los que cada palabra código está formada por un conjunto finito de bits.

En este caso, el código puede interpretarse como una tabla en la que para cada posible mensaje de entrada tenemos almacenado su palabra código correspondiente. De hecho, algunas implementaciones del codificador de canal pueden realizarse mediante accesos a memorias LUT (*look-up tables* – Memorias de asignación). En la figura 8 se muestra un ejemplo de un código de bloque (6,3).

| Mensajes K = 3 bits | | Palabras código n = 6 bits | |
|---|---|---|---|
| $x_0$ | 000 | 000000 | $c_0$ |
| $x_1$ | 001 | 110110 | $c_1$ |
| $x_2$ | 010 | 010101 | $c_2$ |
| $x_3$ | 011 | 100011 | $c_3$ |
| $x_4$ | 100 | 101010 | $c_4$ |
| $x_5$ | 101 | 011100 | $c_5$ |
| $x_6$ | 110 | 111111 | $c_6$ |
| $x_7$ | 111 | 001001 | $c_7$ |

Figura 8. Ejemplo de una tabla de asignación para un código de bloque (6,3)

Veremos a continuación un conjunto de definiciones básicas que se aplican a los códigos de bloques y que utilizaremos posteriormente para derivar sus características para la corrección y detección de errores.

## 5.1. Definiciones básicas

**Distancia de Hamming.** La distancia de Hamming entre dos palabras código se define como el número de componentes en las que las dos palabras son distintas. Tomando como ejemplo el código de la figura 8, tenemos:

$$d(c2,c3) = 4;\ d(c1,c2) = 3$$

**Distancia mínima de Hamming.** La distancia mínima de un código es la menor distancia de Hamming que puede encontrarse entre cualesquiera dos palabras distintas pertenecientes al código. Formalmente:

$$d_{\min} = \min_{\substack{c_i, c_j \\ i \neq j}} d\,(c_i, c_j)$$

**Peso de una palabra código.** El peso de una palabra código es el número de elementos distintos de cero. Para nuestro ejemplo de código de bloques:

$$\omega(c_3) = 3;\ \omega(c_0) = 0$$

**Peso mínimo de un código.** Es la palabra que tiene un peso mínimo, exceptuando la palabra código todo ceros. Es por tanto la palabra que tiene el mínimo número de unos.

## 5.2. Distancia mínima, corrección y detección de errores

La distancia mínima de un código nos indica cuál es el número mínimo de bits que debemos cambiar en una palabra código para que se convierta en otra palabra código.

Según esta definición, parece natural que la distancia mínima determina las capacidades de corrección y detección de errores de un código. En efecto, supongamos que realizamos una decisión hardware y que recibimos una palabra **y** en la que es posible que se hayan producido algunos errores respecto a la palabra **x** transmitida originalmente.

Si sólo deseamos detectar errores, concluiremos que la palabra **y** contiene errores siempre que no coincida con una palabra código. En cambio, si lo que deseamos es corregir errores, el proceso de decodificación más lógico es suponer que la palabra corregida es aquella que presente una distancia de Hamming más próxima a la palabra recibida.

En la figura 9 se muestra un ejemplo gráfico simplificado de un código que tiene una distancia mínima de Hamming igual a 4. Las palabras código se representan con círculos, mientras que las combinaciones binarias que no son una palabra código se representan por puntos. Las rectas unen palabras binarias que difieren en un único bit y que por lo tanto están a una distancia de Hamming unidad. Observemos que, para ir de una palabra código a otra, deben producirse un mínimo de 4 transiciones, lo que nos indica que la distancia mínima de nuestro código es 4.

En esta representación esquemática del código se observa que, si se produce un único error durante la transmisión del mensaje, será posible realizar la corrección utilizando la palabra código que resulte más próxima. Pongamos por ejemplo que se ha transmitido el código **c1** y que se ha producido un error en un bit, por lo que recibimos la palabra **e1**. El sistema puede realizar la corrección sistemática de este error debido a que la palabra código más próxima a **e1** es la palabra correcta **c1**. No obstante, si se producen dos errores alcanzaríamos la palabra **e2**, que está a la misma distancia de **c1** y de **c4**, por lo que no podríamos decidir cuál de las dos palabras ha sido transmitida. Es este caso, parece lógico que, cuando se reciba una palabra que está a la misma distancia de dos palabras código, sólo podamos afirmar que se ha producido un error (detección) pero no podamos corregirlo. En cambio, si se producen tres errores, podemos obtener la palabra **e3**. Si intentamos corregir esta palabra, decidiremos de forma incorrecta, ya que c4 es el código más próximo. En la figura 9 se muestran con círculos discontinuos las regiones en las que se pueden corregir los errores de un único bit.

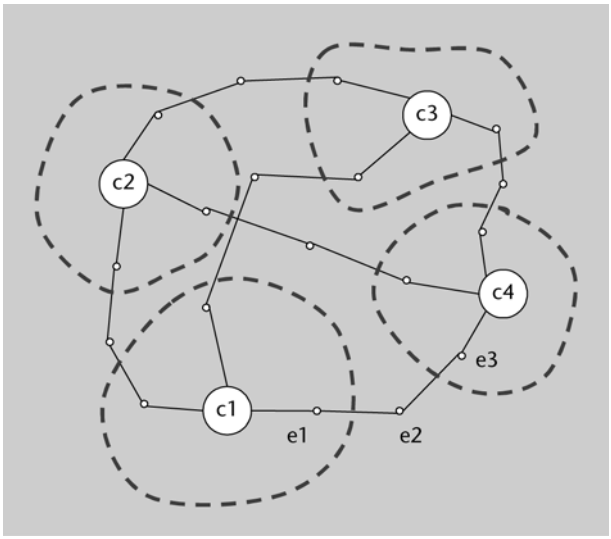


Figura 9. Ejemplo esquemático de un código con distancia mínima 4

Como resumen de este ejemplo, vemos que podemos aplicar dos estrategias básicas para protegernos frente a los errores. En efecto:

**a)** Podemos utilizar una estrategia sólo para detectar los errores. En este caso, seremos capaces de detectar correctamente la existencia de hasta tres errores.

**b)** Podemos utilizar una estrategia para corregir un error y detectar dos.

La elección de una u otra de estas estrategias depende, en parte, de las características del sistema. En sistemas en los que existe un canal de retorno y en los que el tiempo real no es prioritario, puede resultar recomendable aplicar estrategias de detección de errores, ya que es evidente que siempre tendremos una mayor capacidad de detección que de corrección. En sistemas en los que

prima el tiempo real, o en los que no existe un canal de retorno, parece recomendable utilizar estrategias de corrección de errores. En la figura 10 se muestra un gráfico simplificado de las dos estrategias que podemos utilizar en nuestro ejemplo.

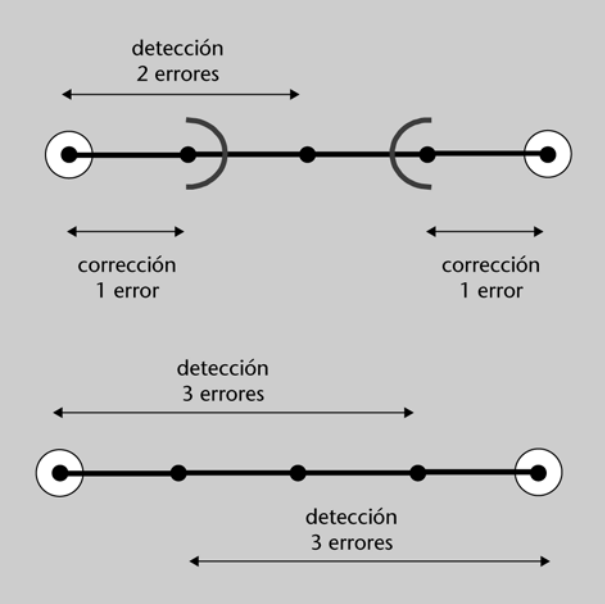


Figura 10. Ejemplo de las capacidades de corrección y detección de errores de un código con distancia mínima 4

Para aclarar algo más la figura 10, veamos con detalle cómo interpretar la estrategia de corrección de un error y detección de 2, que se propone en la gráfica superior.

- Cuando el receptor recibe una palabra, comprueba cuál es la distancia mínima respecto a todas las posibles palabras código que hubiera podido recibir.

- Si la distancia mínima es cero, entonces presuponemos que no se ha cometido error, y no se corrige.

- Si la distancia mínima es uno, entonces corregiremos sustituyendo el mensaje recibido por la palabra código, con la que tiene esta distancia mínima.

- Si la distancia mínima obtenida es dos, entonces no corregiremos pero sabremos que hemos cometido un error.

Este tipo de estrategias se conocen con el nombre de *estrategias de corrección/detección mixtas* debido a que, en función de la distancia mínima obtenida, decidimos si se realizará la corrección o la detección de los errores.

En la práctica existen algoritmos más eficientes que el que hemos propuesto y que estudiaremos en el apartado de decodificación. No obstante, a efectos de comprender las estrategias de corrección y detección, el algoritmo anterior

puede ayudar a entender el procedimiento que se sigue. Debemos mencionar que estas estrategias sólo funcionan si los errores que se han producido en el canal están dentro de las capacidades correctoras del código. Así, si se producen tres errores, estimaremos que la distancia mínima de la palabra recibida con una palabra código es uno y que realicemos una corrección equivocada.

La diferencia entre la estrategia de corrección/detección mixta y la estrategia de sólo detección es que en esta última nunca corregimos: sólo detectamos que se ha producido un error.

Es posible generalizar los resultados anteriores para códigos con cualquier distancia mínima. En el caso de que únicamente realicemos detección de errores, es evidente que el número de errores que podemos detectar es igual a la distancia mínima menos uno.

$$e_d = d_{min} - 1$$

Esto es así porque cualquiera de las $d_{min} - 1$ transiciones que existen entre dos palabras códigos podrán ser detectadas como errores.

Si sólo se realiza corrección, debemos considerar por separado los casos en los que la distancia mínima es un número par o impar. Las relaciones entre la distancia mínima y el número de errores que es posible corregir viene dada por:

$$\begin{cases} 2e_c + 1 \leq d_{\min} & d_{\min}\ impar \\ 2e_c + 1 < d_{\min} & d_{\min}\ par \end{cases}$$

Estas dos ecuaciones pueden agruparse en:

$$e_c = INT\left[\frac{d_{\min} - 1}{2}\right]$$

Donde INT representa la parte entera.

En el caso de que se realice una corrección/detección mixta, puede demostrarse que debe cumplirse que la suma entre los errores que se corrigen más los que se detectan sea igual a la distancia mínima menos uno.

$$e_c + e_d = d_{\min} - 1$$

Observemos que, en el caso del ejemplo con distancia mínima 4, podíamos corregir un error y detectar dos.

En el siguiente ejemplo, intentamos clarificar todas las estrategias de corrección/detección que podemos llevar a cabo con un código con distancia mínima (de Hamming) igual a 5.

### 5.3. Ejemplo 5. Estrategias de corrección/detección de errores

Vamos a determinar todas las posibles estrategias de corrección/detección que podemos aplicar utilizando un código con distancia mínima 5.

Con un código de distancia mínima 5 tenemos las siguientes posibilidades:

**a)** Corregir un máximo de dos errores.

**b)** Corregir un único error y detectar tres.

**c)** Detectar cuatro errores.

Las tres posibilidades se muestran esquemáticamente en la figura 11.

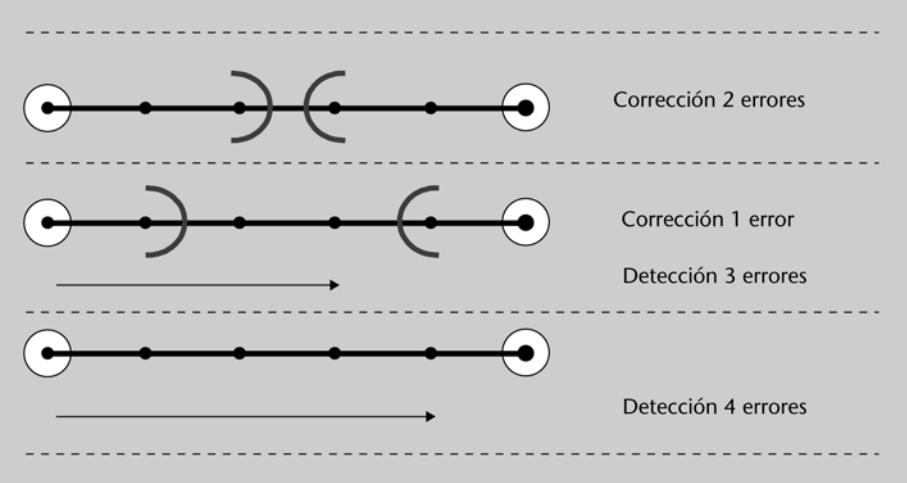


Figura 11. Posibilidades de corrección/detección de errores con un código de distancia mínima 5

### 5.4. Códigos de bloque lineales

Recordemos del apartado anterior que restringimos nuestra discusión de códigos de bloque a códigos binarios, es decir a códigos en los que todas las palabras están formadas por vectores de 1 y 0.

Se dice que un código de bloque es **lineal** si la suma de dos palabras código cualesquiera es también una nueva palabra código. Podemos expresar esta condición matemáticamente como:

Para todo *i,j*, existe *k* tal que $\mathbf{c}_k = \mathbf{c}_i \oplus \mathbf{c}_j$; (1)

Donde el símbolo $\oplus$ indica la suma módulo 2 componente a componente.

En el siguiente ejemplo, intentamos interpretar de manera práctica el concepto de suma en módulo 2 y el de código de bloques lineal.

## 5.5. Ejemplo 6. Código de bloques lineales

**Ejemplo.** Tomemos como ejemplo el código de bloque que se ha mostrado en la figura 8. En este caso, si tomamos las palabras código $c_3$ y $c_4$ obtenemos:

$$\mathbf{c}_3 \oplus \mathbf{c}_4 = [1 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1] \oplus [1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0] = [0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 1] = \mathbf{c}_7$$

Podría comprobarse de forma exhaustiva que, para cualesquiera dos palabras código, cuando realizamos su suma, obtenemos una nueva palabra que también forma parte del código. Por ello, el código anterior es un código lineal.

Observemos cómo se realiza la suma de dos palabras código: el resultado que obtenemos para cada componente es independiente de las demás, la suma de dos símbolos distintos es siempre 1 ($1 \oplus 0$, $0 \oplus 1$) mientras que la suma de dos símbolos iguales es siempre 0 ($0 \oplus 0$, $1 \oplus 1$). La operación de suma es, pues, idéntica a una puerta OR exclusiva aplicada bit a bit.

Este ejemplo puede resultar engañoso, ya que hemos comprobado que un código que originalmente habíamos presentado como código de bloque genérico es también un código lineal, lo que daría a entender que prácticamente todos los códigos de bloques son lineales.

Esto es absolutamente falso: si se eligen al azar las palabras código en un código de bloques lo más probable es que el código resultante no sea lineal. Considerad por ejemplo que en el código de la figura 8 hubiéramos tomado la palabra $c_7$ = [0 1 1 0 0 1]. Puede comprobarse que, ahora, la suma entre $c_3$ y $c_4$ ya no está dentro del código y que tendremos muchas otras sumas que tampoco son palabras código. El código resultante al sustituir esta palabra sería por tanto un código de bloque no lineal.

Una característica específica de todos los códigos de bloque lineales es que contienen la palabra **0** (es decir, el vector que tiene todas las componentes igual a cero). En efecto, como la suma de dos palabras código cualesquiera debe ser también una palabra código, podemos ver que, cuando realizamos la suma de cualquier vector consigo mismo ($c_j \oplus c_j$), siempre obtenemos como resultado la palabra **0**, de manera que esta palabra deberá formar parte del código si queremos que sea lineal.

Un código de **bloque binario** asigna una palabra código (vector) de *n* bits a cada uno de los $M = 2^k$ posibles mensajes de entrada de *k* bits ($k \leq n$; evidentemente, la igualdad no introduce redundancia y no permite la detección/corrección de errores).

Un código de **bloque es lineal** si la suma de cualquiera de dos de sus palabras código es también una palabra código.

Los códigos de bloque sólo tienen que verificar la ecuación (1) para tener todo el carácter y propiedades de los códigos lineales. Sin embargo, en muchos casos, se extiende el concepto de linealidad a la aplicación que nos asocia los mensajes originales a las palabras código. La mayoría de los códigos que se utilizan en la práctica también tienen esta propiedad de linealidad en la transformación, por lo que es habitual que entendamos por un código lineal aquel que verifica la ecuación (1) y además:

$$Si\ \mathbf{x}_i \rightarrow \mathbf{c}_i\ y\ \mathbf{x}_j \rightarrow \mathbf{c}_j\ entonces\ \mathbf{x}_i \oplus \mathbf{x}_j \rightarrow \mathbf{c}_i \oplus \mathbf{c}_j \tag{2}$$

Básicamente, esta ecuación establece que, cuando el mensaje de entrada puede expresarse como la suma de otros dos mensajes, entonces la palabra código también puede expresarse como la suma de las dos palabras código asociadas a cada mensaje.

En el resto del texto, por abuso del lenguaje, nos referiremos a códigos de bloque lineales como aquellos que cumplen la ecuación (1), que nos dice que el código es lineal, y la ecuación (2), que nos dice que la aplicación que asocia los mensajes originales a palabras código es lineal.

## 5.6. Matriz generadora de códigos de bloque lineales

En este apartado veremos una manera sistemática para generar códigos lineales (en el sentido amplio definido en el párrafo anterior). Para ello, debemos identificar la palabra código (*n* elementos) que asociaremos a cada uno de los elementos de la base canónica de los bloques de información (*k* elementos). Recordemos que la base canónica está definida como aquélla cuyos elementos tienen todos los componentes nulos excepto uno de ellos, que toma el valor unidad.

Probablemente se ve más claro en la siguiente ecuación, donde los componentes $\{\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_N\}$ forman la base canónica del los bloques de mensaje.

$$\begin{aligned} \mathbf{e}_1 &= (100\ldots00) \rightarrow \mathbf{g}_1 \\ \mathbf{e}_2 &= (010\ldots00) \rightarrow \mathbf{g}_2 \\ \mathbf{e}_3 &= (001\ldots00) \rightarrow \mathbf{g}_3 \\ &\ldots \\ \mathbf{e}_k &= (000\ldots01) \rightarrow \mathbf{g}_k \end{aligned} \tag{3}$$

Los vectores $\{\mathbf{e}_1, \ldots, \mathbf{e}_k\}$ forman la base canónica del espacio de palabras de entrada al codificador. Cualquier palabra de entrada puede expresarse de manera directa como una combinación lineal de los elementos de la base canónica. Los vectores $\{\mathbf{g}_1, \ldots, \mathbf{g}_k\}$ corresponden a las palabras código que se asocian a cada uno de los vectores de la base canónica. Son, por tanto, vectores con un total de *n* elementos.

Cualquier vector de entrada puede expresarse como una combinación lineal de los vectores de la base canónica. En efecto:

$$\mathbf{x} = (x_1, x_2, \cdots, x_k) = \sum_{i=1}^{k} x_i \cdot \mathbf{e_i} \tag{4}$$

De forma que la palabra código resultante puede expresarse como la misma combinación lineal de las palabras $\mathbf{g_i}$.

$$\mathbf{c} = \sum_{i=1}^{k} x_i \cdot \mathbf{g_i} \tag{5}$$

Resulta útil emplear notación matricial para representar las combinaciones lineales implícitas en las ecuaciones anteriores. Para ello, se define la matriz generadora del código como:

$$\mathbf{G} = \begin{bmatrix} \mathbf{g}_1 \\ \mathbf{g}_2 \\ \vdots \\ \mathbf{g}_k \end{bmatrix} = \begin{bmatrix} g_{11} & g_{12} & \cdots & g_{1n} \\ g_{21} & g_{22} & \cdots & g_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ g_{k1} & g_{k2} & \cdots & g_{kn} \end{bmatrix} \tag{6}$$

Se trata por tanto de una matriz con k-filas y n-columnas en la que cada una de las filas es la palabra código asociada a uno de los vectores de la base canónica. Podemos obtener la expresión (5) en forma matricial utilizando la definición (6), con lo que se obtiene:

$$\mathbf{c = x \cdot G} \tag{7}$$

## 5.7. Ejemplo 7. Matriz generadora de un código de bloques lineal

Consideremos el código (6,3), cuya tabla de asignación se ha definido en la figura 8. Para obtener la matriz generadora del código, debemos identificar los elementos de la base canónica y las palabras código que tienen asignadas. De esta forma:

$$\begin{aligned} e_3 = x_1 = (100) &\rightarrow (101010) = g_3 \\ e_2 = x_2 = (010) &\rightarrow (010101) = g_2 \\ e_1 = x_4 = (001) &\rightarrow (110110) = g_1 \end{aligned} \tag{8}$$

La matriz generadora del código se construye ordenando las palabras código como filas:

$$\mathbf{G} = \begin{bmatrix} \mathbf{g}_3 \\ \mathbf{g}_2 \\ \mathbf{g}_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 \end{bmatrix} \tag{9}$$

Finalmente, para calcular la palabra código asociada al mensaje (1 1 1) podemos utilizar la relación (7). En efecto:

$$\mathbf{c} = \mathbf{x} \cdot \mathbf{G} = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \qquad (10)$$

Que concuerda con la palabra código correspondiente de la figura 8.

## 5.8. Códigos sistemáticos

Los códigos sistemáticos que estudiaremos en este apartado son códigos en los que la parte del mensaje y la parte de la redundancia estructurada pueden identificarse de manera directa. En un código sistemático, los *k* primeros bits de la palabra código corresponden al mensaje, mientras que los siguientes r = n – k corresponden a la redundancia.

En general, los códigos de bloque convierten mensajes de *k* bits de información en palabras código de *n* bits. Se dice que la redundancia añadida es de (n – k) bits, aunque no siempre resulta posible identificar los bits que se corresponden con la redundancia. En efecto, en el ejemplo 7 hemos visto que el mensaje (1 1 1) se convierte en la palabra código (0 0 1 0 0 1), o que el mensaje (1 0 0) se convierte en (1 0 1 0 1 0). Por tanto, que se trata de un código que introduce 3 bits de redundancia adicionales, aunque no es posible identificarlos de modo directo en la palabra código. Podríamos decir que la redundancia está dispersa en todos los bits de la palabra código.

Un caso muy especial de códigos de bloque son los códigos sistemáticos en los que los bits de redundancia pueden localizarse de forma inmediata. En un código sistemático, los *k* primeros bits se corresponden con los del mensaje original y los (n–k) siguientes con la redundancia, en una forma análoga a la que se representa en la figura 3.

Para un código sistemático, la matriz generadora siempre puede descomponerse en la forma siguiente:

$$\mathbf{G} = [\mathbf{I_k} \,|\, \mathbf{P}] \qquad (11)$$

Donde $I_k$ es una matriz identidad de *k* filas y *k* columnas, cuya función es repetir los *k* primeros bits del mensaje original. La matriz P indica cómo calcular los bits de redundancia a partir del mensaje original. Intentemos clarificar estos conceptos mediante un ejemplo.

**Ejemplo**. Consideremos la siguiente matriz generadora de un código sistemático (7,3).

$$\mathbf{G} = \left[\begin{array}{ccc|cccc} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{array}\right]$$

Podemos ver con detalle el proceso de codificación multiplicando la matriz por un mensaje genérico $\mathbf{x} = (x_1, x_2, x_3)$. En efecto:

$$\mathbf{x}\cdot\mathbf{G} = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}\cdot\left[\begin{array}{ccc|cccc} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{array}\right] = \begin{bmatrix} x_1 & x_2 & x_3 & x_1+x_2 & x_2+x_3 & x_1+x_3 & x_1+x_2+x_3 \end{bmatrix}$$

Observamos que los tres primeros bits del mensaje codificado coinciden con los del mensaje original, debido a que la matriz identidad copia el valor original en estas posiciones. El resto de los bits de redundancia se obtiene combinando los valores indicados por la matriz P. Cada columna de esta matriz establece las operaciones que deben realizarse para obtener el bit de redundancia correspondiente. Este resultado sugiere también una implementación del código lineal, mediante la realización de sumas entre los bits de información para obtener los bits de redundancia. Entendemos por *implementación del código lineal* un diagrama de bloques con elementos lógicos que pueden realizarse en hardware o en software. En la figura 12 se representa de forma esquemática una posible implementación del código.

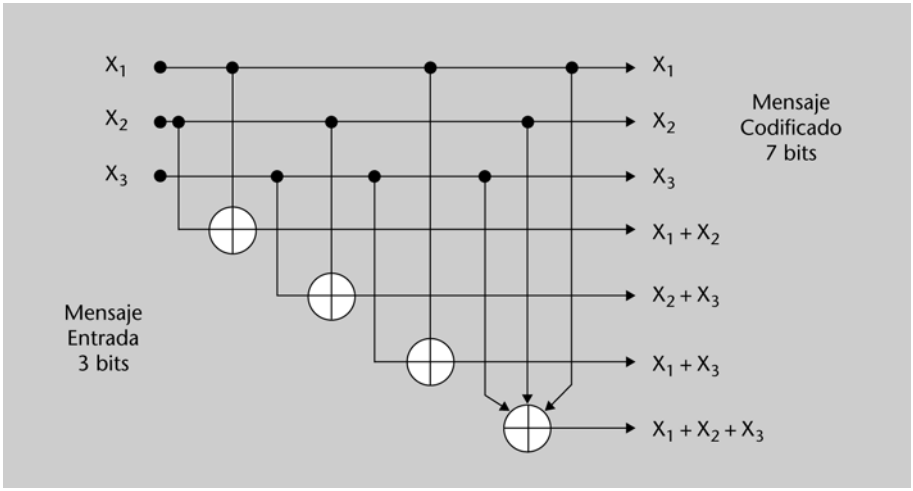


Figura 12. Esquema de implementación de un código de bloque lineal sistemático

## 5.9. Matriz de chequeo de paridad

La matriz generadora de un código de bloque lineal está formada por un total de *k* vectores fila, linealmente independientes y con *n* componentes cada uno de ellos. Por ello, podemos decir que estos vectores fila generan un subespacio vectorial de dimensión *k* (el número de vectores) dentro del espacio vectorial de n-componentes. Así, será posible encontrar un conjunto de (n–k) vectores que sean ortogonales a los vectores generadores y que permitan obtener el complemento ortogonal a este espacio vectorial. Estos (n–k) vectores serán ortogonales a todas las palabras código, de manera que podemos escribir:

$$\mathbf{c}\cdot\mathbf{H}^{\mathbf{t}} = \mathbf{0} \qquad (12)$$

Donde **c** representa cualquier palabra código y $\mathbf{H^t}$ es la matriz cuyas columnas están formadas por los (n–k) vectores que generan el complemento ortogonal de nuestro código. El resultado del producto es un vector fila de (n–k) componentes con todos los elementos igual a cero (ortogonalidad entre **c** i las columnas de $\mathbf{H^t}$). La matriz H se denomina *matriz de chequeo de paridad*, ya que puede utilizarse para comprobar si una palabra es o no es palabra código. En efecto, si el producto de la palabra recibida por la matriz $H^t$ da como resultado un vector con todos los componentes nulos, deduciremos que se trata de una palabra código.

Como la relación (12) se cumple para todas las palabras código, podemos sustituir el vector **c** por la matriz generadora del código. En efecto, $H^t$ deberá ser también ortogonal a todos los vectores base con los que se genera el código. Así pues:

$$\mathbf{G \cdot H^t = 0}$$

Donde, ahora, la matriz **0** es una matriz con *k* filas y (n–k) columnas, con todos los elementos igual a cero.

Para un código sistemático, la matriz de chequeo de paridad viene dada por:

$$\mathbf{H = [P^t \,|\, I_{n-k}]} \qquad (13)$$

La construcción de la matriz de chequeo de paridad para un código sistemático es directa una vez que disponemos de la matriz generadora. Podemos comprobar que, si construimos la matriz de acuerdo con esta ecuación, la matriz de chequeo de paridad y la matriz generadora son ortogonales. En efecto:

$$\mathbf{G \cdot H^t} = \left[\mathbf{I_k} \middle| \mathbf{P}\right] \cdot \begin{bmatrix} \mathbf{P} \\ \mathbf{I_{n-k}} \end{bmatrix} = \left[\mathbf{I_k \cdot P + P \cdot I_{n-k}}\right] = \left[\mathbf{P + P}\right] = \mathbf{0}$$

Observemos que tanto $I_kP$ como $P\, I_{n–k}$ son matrices con *k* filas y (n–k) columnas.

**Ejemplo**. Consideremos la matriz generadora de un código sistemático utilizada en el ejemplo anterior:

$$\mathbf{G} = \left[\mathbf{I_k} \vdots \mathbf{P}\right] = \left[\begin{array}{ccc|cccc} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{array}\right]$$

Podemos obtener la matriz de chequeo de paridad utilizando la ecuación (13), con lo que tenemos:

$$\mathbf{H} = \left[\mathbf{P^t} \vdots \mathbf{I_{n-k}}\right] = \left[\begin{array}{ccc|cccc} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{array}\right]$$

Es posible comprobar que cualquier palabra código es ortogonal a $H^t$. En particular, resulta interesante comprobarlo para el caso general que hemos obtenido en el ejemplo anterior.

$$\mathbf{c}\cdot\mathbf{H^t} = \left[x_1, x_2, x_3, x_1+x_2, x_2+x_3, x_1+x_3, x_1+x_2+x_3\right]\cdot\begin{bmatrix}1&0&1&1\\1&1&0&1\\0&1&1&1\\1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix} =$$

$$= \left[x_1+x_2+(x_1+x_2), x_2+x_3+(x_2+x_3), x_1+x_3+(x_1+x_3), x_1+x_2+x_3+(x_1+x_2+x_3)\right] =$$

$$= [0,0,0,0]$$

## 5.10. Códigos de Hamming

Los códigos de Hamming son códigos de bloque lineales que tienen una distancia mínima 3 y cuya matriz de chequeo de paridad se construye de una manera directa y simple. Al tener distancia mínima 3, pueden utilizarse para corregir un error o detectar dos errores. Se utilizan en un gran número de aplicaciones, ya que pueden realizarse con bajo coste computacional. Si se usan directamente, pueden corregir errores aislados (hasta un bit en cada palabra), pero también pueden usarse conjuntamente con otros códigos para proporcionar robustez al sistema.

Los valores de (n,k) para los que existen códigos de Hamming se obtienen en función de un parámetro adicional *m*, que debe tomar valores mayores o iguales que 2 (ya que la distancia mínima del código debe ser 3). Las relaciones entre *n*, *k* y *m* vienen dadas por:

$$\begin{aligned} n &= 2^{m-1};\ m \geq 2 \\ k &= 2^m - m - 1 \\ d_{\min} &= 3 \end{aligned} \qquad (14)$$

Para construir la matriz de chequeo de paridad de un código de Hamming, debemos poner como columnas todas las posibles combinaciones de (n–k) componentes exceptuando la columna todo ceros.

Consideremos como ejemplo la creación de un código de Hamming sistemático con m = 3. Si sustituimos m = 3 en la ecuación (14), obtenemos un código con palabras de n = 7 bits y mensajes originales de k = 4 bits. La matriz de chequeo de paridad será de (n–k) = 3 filas y n = 7 columnas. Si queremos que el código de Hamming sea sistemático, las últimas tres columnas y filas deberán formar la identidad. Podemos obtener las cuatro primeras columnas utilizando todas las palabras de 3 bits que faltan (suponemos que las tres últimas columnas ya han sido rellenadas) exceptuando la palabra todo ceros. El orden en el que se pongan estas columnas da lugar a distintos ordenamientos de los bits de redundancia, pero en todos los casos se trata

del código de Hamming sistemático de (7,4). Según esto, la matriz de chequeo de paridad viene dada por:

$$\mathbf{H} = \left[\mathbf{P}^{\mathbf{t}} : \mathbf{I}_{\mathbf{n-k}}\right] = \left[\begin{array}{cccc|ccc} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{array}\right]$$

Una vez construida la matriz H, teniendo en cuenta la ecuación (13), podemos identificar las cuatro primeras columnas con la matriz $P^t$. De esta forma, utilizando la ecuación (11) obtenemos la matriz generadora del código de Hamming (7,4).

$$\mathbf{G} = \left[\mathbf{I}_{\mathbf{k}} : \mathbf{P}\right] = \left[\begin{array}{cccc|ccc} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{array}\right]$$

# 6. Decodificación de códigos de bloque lineales

Hemos visto que la matriz generadora de un código nos proporciona un método sistemático para obtener la palabra código de *n* bits asociada a un determinado mensaje de *k* bits. Este método puede implementarse en software o directamente en hardware, utilizando operadores booleanos simples. Por otra parte, en recepción, podríamos utilizar la matriz de chequeo de paridad para comprobar que el mensaje recibido es una palabra código, pues sólo en este caso el producto del mensaje por la matriz H dará un resultado nulo. Si el resultado del producto entre el mensaje recibido y la matriz $\mathbf{H^t}$ –ver ecuación (12)– es diferente de cero, podemos garantizar que se han producido errores en la transmisión del mensaje. Por lo tanto, usando este sencillo procedimiento podemos implementar sistemas de **detección** de errores para códigos de bloque lineales.

Sin embargo, si nuestro objetivo es **corregir** el mensaje, el procedimiento de decodificación es algo más complejo. Podemos intuir que la clave del problema está en analizar el resultado que obtenemos del producto entre el mensaje recibido y la matriz de chequeo de paridad:

$$\mathbf{s} = \mathbf{c} \cdot \mathbf{H^t} \tag{15}$$

Si el resultado del producto es nulo, el mensaje recibido se corresponde con una palabra código. Sabemos también que, si el resultado es distinto de cero, se han producido errores pero no sabemos cómo realizar la corrección. El valor de **s** se conoce con el nombre de **síndrome** y nos proporciona la información suficiente para realizar la corrección del mensaje. No obstante, antes deberemos introducir algunos conceptos y propiedades nuevas que resultan cruciales para realizar la corrección del mensaje.

## 6.1. El estándar Array

El estándar Array es una estructura matricial para códigos de bloque lineales que resulta clave para definir un proceso sistemático de corrección de los mensajes. Esencialmente, es una matriz con $2^k$ columnas y $2^{(n-k)}$ filas en la que aparecen ordenadas todas las palabras binarias de n-bits. Observad que el número de elementos que contiene la matriz es $2^k x 2^{(n-k)} = 2^n$, lo que permite que cualquier combinación de n-bits pueda estar presente en la matriz. La organización de los mensajes binarios en el estándar Array depende del código de bloque, aunque su procedimiento de construcción es sistemático. Puede resumirse en los dos puntos siguientes:

**a)** La primera fila de la matriz contiene $2^k$ palabras de n-bits del código de bloques lineal. El orden de las palabras es indiferente siempre que la primera pa-

labra sea el mensaje nulo (todos los bits igual a cero). Denominaremos este primer elemento como $\mathbf{c_1 = 0}$. El resto de los elementos se denotan como $\mathbf{c_j}$ con $2 \le j \le 2^k$.

**b)** Para construir la fila r (suponemos $r > 1$), debemos examinar todas las posibles palabras de n-bits que aún no hayan sido introducidas en las r–1 filas anteriores. Elegimos una cualquiera de las palabras que tengan menor peso (número de unos) y la situamos como primer elemento de la fila *r*. Denotaremos como $\mathbf{e_r}$ a la palabra seleccionada. El resto de los elementos de esta fila se obtiene como la suma $\mathbf{c_{rj} = e_r \oplus c_j}$ *con* $2 \le j \le 2^k$, donde $c_{rj}$ representa el elemento de la fila *r*, columna *j* del estándar Array, $\mathbf{e_r}$ es el primer elemento de la fila *r* y $\mathbf{c_j}$ es el elemento *j* de la primera fila de la matriz.

En la ecuación (16) se muestra esquemáticamente la organización del estándar Array para un código lineal genérico.

$$\begin{bmatrix} \mathbf{c}_1 & \mathbf{c}_2 & \mathbf{c}_3 & \cdots & \mathbf{c}_{2^k} \\ \mathbf{e}_2 & \mathbf{e}_2 \oplus \mathbf{c}_2 & \mathbf{e}_2 \oplus \mathbf{c}_3 & \cdots & \mathbf{e}_2 \oplus \mathbf{c}_{2^k} \\ \mathbf{e}_3 & \mathbf{e}_3 \oplus \mathbf{c}_2 & \mathbf{e}_3 \oplus \mathbf{c}_3 & \cdots & \mathbf{e}_3 \oplus \mathbf{c}_{2^k} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{e}_{2^{(n-k)}} & \mathbf{e}_{2^{(n-k)}} \oplus \mathbf{c}_2 & \mathbf{e}_{2^{(n-k)}} \oplus \mathbf{c}_3 & \cdots & \mathbf{e}_{2^{(n-k)}} \oplus \mathbf{c}_{2^k} \end{bmatrix} \qquad (16)$$

Cada una de las filas del estándar Array se denomina **COSET**, y al primer elemento de la fila lo denominamos **COSET Leader**.

La construcción del estándar Array puede ilustrarse con un ejemplo que nos ayudará a entender algunas de las propiedades y características que analizaremos posteriormente.

### 6.2. Ejemplo 8. Construcción del estándar Array

Consideremos un código de bloques lineal (4,2) definido por la siguiente matriz generadora:

$$\mathbf{G} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

Para calcular el estándar Array, debemos poner en la primera fila de la matriz las palabras código, situando en primer lugar a la palabra todo ceros. En este caso, las palabras código son {0000, 1010, 0101, 1111}, con lo que tendremos la primera fila de la matriz. Recordad que, para obtener las palabras código, basta multiplicar cada uno de los posibles mensajes **x** de dos bits por la matriz generadora.

Para determinar la segunda fila, debemos elegir una de las palabras de menor peso que aún no han sido puestas en la matriz como COSET Leader. Elegimos

la palabra 1000 y obtenemos el resto de los elementos de la segunda fila mediante la suma directa indicada en la ecuación (16).

Para obtener la tercera fila, debemos buscar las palabras de menor peso que aún no han sido dispuestas. En este momento, están pendientes 0100 y 0001, por lo que podríamos elegir cualquiera de ellas. Elegimos la primera y completamos la tercera fila. Finalmente, para completar la última fila debemos elegir una palabra de peso 2, puesto que todas las palabras con peso unidad ya han sido colocadas. En este caso, la palabra elegida es 1100, con lo que completamos la matriz realizando las sumas directas con los elementos de la primera fila. La matriz final obtenida es:

$$\begin{bmatrix} 0000 & 1010 & 0101 & 1111 \\ 1000 & 0010 & 1101 & 0111 \\ 0100 & 1110 & 0001 & 1011 \\ 1100 & 0110 & 1001 & 0011 \end{bmatrix}$$

Observemos que con este procedimiento no aparece ninguna palabra repetida, y que por lo tanto tenemos todas las posibles combinaciones de 4 bits en el estándar Array.

### 6.3. Propiedades del estándar Array

Vamos a comprobar que el estándar Array tiene un conjunto de propiedades que facilitan la decodificación y corrección de los mensajes recibidos.

Una de las propiedades más importantes es que todos los elementos del estándar Array son distintos (ver demostración en ejercicios resueltos). Podemos comprobarlo en el ejemplo concreto del apartado anterior. Una consecuencia directa de que todos los elementos sean distintos es que el estándar Array incluye todas las posibles palabras de *n* bits. En efecto, el estándar Array, por construcción, tiene un total de $2^k$ columnas y $2^{n-k}$ filas, por lo que el número total de elementos es $2^n$, lo que incluye todas las posibles combinaciones de *n* bits.

Otra propiedad del estándar Array es que todos los elementos pertenecientes a un mismo COSET tienen el mismo síndrome. En efecto, supongamos dos mensajes $\mathbf{y_1}$ e $\mathbf{y_2}$ que pertenecen al mismo COSET, y calculemos su síndrome:

$$\mathbf{y}_1 = \mathbf{e}_l \oplus \mathbf{c}_i \quad \Rightarrow \quad \mathbf{y}_1 \cdot \mathbf{H}^t = (\mathbf{e}_l \oplus \mathbf{c}_i) \cdot \mathbf{H}^t = \mathbf{e}_l \cdot \mathbf{H}^t = \mathbf{s}_l$$
$$\mathbf{y}_2 = \mathbf{e}_l \oplus \mathbf{c}_j \quad \Rightarrow \quad \mathbf{y}_2 \cdot \mathbf{H}^t = (\mathbf{e}_l \oplus \mathbf{c}_j) \cdot \mathbf{H}^t = \mathbf{e}_l \cdot \mathbf{H}^t = \mathbf{s}_l$$

El estándar Array incluye todas las posibles combinaciones binarias de *n* bits (n-tuplas). Las filas del estándar Array se denominan COSET y se caracterizan por que todas las palabras tienen el mismo síndrome. La primera fila de la matriz, con síndrome nulo, está formada por las palabras del código de bloque.

## 6.4. Corrección de un mensaje

Finalmente, supongamos que recibimos una palabra **y**, en la que se han producido errores ya que no concuerda con ninguna de las palabras código. Para realizar la corrección, deberemos determinar aquella palabra código, $\mathbf{c_j}$, cuya distancia a la palabra recibida sea mínima. Es decir, el problema que pretendemos resolver es encontrar el valor de *j* que hace que sea mínima la distancia siguiente:

$$\mathbf{c_j} \textit{ tal que } d\,(\mathbf{y}, \mathbf{c_j}) \textit{ sea mínimo}$$

Como **y** es una n-tupla binaria, podemos encontrarla en el estándar Array, y por tanto podemos escribirla como:

$$\mathbf{y} = \mathbf{e}_l \oplus \mathbf{c}_i$$

Donde $c_i$ es cualquier palabra código y $\mathbf{e_l}$ puede calcularse como el síndrome de **y**. Por lo tanto, debemos buscar el mínimo de la siguiente función de distancia:

$$\mathbf{c}_j \textit{ tal que } d\left(\mathbf{e}_l \oplus \mathbf{c}_i, \mathbf{c}_j\right) \textit{ sea mínimo}$$

La distancia entre dos palabras coincide con el peso de la suma, por lo que podemos escribir:

$$d(\mathbf{e}_l \oplus \mathbf{c}_i, \mathbf{c}_j) = w(\mathbf{e}_l \oplus \mathbf{c}_i \oplus \mathbf{c}_j) = w(\mathbf{e}_l \oplus \mathbf{c}_m)$$

Donde hemos utilizado que, al tratarse de un código lineal, la suma de dos palabras dará lugar a una nueva palabra código. Ahora, el término de la derecha tiene peso mínimo cuando la $c_m$ es la palabra todo ceros. En efecto, recuérdese que todas las n-tuplas corresponden al mismo COSET y que el elemento que tiene peso mínimo (por modo de construir el estándar Array) es el COSET Leader.

Por lo tanto, si $c_m$ debe ser cero, $c_i$ deberá coincidir con $c_j$, por lo que podemos escribir:

$$\mathbf{c}_j = \mathbf{c}_i = \mathbf{c}_i \oplus \mathbf{e}_l \oplus \mathbf{e}_l = \mathbf{y} \oplus \mathbf{e}_l$$

Para entender cómo se deriva esta ecuación, debemos tener en cuenta que siempre que sumamos el mismo vector dos veces estamos sumando ceros, por

lo que no modificamos el valor original. La ecuación anterior nos dice que el valor $\mathbf{c_j}$ que deberemos decodificar cuando recibamos cualquier mensaje *y* es el resultado de sumar el mensaje con su COSET Leader.

Simplificándolo, podemos descomponer el procedimiento en los siguientes pasos:

- Determinar el síndrome del mensaje recibido:

$$\mathbf{s_1 = y \cdot H^t}$$

- Determinar el COSET Leader que tiene el mismo síndrome:

$$\mathbf{s_1 = e_1 \cdot H^t}$$

- Corregir el mensaje recibido:

$$\mathbf{c_j = y \oplus e_1}$$

Este resultado es ciertamente interesante, ya que nos indica cómo utilizar la información que proporciona el síndrome para realizar la corrección de la n-tupla recibida. El procedimiento consiste en calcular el síndrome del mensaje recibido y en sumar al propio mensaje la n-tupla de peso mínimo que tenga el mismo síndrome. Puede encontrarse cierta analogía con algunas técnicas médicas en las que, una vez conocidos los síntomas que presenta la enfermedad (síndrome), se aplica un elemento correctivo que produce los mismos síntomas con el objeto de realizar la cura.

## 6.5. Ejemplo 9. Decodificación de una palabra errónea

Veamos cómo se realiza el proceso de decodificación de una palabra errónea, utilizando el mismo código del ejemplo previo. Supongamos que hemos recibido la palabra y = 0111, que no se corresponde con ninguna de las palabras código.

A partir de la matriz generadora y utilizando la ecuación (13), podemos obtener la matriz de chequeo de paridad. En nuestro ejemplo, debido a las simetrías de este código concreto, las dos matrices son idénticas:

$$H = G = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

Si calculamos el síndrome del mensaje recibido, obtenemos:

$$\mathbf{S} = [0111] \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} = [10]$$

El valor del síndrome coincide con el síndrome del COSET Leader 1000, por lo que deberíamos decodificar la palabra como c = 0111 + 1000 = 1111.

Una manera alternativa y más simple de realizar la decodificación es buscar el mensaje en el estándar Array y dar como resultado de la decodificación la primera palabra código de la misma columna.

Debemos dejar claro que únicamente hemos utilizado este ejemplo para ilustrar el proceso sistemático de decodificación. En nuestro ejemplo no deberíamos realizar la corrección de las palabras recibidas debido a que estamos utilizando un código con distancia mínima igual a 2, que puede detectar dos errores pero no puede realizar ninguna corrección. En efecto, observad que el resultado que hemos obtenido al aplicar el procedimiento de decodificación a 0111 es la palabra 1111, que está a una distancia unidad de la palabra recibida. No obstante, 0101 también es una palabra código que está a una distancia unidad de la palabra que hemos recibido.

# 7. Códigos cíclicos

Los códigos cíclicos son un caso particular de los códigos de bloque lineales para los que existen algoritmos muy eficientes tanto para la codificación como para la decodificación de los mensajes. Los procesos de codificación y decodificación de los códigos de bloque vistos en nuestros ejemplos pueden inducir a pensar que la complejidad de los sistemas de corrección y detección es moderada o baja, ya que siempre hemos tratado con ejemplos manejables, con un número reducido de palabras. No obstante, especialmente en el proceso de decodificación, la complejidad computacional puede ser muy importante, sobre todo en aquellos casos en los que los valores de $n$ y $k$ y (n–k) son elevados, ya que se requiere realizar procesos de búsqueda de palabras código o de síndromes que estarán almacenadas en tablas de memoria de gran tamaño.

En este apartado, nuestro objetivo es el de introducir propiedades básicas de los códigos cíclicos, para que os familiaricéis con ellos y tengáis una perspectiva general de sus características. No obstante, no queremos abordar un estudio detallado de estos códigos, que exigiría una matemática rigurosa y relativamente compleja. Lamentablemente, este enfoque nos impedirá abordar la descripción de los algoritmos de decodificación que pueden utilizarse. Estos algoritmos, aunque son muy simples desde el punto de vista computacional, requieren una fuerte componente matemática para su comprensión. Desde el punto de vista tecnológico, los códigos cíclicos son muy importantes debido a que se utilizan en diferentes aplicaciones, especialmente las variantes de los códigos de BCH y las de los códigos de Reed-Solomon.

Un código cíclico es un código lineal de bloque que se caracteriza por que, si $\mathbf{c}$ es una palabra código, entonces cualquier desplazamiento circular de $\mathbf{c}^{(k)}$ también es una palabra código. Entendemos por desplazamiento circular de una palabra el hecho de que todos los bits se desplazan hacia la posición adyacente y el último bit se desplaza a la primera posición. En la Figura 13 se muestra un ejemplo de un desplazamiento circular hacia la derecha.

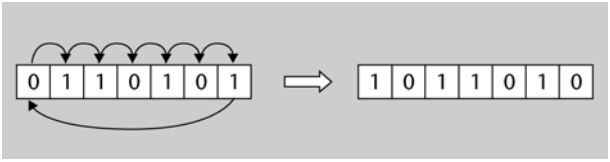


Figura 13. Ejemplo de desplazamiento circular de una palabra hacia la derecha

## 7.1. Ejemplo 10. Códigos cíclicos

El código {000, 110, 101, 011} es un código cíclico, ya que cualquier desplazamiento circular de una palabra produce otra palabra código. En efecto, el des-

plazamiento circular de la palabra 000 da lugar a la palabra 000, que es una palabra código. El desplazamiento circular de 110 da lugar a 011, que también es palabra código. Lo mismo ocurre para todas la palabras del código.

En cambio, el código {000, 010, 101, 111} no es un código cíclico, ya que, por ejemplo, el desplazamiento circular de 010 en 001 no es una palabra código.

## 7.2. Representación de las palabras código como polinomios

Para tratar matemáticamente los códigos cíclicos, resulta conveniente expresar las palabras código mediante polinomios. En efecto, puede establecerse una relación uno a uno entre el espacio vectorial de n-tuplas de n-bits y el espacio vectorial de los polinomios de grado n–1 de acuerdo con la siguiente expresión:

$$\mathbf{c} = (c_1, c_2, \ldots, c_n) \quad \Leftrightarrow \quad \mathbf{c}(p) = \sum_{i=1}^{n} c_i \cdot p^{n-i} = c_1 \cdot p^{n-1} + c_2 \cdot p^{n-2} + \ldots + c_{n-1} \cdot p + c_n$$

Aclaremos esta definición con un ejemplo. De acuerdo con esta definición, para una palabra código de 7 bits (1, 0, 0, 1, 1, 0, 1), el polinomio asociado será de orden 6. El primer término de la n-tupla binaria se corresponde con el primer coeficiente de la n-tupla, y así sucesivamente.

En definitiva, el polinomio asociado será:

$$1 \cdot p^6 + 0 \cdot p^5 + 0 \cdot p^4 + 1 \cdot p^3 + 1 \cdot p^2 + 0 \cdot p^1 + 1 = p^6 + p^3 + p^2 + 1$$

Veamos cómo podemos expresar matemáticamente el desplazamiento cíclico. Consideremos:

$$c^{(1)}(p) = c_2 \cdot p^{n-1} + c_3 \cdot p^{n-2} + \ldots + c_n \cdot p + c_1$$

Como estamos operando con aritmética binaria, podemos añadir el factor $c_1 p^n$ dos veces, que equivale a cero:

$$c^{(1)}(p) = c_1 \cdot p^n + c_2 \cdot p^{n-1} + c_3 \cdot p^{n-2} + \ldots + c_n \cdot p + c_1 + c_1 \cdot p^n$$

Agrupando los *n* primeros términos del segundo miembro de la igualdad, tenemos:

$$c^{(1)}(p) = p \cdot c(p) + c_1 (1 + p^n)$$

Sumando $c_1(1+p^n)$ al último término de la igualdad, obtenemos:

$$p \cdot c(p) = c^{(1)}(p) + c_1(1 + p^n) \qquad (17)$$

Esta igualdad suele interpretarse diciendo que el polinomio correspondiente al desplazamiento circular de una palabra código puede obtenerse mediante el siguiente procedimiento:

- Multiplicar el polinomio de la palabra origen por $p$ (la variable independiente del polinomio)
- Dividir el polinomio resultante por $(1 + p^n)$
- El polinomio asociado a la palabra desplazada es el resto de la división.

De acuerdo con este procedimiento, la ecuación (17) se escribe generalmente como:

$$c^{(1)}(p) = p \cdot c(p) \ (\mathbf{mod}\ (p^n + 1))$$

Y puede generalizarse para cualquier número de desplazamientos de una palabra código como:

$$c^{(r)}(p) = p^r \cdot c(p) \ (\mathbf{mod}\ (p^n + 1))$$

A partir de esta expresión general, es evidente que, cuando hemos realizado $n$ desplazamientos circulares, recuperamos el polinomio inicial. En efecto:

$$c^{(n)}(p) = p^n \cdot c(p) \ (\mathbf{mod}\ (p^n + 1)) = p^n \cdot c(p) + c(p) + c(p) \ (\mathbf{mod}\ (p^n + 1)) =$$
$$= (p^n + 1)c(p) + c(p) \ (\mathbf{mod}\ (p^n + 1)) = c(p) \ (\mathbf{mod}\ (p^n + 1)) = c(p)$$

### 7.3. Ejemplo 11. Desplazamiento en códigos cíclicos

**Ejemplo.** Supongamos una palabra binaria de 5 bits 01001. El polinomio asociado a esta palabra código será:

$$c(p) = 0 \cdot p^4 + 1 \cdot p^3 + 0 \cdot p^2 + 0 \cdot p^1 + 1 = p^3 + 1$$

Para obtener el polinomio asociado cuando realizamos un desplazamiento circular de un bit, debemos multiplicar por $p$ y dividir por $p^5+1$. En este caso, el orden del polinomio resultante es menor que 5, por lo que no es necesario realizar la división. El polinomio resultante y su palabra código asociada es:

$$c^{(1)}(p) = p^4 + p \quad \Rightarrow \quad (10010)$$

Para obtener un nuevo desplazamiento, multiplicamos de nuevo por $p$. Ahora sí que debemos calcular el valor resultante de la división:

$$c^{(2)}(p) = p^5 + p^2 \quad (\mathrm{mod}(p^5 + 1)) = p^2 + 1 \quad \Rightarrow \quad (00101)$$

## 7.4. Polinomio generador de un código cíclico

En un código cíclico, todas las palabras código pueden expresarse como el producto entre el polinomio asociado a la n-tupla de información (orden k – 1) y un polinomio de orden (n – k), denominado polinomio generador y que tiene la propiedad de ser un divisor del polinomio $p^n + 1$.

Así, podemos escribir el polinomio asociado a la secuencia de información como:

$$X(p) = x_1 \cdot p^{k-1} + x_2 \cdot p^{k-2} + \ldots + x_{k-1} \cdot p + x_k$$

El polinomio generador puede expresarse como:

$$g(p) = p^{n-k} + g_2 \cdot p^{n-k-1} + g_3 \cdot p^{n-k-2} + \ldots + g_{n-k} \cdot p + 1$$

De manera que podemos obtener el polinomio asociado a una palabra X, realizando el producto entre los dos polinomios:

$$c(p) = X\,(p) \cdot g\,(p)$$

## 7.5. Ejemplo 12. Polinomio generador de códigos cíclicos

Vamos a obtener un código cíclico de (7,4). Para ello necesitamos un polinomio generador de grado 3. El polinomio generador debe ser un divisor perfecto de $p^7+1$. Si descomponemos este polinomio en factores, obtenemos:

$$p^7 + 1 = (p+1)\,(p^3 + p^2 + 1)\,(p^3 + \mathrm{p} + 1)$$

Lo que nos indica que podemos utilizar cualquiera de los dos últimos polinomios de grado 3 como polinomio generador de un código cíclico (7,4). Tomaremos como generador el último de los dos polinomios (pero podríamos haber tomado el del medio):

$$g(p) = p^3 + p + 1$$

Ahora, para obtener la palabra código asociada a un mensaje genérico deberemos multiplicar el polinomio asociado a este mensaje por g(p). Calculemos en primer lugar el polinomio asociado al mensaje de información X:

$$(x_1, x_2, x_3, x_4) \quad \Rightarrow \quad X(p) = x_1 \cdot p^3 + x_2 \cdot p^2 + x_3 \cdot p + x_4$$

Con lo que las diferentes palabras código se obtienen realizando el producto entre los dos polinomios.

$$c(p) = X\,(p) \cdot g\,(p)$$

Supongamos que el mensaje que deseamos transmitir es (0110). En este caso, el polinomio asociado al mensaje es X(p) = $p^2$+p. Cuando multiplicamos este polinomio por el polinomio generador, obtenemos:

$$c(p) = (p^2 + p) \cdot p^3 + (p^2 + p)p + (p^2 + p) =$$
$$(p^5 + p^4) + (p^3 + p^2) + (p^2 + p) = p^5 + p^4 + p^3 + p$$

Los polinomios se multiplican siguiendo las reglas elementales del cálculo pero teniendo en cuenta que los coeficientes sólo pueden ser binarios (0 ó 1). Por ello, $p^k + p^k = 0$.

De acuerdo con el resultado que hemos obtenido, podemos pasar del polinomio a la palabra de 7 bits realizando la asociación inversa entre (0111010).

De forma análoga podríamos completar la tabla de todos los 16 mensajes del código cíclico.

## 7.6. Matriz generadora de códigos cíclicos

En este apartado veremos cómo obtener la matriz generadora de un código cíclico sistemático a partir del polinomio generador. La matriz generadora puede descomponerse como:

$$\mathbf{G} = [\mathbf{I_k} \mid \mathbf{P}]$$

De acuerdo con esta estructura, la i-ésima fila de la matriz viene dada por:

$$g_i = (0, 0, \ldots, 1, \ldots, 0, P_{i,1}, P_{i,2}, \ldots, P_{i,n-k})$$

Si expresamos este vector como un polinomio y lo identificamos con una palabra código X(p)g(p), obtenemos:

$$g_i(p) = p^{n-i} + P_{i,1} \cdot p^{n-k-1} + \ldots + P_{i,n-k} = X(p) \cdot g(p)$$

Por lo que podemos escribir:

$$p^{n-i} = p_{i,1} \cdot p^{n-k-1} + \ldots + p_{i,n-k} + X(p) \cdot g(p)$$

O de una forma más compacta:

$$p_{i,1} \cdot p^{n-k-1} + \ldots + p_{i,n-k} = p^{n-1}(\mathbf{mod}(g(p))$$

## 7.7. Ejemplo 13. Matriz generadora de código cíclico

Es importante notar que el código cíclico que obtendremos en este apartado es un código sistemático, que aunque se construya con el mismo polinomio

que hemos utilizado en el apartado anterior, no da exactamente el mismo código. En efecto, el código que hemos obtenido en el apartado anterior no es sistemático y este sí que lo será. De hecho, el código que obtendremos tiene las mismas palabras código, pero ordenadas de una manera diferente, es decir, es el mismo conjunto de palabras código pero asignadas a mensajes diferentes. Sigue siendo, por tanto, un código cíclico, aunque esta vez es sistemático.

Determinaremos la matriz generadora asociada al código cíclico (7,4) cuyo polinomio generador es $g(p) = p^3 + p + 1$.

$$\begin{aligned}
&p^6 \mod(p^3+p+1) = p^2+1 && \Rightarrow \quad FILA\,1: \quad (1000101)\\
&p^5 \mod(p^3+p+1) = p^2+p+1 && \Rightarrow \quad FILA\,2: \quad (0100111)\\
&p^4 \mod(p^3+p+1) = p^2+p && \Rightarrow \quad FILA\,3: \quad (0010110)\\
&p^3 \mod(p^3+p+1) = p+1 && \Rightarrow \quad FILA\,4: \quad (0001011)
\end{aligned}$$

Por lo que la matriz generadora será:

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

## 7.8. Codificación de los códigos cíclicos

La principal ventaja de los códigos cíclicos es que tienen una estructura matemática más rígida que la de los códigos de bloque convencionales, con unas propiedades bien definidas que facilitan encontrar mecanismos sistemáticos que faciliten la generación y decodificación.

En efecto, hemos visto que las palabras código pueden obtenerse multiplicando el polinomio generador con el polinomio de la información que se va a codificar. El producto entre dos polinomios puede considerarse como una operación de convolución entre dos secuencias: la secuencia de los datos y la formada por los coeficientes del polinomio generador. Según esto, es posible interpretar la codificación como un filtrado de la secuencia de información mediante un registro de desplazamiento (filtro) cuyos coeficientes son los elementos del polinomio generador. Así pues, en resumen, los códigos cíclicos pueden implementarse mediante registros de desplazamiento.

## 7.9. Códigos BCH

Los códigos BCH son una subclase de códigos cíclicos que permiten corregir un número arbitrario de errores *t*. Los códigos fueron propuestos por Bose, Chaudhuri y Hocquenghem, a los que deben su nombre. Estos códigos tienen una gran versatilidad para el diseño, ya que existe un gran número de polino-

mios generadores previamente tabulados. De esta forma, el usuario sólo debe determinar unos parámetros de diseño, que dependen del número de errores que desee corregir y la longitud de los bloques y, posteriormente, con estos parámetros, buscar el polinomio generador en una tabla. Existen también algoritmos eficientes para su decodificación.

Los parámetros de diseño se determinar a partir de las ecuaciones:

$$n = 2^m - 1$$
$$\mathrm{n} - \mathrm{k} = \mathrm{mt}$$
$$d_{min} = 2t + 1$$

De acuerdo con estos parámetros, si queremos utilizar un tamaño de bloque de 31 bits (n = 31) y corregir un total de tres errores (t = 3), encontraríamos un único polinomio en las tablas cuyo valor de *k* es 16. Así, el código BCH con un tamaño de bloque 31 que permite corregir tres errores contiene un total de 16 bits de información y 15 de redundancia. La tasa del código es R = 16/31. El polinomio generador que obtenemos en las tablas está expresado generalmente en octal. En nuestro caso concreto es $G = 107657_{OCT}$.

Podemos pasar este polinomio a forma binaria directamente: G = 1 000 111 110 101 111, o expresarlo directamente como un polinomio convencional:

$$G(p) = p^{15} + p^{11} + p^{10} + p^9 + p^8 + p^7 + p^5 + p^3 + p^2 + p + 1$$

## 7.10. Códigos de Reed-Solomon

Los códigos de Reed-Solomon son una variante de los códigos BCH y por tanto también son una subclase de los códigos cíclicos. Se trata en este caso de unos códigos que encontramos en muchas aplicaciones, como por ejemplo las codificaciones de canal en el CD-Audio, el MiniDisc, el DAT, el DVD-Vídeo, los diferentes sistemas de difusión de señales audiovisuales DVB-T, DVB-S, DVB-C, etc. Desafortunadamente, se trata de unos códigos que requieren una fuerte componente matemática para comprender sus detalles y propiedades, por lo que nos limitaremos a enunciar algunas de sus características sin demostrarlas.

La característica más específica de estos códigos es que trabajan a nivel de símbolo y no a nivel de bit. Es decir, las palabras sobre las que se aplica el código pertenecen a un alfabeto con un número finito de símbolos. En la mayoría de las aplicaciones prácticas, el símbolo más utilizado es el byte, es decir, un símbolo es una palabra de 8 bits. Los bits en la entrada del codificador se agrupan en palabras de 8 bits (bytes). El código de Reed-Solomon toma un conjunto de K bytes en la entrada y general un total de N bytes en la salida. Observemos que la idea general es muy parecida a lo que hemos visto hasta ahora pero que, en vez de trabajar a nivel de bit, se trabaja a nivel de byte. La tasa del código es R = K/N.

Los códigos de Reed-Solomon pueden corregir símbolos completos. Esto significa que, si un código de Reed-Solomon está diseñado para trabajar con tres errores por paquete, podrá corregir 3 bytes erróneos completos, independientemente de los bits erróneos que existan en cada byte. Así, para el código de Reed-Solomon no supone ningún problema que todos los bits que forman el byte sean erróneos. La corrección es tan factible como si sólo existiera un bit erróneo en el byte. Esta característica hace que los códigos de Reed-Solomon sean muy adecuados en aquellas aplicaciones en las que pueden aparecer errores en forma de ráfaga, que afectan a varios bits consecutivos. Esta situación es muy habitual en muchas aplicaciones. En efecto, en el CD-Audio los errores en la lectura de los bits del soporte se producen debido a la acumulación de polvo o rayadas en el disco que afectan a varios bits consecutivos. El DAT es un sistema de registro de audio digital en cinta magnética en la que los errores aparecen debido a la pérdida de parte del material magnético en la cinta y que también produce errores en más de un bit. En los sistemas de radiodifusión pueden aparecer interferencias o desvanecimientos que también afecten a varios bits consecutivos. En muchas de estas aplicaciones, los códigos de Reed-Solomon se utilizan en conjunción con otros códigos orientados a la corrección de errores individuales.

Los parámetros que definen a un código de Reed-Solomon (RS) son:

$$N = 2^k - 1$$
$$K = 1,3,\ldots,N-2$$
$$D_{\min} = N - K + 1$$
$$R_c = K / N$$

**Longitud del bloque**

Observemos también que la longitud del bloque depende directamente del número de bits que forman un símbolo. En el caso más habitual, k = 8 (los símbolos son de un byte) y el número total de posibles símbolos será de 255. K es el número de símbolos de información útil que se utilizan. A medida que aumentamos K, aumenta la tasa del código pero disminuye la capacidad de corrección ($D_{min}$). El número de símbolos erróneos que pueden corregirse viene dado por:

$$t = \frac{D_{\min} - 1}{2}$$

## Actividades

**1.** Demostrad la siguiente propiedad del estándar Array.

**Propiedad 1**. Todos los elementos del estándar array son distintos.

**Demostración**. Supongamos que existen dos elementos iguales, $\mathbf{y_i}$ e $\mathbf{y_j}$. Estos dos elementos pueden pertenecer al mismos COSET o a COSETs distintos. En el primer caso, como pertenecen al mismo COSET deben poder ser expresados como:

$$\left.\begin{array}{l}\mathbf{y}_{\mathbf{i}} = \mathbf{e}_{\mathbf{k}} \oplus \mathbf{c}_{\mathbf{i}} \\ \mathbf{y}_{\mathbf{i}} = \mathbf{e}_{\mathbf{k}} \oplus \mathbf{c}_{\mathbf{j}}\end{array}\right\} \Rightarrow \quad y_i = y_j \quad \Rightarrow \quad \mathbf{e}_{\mathbf{k}} \oplus \mathbf{c}_{\mathbf{i}} = \mathbf{e}_{\mathbf{k}} \oplus \mathbf{c}_j \quad \Rightarrow \quad \mathbf{c}_{\mathbf{i}} = \mathbf{c}_{\mathbf{j}}$$

Lo que resulta absurdo por cuanto $c_i$ y $c_j$ son, por construcción del estándar Array, palabras código distintas.

Por otra parte, si suponemos que $\mathbf{y_i}$ e $\mathbf{y_j}$ son iguales y pertenecen a distintos COSETs, tendremos:

$$e_l \oplus c_i = e_k \oplus c_j \Rightarrow e_l = e_k \oplus \left(c_j \oplus c_i\right) = e_k \oplus c_m$$

Donde deducimos que es contradictorio que pertenezcan a distintos COSETs.

# Ejercicios de autoevaluación

**1.** Se desea realizar la codificación de 25 bits de información útil mediante un código rectangular con paridad par. Para ello, los 25 bits se organizan en una matriz de cinco filas y columnas columnas, como la que se muestra en la figura siguiente. Se os pide:
**a)** Determinar los bits de paridad necesarios para construir el código rectangular.
**b)** Calcular la tasa de redundancia.
**c)** Determinar la tasa del código.
**d)** Comparando este código con el del ejemplo 2, indicar cuál de ellos tiene una mayor protección frente a posibles errores y cuál de ellos supone un aumento mayor del ancho de banda.

**2.** Suponed un codificador que admite las siguientes palabras código: {0000} {1010} {0101} y {1111}. Los bits transmitidos son enviados por un canal con ruido blanco gaussiano a un receptor que detecta un valor de tensión de 2 voltios de media cuando transmitimos un 1 y un valor medio de –2 voltios cuando transmitimos un 0. Recibimos los valores de tensión {0.1, 2.2, –0.2, 3}. Se os pide:
**a)** ¿Que valores decidiríais que se han transmitido si utilizáis un decodificador por decisión *hard*? ¿Existe ambigüedad?
**b)** ¿Que valores decidiríais que se han transmitido si utilizáis un decodificador por decisión *soft* (mínima distancia euclídea)? ¿Existe ambigüedad?

**3.** Enumerad y discutid las posibles estrategias de corrección y detección de errores que pueden realizarse con un código de bloques de distancia mínima 7.

**4.** Determinad el estándar Array de un código (6,3) con la siguiente matriz generadora:

$$\mathbf{G} = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

**5.** Considerad un código de bloque lineal (8,3) en el que las palabras código asociadas a los mensajes de la base canónica son:

$$\mathbf{e}_1 = [1,0,0] \Rightarrow \mathbf{g}_1 = [1,0,0,1,0,1,0,1]$$
$$\mathbf{e}_2 = [0,1,0] \Rightarrow \mathbf{g}_2 = [0,1,0,0,1,0,1,0]$$
$$\mathbf{e}_3 = [0,0,1] \Rightarrow \mathbf{g}_3 = [0,0,1,0,1,1,1,0]$$

Se os pide:
**a)** Determinar la matriz generadora del código lineal.
**b)** Calcular la distancia mínima del código lineal.
**c)** Determinar la matriz de chequeo de paridad.
**d)** Representar esquemáticamente cómo se obtienen los bits de redundancia a partir de los bits del mensaje.

**6.** Determinad la matriz de chequeo de paridad y la matriz generadora de un código de Hamming con parámetro m = 2. Escribid las dos palabras código y comprobad que la distancia mínima del código es 3.

**7.** Determinad la matriz de chequeo de paridad y la matriz generadora de un código de Hamming obtenido con un parámetro m = 4.

**8.** Considerad un código de bloque lineal con la siguiente matriz generadora:

$$\mathbf{G} = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

Determinad:
**a)** La tasa del código y su distancia mínima.
**b)** La matriz de chequeo de paridad.
**c)** El estándar Array.
**d)** Si se recibe el mensaje y = 11111, determinad el síndrome e indicad si se ha producido algún error.
e) Describid un procedimiento sistemático que utilizaríais para realizar la corrección de la palabra. ¿Cuál es la palabra código que obtendríais?

**9.** Obtened la tabla completa del código cíclico (7,4) asociado al siguiente polinomio generador:

$$g(p) = p^3 + p^2 + 1$$

# Bibliografía

## Bibliografía básica

**Benedetto S.; Biglieri, E.** (1999). *Principles of Digital Transmission*. Kluwer Academic Press / Plenum publishers.

**Proakis, J. G.** (2003). *Digital Communications* (4.ª ed.). McGraw Hill.

**Proakis, J. G.; Salehi, M.** (2002). *Communication Systems Engineering* (2.ª ed.). Prentice Hall.

## Bibliografía complementaria

**Carlson, A. B.** (2001). *Communication Systems: An Introduction to Signals and Noise in Electrical Communication* (4.ª ed.). McGraw Hill.

**Gibson Jerry D., y otros** (1998). *Digital Compression for Multimedia: Principles & Standards*. Morgan Kauffman.

**Stix, G.** (1991, septiembre). "Encoding the Neatness of Ones and Zeroes". *Scientific American* (pág. 54-58).