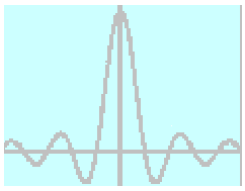
PROYECTO FIN DE CARRERA

Ingeniería Superior de Telecomunicación



Diseño e implementación de un módem APK mediante SoundBlaster

Autor: José Miguel Moreno Pérez

Director: José Ramón Cerquides Bueno

Escuela Superior de Ingenieros

Universidad de Sevilla

Marzo 2004

Reflexión personal

Un poeta inglés, ante una duda, respondió:

" Me contradigo, y qué,
lo reconozco, tengo multitudes,
y todas tienen derecho a manifestarse".

Yo, al igual que el poeta, también tengo multitudes, y todas se manifestaron, aunque débilmente, al empezar el proyecto.

Luego, una vez que éste fue tomando forma, las dudas se disiparon, y fui siendo uno solo, con una idea más definida y propia.

Quizás por eso he desarrollado y mantenido la memoria en ***primera persona del plural***, como homenaje a todas esas ideas afines y contradictorias que fueron surgiendo en mi persona.....como agradecimiento a todas esas personas que alguna vez creyeron en mí, por encima de mis dudas.....gracias.

José Miguel Moreno Pérez.

ÍNDICE

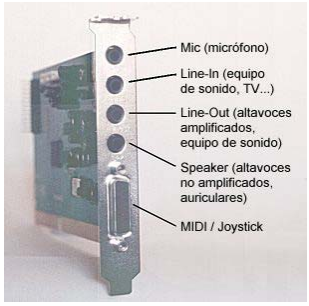
I Introducción

La tarjeta de sonido Sound Blaster de Creative Labs y similares nos ofrecen un abanico de alternativas a su uso habitual. Una de estas posibilidades es la aplicación práctica de la gran cantidad de fundamentos teóricos propios de la Ingeniería de Telecomunicaciones estudiados a lo largo de la carrera.

El objetivo principal de este proyecto es el diseño e implementación de un módem, usando una modulación digital en amplitud y fase (APK: amplitud phase Keying). Dicha implementación se hará vía software usando una herramienta muy conocida en el mundo de la ingeniería: Matlab (versión 6.1). Luego usaremos un compilador interno a este programa, para transcribir el código a lenguaje C/C++, que por ser externo a Matlab, se puede ejecutar fuera de éste y ello nos proporciona, en teoría, una gran cantidad de ventajas como velocidad de ejecución. Finalmente comprobaremos la eficacia de nuestro módem intercomunicando dos ordenadores mediante la transmisión de ficheros (bytes de datos) a través de las líneas LINE IN y LINE OUT de sus respectivas tarjetas de sonido.

Durante la memoria desarrollaremos los cálculos de todos los posibles parámetros (frecuencia de muestro, valores de Roll-off, longitud de tramas, nº de bits/símbolo) que influyen en la velocidad de transmisión, así como las causas de sus limitaciones (que originarán errores en la decisión de los bits) con el único objetivo de optimizar la transmisión en el sentido de maximizar la tasa de bits y minimizar la probabilidad de error de bit. También realizaremos un estudio del canal de comunicaciones (que para nuestro caso se trata de un simple cable de audio que unirá la entrada LINE IN de la tarjeta de sonido transmisora con la entrada LINE OUT de la tarjeta de sonido receptora, y abordaremos problemas derivados de este estudio como la limitación en banda del canal, o el cálculo de la frecuencia de portadora).

De igual forma trataremos, entre otras muchas cosas, conceptos claves en los sistemas de comunicaciones, que hemos adquirido durante la carrera, tales como problemas de sincronismo y formas de recuperación, formación de tramas IEEE 802.3, (con las modificaciones necesarias para este tipo de transmisión), sistemas M-arios, codificación de línea, modulación digital , constelaciones, interferencia intersimbólica (ISI) y su posible solución con el uso de filtros de coseno alzado, ecualización e igualación de canal y sus respectivos procesos inversos de recuperación de la señal (demodulación, detección, decisión y decodificación de la señal en recepción).



Tarjeta de sonido SoundBlaster

II Estructura del Proyecto

El proyecto lo podríamos dividir en 5 apartados fundamentales, a continuación haremos un breve resumen de ellos:

1.- Estudio del canal de comunicaciones.

Este apartado tiene como objetivo hallar la respuesta impulsiva del canal, para deducir su respuesta en frecuencia y obtener así su forma espectral (paso de baja, paso de banda, etc...), su ancho de banda y derivar unas primeras conclusiones en nuestro posterior diseño del módem, como la frecuencia de portadora. También estudiaremos el nivel de ruido, para tener una ligera idea acerca del ecualizador a implementar.

2.- Diseño del sistema de comunicaciones.

Este es el apartado más extenso de toda la memoria. En él concentraremos todos los conceptos teóricos que puedan surgir en nuestro proyecto. Se dividirá en los siguientes bloques:

2.1.- Introducción a la modulación digital.

Empezaremos el diseño haciendo una pequeña introducción a la modulación digital, y su desarrollo en las últimas décadas, para acabar hablando en este subapartado de las técnicas digitales de transmisión más comunes.

2.2.- Justificación comparativa del la modulación APK.

En este punto, se tratará de justificar el uso de la técnica de modulación APK, frente a otras como la ASK o la PSK. También comentaremos la posible transmisión en banda base mediante codificaciones como la de tipo manchester.

2.3.- Diseño de nuestro sistema de comunicaciones.

En este apartado describiremos detalladamente cada componente de los Bloques Transmisor y Receptor, analizando problemas y soluciones adoptadas en cada caso.

Asimismo se procederá a la selección de los parámetros que intervendrán en la transmisión de los datos. Para dicha selección, nos basamos, como requerimiento principal, en intentar optimizar los siguientes factores:

- La velocidad de transmisor.
- La probabilidad de error de bit.
- Aprovechamiento del espectro disponible.

2.4.- Anomalías en nuestro sistema.

Aquí recapitularemos los problemas con los que nos hemos encontrado a la hora de conseguir una transmisión correcta, y las soluciones que hemos desarrollado para paliar dichos efectos indeseados, ocupándonos principalmente del problema del sincronismo.

2.5.- Estructuración de las Tramas.

Por otra parte, y como consecuencia inmediata de la ausencia de sincronismo entre la tarjeta transmisora y la tarjeta receptora (como comprobaremos más adelante), contemplaremos la posibilidad de dividir el fichero en distintas tramas. Usaremos tramas Ethernet, por lo que nos basaremos en la recomendación IEEE 802.3., así como el uso de sistema de tratamiento de errores mediante códigos cíclicos, para detectar los posibles fallos en la decisión de los bits.

3.- Implementación en Matlab.

Aquí generaremos todas las rutinas para la transmisión de los datos. Es decir, mediante Matlab 6.1 desarrollaremos, vía software, el Transmisor y el Receptor, cuyas fases en las que se dividen ambos extremos de transmisión se describirán con todo detalle durante el desarrollo de este apartado.

Durante las fases de prueba se ha probado con todo tipo de formatos de archivos (texto, audio, imágenes), pues al fin y al cabo se trata de archivos de datos.

4.- Implementación en Lenguaje C.

Para concluir la fase práctica, procederemos a la trascripción del código desplegado en Matlab a un código en lenguaje C, obteniendo así un par de aplicaciones externas (transmisor y receptor) que se ejecutarán fuera de Matlab.

5.- Conclusiones finales y propuestas de mejora.

Para concluir, haremos un breve resumen, paso a paso, destacando las principales conclusiones que sobresalen en la ejecución del proyecto, y propondremos una serie de mejoras para futuras ampliaciones del proyecto en cuestión.

Antes de terminar este apartado, nos gustaría añadir simplemente que la estructura que vamos a desarrollar en la realización de esta memoria también es la que hemos seguido a la hora de su diseño e implementación práctica.

III Desarrollo de la memoria

III.1.- ESTUDIO DEL CANAL DE COMUNICACIONES

El primer paso que hay que dar antes de realizar ningún tipo de pruebas en el canal, es montar el propio Sistema de Comunicaciones. Esto es, unir el transmisor (o tarjeta transmisora) con el receptor (o tarjeta receptora) a través de un canal, que precisamente va a ser objeto de estudio en este primer apartado.

Para ello, usamos como canal de comunicaciones un CABLE DE AUDIO, y conectamos un extremo al puerto de salida LINE OUT de la tarjeta de sonido transmisora, y el otro extremo lo introducimos en la entrada Line IN de la tarjeta receptora.

Existen dos formas de montar dicho sistema, dependiendo de la capacidad que tenga la tarjeta para recibir y transmitir datos:

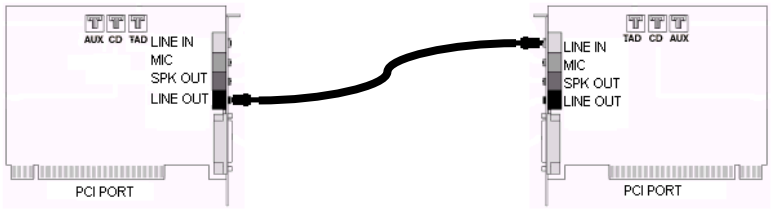


Fig. 1.- Esquema de conexión

- Si la tarjeta es ***full dúplex***, es decir, si puede transmitir y recibir al mismo tiempo o, dicho de otra manera, si es capaz de reproducir y grabar datos simultáneamente, entonces podemos montar nuestro sistema en un mismo PC, comunicando, con el cable de audio o canal, las líneas de entrada y salida a las que hicimos alusión en el párrafo anterior. De este modo, los Bloque Transmisor y Receptor se encuentran en la misma tarjeta de sonido. Ese es nuestro caso, y, por tanto, el proyecto lo hemos podido llevar a cabo en nuestro PC. Véase aquí un esquema ilustrativo:

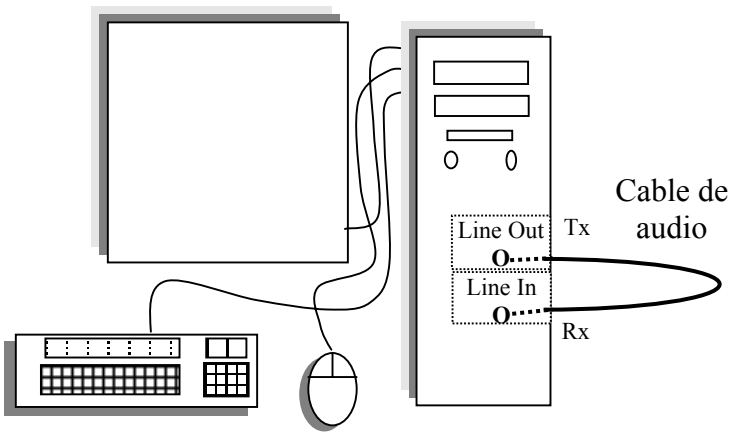


Fig. 2.- Tarjeta Full Dúplex

- Si, por el contrario, la tarjeta de sonido es ***semidúplex***, esto es, tan sólo es capaz de transmitir o recibir en un mismo intervalo de tiempo, entonces tendremos que recurrir a dos tarjetas de sonido, que generalmente se hallarán en PC's distintos. Así, si queremos que la comunicación sea bidireccional, necesitaremos dos cables de audio para comunicar, por un lado, la línea LINE OUT de la tarjeta de sonido del PC Transmisor, y por otro, la línea LINE IN de la tarjeta del PC Receptor. Pero para simplificar las cosas, supondremos que la comunicación será unilateral, por lo que sólo necesitaremos un cable de audio.

Véase aquí otro ejemplo ilustrativo:

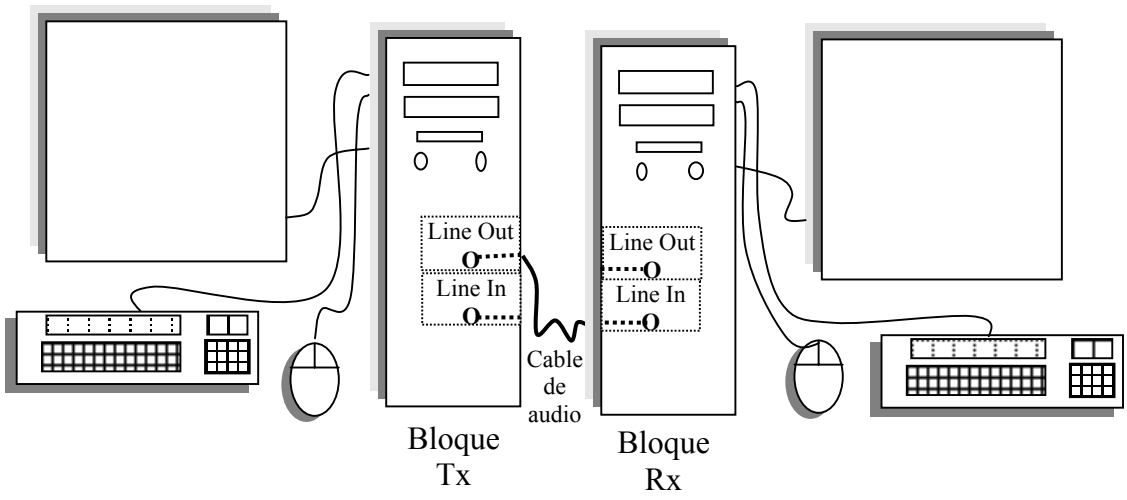


Fig. 3.- Tarjeta Full Dúplex

En cualquier caso, si el reloj de la parte reproductora de datos es ligeramente distinto del reloj de la unidad receptora de datos (ni que decir tiene, que, en el primer caso, la tarjeta full dúplex constaría de dos relojes), aparece en la transmisión un problema derivado de ese desajuste (que generalmente no va más allá de 0.5 hz. en las tarjetas normales): el ***sincronismo***. Pero este problema lo abordaremos más adelante.

Ahora lo que nos interesa es ver la forma de onda de la respuesta impulsiva del canal, y su respuesta en frecuencia. Para ello, la forma más fácil es transmitir una delta de Dirac a través del cable, y observaremos los resultados en recepción.

Pero antes de nada, vamos a realizar una serie de ajustes en el mezclador (o mezcladores, si estamos en el segundo caso), para adaptar la señal a transmitir a valores permisibles en el cable de audio, en la propia tarjeta, y en funciones de reproducción y grabación propias de las librerías de Matlab. Así, ya que el único hardware que vamos a usar para implementar el módem es la tarjeta de sonido, vamos a conocerla un poco por dentro. En la página siguiente intentaremos realizar un resumen gráfico de los componentes esenciales de una dicha tarjeta:

III.1.1.- Resumen esquemático y configuración de una tarjeta de Sonido

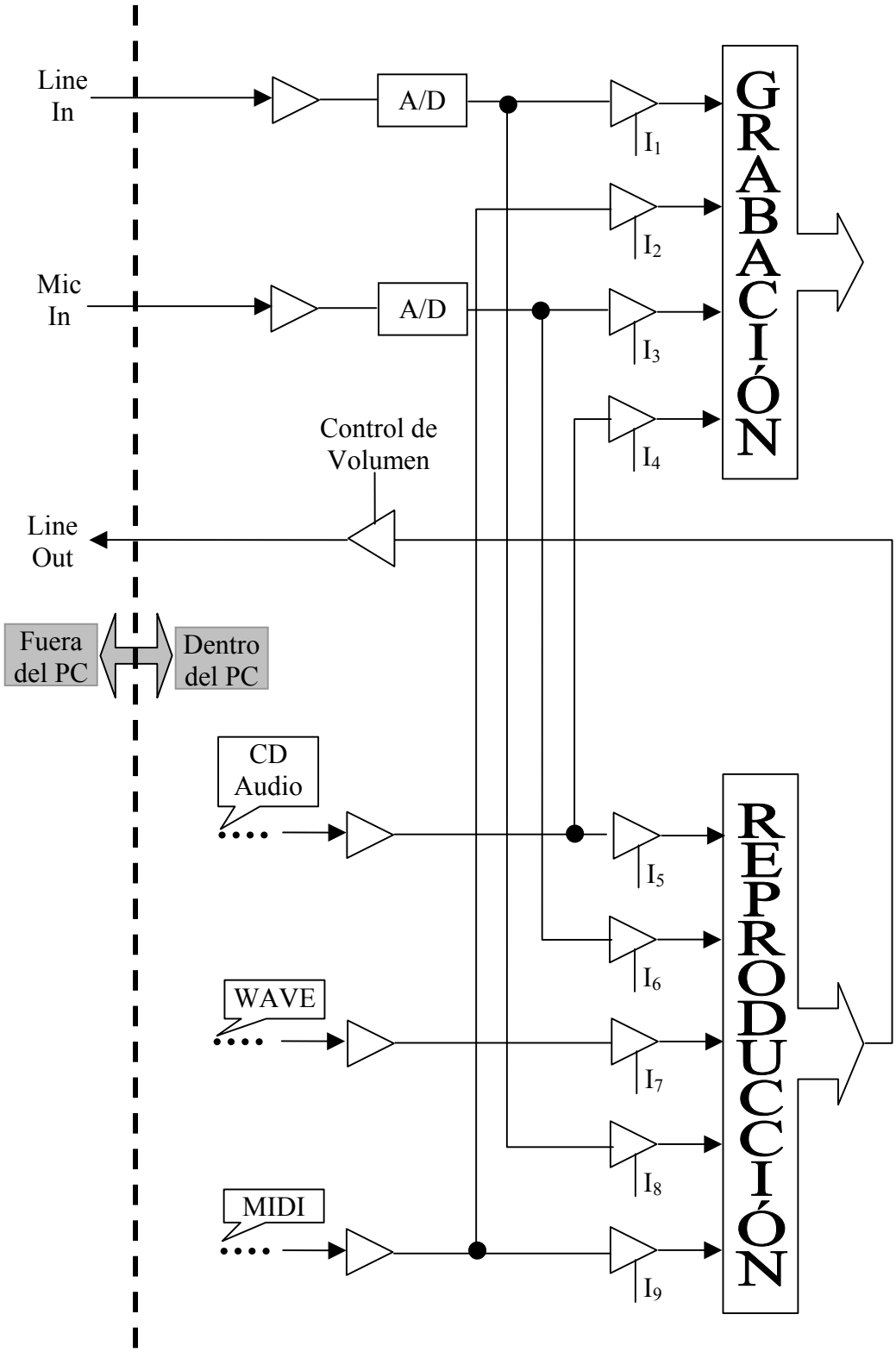


Fig. 4.- Tarjeta de sonido

A simple vista, cabe destacar, en el esquema anterior, que la tarjeta de sonido tiene dos bloques mezcladores bien diferenciados[†8]:

- El bloque de ***Grabación***. En él se controla la fuente externa de sonido que se quiere grabar.
- El bloque de ***Reproducción***. En él se controlan las diversas fuentes de sonido que queremos enviar al exterior, mediante el puerto de salida LINE OUT. Cabe Resaltar que éste es el verdadero mezclador, puesto que pueden seleccionarse varias fuentes simultáneamente, en contraste con el anterior bloque.

Hay que hacer notar que hemos citado, de entre los dispositivos que acceden a la tarjeta de sonido, a los más usuales. Aquí también hay que hacer otra distinción, la que existe entre las fuentes de sonido que proceden la zona externa al PC, y las que están ubicadas en la zona interna al equipo[†8] [†7]:

- Las *fuentes externas* acceden a la tarjeta de sonido a través de los puertos de entrada LINE IN y MIC IN. Por ejemplo, MIC IN constituye la entrada específica del micrófono.
- Las *fuentes internas* son las que proceden de dentro del equipo, como WAVE, MIDI y CD AUDIO. Esta última procede de un dispositivo periférico, pero se considera interna, porque la conexión del reproductor /grabador de CD's es interna al PC.

Hemos querido reflejar de forma gráfica, mediante salidas en triestado, la manera de seleccionar (o silenciar) los dispositivos que accederán, en un momento dado, a cualquiera de los dos bloques del mezclador. Según el valor de los interruptores 'I_n' (con 'n' tomando cualquier valor entero entre 1 y 9), su correspondiente dispositivo estará seleccionado o no, esto es:

- Si 'I_n'='1'→ su dispositivo correspondiente está seleccionado.
- Si 'I_n'='0'→ su dispositivo en cuestión está en estado de alta impedancia y no accederá a la tarjeta de sonido. Se dice entonces que está silenciado.

Nuestro objetivo no es otro que configurar la tarjeta de sonido de manera que consiga transmitir, a través de la línea LINE OUT, los datos que hemos generado en el PC, y recoger dichos datos a través del puerto LINE IN. Para ello, la posición de los interruptores debe ser la siguiente:

- En reproducción:
  - 'I_7'='1'. WAVE seleccionado. Así podremos enviar datos almacenados en el PC a través de la línea LINE OUT.
  - 'volumen de control'='1', pues por esa puerta pasa todo lo que sale del bloque reproductor a través de LINE OUT. Si se deja abierto, no saldrá nada por dicho puerto de salida.
  - Todas las demás fuentes que accedan al reproductor deben estar con su correspondiente etapa de triestado en alta impedancia.

- En grabación:
  - 'I_1'='1'. LINE IN seleccionado.
  - Al seleccionar LINE IN, las demás fuentes se deseleccionan automáticamente.

Veamos, en Windows 2000, las ventanas donde se encuentra el software que selecciona los distintos dispositivos que acceden a la tarjeta de sonido, y desde donde se controla su volumen:

➢ **Control de Reproducción**:

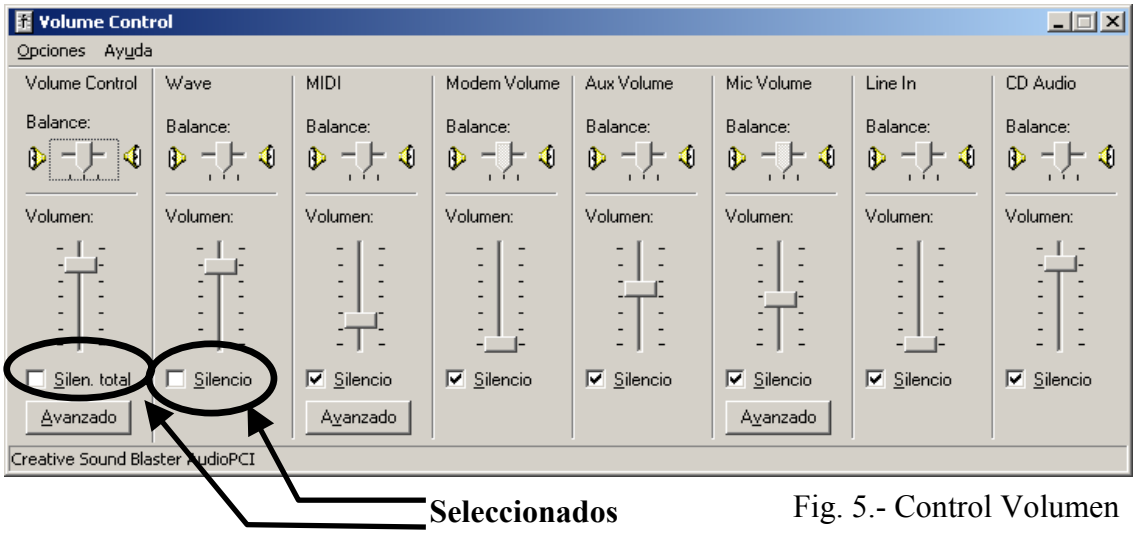


Fig. 5.- Control Volumen

Como podemos comprobar, están silenciados todas las fuentes de sonido excepto la que proviene del disco duro del PC (WAVE) y, lógicamente, el 'Control del Volumen', que amplifica o atenúa todo el sonido que sale del reproductor.

➢ **Control de Grabación**:

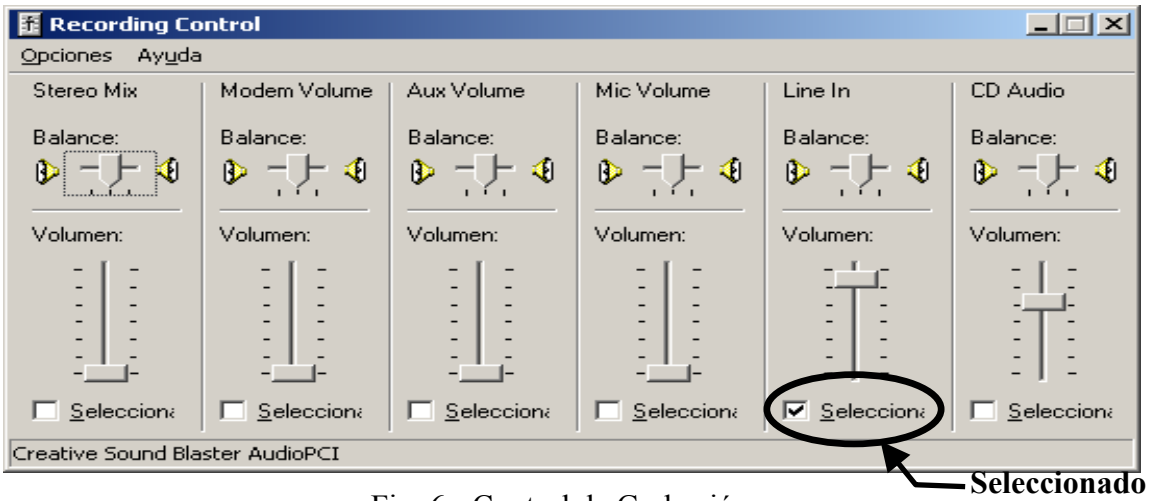


Fig. 6.- Control de Grabación

Un detalle **MUY IMPORTANTE** es, antes de nada, ***anular todo tipo de efectos 3D*** que formen parte de las propiedades de la tarjeta. Si no se hace, se producirán efectos no deseados en la transmisión, que provocarán el fallo de nuestro sistema[†7] .

Un ejemplo claro de estos efectos 3D es el ***eco***. Cuando empezamos a estudiar la forma del canal, procedimos a transmitir un impulso delta de Dirac, para observar la respuesta impulsiva, sin embargo obtenía dicha respuesta por duplicado, un cierto intervalo de tiempo posterior a la llegada de la primera respuesta.

Empezamos a investigar, y abrimos la ventana de CONTROL DE REPRODUCCIÓN (la que hemos mostrado un poco más arriba), y pulsamos el botón ***Avanzado*** que está en la columna CONTROL DE VOLUMEN. Se nos volvió a abrir una nueva ventana, en la que aparecía una pestaña que estaba activada, la desactivé, y efectivamente, correspondía al control del 'eco'. Esta es la ventana en cuestión:

Además, esta es la posición de los tonos que mejor se ajusta a una especie de sampling más o menos simétrica en sus lóbulos.

Esta es la pestaña que hay que desactivar para eliminar el efecto del eco.

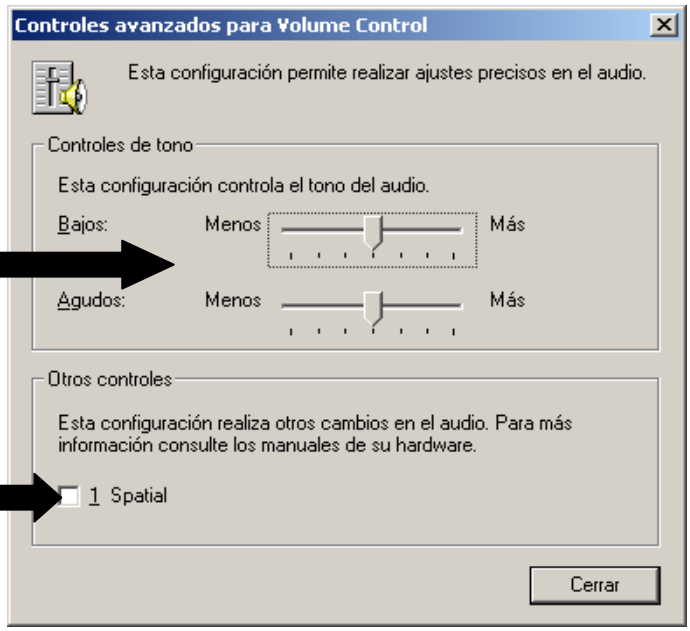


Fig. 7.- Control avanzado de Volumen

NOTA: En el gráfico anterior, hacemos un comentario sobre el ajuste de los tonos. Este comentario es producto de posteriores pruebas, en la que concluimos que la respuesta impulsiva deberá tener la forma de una función sampling, y con esta configuración conseguimos lo más parecido a una función de estas características.

III.1.2.- Formato de Reproducción /Grabación

Una vez hemos establecido todos los ajustes de la tarjeta de sonido, procederemos a escoger el formato que usaremos en la reproducción /grabación. Usaremos el conocido ***formato de audio PCM***, con un conjunto estandarizado de atributos, denominado ***Calidad de CD monocanal***, con los siguientes valores[†7]:

- ***Formato: PCM***, como hemos dicho anteriormente.

- ***Frecuencia de muestreo: 44100 hz***.
- ***Nº de bits de resolución: 16 bits/ muestra***, son los bits que se usarán para poder codificar las muestras analógicas que salen del CDA del transmisor hacia el canal, o que acceden al CAD del receptor proveniente del cable de audio. Así el cuantizador tendrá 2^{16} ***niveles de cuantización***.
- ***nº de canales: monocanal***., ya que sólo vamos a emplear un canal, aunque el cable puede ser, indistintamente, mono o estéreo. Sin embargo, como ya hemos comentado, sólo usaremos un hilo de transmisión.

Hemos escogido unos valores relativamente elevados para la frecuencia de muestreo y para la resolución de las muestras cuantizadas con el único objetivo de optimizar el sistema en cuanto a aumentar su velocidad de transmisión y disminuir la probabilidad de error de bit en dicho sistema.

De todos modos, ambos parámetros son configurables durante la ejecución del programa que hemos diseñado, para hacerlo así más versátil y adaptable a otras tarjetas de audio más antiguas, que no admitan estos valores extremos.

III.1.3.- Ajuste de Amplitudes

En el anterior apartado, concluimos diciendo que para la transmisión de la señal vamos a usar un sistema PCM. El esquema básico en ambos extremos, es el siguiente:

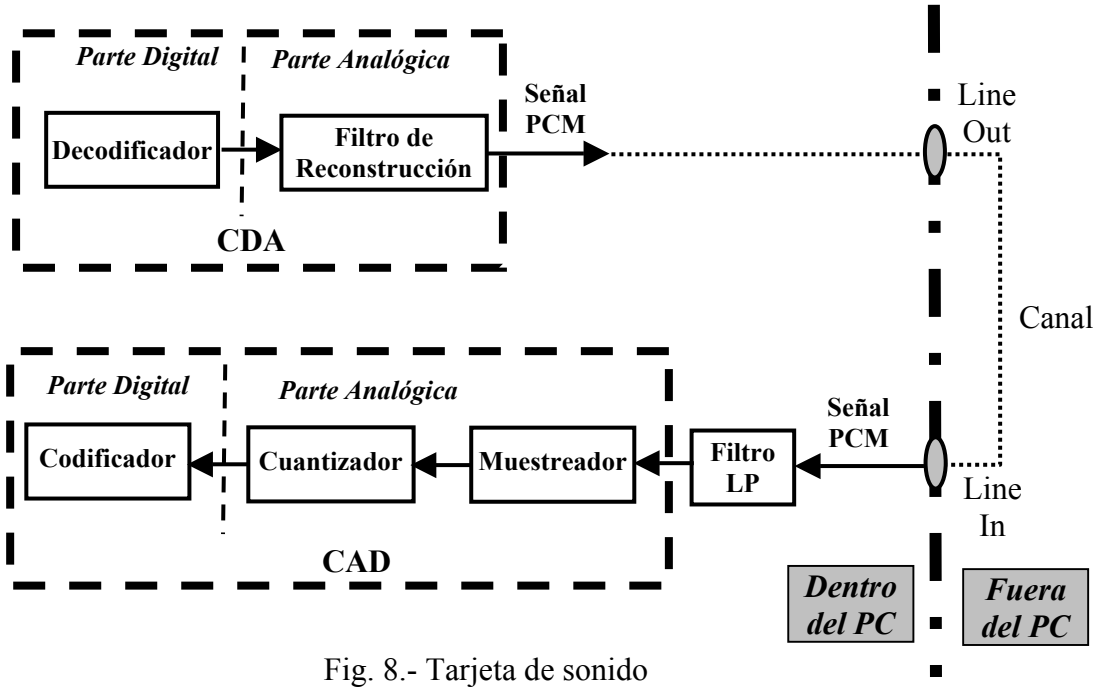


Fig. 8.- Tarjeta de sonido

El rango de salida la zona digital es el intervalo [-1,1]. Por este motivo, al usar cualquier función de Matlab que transmita datos desde la tarjeta de sonido a través de LINE OUT, dicha función recorta siempre los valores que excedan de este rango, es decir[†6]:

- Los valores > 1 serán redefinidos a este valor.
- los valores < -1 serán recortados a este valor.

El siguiente gráfico es un esquema de lo que podría ser la característica de transferencia del cuantizador, donde m(t) es la señal de entrada al cuantizador y $m_q(t)$ es dicha señal cuantizada. En este caso hemos dibujado 5 niveles de decisión, para simplificar la complejidad de un cuantizador con 2^{16} niveles como el nuestro[†1]:

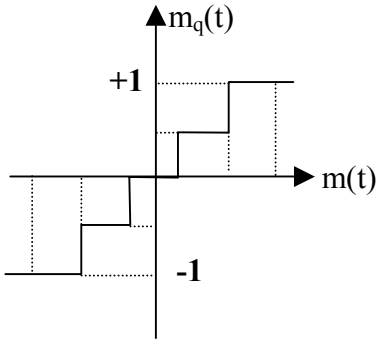


Fig. 9.- Cuantizador

Se puede comprobar que los niveles discretos del cuantizador están normalizados entre 1 y -1, y para valores mayores en valor absoluto que éstos la salida satura.

El formato usado en codificación es 'complemento a2' para nº binarios con signo. Así, los valores máximo y mínimo de la tabla del codificador serán los siguientes:

	Valor mostrado en Pantalla	Entrada del codificador
nivel max.	+1	2^{15}-1
nivel min.	-1	-2^{15}

Tabla 1.- Codificador Tarjeta

En definitiva, la parte digital está normalizada entre –1 y 1, esto es, tenemos 2^{16} niveles distribuidos entre dichos valores –1 y 1, y ante esta situación, tenemos que ajustar los niveles a la entrada y salida de la tarjeta, para no recibir una señal demasiado atenuada, ni recibir una señal saturada y recortada (con la pérdida de información que esto conlleva).

Para ajustar estos niveles generaremos una senoide de 10 Khz. (que estará situada en la mitad del ancho de banda del canal, como veremos posteriormente) y de amplitud unidad (el máximo a transmitir sin que sobre pase dicho umbral de saturación). De esta manera vamos regulando el valor del CONTROL DE VOLUMEN y el de WAVE en reproducción y el de LINE IN en grabación, hasta conseguir aproximadamente un valor entre 0,6 y 0,8.

PASOS PARA TRANSMITIR Y RECIBIR UNA SEÑAL

Lo principal es tener dos reproductores /grabadores de sonido. Particularmente hemos usado la grabadora de sonido de Windows, y un programa grabador, mezclador y reproductor de sonidos: '***Cool edit'*** ,el cual nos ha sido de mucha ayuda en las pruebas previas y durante el diseño del módem, ya que nos aporta una forma más dinámica y potente de análisis que las funciones gráficas de Matlab.

Generamos la senoide de 10 Khz. Grabamos su contenido en el fichero "seno10.wav", iniciamos sesión en la Grabadora de sonidos de Windows, abrimos el archivo, lo transmitimos, lo recibimos con la grabadora de *'Cool Edit'* y lo generamos como *'seno10rx*'. Ya tenemos listo el archivo *'seno10rx'* para abrirlo y analizar, en este caso concreto, los valores extremos de sus amplitudes.

III.1.4.- Respuesta impulsiva del Canal

Ya hemos comentado que para hacer un estudio del canal de comunicaciones, lo mas obvio es estudiar su respuesta impulsiva. Para ello transmitiremos una señal delta de Dirac a través del cable de audio y recibiremos justamente la respuesta impulsiva a la entrada del receptor[†2]. Gráficamente:

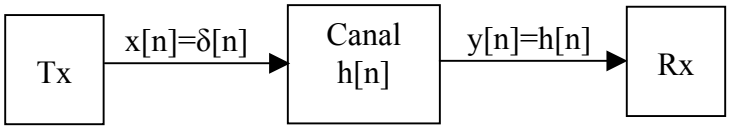


Fig. 10.- Sist. comunicación

Para generar una delta de Dirac, hay que recurrir ya al programa Matlab, y a su particular lenguaje de programación. Existen muchas formas de generar y transmitir dicha delta, he aquí dos ejemplos:

Ejemplo nº 1

```
%delta de dirac
%ejemplo 1º
d=1;
sound(d,44100,16);
```

Ejemplo nº2

```
%delta de dirac
%ejemplo 2º
d=1;
wavwrite(d,44100,'delta.wav');
```

En el EJEMPLO Nº 1, creamos un vector de un solo símbolo unidad, y hacemos uso de la función ***sound*** para transmitir desde WAVE (a través de LINE OUT) ese vector, y recogerlo en la grabadora de 'Cool Edit', como fichero ***'.WAV'***. (*'deltaRx.wav'*).

En el EJEMPLO Nº 2,creamos el mismo vector, pero en lugar de transmitirlo, lo generamos en un archivo *'.WAV'* del disco duro. Luego usamos la grabadora de sonido para reproducir dicho fichero, recogerlo con Cool Edit, y guardarlo en *'deltaRx.wav'*.

Una cosa importante a tener en cuenta es que, para evitar transitorios indeseados durante la grabación, es conveniente grabar con segundos de antelación a la reproducción de los datos.

Las siguientes gráficas se corresponden con la respuesta impulsiva del sistema, muestra a muestra (usando el comando ***stem***), y de forma interpolada (mediante los comandos ***interp*** y ***plot***).

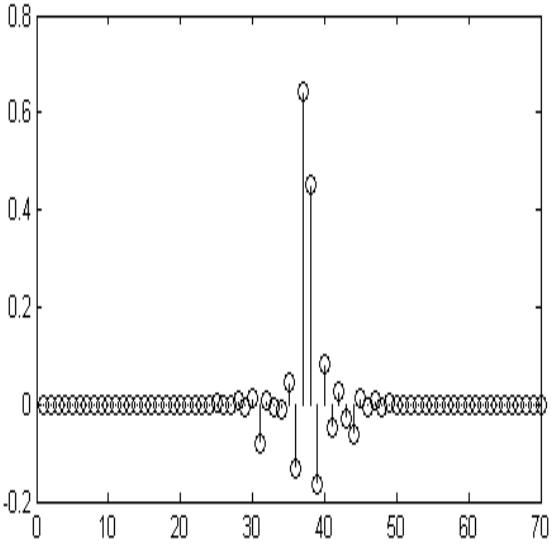


Fig. 11.- h[n], forma discreta

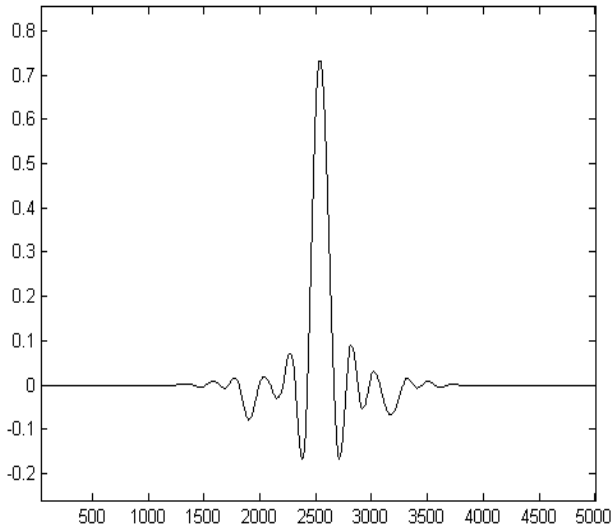


Fig. 12.- h[n], forma interpolada

En las gráficas podemos destacar dos aspectos fundamentales:

- El gran parecido de h[n] con la función sampling.
- una ligera atenuación, pues el máximo no alcanza la unidad.

Para indagar más en el estudio y caracterización del canal, vamos a pasar a estudiarlo en el dominio de la frecuencia. Hallemos la ***respuesta en frecuencia*** del canal mediante la DFT de su respuesta impulsiva. Usaremos el conocido algoritmo ***fft***[†4]. La secuencia de comandos sería:

(***Nota***: '%' inicia un comentario hasta el final de línea)

```
% Leemos del disco duro el fichero donde residen las muestras de h[n].
   h=wavread('deltaRx.wav');       %h: respuesta impulsiva
% hallamos la DFT de h[n], usando zero-padding hasta llegar a 2^16 puntos
   H=fft(h,2^16);   %H: respuesta en frecuencia
% la frecuencia digital ω tomará valores discretos, ωk, en función de 'k':
% ωk=2Πk/(2^16) ;  k=0,1,2,.... 2^16-1
% hallamos los valores discretos de la frecuencia digital ωk
   w=freqspace(2^16,'whole')*pi;
% la relación entre la frecuencia analógica y la digital es:
% f= ω·Fs/2Π;  donde Fs= frec. muestreo=44100.
% así, para cada valor ωk, ob tenemos un valor fk
   f=w*44100/(2*pi);
% ya podemos dibujar la respuesta en frecuencia del sistema.
```

*subplot(2,1,1);plot(f,20*log(abs(H)));*
subplot(2,1,2);plot(f,angle(H));

La figura que resulta es la siguiente:

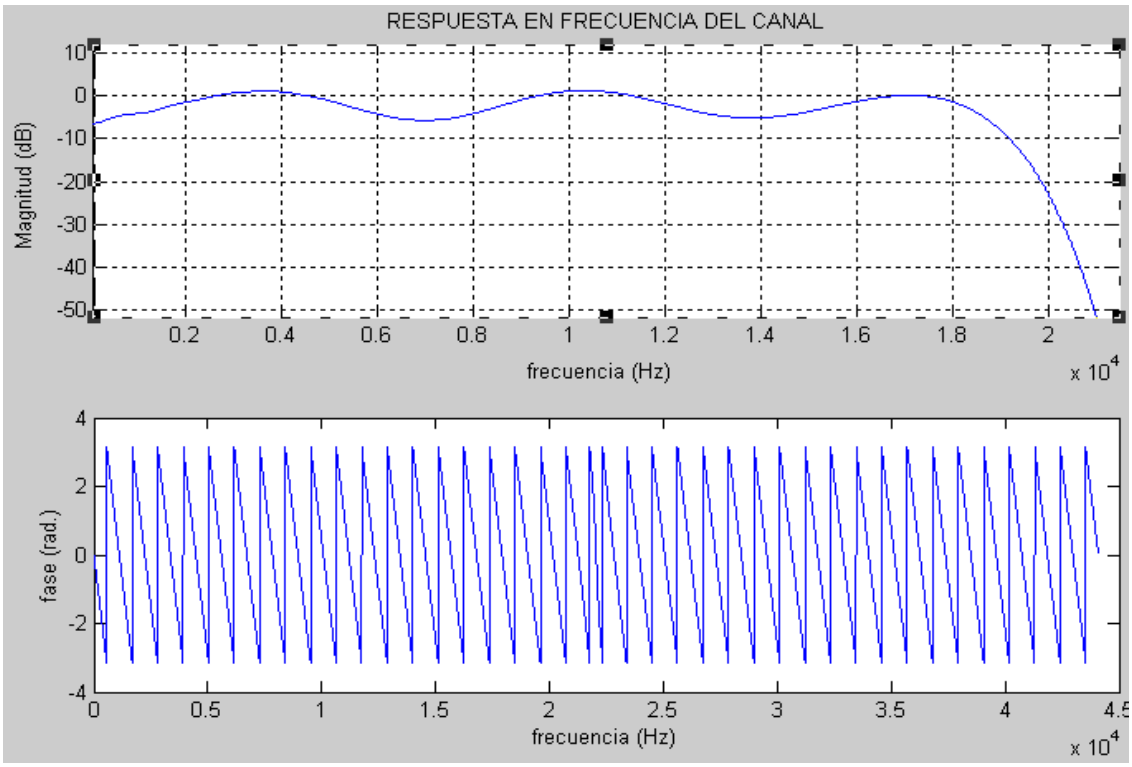


Fig. 13.- Respuesta frecuencial del canal

Sobre la ***fase de la respuesta en frecuencia del canal***, cabría comentar, que el ***retraso del canal*** es ***constante***, pues la fase es función lineal de la frecuencia angular y, por tanto, dicho retraso de canal afecta por igual a todo el rango de frecuencias del ancho de banda que abarca el canal. Veamos, a partir de la gráfica, y a modo de ejercicio teórico,cómo hallar dicho retraso:

- El retraso, τ, la fase de la respuesta en frecuencia,Φ, y la frecuencia angular, Ω, se relacionan de la siguiente manera[†2]:

$$\tau = -\Delta\Phi/\Delta\Omega \text{ (seg.)} \qquad \Phi = -\tau\Omega \text{ (rad.)}$$

- Valiéndonos de los datos del script de la página anterior, creamos un fichero que contenga los datos de la fase, para estudiar uno de sus ciclos:

```
% Atenuamos la señal por cuatro para que no sature, al salirse del
  rango [-1,1]
% Recordemos que H era la variable que contenía la respuesta en frec.
  wavwrite(angle (H)/4,44100,16,'faseDeltaRx.wav');
```

- Hay que hacer notar que la imagen, de un solo ciclo, que hemos impreso en esta página está hecha con el editor de imágenes de Matlab, y aquí la señal se presenta sin atenuar, pero que el estudio de las muestras de la función de fase está hecho mediante el programa 'Cool Edit', pues se realiza de manera más rápida:

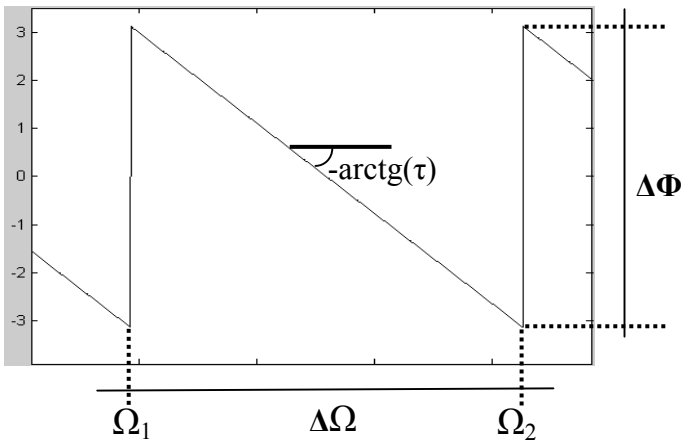


Fig. 14.- Ciclo de la fase

- Mediante el estudio de un ciclo en 'Cool Edit', obtenemos los siguientes resultados:

  - $\Delta\Phi/4$ es aproximadamente $-\Pi/2=-1{,}5708$. Sin tener en cuenta la atenuación provocada → $\Delta\Phi= -2\Pi$. Como cabía esperar.
  - A partir de sendas posiciones discretas k_1 y k_2 correspondientes a Ω_1 y Ω_2, y haciendo uso de las relaciones entre las distintas variables (que se detallan a continuación), obtenemos los siguientes valores:

$\Omega=\omega\cdot Fs$

$\omega_k=2\Pi k/(2^{16})$

$\Omega=2\Pi\cdot f$

Ω: frecuencia angular analógica. (rad/sg)
ω_k: frecuencia angular digital discreta.(rad/muestra)
ω: frecuencia angular digital continua.(rad/muestra)
Fs: frecuencia analógica de muestreo (muestras/sg)
f: frecuencia analógica (muestras/sg.)

  - $k_1 = 25800$ → $\omega_{k1}=2{,}4735$ rad/muestra → $f_1=17361$ rad/sg
  - $k_2 = 27465$ → $\omega_{k2}=2{,}6332$ rad/muestra → $f_2=18482$ rad/sg

  - Con estos valores, ob tenemos $\Delta\Omega$ y τ :

    - $\Delta f = f_2-f_1 = 1121$ hz → $\Delta\Omega = 7050$ rad/sg
    - $\tau = -\Delta\Phi/\Delta\Omega = 2\Pi / 7050 = 8{,}9\cdot10^{-4}$ sg

Así podemos concluir con el ***retraso*** realizando las siguientes afirmaciones:

- Es constante e independiente de la frecuencia.
- El retraso de fase coincide, por tanto, con el de grupo.
- Es inapreciable, pues su valor es ínfimo.

Respecto a la ***amplitud de la respuesta en frecuencia del sistema***, podríamos formalizar los siguientes comentarios:

- A simple vista, parece que nuestro canal constituye un sistema LPF(Low Pass Filter).
- Pero si hacemos un *'zoom'* en torno a la frecuencia de DC (0 hz.), podemos apreciar con más detalle que existe una caída de la amplitud de la respuesta en frecuencia, por lo que podemos sospechar que se trata de un ***sistema BPF***.
- Para cerciorarnos de la anterior afirmación, transmitiremos a través del canal una señal continua de máxima amplitud:

```
a=ones(1,1000);
sound(a,44100,16);
```

En 'Cool Edit' recibimos lo siguiente:

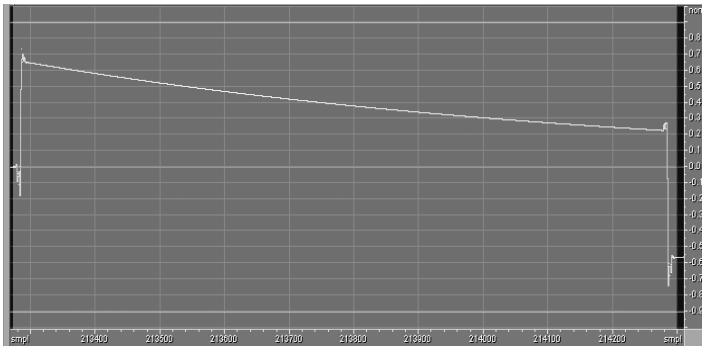


Fig. 15.- Escalón unitario a la Salida del canal

- Como presentíamos, el canal bloquea toda señal de continua, de manera que sólo existe salida en régimen transitorio, con una constante de tiempo inversa al ancho de Banda del Sistema. Es un resultado muy parecido a la respuesta de un sistema de primer orden, como un circuito RC, y más concretamente cuando a la entrada generamos una tensión constante mayor que la tensión del condensador y medimos el voltaje en la resistencia[†9].
- Así que podemos concluir que se trata de un ***sistema BPF***, cuya frecuencia de paso está en el intervalo de frecuencias ***de 20 hz. a 20 Khz.***, que coincide con el rango de frecuencias audible para el ser humano, lo cual es obvio al tratarse de un cable de audio.
- Ya podríamos fijar un parámetro de diseño: la ***frecuencia de portadora modulada***. Un valor interesante sería ***11025 hz.***, ya que se encuentra aproximadamente en el centro de la banda de paso, y además la frecuencia de muestreo, 44100 hz., es un múltiplo de dicho valor frecuencial, con las ventajas visuales que esto nos traerá más adelante.

- Por otra parte, podemos ver que existe un pequeño rizado en la banda de paso del módulo de la respuesta en frecuencia. En ausencia de este rizado, la respuesta en frecuencia, en módulo, constituiría una función rectángulo y, en consecuencia, dada la dualidad función rectángulo-función sampling[†2], la respuesta impulsiva del canal constituiría una función sampling. Pero si nos volvemos a fijar en dicha respuesta impulsiva, los lóbulos no se van atenuando progresivamente con el tiempo, sino que aparece una caída pronunciada en el tercer lóbulo secundario (siendo éste mayor que el anterior), que demuestra que no se trate exactamente de una función sampling. Por tanto, esa degeneración de la función sampling en la respuesta impulsiva del sistema, se traduce en un rizado en su respuesta en frecuencia.

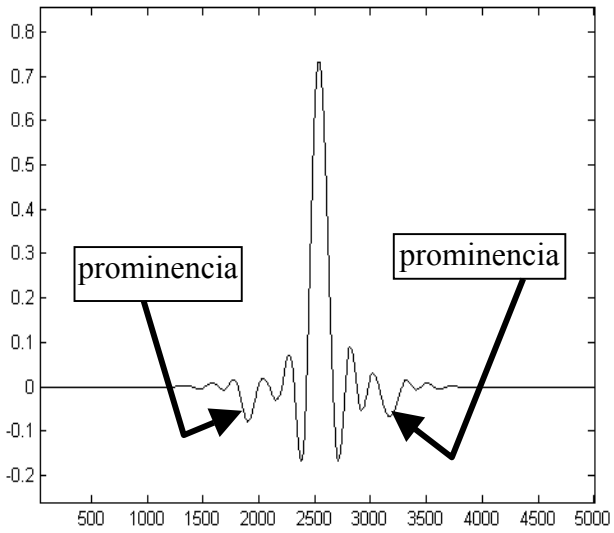


Fig. 16.- Respuesta impulsiva

- El hecho de que la respuesta en frecuencia, en su banda de paso, no sea totalmente plana, implica que el canal atenuará de distinta forma la señal que transmitamos a través de él. Por tanto, existirá una ***distorsión en la amplitud***, porque dicha atenuación dependerá del rango de frecuencias del espectro de la señal de entrada.
- Así, y para concluir el estudio de la respuesta en frecuencia, podemos adelantar que tendremos que introducir, a la entrada del receptor, un ***igualador de canal o ecualizador***, para eliminar la distorsión en amplitud que introduzca el canal. Se trata de un sistema inverso al del canal, tal que, en conjunto, el sistema en cascada se comporte como un sistema ideal [†1][†2] (en la banda de paso):

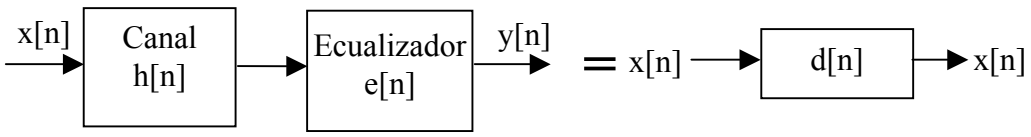


Fig. 17.- Sist. Comunicaciones

III.1.5.- Análisis de ruido del Sistema

La necesidad de introducir un ecualizador a la salida del canal, nos lleva a analizar el ruido del sistema, con el fin de ver si sus efectos son significativos. Si fuera este el caso, tendríamos que implementar un filtro de Wiener, que se usa en sistemas con un ruido considerable. Sin embargo, si el ruido existente en nuestro canal es despreciable, podemos utilizar filtros que no tengan en cuenta dicha existencia parásita, como un filtro inverso o un filtro de cero forzado(ZF) [†4].

Vamos a llevar a cabo el análisis, que en nuestra opinión, nos parece el más intuitivo: ***la transmisión en vacío***. Esta prueba consiste en transmitir una cadena larga de ceros a través del canal. Esto equivale a no transmitir nada. De esta manera, lo que recibimos en el otro extremo será el ruido presente en el canal. También analizaremos la posible existencia de interferencias procedentes de los principales armónicos de la red eléctrica:

- 50 hz.
- 100 hz.
- 150 hz.

Un modelo aproximado de un canal con ruido sería el siguiente:

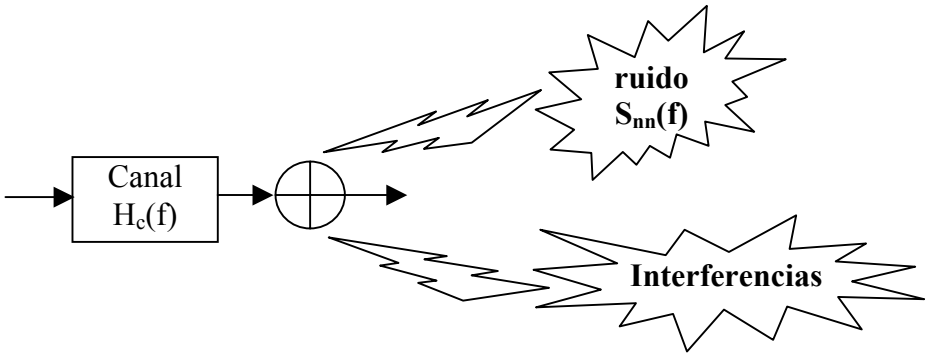


Fig. 18.- Ruido e Interferencias

Para la transmisión en vacío, volvemos a echar mano de la herramienta 'Cool Edit', aunque también se podría haber estudiado en Matlab, pero insistimos en que el editor de imágenes del programa que usamos es más eficiente.

- Transmisión en vacío

```
% generamos tres segundos de ceros.
% transmitimos el vector de ceros y lo recogemos en el receptor
a=zeros(1,44100*3);
sound(a,44100,16);
```

A simple vista, parecía que no se había grabado nada, pero al hacer un *'zoom'*, veíamos que, en efecto, el ruido estaba inmerso en el canal. La gráfica del ***ruido recibido***, ampliada con el *'zoom'* es la siguiente:

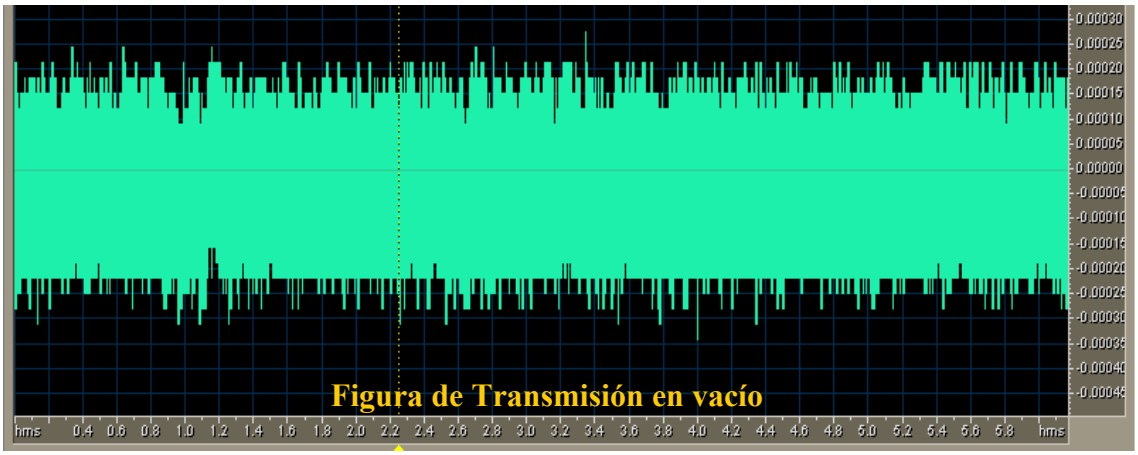


Fig. 19.- Transmisión en vacío

Se puede apreciar que los valores máximos y mínimos están en torno a ±0.00025. Por tanto, concluimos que el rango de amplitudes del ***ruido*** es ***poco influyente***, frente a las que tendremos para los símbolos que vamos a transmitir a través del canal (el más pequeño, para M=256 niveles/ símbolo, será de $1/(15\cdot\sqrt{2})= 0.0471$, como ya veremos al analizar la constelación de símbolos de nuestro sistema).

Así, desecharemos el uso de un filtro Wiener, y nos decantaremos por la diseño y implementación de un ***filtro de cero forzado(ZF)*** [†4].

También sería importante ver la ***distribución en frecuencia de la potencia de ruido recibido***[†2], para ver, entre otras cosas, la existencia de posibles interferencias, como ya comenté anteriormente. Para ello calcularemos su densidad espectral. Además, nos servirá como ejercicio práctico con funciones del programa Matlab, para ir adentrándonos, un poco más, en el cavilante mundo de la programación:

- Densidad espectral de la potencia del ruido recibido:

```
% Primero grabamos el ruido recibido en el fichero 'ruidoRx.wav':
% El script creado es el siguiente:
% Leemos el ruido recibido registrado en el fichero 'ruidoRx.wav'
    ruido=wavread('ruidoRx');
% estimacion de Snn(f) a la salida
% Snno: Densidad espectral de potencia a la salida del canal
% F: Rango de frecuencias en Hz.
    [Snno,F]=PSD(ruido,2046,44100);
% Imagen de dicha PSD a la salida, en decibelios
    plot(F,10*log10(Snno));grid;
    xlabel('Frecuencia');
    ylabel('Densidad espectral de potencia de ruido (db)');
```

A continuación mostraremos el resultado gráfico de dicho script:

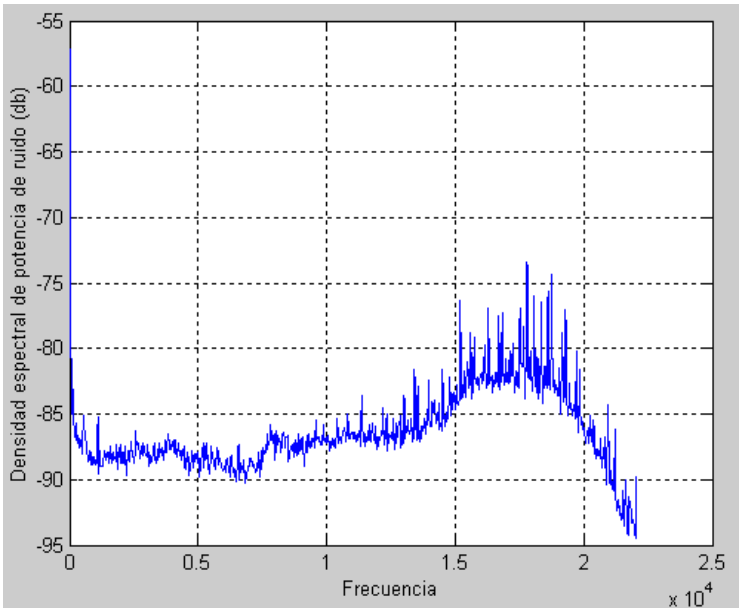


Fig. 20.- PSD ruido recibido

Obsérvese aquí también la escasa influencia del ruido, con unos valores ínfimos medidos en db. de potencia. También es apreciable que el ruido se concentra en frecuencias altas.

Para estudiar las posibles ***interferencias*** provocadas por la ***red eléctrica***, haremos uso de la escala logarítmica que nos proporciona el programa 'Cool Edit', para comprobar si realmente aparecen dichas anomalías no deseadas:

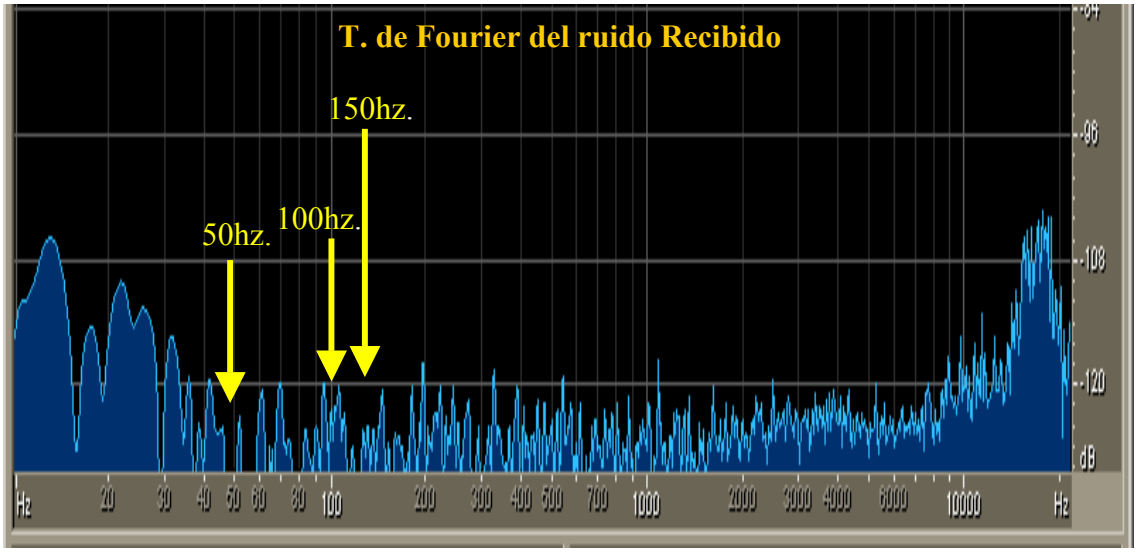


Fig. 21.- Interferencias red eléctrica

Podemos concluir, que, ante la ausencia de valores elevados a la frecuencia de estos armónicos, ***no existen interferencias provenientes de la red eléctrica***. Además, dichos valores no nos afectarían significativamente, pues nuestro espectro de frecuencias estará centrado en torno a los 11025 hz., como ya hemos comentado anteriormente.

Para terminar con el estudio del canal, vamos a ***confirmar la veracidad de la respuesta en frecuencia*** obtenida anteriormente mediante la FFT de su respuesta impulsiva. Además, vamos a ver conocer otra alternativa para hallar dicha respuesta en frecuencia, a través del análisis de un ruido blanco que introduciremos en el canal.

El ejercicio práctico de la siguiente alternativa se basa en la relación que existe entre las densidades espectrales de potencia (DSP) a la entrada y salida de un sistema (nuestro canal, en este caso), y la propia Respuesta en frecuencia del sistema. Dicha relación la hemos plasmado en la siguiente fórmula[†1]:

$$Snn_o(f)= |H_c(f)|^2 \cdot Snni(f)$$

- $Snn_i(f)$: DSP del ruido a la entrada
- $H_c(f)$: Rpta. en frec. del canal
- Snn_o: DSP del ruido a la salida

Los pasos a seguir serán:

- crearemos, con un editor de sonido, como 'Cool Edit', un ruido blanco.
- Guardaremos su contenido en el fichero *'ruidoblanco'*.
- Transmitiremos el fichero mediante la grabadora de sonido, recogeremos el ruido a la salida del canal mediante 'Cool Edit'.
- Grabaremos su contenido en el fichero *'ruidoblancoRx'*.
- Con las muestras de ambos ruidos, procederemos a editar un *'script'* para hallar la $|H_c(f)|$ del canal y dibujarla en la pantalla. Dicho *'script'* se muestra a continuación:

  - Cálculo de $|H_c(f)|$ a través de la transmisión de ruido blanco

```
% Leemos el fichero que contiene el ruido a la entrada.
% r: ruido a la entrada del canal.
   r=wavread('ruido blanco');
% Leemos el fichero que contiene el ruido a la salida.
% rx: ruido recibido
   rx=wavread('ruidoblancoRx');
% Sni: PSD del ruido a la entrada.
   [Sni,F ]= PSD(r,2046,44100);
% Sno: PSD del ruido a la salida.
   [Sno,F]=PSD(rx, 2046, 44100);
% H_cuad: módulo al cuadrado de la respuesta en frecuencia.
% Hacemos uso de la fórmula comentada mas arriba.
   H_cuad=Sno./Sni;
%representacion de H_cuad en db.
   plot(F,10*log10(H_cuad));grid;
```

En la página siguiente podemos comprobar la semejanza entre esta figura y la anteriormente obtenida mediante la FFT de la respuesta impulsional del sistema. En esta ocasión la figura aparece como distorsionada. Lo que ocurre es que como se ha obtenido de señales aleatorias, pues esta función también tiene carácter aleatorio, esto es, que contiene varianza no nula. Por tanto, la media de esta función es la que se parecería con mayor exactitud a la gráfica anterior[†1].

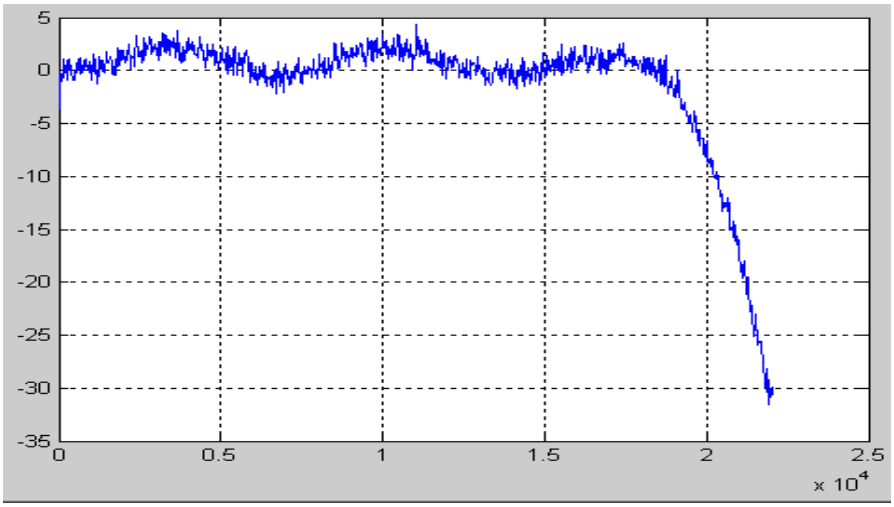


Fig. 22.- Respuesta frecuencial del canal

III.1.6.- Conclusiones definitivas sobre el estudio del canal

Este es el último punto de este apartado III.1 sobre el estudio del cable de audio o canal, y en él simplemente enumeraremos las **conclusiones** más relevantes sobre el estudio de nuestro canal:

- Se trata de un ***canal paso de banda***.
- Su ***ancho de banda*** está comprendido ***entre 20 hz. y 20 Khz***.
- Como consecuencia inmediata, nuestra ***frecuencia de portadora*** será de ***11025 hz***.
- Existe un ***rizado*** en la ***magnitud de la respuesta en frecuencia***, que provoca ***distorsión en amplitud***.
- Existe, por tanto, la necesidad de implementar un ***ecualizador***, como primer bloque del receptor.
- El ***ruido*** es ***poco determinante*** para el diseño del ecualizador, por lo que utilizaremos un ***filtro inverso o de cero forzado (ZF)***.
- Las interferencias de la red eléctrica tampoco aparecen en nuestro sistema. No hay que implementar filtros peine, ni tampoco hay que recurrir a Métodos como el de Estimación de máxima verosimilitud, para eliminar dichas interferencias (ambas soluciones no las mencionamos durante el apartado, por eso las comentamos aquí sin entrar en profundidad, porque no usaremos nada de esto).

III.2.- DISEÑO DEL SISTEMA

En este segundo apartado de la memoria procederemos a ***diseñar el sistema de comunicaciones***, dejando todos los ***parámetros***, tanto los fijos como los variables, totalmente definidos. También abordaremos los posibles ***problemas*** con los que nos encontraremos en la implementación del modulador, como la falta de sincronismo o la no idealidad del canal, comentando su origen, sus consecuencias perjudiciales de cara a la correcta transmisión de los datos, y las distintas ***soluciones*** que podríamos tomar.

Los puntos a tratar serán, por este orden, los siguientes:

- Haremos una breve ***introducción a la modulación digital***, exponiendo sus ventajas e inconvenientes respecto a la analógica, y mencionando de forma breve las técnicas más usadas.
- Realizaremos una ***justificación comparativa*** del uso de esta técnica de modulación M-QAM en detrimento del resto de modulaciones comúnmente usadas.
- Posteriormente entraremos de lleno en el ***diseño de nuestro sistema de comunicaciones***. Dividiremos nuestro sistema de comunicaciones en dos bloques bien diferenciados: el receptor y el transmisor, e iremos desgranando cada uno de ellos, analizando los problemas con los que nos vamos encontrando, exponiendo las soluciones que vayamos adoptando y la consecuente elección de los parámetros de diseño. En el apartado III.3 de la memoria se implementará, mediante funciones de Matlab, todo lo que diseñemos en este apartado III.2.
- Paralelamente al anterior punto, iremos añadiendo imágenes ilustrativas de señales, mediante ***ejemplos prácticos***, para un afianzamiento de los conceptos que vayan surgiendo.
- Dedicaremos un punto de este apartado a enumerar los posibles problemas que puedan amenazar el correcto funcionamiento de nuestro sistema de comunicaciones, examinando más profundamente el más perjudicial de todos: la ***falta de sincronismo***, y las distintas formas de combatirlo.
- Este punto surge como consecuencia del anterior. Para paliar los efectos de la ausencia de total sincronismo, dividiremos la información en tramas. En esta sección explicaremos todo lo referente a la ***formación de las tramas***.

Antes de comenzar, nos gustaría añadir que eludiremos, en la medida de lo posible, el desarrollo de ecuaciones analíticas, para explicar los conceptos que vayan acaeciendo en los siguientes puntos de este capítulo, tratando de recurrir a ***opciones explicativas lo más intuitivas posibles***.

III.2.1.- Introducción a la modulación digital

Un ***sistema de comunicaciones*** (como el nuestro), tiene como objetivo transportar una señal portadora de información de una fuente a un destino. Los dos tipos de sistemas que existen son[†4] [†1]:

- ***Sist. comunicación analógico***: en este tipo de sistemas, la señal de información viaja de forma continua tanto en su amplitud como en el tiempo, y dicho cambio de la señal se refleja en la modificación de alguna de las características de la portadora modulada (amplitud, fase o frecuencia).
- ***Sist. comunicación digital***: se trata de un sistema más complejo, donde a la señal analógica se le aplica un convertidor analógico/digital (muestreo, cuantización y codificación), para convertirla en digital. Dicha señal resultante, contenedora de muestras de la señal original analógica, es procesada para ser representada mediante una secuencia de símbolos discretos.

Debido al ***origen analógico*** de las señales presentes en la naturaleza, los sistemas más antiguos son los analógicos, por razones intuitivas. Pero en la segunda mitad del siglo XX, el desarrollo de los sistemas digitales ha sido tan espectacular, que a esta época se la considera como la ***era de las comunicaciones digitales***[†4] [†1].

Los ***motivos del éxito*** de este tipo de sistemas en detrimento del anterior son[†1]:

- Las ***ventajas del sistema digital*** frente al analógico, que repercute en la mejora de la calidad de la transmisión. Enumeraremos las más importantes:
  - ***Mayor inmunidad*** a los efectos del ruido e interferencias.
  - Flexibilidad en la creación de un ***formato digital común***, para la transmisión simultánea de diferentes fuentes de señales, en el mismo sistema. De ahí que, aunque nuestro canal sea de audio, podemos transmitir cualquier tipo de archivos (voz, imagen, texto, ...), pues la información va inmersa en una secuencia binaria, que es lo que realmente transmitimos, y nuestro formato binario es independiente del tipo de archivos que transmita.
  - ***Seguridad y privacidad*** en la comunicación, mediante la posibilidad de control de errores y encriptación de datos. En nuestro caso no tenemos necesidad de encriptación, pero haremos uso de un código redundante cíclico para la identificación de posibles errores en la transmisión.
  - ***Regeneración eficiente*** de la señal codificada. En nuestro caso, debido a la pequeña longitud del canal, hemos descartado el uso de repetidores (sería absurdo colocar otro ordenador para implementar, vía software, un repetidor regenerativo, lo cual implicaría usar un canal -cable de audio-hasta el repetidor, y otro hasta la tarjeta receptora).

- El gran ***impacto de las computadoras*** en la sociedad actual, tiene dos consecuencias inmediatas para el progreso de los sistemas digitales:

  - La ***naturaleza digital*** de las ***fuentes de datos*** de los computadores. En nuestro particular sistema de comunicaciones, nuestra fuente de datos son los bytes de los archivos que queremos transmitir.
  - El uso de dichas máquinas como ***herramientas de comunicación***. Tal es el caso de nuestro proyecto.

- La aparición de ***canales de banda ancha*** (cable coaxial, fibra óptica,...), que tuvieran la suficiente capacidad para albergar el mayor espectro de frecuencias de la señal digital.
- El nacimiento de la ***tecnología digital integrada***, que posibilitó la mayor complejidad de que constaban los sistemas digitales, y la reducción del coste de dichos sistemas.
- Gracias a estos dos últimos sucesos, el paralelo nacimiento de desarrollos sobre la ***teoría digital*** (Nyquist, Neyman, Pearson..), pudo ponerse en práctica.

Adentrándonos ya en la ***comunicación digital***, podremos optar por dos tipos de transmisiones, dependiendo de la naturaleza del canal[†1]:

- Si el canal tiene su banda de paso centrado en el origen, no hay necesidad de usar osciladores para modular. En este caso se habla de ***transmisión en banda base***, porque no existe desplazamiento en frecuencia de la señal original. Se recurre en este caso a técnicas de ***codificación de línea***.
- Si, por el contrario, el canal es paso de banda, tendremos que usar ***técnicas de modulación digital***.

Nuestro canal es un tanto peculiar, pues es paso de banda, pero la frecuencia de corte más baja la tiene en torno a 20 Hz.(como pudimos comprobar en el anterior capítulo), con lo que el problema se podría abordar desestimando el uso de osciladores. De hecho, en un proyecto llevado a cabo en la universidad de Sevilla, denominado "Transmisión de Datos mediante SoundBlaster", ya se abordó este problema transmitiendo la señal mediante el código de línea Manchester, cuyo espectro de potencia no tiene componentes de continua[†5].

Sin embargo, la solución que adoptaremos es la implementación de un modulador digital, para trasladar nuestra señal al centro de la banda de paso del canal. En nuestro caso hemos optado por centrar la señal en torno a los ***11025 hz***., por tratarse de una frecuencia submúltiplo de la frecuencia de muestreo que usamos, con las ventajas que ya veremos.

En la ***modulación digital***, la información se codifica con una secuencia de símbolos M-arios (M posibles puntos de mensaje), y cada símbolo se va a corresponder con uno de los M estados discretos del parámetro de la portadora que queramos modular.

Existen muchos técnicas de modulación digital M-aria, pero sólo vamos a mencionar, muy por encima, las más básicas[†4] [†1]:

- ***M-ASK (M-ary Amplitude Shift Keying):*** El parámetro a modular será la amplitud, la cual tendrá M niveles discretos, uno para cada distinto símbolo del mensaje. Una posible representación, para cualquiera los M posibles puntos de mensaje, $S_i(t)$, podría ser:

$$S_i(t) = Re\,[A_i \cdot g(t) \cdot \exp(j \cdot w_c \cdot t)] = Ai \cdot g(t) \cdot \cos(w_c \cdot t)\ ; \quad i=0,1...M-1;\ \ 0 \le t \le T$$

Como refleja la expresión, la señal M-ASK, en cada intervalo de símbolo, T, puede adoptar M posibles símbolos, $S_i(t)$, que difieren unos de otros por su amplitud, quedando fijas la frecuencia y la fase de la portadora: $\cos(w_c \cdot t)$.

Podemos observar también la expresión del ***filtro conformador*** del Bloque Transmisor, ***g(t)***, que, como su nombre indica, es la forma que adoptará cada símbolo antes de pasar por el modulador.

- ***M-FSK (M-ary Frequency Shift Keying):*** En este caso, modularemos la frecuencia. Este tipo de señales podría adoptar la siguiente expresión:

$$S_i(t) = Re\,[\sqrt{(2E/T)} \cdot \exp(j \cdot w_i \cdot t)] = \sqrt{(2E/T)} \cdot \cos(w_i \cdot t)\ ; \quad i=0,1...M-1;\ 0 \le t \le T$$

En este caso, cada símbolo, $S_i(t)$, adopta una frecuencia de portadora distinta, w_i , de valor $w_i = w_c + \Delta w_i$, conservando la amplitud y la fase de la portadora en cada intervalo de símbolo.

- ***M-PSK (M-ary Phase Shift Keying):*** Se trata de una modulación en fase. Una representación analítica válida para esta señal sería:

$$S_i(t) = Re\,[g(t) \cdot \exp(j \cdot (w_c \cdot t + \theta_i))] = g(t) \cdot \cos(w_c \cdot t + \theta_i)\ ; \quad i=0,1...M-1;\ 0 \le t \le T$$

En esta modulación, cada símbolo se corresponderá con una fase distinta, θ_i, de valor $\theta_i = 2 \cdot \pi \cdot i/M$, manteniendo la misma amplitud y frecuencia de portadora en cada intervalo de símbolo.

En nuestro sistema, sin embargo, implementaremos una técnica de modulación híbrida, ya que realiza la modulación conjunta de amplitud y fase de la portadora. Dicha técnica es conocida bien como ***M-APK (M-ary Amplitude Phase Keying)***, o bien como ***M-QAM (M-ary Quadrature Amplitude Modulation)***, y hablaremos más detenidamente de este tipo de modulación en posteriores apartados. De todas formas, su expresión analítica podría ser la siguiente:

$$S_i(t) = Re\,[a_i \cdot g(t) \cdot \exp(j \cdot (w_c \cdot t + \theta_i))] = \underbrace{A_i \cdot g(t) \cdot \cos(w_c \cdot t)}_{\textbf{M-ASK en fase}} + \underbrace{B_i \cdot g(t) \cdot sen(w_c \cdot t)}_{\textbf{M-ASK en cuadratura}}\ ;$$

con $i=0,1...M-1;\ \ 0 \le t \le T$

Respecto a su ***nomenclatura***, cabe comentar que la primera definición (M-APK) es intuitiva, la segunda (M_QAM) tiene la siguiente explicación:

- Al observar la señal M-QAM, podemos comprobar que es equivalente a la ***superposición de dos modulaciones L_1-ASK*** y ***L_2-ASK***, una realizada sobre una portadora en fase ,$\cos(w_c \cdot t)$, y la otra realizada sobre una portadora en cuadratura, $sen(w_c \cdot t)$. De ahí el término 'Amplitude Modulation'. El término 'Quadrature' lo adopta porque se añade, a la modulación M-ASK original, otra modulación M-ASK en cuadratura.

III.2.2.- Justificación comparativa del formato de modulación APK

Antes de realizar ninguna comparación entre la técnica de modulación que vamos a diseñar y cualquier otra forma de transmisión, nos gustaría enfatizar que el reto que nos hemos planteado en este proyecto se puede conseguir con cualquier técnica de modulación digital, e incluso mediante transmisión en banda base, restringiendo el tipo de códigos de línea a aquellos cuyo espectro de densidad de potencia no tenga componentes de continua, como ya comentamos al referirnos al proyecto "Transmisión de datos mediante SoundBlaster" [†5] durante el apartado III.2.1.

Ahora nosotros afrontaremos este desafío mediante el empleo de osciladores, utilizando una técnica de modulación digital (entre las muchas posibles, como ya hemos dicho), es decir, implementaremos un módem APK vía software.

Esta medida de la eficacia entre distintos sistemas es una forma de reflejar una lista con las distintas alternativas a la técnica que usamos, de otras vías que serían eficaces a la hora de abordar nuestro proyecto.

En primer lugar comenzaremos realizando una ***comparación*** con las señales de formato de codificación de línea, luego compararemos la técnica APK con las distintas técnicas de modulación digital básicas anteriormente citadas[†4] [†1].

Esta serie de comparaciones van a ir siempre enfocadas a las ***ventajas e inconvenientes*** que tendremos al usar nuestra técnica de modulación en lugar de cualquiera de estos otros métodos de transmisión.

Para ello nos basaremos en una serie de ***parámetros comparativos*** ya conocidos:

- La ***probabilidad de error de símbolo***.
- El ***ancho de banda*** de la señal.
- ***Energía media*** y ***energía pico*** del sistema.
- ***Tasa de bits*** del sistema.
- ***Nº de ptos del mensaje*** del sistema.

Pero antes de nada, es conveniente que hagamos un pequeño ***dimensionamiento del sistema***, dando unos valores fijos a ciertos parámetros de éste que, aunque serán modificables durante la ejecución de nuestro programa en Matlab, nos servirán para intuir valores reales de factores como ***ancho de banda*** o ***tasa de bits***, en cualquiera de las hipotéticas implementaciones (Banda base, APK, ...) con las que vamos a medir nuestro sistema, con el fin de que dicha comparación sea lo más real posible.

También, al tiempo que dimensionamos algunos parámetros, vamos a establecer las ***relaciones*** que existen ***entre*** los ***distintos parámetros*** del sistema, para ver la dependencias que existen entre unos y otros y las consecuencias de tomar ciertas decisiones sobre algunos parámetros en cuestión.

Dimensionamiento y parametrización del Sistema

- El ***ancho de banda, WB***, de las señales que entren en el canal, independientemente de su formato, vendrá determinado por la ***tasa de símbolos del sistema***, ***D***,y ésta queda fija al definir dos parámetros:
  - La ***frecuencia de muestreo***, ***Fs***, que será de ***44100 hz***.

- El ***nº de muestras por símbolo***, ***N***, que, para esta serie de comparaciones podría ser de ***10***.

Para dichos valores de Fs y N, la ***tasa de símbolos***, ***D***, tendrá el siguiente valor:

- $D = Fs / N = 4410$ hz.

El ancho de banda de la señal que entra en el canal es por tanto función de la tasa de símbolos, $D_{[\dagger 4]\,[\dagger 1]}$:

- $WB = f(D, \alpha)$; α: Factor de Roll-off (caso de usar coseno alzado)

Así, tenemos que dicho ***Ancho de Banda*** será ***fijo*** para estos valores de Fs y N, a menos que se usen funciones del tipo ***coseno alzado***, que permiten modificar el ancho de banda, sin necesidad de variar $D_{[\dagger 4]\,[\dagger 1]}$.

- Por otra parte, si nos encontramos en una ***transmisión M-aria***, a cada símbolo se le hará corresponder con un bloque fijo de ***k bits***, mediante la siguiente relación combinatoria$_{[\dagger 4]\,[\dagger 1]}$:

  - $M = VR_{2,k} = 2^k$ niveles de representación $\Leftrightarrow k = \log_2(M)$ bits / símbolo

  Lo cual quiere decir que combinando el bloque de 'k' bits, podremos representar los $\boldsymbol{M = 2^k}$ ***símbolos*** o puntos de mensaje, esto es, M niveles discretos de codificación distintos.

- La ***modificación*** de esta variable del sistema ,***'M'***, afecta a otros factores importantes como son$_{[\dagger 4]\,[\dagger 1]}$:

  - La Tasa de bits, del sistema, R_b.
  - La Probabilidad de error de símbolo, P_e.
  - La Energía media y Energía pico del sistema.

- La ***Tasa de bits, R_b ,*** y de ***símbolo***, ***D,*** guarda una estrecha relación analítica, consecuencia de la representación M-aria mediante la secuencia de k bits/símbolo. Sus correspondientes magnitudes inversas, ***intervalo de bit***, $\boldsymbol{T_b}$, e ***intervalo de símbolo***,***T*** , encierran, obviamente, análoga relación:

  - $R_b\,(bits/sg) = k \cdot D \Leftrightarrow T\,(sg / símbolo) = k \cdot T_b$

  Para ***optimizar*** nuestro sistema en cuanto a ***velocidad de transmisión***, nos interesará ***aumentar 'M'***. Si hacemos esto:

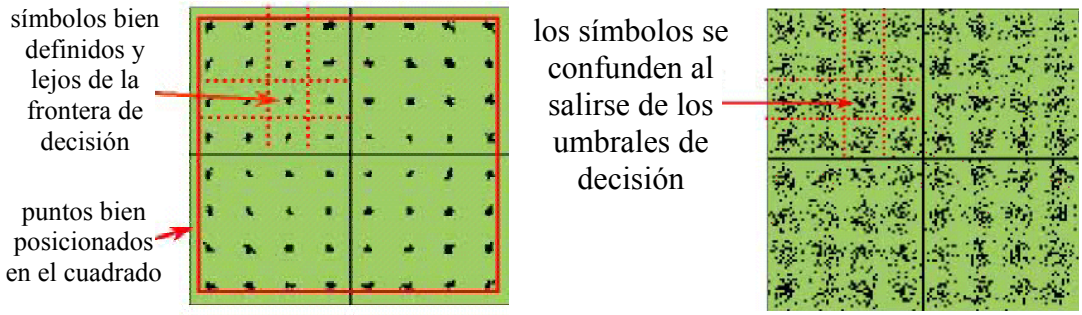
  - La ***tasa de símbolos,D***, permanecerá ***intacta***, pues depende directamente de la frecuencia de muestreo, Fs, y del nº de muestras por símbolo, N, como ya comentamos con anterioridad. El ancho de banda seguirá siendo el mismo, por su dependencia directa con D, a menos que usen filtros de tipo coseno alzado, como ya anticipamos.
  - Sin embargo, la ***tasa de bits del sistema***, $\boldsymbol{R_b}$, se verá afectada por su relación con D, de la siguiente manera:

- $M\uparrow \rightarrow k\uparrow \rightarrow R_b\uparrow$

Conseguimos así ***aumentar*** R_b, y por tanto, la velocidad de transmisión del sistema.

- Pero si atendemos a la ***Probabilidad de error de símbolo,*** P_e (o a la ***probabilidad de error de bit***, ya que ambas están íntimamente unidas por el parámetro M, pues es obvio que si existe fallo en un bit, originará un error en su símbolo correspondiente), y echamos una ojeada a cualquier libro de teoría de la señal digital[†4][†1], podremos comprobar que existen expresiones estadísticas que relacionan P_e **,** ***M*** y la ***energía de símbolo, E***, y que reflejan que, por lo general, al aumentar M, y mantener E constante, se produce un incremento en la probabilidad de error de símbolo y del sistema en general. Esto se puede explicar siguiendo un razonamiento muy intuitivo:

  - De todos es sabido que la Energía de un símbolo es su módulo al cuadrado, esto es, el cuadrado la distancia del punto de mensaje al origen.
  - Si ***aumenta M***, manteniendo la energía pico del sistema (la distancia correspondiente al símbolo más lejano al origen en la constelación de símbolos), la constelación empezará a cargarse de puntos de mensaje.
  - Entonces, la distancia entre cada uno de ellos comenzará a disminuir, lo cual hace que se ***decremente*** la ***región de decisión*** para cada símbolo.
  - También sabemos que, por alteraciones que el canal introduce en nuestra señal, la posición de los símbolos recibidos no es idéntica a la de los símbolos transmitidos, pues puede existir una desviación tanto en el módulo del símbolo como en su fase (o bien una desviación de su posición en fase y/o en cuadratura, para el caso de una o dos dimensiones).
  - Veamos un ***ejemplo*** con 2 gráficas sobre una constelación de ***puntos de mensaje recibidos*** de una señal ***64- QAM*** y su desviación respecto a su posición original debido a la ***distorsión del canal***. Se puede apreciar que los efectos de dicha distorsión en la primera gráfica no impide la detección de los símbolos, pero en la segunda gráfica, los efectos de ésta son alarmantes, y la detección será prácticamente imposible.



Constelación 64-QAM "limpia" **Constelación 64-QAM "sucia"**

Fig. 23.- Constelaciones sucia y limpia

- Además, para nuestro diseño nos basaremos en un ***detector de máxima verosilitud***[†4][†1], donde el criterio de decisión consiste en elegir como símbolo transmitido a aquel cuya distancia euclidiana sea mínima respecto al vector de recepción o símbolo recibido.
- Por tanto, si aumenta M, crece la posibilidad de que el símbolo recibido caiga dentro de otra región de decisión distinta a la suya, ***acentuándose*** así la probabilidad de una mala decisión, y la consecuente ***probabilidad de error de símbolo***, y la del sistema en general[†4] [†1].
- Si queremos aumentar el nº de símbolos en el sistema, ***manteniendo*** la ***probabilidad de fallos en el sistema***, o bien escogemos otra técnica de modulación (que implica una constelación o distribución de símbolos distinta), o bien separamos los símbolos, ***aumentando la energía de cada símbolo, E,*** y, por tanto, la energía media y pico del sistema. Pero en nuestro caso, la ***energía de transmisión*** está ***limitada*** por la tarjeta de sonido, que sólo acepta niveles entre 1 y -1, y este es el rango de valores que se tienen que repartir los distintos símbolos, tanto para una dimensión (fase), como para dos dimensiones (fase y cuadratura).

- Por tanto, podemos concluir con una ***regla de diseño*** bastante lógica a la hora de diseñar nuestro sistema de comunicaciones digital:

  - Si lo que prima en nuestro sistema es la ***velocidad*** de transmisión, es preferible aumentar M.
  - Si la velocidad no interesa tanto, e impera una necesidad de transmitir con una ínfima ***probabilidad de error*** de bit, podemos usar una transmisión binaria (M=2), o bien implementar un sistema M-ario con un valor de 'M' relativamente bajo.

  En nuestro caso, hemos considerado ambos parámetros como importantes, es decir, queremos transmitir bytes lo más rápidamente posible, conservando una Probabilidad de error de símbolo bastante baja. De ahí que hayamos optado por la ***modulación M-QAM***, pues como veremos, aún aumentando M, los símbolos se mantienen bastante separados en su constelación correspondiente, pudiendo llegar a valores de M, que el resto de los hipotéticos formatos de codificación apenas resistirían sin que se disparase su probabilidad de error del sistema.

- Ni que decir tiene que, para ***aumentar la tasa de bits*** del sistema, sin incrementar M (y por tanto, su Probabilidad de error de símbolo), deberíamos ***acrecentar su tasa de baudios, D***, bien elevando la ***frecuencia de muestreo, Fs***, o bien disminuyendo el ***nº de muestras por símbolo, N***.

- Pero si tenemos en cuenta que el máximo de ancho de banda lo tenemos para un factor de Roll-off unidad, y su expresión depende de D:

  - WB_{max} = (fc + D) (siendo fc=11025hz un valor fijo) ;

  , podremos hallar un ***mínimo teórico para N***, si tenemos en cuenta varios aspectos de teoría de la señal:

- Si atendemos al efecto denominado ***'aliasing'*** o solapamiento[†2], para que este fenómeno no se produzca se debe cumplir que:

  - $Fs > 2\cdot(fc + D) \rightarrow D < 11025$ hz. $\rightarrow Fs/N < 11025$ hz. $\rightarrow \boldsymbol{N > 4}$

- Por otra parte, si atendemos a la ***anchura del canal***, para que la señal quede dentro del rango de paso de banda del canal, se debe cumplir que:

  - 20 Khz. $> (f_c + D) \rightarrow D < 8975$ hz. $\rightarrow Fs/N < 8975$ hz. $\rightarrow \boldsymbol{N > 5}$

Así, nos quedamos con el valor de N teórico más restrictivo, y consideramos la anchura de canal y no el posible "alising" como consecuencia de la limitación de N. Aunque bien es cierto que en la práctica, hemos comprobado que, para un valor de Roll-off unidad (que maximiza el ancho de banda de nuestra señal) y M=256 (que maximiza la probabilidad de error en la detección de un símbolo), el ***valor mínimo de N*** sin que se produzcan errores es ***N = 9***, y esto es lógico si pensamos que, tanto la frecuencia superior de corte de nuestro medio de transmisión (que realmente está en torno a 18 Khz.) como la frecuencia de corte de nuestra señal son valores teóricos. Además, apurar tanto el valor de N nos llevaría a un problema todavía más grave que es el ***problema del sincronismo***, del que ya nos ocuparemos más adelante, y que puede que sea otra razón de que exista tanta diferencia entre el valor teórico de N y el valor práctico.

- De la misma forma, y teniendo en cuenta el valor mínimo empírico de N, N=9, y para un caso extremo de ancho de banda, es decir, factor de Roll-off unidad, podemos hallas un ***rango de actuación teórico para*** nuestra frecuencia ***Fs***:

  - Para hallar el ***valor mínimo teórico de Fs***, para un número de muestras por símbolo dado, N, tenemos que tener en cuenta que si Fs disminuye, aunque también disminuya el ancho de banda de nuestra señal (porque disminuiría D), llegará un momento en que no se cumpla el ***teorema de Nyquist del muestreo***[†2]. Es decir, la condición para que no exista solapamiento determinará el Fs mínimo:

$$\left.\begin{array}{l} Fs > 2\cdot(f_c + D) \\ D = Fs/N \end{array}\right\} \rightarrow Fs > 2\cdot N\cdot fc / (N-2)$$

Así los valores mínimos de Fs típicos, es decir, para un número de muestras / símbolo, N, lo más bajo posible, serán:

- $N=9 \rightarrow Fs|_{min} = 28350$ hz.
- $\boldsymbol{N=10 \rightarrow Fs|_{min} = 27563\ hz}$.

Para un número elevadísimo de N, que obviamente no deseamos, existiría un límite asintótico de Fs:

- $(N \rightarrow \infty) \rightarrow Fs|_{min} = 22050$ hz.

- Por otro lado, para hallar el ***máximo teórico de Fs***, tendríamos que tener en cuenta la ***anchura del canal*** (pongamos ***18 Khz***., para ser restrictivos), ya que al aumentar Fs, aumentamos también la anchura de nuestra señal:

$$\left.\begin{array}{l} WB|_{canal} > (f_c + D) \\ D = Fs/\ N \end{array}\right\} \rightarrow Fs < N \cdot 6975 \text{ hz.}$$

De este modo, para valores típicos de N, los valores máximos teóricos de Fs serán:

- N=9 → $Fs|_{max}$ = 62775 hz.
- ***N=10 → $Fs|_{max}$ = 69750 hz***.

Para un número elevadísimo de N, que reiteramos no desear, no existiría un límite asintótico de Fs, pues la anchura de nuestra señal sería prácticamente nula.

- Pero huyamos un poco de toda esta teoría, y acerquémonos a la realidad, estableciendo el rango de velocidades de nuestro sistema, que está, como sabemos unido al rango de variación de M. El mínimo nº de niveles que usamos en nuestro sistema es M=4, que será el más lento en cuanto a tasa de bits, pero el más robusto frente a errores de transmisión. El máximo valor que hemos podido alcanzar, con un nº bajo de errores es M=256, que obviamente, es el más rápido.

  - M = 4→ Rb= k·D = 2 bits/símb· 4410 símb/sg = 8820 bits/sg.
  - M = 256→ Rb= k·D = 8 bits/símb· 4410 símb/sg = 35280 bits/sg.

  Así, el ***rango de velocidades*** de nuestro sistema ***M-QAM*** , para N=10, estaría entre ***8820 bits/sg y 35280 bits/sg***, que abarcará también todas las posibles velocidades de transmisión de los demás sistemas a comparar, debido a que el rango de variación de M en nuestro formato de modulación es mayor que en el resto, como ya veremos.

TRANSMISIÓN EN BANDA BASE

Atendiendo a la respuesta en frecuencia del canal, que rechaza la transmisión de señales cuyo espectro quede fuera del rango de 0 a 20 hz., ya hemos hecho hincapié en que los únicos formatos de codificación en banda base que serían aptos para usarse en este proyecto serían aquellos cuyo espectro de frecuencia careciera de componente continua o DC, para evitar así la pérdida de información en su paso por el canal[†1] [†4].

De entre los códigos de línea que cumplen este requisito, los más populares son:

- ***Formato Manchester.***
- ***Formato NRZ Bipolar.***

, cuyos ***espectros de potencia***, normalizados tanto en frecuencia como en fase, se muestran un poco más adelante. Para representar dicha gráfica, hemos recurrido a sus respectivas expresiones analíticas originales, que vienen en cualquier libro de texto[†1] :

- Manchester: $Sx(f) = a^2 \cdot T \cdot \text{sinc}^2(f \cdot T/2) \cdot \text{sen}^2(\pi \cdot f \cdot T/2)$
- Bipolar NRZ: $Sx(f) = a^2 \cdot T \cdot \text{sinc}^2(f \cdot T) \cdot \text{sen}^2(\pi \cdot f \cdot T)$

, donde 'a' es la amplitud del símbolo y 'T' es la duración del mismo. Luego hemos normalizado en amplitud y frecuencia, y hemos ejecutado los siguientes comandos de línea en Matlab:

- Generación de PSDs Normalizados Manchester y Bipolar NRZ

```
% x: variable normalizada. x=f·T. Representamos 2000 muestras.
   x=0:0.001:2;
% bipnrz: PSD de formato bipolar NRZ normalizada
   bipnrz=(sinc(x).*sin(pi*x)).^2
% manch: PSD de formato manchester normalizada
   manch=(sinc(x/2).*sin(pi*x/2)).^2;
% dibujamos ambas señales con el comando plot.
   plot(x,s,x,t);
```

El resultado de dicho 'script' lo mostramos en la siguiente página. En dicha gráfica, es evidente que la señal ***Manchester*** ocuparía en el canal el ***doble de ancho de banda*** que la señal Bipolar NRZ. Esto es debido a que el pulso básico de codificación Manchester equivale a dos semipulsos de mitad de intervalo de símbolo y de amplitud distinta. Este hecho representa una desventaja para el uso de la codificación Manchester, puesto que nos interesa minimizar este parámetro.

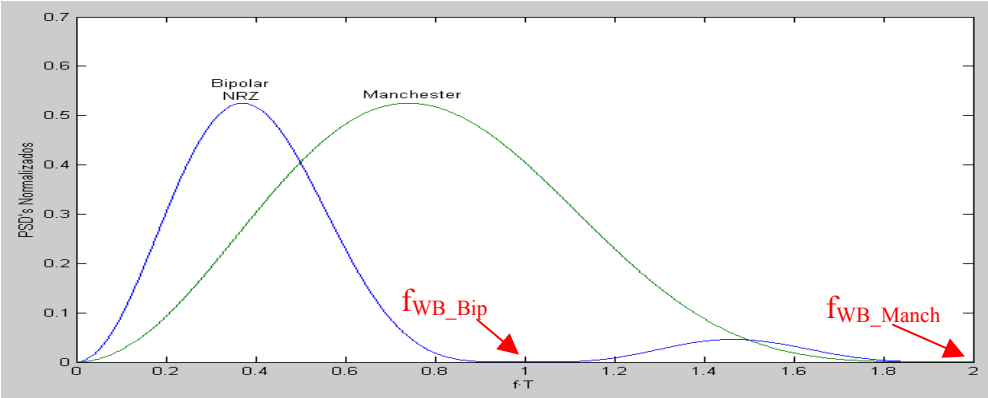


Fig. 24.- Bipolar y Manchester

Pero si dimensionamos el eje de abcisas de la gráfica en cuestión, nos daremos cuenta de que tampoco es un problema muy importante, como para descartar la codificación Manchester en favor de la bipolar NRZ. Veámoslo:

- Supongamos los siguientes valores para una hipotética implementación:
  - Fs = 44100 hz.
  - N = 10 muestas/símbolo.
  - D = Fs / N = 4410 baudios

- Si despreciamos los lóbulos secundarios, el ancho de banda de ambas señales lo podríamos aproximar al ancho del lóbulo principal , que comprende un rango de 0 hz. hasta el ***primer cruce por cero***, f_{WB} .

  - ***Manchester***: $f_{WB} \cdot T = 2 \rightarrow f_{WB} = 2 / T = 2 \cdot D \rightarrow f_{WB} = 8820$ hz.
  - ***Bipolar NRZ:*** $f_{WB} \cdot T = 1 \rightarrow f_{WB} = 1 / T = D \rightarrow f_{WB} = 4410$ hz.

- Así, el ancho de banda para ambas señales será de:

  - ***Manchester***: $WB_{manch} = 8820$ hz.
  - ***Bipolar NRZ:*** $WB_{bipNRZ} = 4410$ hz.

- Para la señal ***Bipolar NRZ***, el resto de frecuencias que determinan los lóbulos secundarios (dentro del rango espectral de nuestro canal) estarán situados en posiciones n·D, siendo 'n' un nº natural, esto es, las ***frecuencias de cruce por cero*** serán:

  - f = 8820 hz., 13230 hz. y 17640 hz.

- Para la señal ***Manchester***, estas ***frecuencias de cruce por cero*** estarán separadas 2·D entre cada una de ellas, o sea, sus frecuencias de cruce por cero serán:

  - f = 8820 hz. y 17640 hz.

- Así, podemos concluir, que ambos espectros, llegan muy atenuados a la frecuencia de corte del canal ,20 Khz., por lo que la transmisión en ambos casos se podría realizar sin eliminar mucha información, ya que las ***colas*** de dichos espectros se suponen ***despreciables*** en los dos casos.

Así, al tener un canal lo suficientemente ancho, nos decantaríamos por la señal Manchester, cuyo lóbulo principal está más alejado del origen de frecuencias y, por tanto, se ve menos afectada por la frecuencia de corte de 20 hz. del canal.

Por tanto, la señal a comparar con nuestro sistema M-APK sería la señal Manchester, sin que suponga pérdida de generalidad respecto al formato Bipolar NRZ descartado.

Ventajas de una Transmisión en Banda Base respecto a M-APK

- En primer lugar, al usar dicho código de línea, transmitiríamos en banda base, por lo que podríamos prescindir del uso de los ***bloques modulador***, y ***demodulador*** (con sus correspondientes ***osciladores***), simplificándose de este modo el diseño e implementación de nuestro sistema.
- Además, para evitar la interferencia entre símbolos, vamos a usar, en nuestro sistema M-QAM, dos ***filtros raíz de coseno alzado***, uno como pulso conformador en el transmisor, y otro como filtro adaptado en el receptor. Sin embargo, si usáramos la codificación Manchester, no necesitaríamos usar dichos pulsos, con lo que el sistema se reduciría aún más.

- Además, el desfase que existe entre los relojes del transmisor y receptor, si es muy elevado, puede afectar no sólo ya al muestreo desigual de ambos bloques (cosa a la que también se expone el sistema Manchester), sino que esa ***falta de sincronismo*** podría afectar a los ***osciladores***, por lo que la frecuencia de portadora no sería idéntica y los problemas serían ya mucho más graves. Este tema del sincronismo lo veremos más adelante.

Ventajas de una Transmisión M-APK respecto a Banda Base

- Sin embargo, y como ya veremos, el problema del sincronismo lo podremos paliar mediante la ***división de la información en tramas*** relativamente cortas, donde el efecto de la falta de sincronismo apenas se note.
- Respecto al ***ancho de banda***, ya dijimos que la codificación Manchester ocuparía un ancho de banda de $2 \cdot D = 8820$ hz. Sin embargo el ancho de banda de una señal paso de banda tipo M-APK varía según el factor de roll-off, α, que usemos. Los valores extremos para este ancho de banda serán:

  - $\alpha = 0$, WB=D= 4410 hz.
  - $\alpha = 1$, WB=2·D= 8820 hz.

  Ambas señales caben perfectamente en la banda de paso del canal, pero la señal M-APK, al estar centrada (y por tanto, más alejada de la frecuencia del corte en torno a 20 hz.) y tener, como ya veremos, una forma limitada en frecuencia (al contrario, que el código de línea Manchester, cuyo espectro en frecuencia es ilimitado, como vimos anteriormente), no tenemos que despreciar ninguna cola del espectro que sobresalga de los límites del canal (como hacíamos en la codificación Manchester).

- Además, existe un parámetro denominado ***'eficiencia del ancho de banda'***, ***ρ***, que mide la relación entre la tasa de bits, R_b, y el ancho de banda, WB, de un sistema[†1] [†4].

  - Para el sistema ***Manchester***, el valor de este parámetro es:

    - $\rho = R_b/WB = k \cdot D / 2 \cdot D \rightarrow \rho = k / 2$

  - Para el sistema M-APK, este parámetro variará entre:

    - $(\alpha = 0) \rightarrow \rho = R_b / WB = k \cdot D / D \rightarrow \rho = k$
    - $(\alpha = 1) \rightarrow \rho = R_b / WB = k \cdot D / 2 \cdot D \rightarrow \rho = k / 2$

  Siendo $k=\log_2 (M)$, se entiende que conforme aumente M, el sistema será más eficiente en cuanto a que aumenta su tasa de bits sin aumentar su ancho de banda. Por tanto, el sistema M-APK es más eficiente, en cuanto que soporta unos valores de M mayores.
- El anterior punto nos lleva al análisis de otro parámetro importante del sistema: la **velocidad de transmisión**. El parámetro que mide dicha

característica es la ***Tasa de bits***, y como ya sabemos, está íntimamente ligada al valor de M.

- En la codificación ***Manchester***, el máximo valor de M que podríamos alcanzar, sin originar una tasa elevada de error de símbolos, sería de ***M=16***, por tanto, la ***máxima tasa de bits*** que alcanzaríamos sería de :
  - $R_b|_{max} = k \cdot D = 4 \cdot 4410 = 17640$ bps
- En nuestro sistema ***M-QAM***, en principio, ***para esta misma tasa de símbolos***, el máximo valor de M que podríamos alcanzar, sin originar una tasa elevada de error de símbolos, sería de ***M=256. P***or tanto, la ***máxima tasa de bits*** que alcanzaríamos sería de :
  - $R_b|_{max} = k \cdot D = 8 \cdot 4410 = 35280$ bps
- ***Sin embargo***, esto sería así para una misma tasa de bits, ***D***. Pero no olvidemos que D también depende de la frecuencia de muestreo, Fs y del ***nº de muestras por símbolo***, ***N***. Ya hemos comentado que nuestro ***sistema APK no soportará un nº de muestras N<9***, sin que se produzcan errores debido a la falta de sincronización entre otros muchos factores. Por el contrario, para la ***modulación Manchester***, al emplear pulsos rectangulares como filtros conformadores, se puede conseguir un ***nº menor de muestras***, pudiendo incluso llegar a ***N=2***, para un nº M de símbolos bajo. De este modo, estamos multiplicando la tasa de baudios, D, por un factor mayor de 4, con respecto al máximo D=4900 baudios que obtenemos para N=9 en nuestro sistema.
- Aunque también es cierto que en el formato Manchester, para N=2 tendríamos que quedarnos en un nº M de símbolos muy bajo, en torno a M=4, para evitar errores en la decisión, siendo muy perjudicial también la ausencia de sincronismo para la detección de dichos símbolos. Por otro lado, en nuestro sistema APK, el límite para N se puede rebajar hasta N=7, aunque para un nº de símbolos, M, relativamente más bajo que para M=256, del orden de M=32 o M=64.
- De todos modos, el sistema en ***formato Manchester*** proporciona una ***tasa de bits más elevada***, a juzgar por los resultados obtenidos en el proyecto ya comentado "Transmisión de datos mediante SoundBlaster", realizado en la universidad de Sevilla.
  Si bien es cierto que ***nuestro sistema APK*** tiene mucha más riqueza en lo que a constelaciones se refiere, ya que es ***mucho más versátil*** tanto en su número como en su amplia gama de posibles fisonomías a adoptar.

- Por último, y para captar mejor la idea de por qué la modulación digital M-APK acepta tan elevado nº de símbolos, M, en comparación con la señal Manchester, vamos a hablar un poco sobre las ***constelaciones*** de ambos sistemas[†1] [†4]:
  - Para implementar un ***sistema M-ario*** utilizando la codificación ***Manchester***, tendríamos que recurrir a una técnica digital de

modulación denominada ***M-PAM***, que es un concepto genérico de M-ASK, donde los bits que constituyen la información codificada, se agrupan en un nº fijo para formar los M símbolos del sistema, y a cada uno de ellos se le hace corresponder con una amplitud determinda del pulso básico. De ahí el nombre de M-PAM, ***M-ary Pulse Amplitude Modulation***. La expresión matemática de cada símbolo es la siguiente:

- $S_i(t) = A_i \cdot \Phi(t)$ $\quad i = 0,1,.....M-1 \quad ; \quad 0 \leq t \leq T$

Donde $\Phi(t)$ es el vector que representa el pulso básico normalizado, y A_i es una de las M posible amplitudes de codificación del pulso.
Es evidente que se trata de un tipo de ***señalización unidimensional***. Así los M símbolos tendrán que ubicarse en posiciones equidistantes entre los valores 1 y -1, que constituyen los extremos del rango permitido de la tarjeta. Por tanto, una posible representación de la constelación del sistema M-PAM, para distintos valores de M sería:

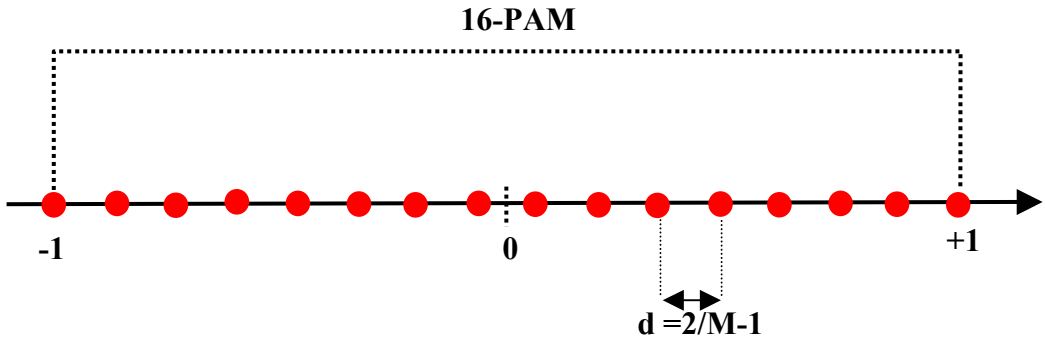


Fig. 25.- 16-PAM

La distancia mínima entre cada par de símbolo para nuestro sistema M-PAM sería:

- $d = 2 / M-1$

Por lo que, se puede ver que, conforme aumente M, dicha distancia entre cada símbolo disminuirá.
La distancia entre un símbolo y su frontera de decisión será de d/2. Este será el margen de error máximo que puede tener el símbolo recibido respecto al transmitido, al desviarse bien a la derecha o bien a la izquierda de su posición original (por interferencias del canal), sin que se produzca error en la decisión del símbolo.
Así, para M=16, tenemos ya una distancia entre símbolos, d, muy pequeña:

- M=16 → d = 0.1333

Si siguiéramos incrementando M, los errores de decisión serían ya perceptibles.
De hecho, si igualamos la distancia mínima de un sistema M-PAM, $d|_{M\text{-}PAM}$, con la distancia mínima de nuestro sistema 256-QAM, $d|_{256\text{-}QAM}$, que obviamente es la más pequeña que hemos podido conseguir sin que existan muchos errores, obtendremos el valor de M para el sistema M-PAM, tal que la distancia mínima entre símbolos sea la misma:

- $d|_{M\text{-}PAM} = d|_{256\text{-}QAM} \rightarrow 2 / (M-1) = 2 / (15 / \sqrt{2}) \rightarrow M \sim 22$

Obtenemos un valor aproximado de M=22, y como el valor de M siempre es potencia de dos, es coherente la elección de un máximo de valor de M=16 para la técnica M-PAM.

- Sin embargo, nuestro sistema M-ario M-APK, modula tanto en amplitud como en fase, o sea, que cada uno de los M símbolos tiene una pareja de amplitud y fase distinta. Se trata de una ***señalización bidimensional***, donde cada símbolo adoptaría la siguiente expresión matemática:

- $S_i(t) = \underbrace{A_{i1} \cdot \Phi_1(t)}_{\text{Fase}} + \underbrace{A_{i2} \cdot \Phi_2(t)}_{\text{Cuadratura}} ; \quad i = 0,1,.....M-1 \;\; 0 \;\; ; \quad 0 \leq t \leq T$

, donde ambos sumandos están desfasados 90º, como ya apuntamos al final de la sección III.2.1, donde definíamos una señal M-QAM utilizando otra expresión equivalente a ésta, donde la señal en fase correspondía a la modulación en amplitud de un coseno, y la señal en cuadratura pertenecía a la modulación en amplitud de un seno. Por tanto, y como ya reseñábamos anteriormente, esta señal es equivalente a una ***superposición de 2 señales M-ASK, desfasadas 90º***, o sea, que cada sumando no es más que una señal M-ASK o M-PAM. Con esto queremos decir que si en M-PAM podíamos llegar a un nº de M=16 niveles, en el nuestro se podría llegar a un nº $M = 16 \cdot 16 =$ ***256 niveles***, como realmente ha sido.
En esta constelación, donde cada símbolo viene determinado por la suma de un vector en fase y un vector en cuadratura, los puntos de mensaje se reparten en el plano, pudiendo situar una mayor cantidad de ellos, sin que se reduzca tan considerablemente la región de decisión.
Nos gustaría hacer notar que a los símbolos más alejados al origen, le doy amplitud unidad, y para conseguir esto, ***escalamos los niveles M-ASK*** tanto de fase como de cuadratura, dividiendo por $\sqrt{2}$ dichos niveles. Así los ***extremos*** de los ejes no estarán en 1 y -1, sino en $\mathbf{1/\sqrt{2}}$ y $\mathbf{-1/\sqrt{2}}$ respectivamente.
Así, la distancia mínima entre dos símbolos consecutivos la encontramos para el valor de M más alto que admite el sistema, sin

que ocurra un índice de fallos considerable en el sistema, esto es, para M=256. Su valor será:

- $d = 2/((M-1)\cdot \sqrt{2}) \rightarrow d = 0.0943$

Veamos un ejemplo de una constelación M-QAM rectangular (k par, siendo $k = \log_2 M$), que es la más típica y la más usada para el estudio y caracterización de este tipo de modulaciones.
Para M=256, la constelación adopta el siguiente aspecto:

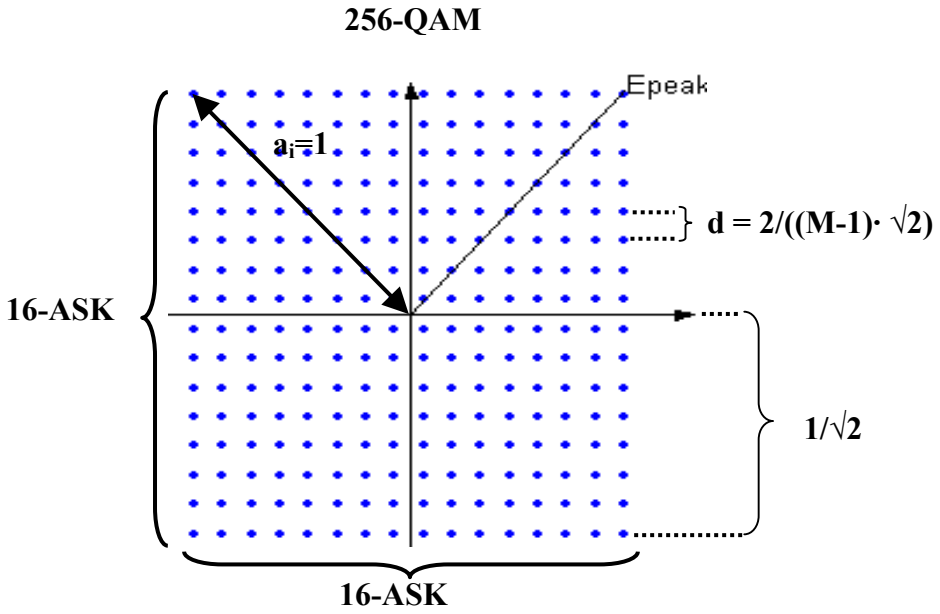


Fig. 26.- 256-QAM

MODULACIÓN DIGITAL FSK

Ya definimos este tipo de modulación en el apartado III.2.1. Se trata de un esquema igual al usado para generar una señal ***FM analógica***[†2], donde la diferencia estriba en que ésta última modula señales analógicas, y la ***modulación FSK*** se usa para señales digitales.

Por tanto , en este tipo de modulación, la envolvente de cada símbolo es constante y lo mismo ocurre con su fase inicial. Los símbolos que representan los distintos niveles de decisión se harán corresponder con ***diferentes frecuencias de portadora***, por lo que ocuparán diversas zonas del ancho de banda.

Este hecho de usar distintas frecuencias para alojar las envolventes complejas de los distintos símbolos, hace que la ***eficiencia del ancho de banda, ρ***, sea ***muy pobre***, en comparación con la eficiencia espectral de nuestro sistema de modulación M-QAM, ya que, para garantizar la ortogonalidad de las distintas señales del sistema M-FSK, la distancia entre cada frecuencia de símbolo ha de ser D/2, donde D = 1/T, es la tasa del símbolos del sistema, y conforme aumenta M, se incrementa el ancho de banda de la señal a transmitir.

Si tenemos M frecuencias distintas y equidistantes en nuestro espectro, separadas D/2 hz., el ancho de banda aproximado será de (M-1)·D / 2, pero para calcular la eficiencia del ancho de banda, ρ, nos basaremos en una ***estimación***: supondremos que el ***ancho de banda*** es igual $a_{[\dagger 1]}$:

- $WB|_{M\text{-}FSK} = M \cdot D / 2$ hz.

Así, el valor de dicha ***eficiencia espectral*** será:

- $\rho = R_b / WB = k \cdot D / (M \cdot D/2) = 2 \cdot k / M \rightarrow \rho = 2 \cdot \log_2 M / M$

Se puede ver que, si M aumenta, ρ disminuye, lo cual refleja lo que ya referíamos: la ineficiencia espectral de este sistema.

Además, si ***dimensionamos*** este sistema en cuanto a ***ancho de banda***, usando los parámetros hasta ahora utilizados en los anteriores cálculos, esto es, D =4410 hz., e igualamos el valor estimado de $WB|_{M\text{-}FSK}$ a los 20 khz. que comprende nuestro canal, nos daríamos cuenta de que, a lo sumo, el máximo número de símbolos que aceptaría el canal sería de, M = 8. Por lo tanto, la tasa máxima de bits de este sistema M-FSK, para M=8, sería de, R_b = 13230 hz.

Concluimos así que dada la poca velocidad de este sistema en comparación a los 35 kbps que ofrece nuestra técnica de modulación 256-QAM, es una justificación más que suficiente para optar por este último método de codificación.

De todos modos, podríamos destacar entre las ***ventajas*** de usar una señalización ***M-FSK***, la de que es ***menos susceptible a anomalías de fase y amplitud*** del canal que otras codificaciones digitales como M-QAM o $M\text{-}PSK_{[\dagger 1]}$.

Además, otra característica que favorece el uso de este tipo de formato deriva de la relación existente entre la probabilidad de error de símbolo, P_e , el número de símbolos, M, y la relación señal a ruido, SNR= E/N_o, ya que , para este tipo de señales, si aumento M (para conseguir una tasa de bits más elevada), para una P_e dada, los requerimientos de potencia disminuyen para mantener dicha P_e, pero eso sí, a costa de agrandar el ancho de $banda_{[\dagger 1]}$. Sin embargo, para el resto de modulaciones digitales, si queríamos incrementar M, para conseguir un mayor valor de R_b, teníamos que acrecentar la SNR por símbolo.

Así, el tipo de modulación digital FSK se usa en ***sistemas limitados en Potencia***, donde el ancho de banda no sea una limitación del sistema de comunicaciones, y este no es el caso, ya que tenemos un ancho de banda del canal limitado a 20 Khz., y el ruido es muy bajo, como para tener que aumentar mucho la potencia de la señal.

MODULACIÓN DIGITAL ASK

Este tipo de señales, definido en el apartado III.2.1, es ***análogo al sistema*** codificación ***M-PAM*** usado para la transmisión en Banda Base, con la salvedad de que se le añade un ***oscilador*** a la frecuencia de portadora, que provoca su correspondiente desplazamiento en el espectro a dicha frecuencia de portadora. Por tanto, la representación matemática es la misma que la de una señal M-PAM, pero el vector de la base, Φ(t), vendrá multiplicado por un $\cos(w_c \cdot t)$, siendo w_c la frecuencia angular de la portadora. De este modo, ambas constelaciones serán idénticas. Por este motivo, comentaremos muy brevemente las consecuencias de haber usado esta señal digital.

En cuanto a ***ancho de banda***, como el pulso básico es una función rectángulo de amplitud constante, y duración de símbolo T sg., la anchura espectral de nuestra señal a transmitir hubiera sido de 2·D = 2 / T hz., correspondiendo con la anchura del lóbulo principal de la función sampling (no olvidemos la dualidad función rectángulo- función sampling). Teniendo en cuenta que la señal hubiera estado centrada en la frecuencia de portadora f_c=11025 hz., el ancho de banda hubiera ocupado desde f_c-D = 6615 hz, hasta f_c+D = 15435 hz. Los siguientes cortes por cero de dicho espectro hubieran estado en f_c-2·D = 2205 hz y f_c+2·D = 19845, con lo cual, los lóbulos secundarios también estarían dentro de la banda de paso del canal, por lo que, al igual que hicimos cuando tratamos la codificación en banda base, podemos suponer que todo el espectro de potencia está dentro del ancho de banda permisible del canal. Si se produjera ISI, tendríamos que usar pulsos en coseno alzado, y el ancho de banda oscilaría entre D y 2·D, dependiendo del valor del factor de Roll-off.

Respecto a la máxima tasa de bits aceptable por este sistema, decir que, al disponer de idéntica constelación unidimensional que la de una señalización M-PAM, las conclusiones que obtuvimos para ésta también nos vale para la modulación digital M-ASK. Así, el máximo valor de M, sin incrementar la energía de los símbolos (puesto que como ya hemos comentado en reiteradas ocasiones, la ubicación de éstos se tiene que hacer entre 1 y –1), sería M = 16, con lo que la velocidad máxima que alcanzaríamos sería de Rb = 4·4410 = 17640 bps, que, lógicamente es la misma que para un sistema M-ario de codificación de línea, y muy inferior del que alcanzamos para un sistema 256-QAM.

MODULACIÓN DIGITAL PSK

De todas las hipotéticas implementaciones, hubiera sido la más firme candidata a implementar en lugar de nuestro sistema M-QAM.

Se trata, como ya definimos en el apartado III.2.1, de variar la fase de la portadora entre M posibles valores discretos, que corresponda cada uno de ellos a uno de los M distintos símbolos del sistema.

La ***ecuación*** que describe este tipo de modulación ya la dimos a conocer anteriormente, y es la siguiente[†1] [†4]:

- $S_i(t) = Re\,[g(t)\cdot\exp(j\cdot w_c\cdot t+\theta_i)] = g(t)\cdot\cos(w_c\cdot t+\theta_i)$; $i = 0,1...M-1;\ 0 \le t \le T$

,donde la ***fase*** tendrá estos M posibles valores:

- $\theta_i = 2\cdot\pi\cdot i / M;$ $i= 0,1, \ldots M-1$

En lo que respecta al ***ancho de banda*** y a la ***eficiencia espectral***, se puede comprobar que ambos sistemas son igual de eficientes.

También, para ambos formatos, si incrementamos el número de puntos de mensaje del sistema, M, aumentará la ***probabilidad de error***, y para mantener ésta constante, necesitamos aumentar la ***energía media*** del sistema.

Si nos fijamos en sus ***constelaciones***, ambas son ***bidimensionales***, esto es, la base vectorial la componen un vector en fase, $\Phi_1(t)$, y otro en cuadratura,$\Phi_2(t)$, pero existe una diferencia fundamental, y es que mientras el sistema ***M-QAM*** tiene ***dos grados de libertad*** (amplitud y fase), el ***M-PSK*** tiene sólo ***un grado*** (fase). Por eso, las constelaciones de las señales M-PSK para cualquier M, tienen los puntos distribuidos a

la misma distancia (misma energía de símbolo) y distinta fase (siempre formando una circunferencia); sin embargo, los puntos de mensaje de la señal M-QAM se encuentran distribuidos en diferentes distancias al origen y distintas fases. Así, los símbolos de las señal M-QAM pueden distribuirse más y mejor en el plano de la constelación, sin estrecharse, y por tanto, sin aumentar la probabilidad de error en la decisión de un símbolo[†1] [†4].

De hecho, para M>4, y conforme M aumenta, ya se ve que la distancia entre cada elemento se reduce considerablemente más en la constelación M-PSK, en comparación a la constelación M-QAM.

Veamos a continuación unos ejemplos ilustrativos de las representaciones vectoriales de los símbolos de estas dos técnicas de modulación digital, para diferentes valores de M, donde se refleja con bastante evidencia la distancia entre símbolos en un sistema de modulación y en otro.

Hay que destacar que la energía de pico, E_{peak}, es la misma para ambas señales, y tiene un valor igual a 1, cuya raíz cuadrada, o sea, la amplitud del símbolo más alejado, coincide con el valor máximo de la tarjeta de sonido.

- Para M = 4:

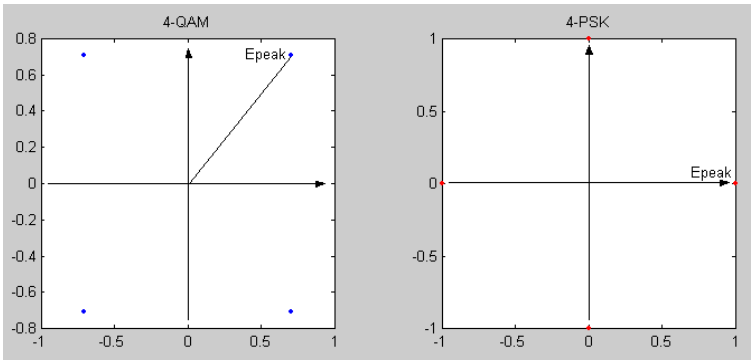


Fig. 27.- 4-APK vs. 4-

Aquí todavía existe semejante probabilidad de error, ya que ambas constelaciones son idénticas, salvo que giradas π / 4 radianes.

- Para M = 8

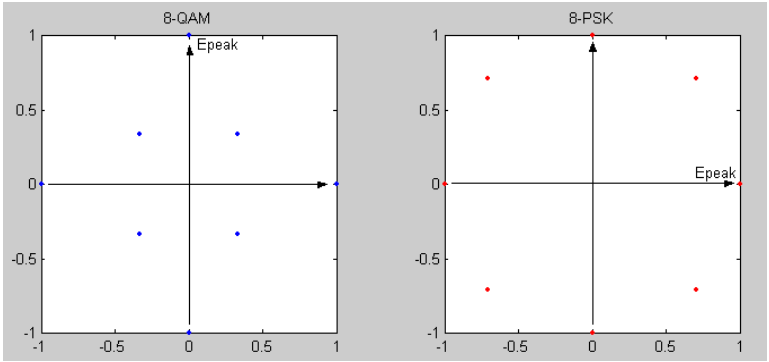


Fig. 28.- 8-APK vs. 8-

En este caso, la distancia entre los símbolos de la constelación 8-PSK todavía es considerable, en relación a la distancia entre símbolos de la constelación 8-APK, si bien la energía media del sistema en esta última es menor que la energía media de la constelación 8-PSK. Para este valor de M, hemos optado por una constelación no rectangular en la distribución 8-APK.

- Para M = 16

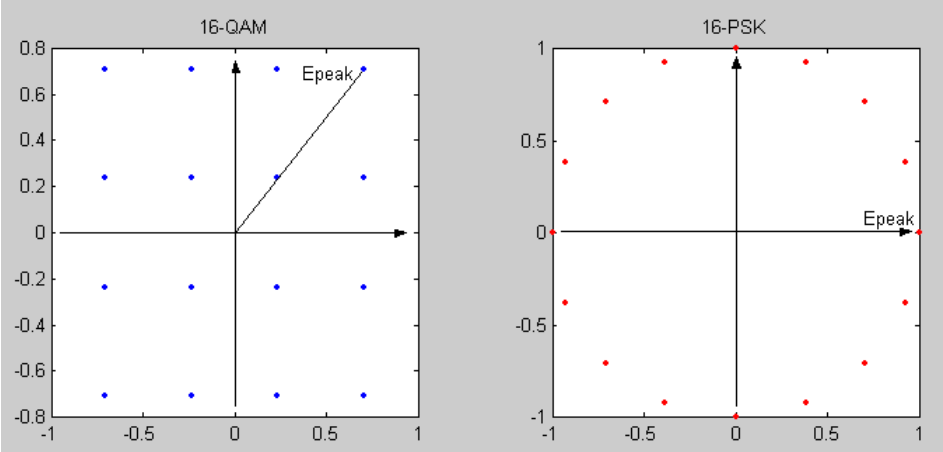


Fig. 29.- 16-APK vs. 16-PSK

En este gráfica, la distancia mínima entre símbolos empieza a ser visiblemente más pequeña en la constelación 16-PSK.

- Para M = 32

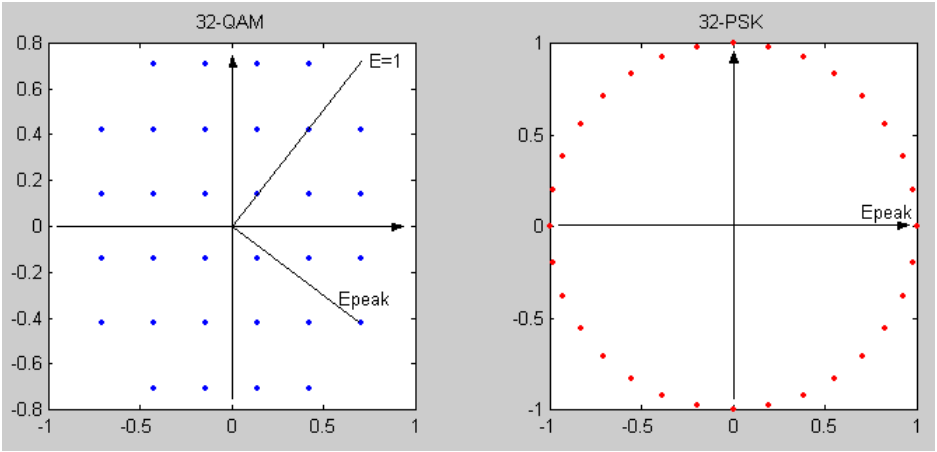


Fig. 30.- 32-APK vs. 32-PSK

La separación entre símbolos adyacentes es ya considerablemente mayor en la constelación 32-QAM. Comentar en este caso, que esta última constelación proviene de una 36-QAM rectangular, a la que se le quitan los 4 símbolos con más energía (los de las esquinas), para conseguir un número de símbolos potencia de dos.

- Para M = 64, M = 128 y M =256, sencillamente la distancia entre los símbolos de la señalización M-PSK se va reduciendo de forma abismal, hasta hacerse ínfima. Así, para la misma energía pico, es notoria la mejor distribución de la constelación M-QAM, lo que reduce su Energía media, y la probabilidad de error de símbolo del sistema[†1] [†4]. Veamos esas gráficas en cuestión:

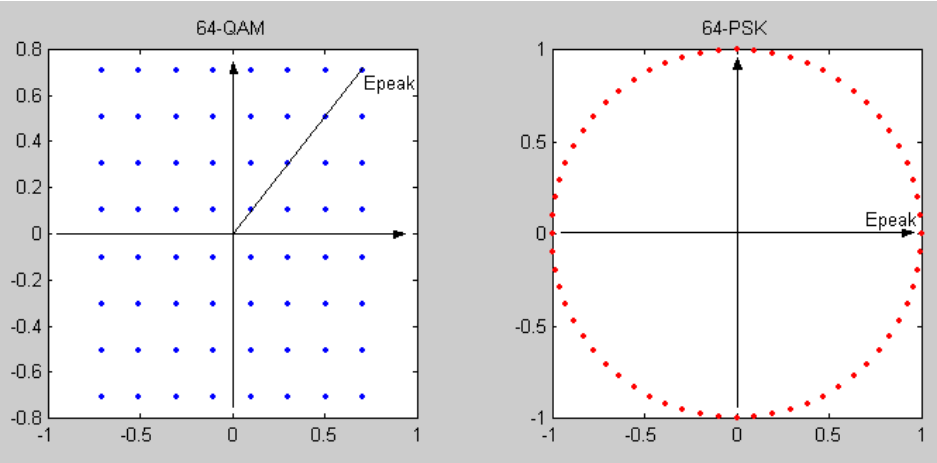


Fig. 31.- 64-APK vs. 64-PSK

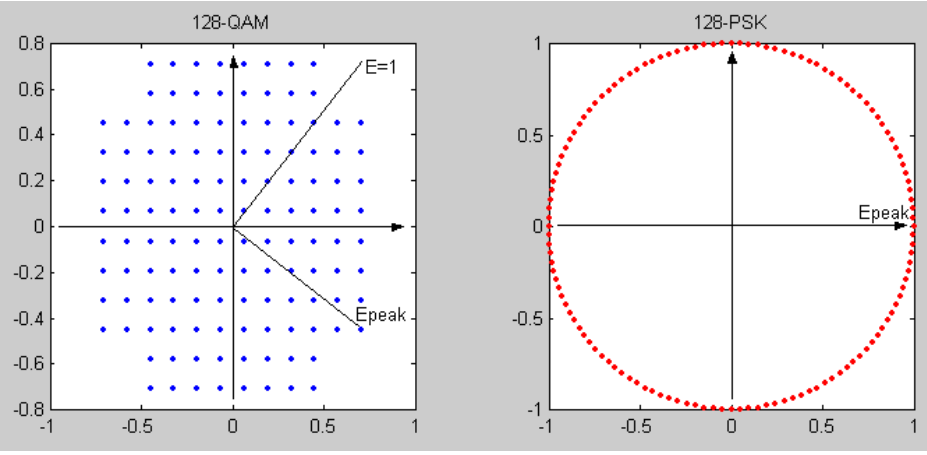


Fig. 32.- 128-APK vs. 128-PSK

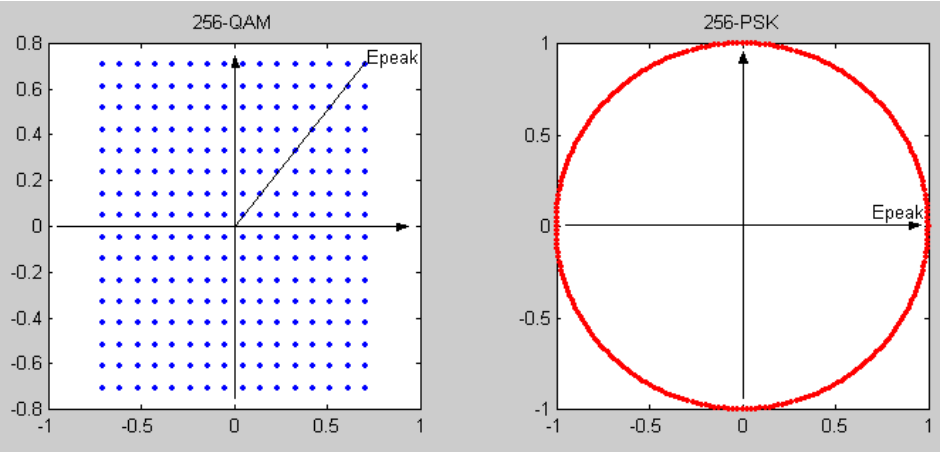


Fig. 33.- 256-APK vs. 256-PSK

Al igual que hicimos con la señalización M-PAM, vamos a igualar la distancia mínima entre símbolos, para un valor genérico de M, a la distancia mínima entre símbolos más pequeña de nuestro sistema M-ario, que se da, como ya sabemos, para M = 256, esto es, para una constelación de una modulación 256- QAM.

En primer lugar, veamos la distancia mínima para un sistema genérico M-APK, aunque como ilustración, usaremos la 16-PSK.

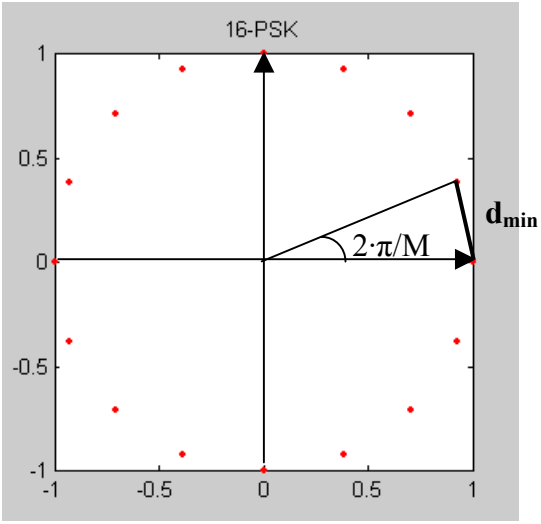


Fig. 34.- 16-PSK

Sin pérdida de generalidad, ya que todos los símbolos son equidistantes, tomemos la distancia mínima, d_{min}, la que existe entre el símbolo de fase cero, y el de fase $2\cdot\pi/ M$.

Ambos símbolos tienen módulo unidad.Con estos datos, y recurriendo a los métodos trigonométricos habituales, llegamos a que esa distancia mínima es:

- $d_{min} = 2 \cdot \text{sen}\ (\pi / M)$

E igualando ambas distancias mínimas:

- $d|_{M\text{-}PSK} = d|_{256\text{-}QAM} \rightarrow 2 \cdot \text{sen}\ (\pi / M) = 2 / (15 / \sqrt{2}) \rightarrow M \sim 66$

De este modo tenemos que, para un valor aproximado de M=66, la distancia entre símbolos adyacentes es idéntica a la distancia entre símbolos vecinos para un sistema 256- QAM. Por lo tanto, mediante este modo un tanto atípico de intuir la probabilidad de error de un sistema, aproximamos el valor máximo de M=64, para un sistema de modulación digital de fase.

De esta manera, conseguimos una velocidad de transmisión de:

- $R_b = k \cdot D = 6 \cdot 4410 = 26460$ bps

Concluimos entonces que, de todas las hipotéticas alternativas de implementación, resulta la más rápida, aunque sigue siendo un sistema más lento que nuestro bloque M-QAM.

Por último, añadir respecto a la señalización M-QAM, que esos dos grados de libertad a los que hicimos mención (fase y amplitud), también tiene sus pequeños inconvenientes, y es que la modulación M-QAM es la más sensible a las no linealidades del canal, ya que la información va inmersa en dos de los parámetros de la portadora.

III.2.3.- Diseño de nuestro Sistema de Comunicaciones

Dejada atrás la tediosa labor comparativa y teórica, nos meteremos de lleno en el diseño de nuestro ***sistema de comunicaciones***.

Lo más razonable es que dividamos esta sección en dos grandes bloques bien diferenciados:

- En primer lugar, nos ocuparemos de la ***parte transmisora***. Mostraremos un dibujo ilustrativo del Transmisor, desde la fuente digital donde tomamos los datos, hasta la zona específica de transmisión de las muestras, e iremos desgranando cada parte de este bloque, profundizando aún más en las características de nuestra señal, y comentando los problemas que nos hemos ido encontrando, las soluciones que les hemos dado y los parámetros por los que nos hemos decantado.
- Posteriormente, trataremos la ***fase receptora***, dibujando primero un esquema orientativo, y desmenuzando igualmente cada parte del tramo desde el ecualizador o igualador de canal, hasta la definitiva formación del archivo en el destino, a partir de los datos recibidos.
- Paralelamente, iremos desarrollando una ***aplicación práctica*** al final de cada módulo, para mostrar el aspecto que la señal va tomando, y así poder tener una idea más intuitiva de cada parte del sistema.

En el siguiente capítulo, el III.3, procederemos a implementar vía software, con la ayuda del programa Matlab, todo lo modelado en esta etapa de la memoria. De ahí la importancia de las decisiones que tomemos en esta sección.

Veamos, antes de entrar en detalles, una representación global de nuestro sistema de comunicaciones:

Fig. 35.- Sist. Comunicaciones

BLOQUE TRANSMISOR

Hemos considerado nuestro ***Bloque Transmisor*** como el tramo desde la fuente digital de datos hasta la salida del convertidor digital/analógico de la tarjeta de sonido.

Si fuéramos un poco exhaustivos, nos veríamos obligados a decir que realmente, de las cinco subetapas en que dividimos este Bloque Transmisor, la única considerada como parte puramente transmisora es la correspondiente al ***Modulador APK***, ya que si observamos en cualquier libro de texto[†1] [†4], la ***Fuente digital*** siempre es considerada como bloque ajeno al transmisor en cualquier Sistema de comunicaciones. Por otra parte, las ***fases de entramado***, como ya veremos, son el resultado de la necesidad de dividir la información en tramas, como consecuencia del imperfecto sincronismo entre los relojes del transmisor y receptor, y no constituyen una parte puramente del transmisor. Por último, el ***Convertidor digital / analógico*** es un componente del puerto LINE OUT de la tarjeta de sonido[†7], y aunque se trata de una etapa transparente a nuestro diseño, dicho convertidor posee varios parámetros que controlamos mediante

variables, y limita algunas características de nuestro sistema, por lo que hemos decidido ubicarlo en nuestro dibujo aunque con línea discontinua.

Aquí mostramos un ***esquema del Bloque Transmisor*** en cuestión, donde también hemos introducido el ***canal***, con su inevitable ***fuente de ruido***:

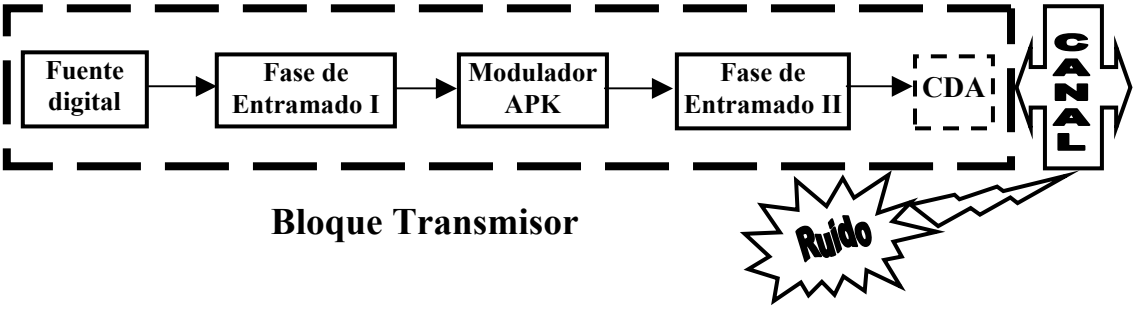


Fig. 36.- Bloque Transmisor

Procedamos ya a entrar, con más detalle, en cada una de las fases de esta etapa transmisora:

- Fuente digital

Esta subetapa la constituyen los ***bytes*** de los ***ficheros*** a transmitir en sí. Hay que señalar que, al tratarse de una fuente netamente digital, no es necesario convertir ninguna señal analógica en digital, por lo que no hace falta introducir ningún sistema tipo ***PCM***[†1] (modulación por codificación de pulsos) a la salida de la fuente.

Por otra parte, tampoco hemos estimado necesario ningún tipo de ***compactación*** de datos, ni de ***encriptación*** de los mismos, por lo que la implementación de cualquier tipo de ***codificación fuente***[†1] está totalmente descartada, ya que resultaría ridículo pensar que alguien pudiera estar interesado en filtrar los datos transmitidos. Además, la conexión es punto a punto.

Respecto a los ***datos originales***, cabe decir que se hallarán almacenados en ***bytes***, en un medio de almacenamiento, que generalmente se corresponderá con el ***disco duro***, aunque también pudieran estar alojados en un ***diskette***.

Se accederá a estos datos mediante su ***lectura***, pero para poder leerlos, primero hay que proceder a la ***apertura*** del fichero que contiene dichos bytes. Al abrirlo, nos dará un ***identificador de fichero***, ***fid***, que usaremos para leer su información.

La implementación en Matlab de ambos procesos de acceso al fichero se ha realizado mediante dos funciones propias de este lenguaje:

- *fopen*: para abrir el fichero y obtener su identificador.
- *fread*: para leer los bytes del fichero en cuestión.

Por otra parte, mediante la configuración de los parámetros de la función *fread*, también realizamos la ***separación en tramas de la información***, que es primordial para el buen funcionamiento de nuestro sistema, como veremos más adelante.

Usaremos un *bucle*, y en cada ciclo de éste, leeremos una trama, y la haremos pasar por la fase de entramado I, el modulador APK, y la fase de entramado II. En el

siguiente ciclo volveremos a leer otra trama, y le aplicaremos el mismo procedimiento, y así sucesivamente, hasta terminar de leer el fichero. Es, mediante este bucle y la ayuda de *fread*, como logramos simular la separación de la información en tramas.

También es interesante comentar que a la salida de la función *fread* tenemos una columna de números en decimal, que corresponden al valor de los bytes leídos. Necesitaremos, por tanto, convertir esos números decimales en números binarios de 8 bits cada uno, y darles un formato de ***secuencia binaria*** de unos y ceros. Para ello recurriremos a las funciones de Matlab *dec2bin* y *reshape*, respectivamente.

Un ejemplo ilustrativo, para una trama en concreto (obviando el bucle), podría ser el siguiente. Supongamos que los 3 primeros bytes de la trama son: 4, 5 y 7.

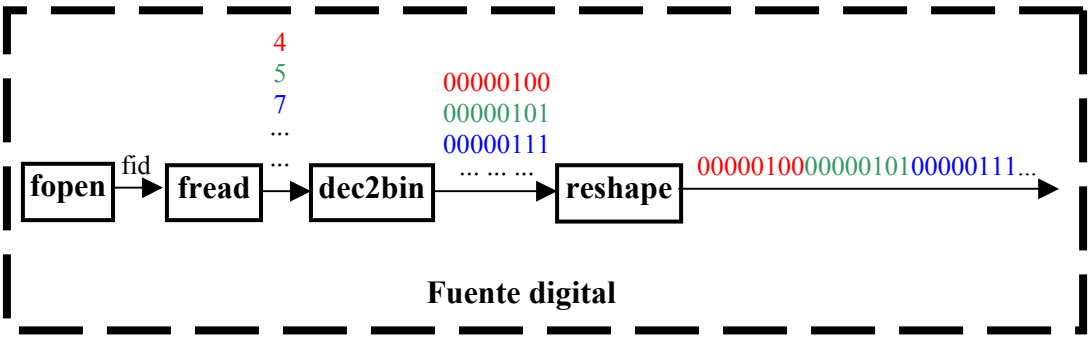


Fig. 37.- Fuente digital

- Fase de Entramado I

Ya hemos comentado que, debido a que las frecuencias de reloj del transmisor y del receptor no son exactamente iguales, existirá ***falta de sincronismo*** en el sistema. Este problema lo analizaremos con más detenimiento en otro apartado posterior.

En dicho apartado dedicado al sincronismo, veremos que la diferencia entre la frecuencia de reloj de ambos bloques es de 0.5 hz. aproximadamente. Entonces es obvio concluir que cada 2 segundos se ganará una muestra espuria.

Para paliar estos efectos, hemos optado por ***dividir la información en tramas***, de manera que al receptor no le dé tiempo a desincronizarse con el transmisor hasta el punto de ganar una muestra espuria.

Concretamente, se puede dar por tolerable una ***desviación en los relojes*** de ambos bloques de un ***1%***, respecto del 100% que supondría la una desviación igual a un ciclo de reloj. Si para tener una desviación del 100% tienen que transmitirse 88200 muestras (a la frec. de muestreo Fs = 44100 hz., esto es, tienen que transcurrir dos segundos de transmisión), es indudable que para que exista únicamente una desviación menor o igual al 1%, tendrán que transmitirse aproximadamente ***882 muestras*** por cada trama.

Al existir una división en tramas, surge la necesidad de dotar de ***información redundante*** a dicha trama, para su correcta identificación en el receptor. Para ello hemos optado por usar como trama, la ***recomendación IEEE 802.3***, conocida más popularmente como ***Ethernet***[†10].

Dedicaremos más adelante un apartado para hablar acerca de este modo de estructuración de la información. Aún así, y para continuar con el análisis, paso a paso, de nuestro sistema, vamos a comentar brevemente los ***nuevos campos*** que añadiremos a la cadena binaria de información proveniente de la salida de la Fuente digital:

- ***LONG*** : indica la ***longitud***, en número de ***bytes***, del campo DATOS. Este campo ocupará dos bytes. Podríamos usar perfectamente un byte, puesto que con dicho byte podría codificar una longitud del campo DATOS de hasta hasta 255 bytes, y teniendo en cuenta que cada símbolo va a ser representado por unas diez muestras, y que en el mejor caso (M = 256 símbolos) de que esos 255 bytes se correspondan con el menor número de símbolos (exactamente 1 símbolo/byte, para este valor de M), y por tanto, en el menor número de muestras, dicha cifra sería igual a 2550 muestras, y eso sin contar el resto de muestras de la trama, que no son puramente de información, que elevaría la cantidad aún más. Este valor, de por sí, es mucho mayor de lo que nos está permitido transmitir por cada trama (882 muestras), pero, por guardar el parecido con la trama Ethernet, y por si en un futuro se mejorara la eficiencia de nuestro sistema en cuanto a número de muestras a transmitir por trama, vamos a conservar este campo con una longitud ***de 2 bytes***.

- ***DATOS***: aquí alojaremos la ***información propiamente dicha***, los bytes leídos de los ficheros en la fuente. El número de bytes que habrá en dicho campo es un parámetro de diseño variable de nuestro sistema. Este parámetro está limitado por las 882 muestras totales por trama que podemos transmitir como máximo, para no sobrepasar el 1 % de la desviación entre ambos relojes.
  De todos modos, intentaremos aumentar un poco este número de muestras a transmitir, pues esto conlleva menor número de tramas a transmitir y , por consiguiente, menor número de bytes de control, para procesar y transmitir.

- ***CRC***: Corresponde al ***código de redundancia cíclica***. Sirve para detectar los errores de decisión de los bits en recepción del campo DATOS, debido a la presencia de ruido en el canal o a una mala ecualización. Al final, hemos optado por introducir sólo dos bytes en este campo, tratándose, por tanto, de un CRC-16, como ya veremos. Este campo sí corresponde a una etapa propia de la transmisión, tal y como aparece en los libros de texto: la ***codificación de canal***[†10], que consiste en introducir información redundante, para detectar e incluso corregir (este no es el caso) errores en la recepción de los datos.

Vamos a mostrar, seguidamente, un ***ejemplo*** de cómo quedaría nuestra trama, partiendo de la hipótesis anterior, de que los primeros 3 bytes leídos eran 4, 5 y 7, y suponiendo además que se han leído 63 (111111_b) bytes de información:

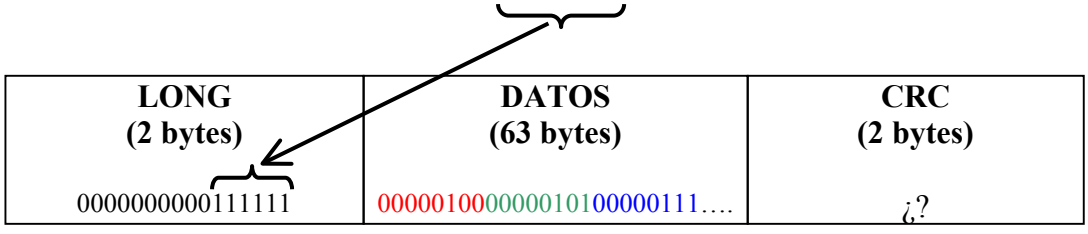


Fig. 38.- Fase Entramado I

- Modulador APK

Este es la ***etapa estrella*** de nuestro sistema de comunicaciones, puesto que en ella diseñamos el ***módem*** en sí. Las etapas previas cumplen funciones meramente acondicionadoras, que son necesarias para el buen funcionamiento de nuestro sistema, tal y como lo hemos concebido, como pueden ser la de proporcionar a los datos un formato de bits o la formación de las tramas. En esta fase, sin embargo, procederemos al diseño de la ***técnica de modulación M-QAM***.

En primer lugar, expondremos un dibujo donde se verán reflejadas todas las partes de este sistema modulador, y posteriormente analizaremos, paso a paso, cada una de ellas, detallando y fijando cada uno de los conceptos o parámetros de diseño que vayan surgiendo en nuestro análisis.

Pero antes de nada, es importante hacer un ***análisis de este tipo de modulación***, para concentrar todos los conceptos sobre esta señal, tratados en apartados anteriores, y profundizar un poco en nociones tan importantes para nuestro desarrollo práctico, como puede ser la representación vectorial de la señal M-APK, ya que para conseguir nuestra meta, que es el diseño de un módem M-APK, es imprescindible conocer la forma física y analítica de la señal que queremos obtener a la salida del Modulador APK.

Introducción a la Modulación M-QAM

Como ya hemos comentado, se trata de una ***modulación digital híbrida***, esto es, en ***amplitud y fase***. También hemos mencionado que su señal es equivalente a la superposición de ***dos señales PAM en quadratura***.

Vimos que la ***expresión*** para uno de los M posibles símbolos de esta representación M-aria es la siguiente[†1] [†4]:

- $S_i(t) = \mathrm{Re}\,[a_i \cdot g(t)\cdot \exp(j\cdot(w_c\cdot t+\theta_i))] = \underbrace{A_i\cdot g(t)\cdot\cos(w_c\cdot t)}_{\textbf{M-ASK en fase}} - \underbrace{B_i\cdot g(t)\cdot \mathrm{sen}(w_c\cdot t)}_{\textbf{M-ASK en cuadratura}}$

  con $i=0,1...M-1$; $0 \le t \le T$

Vamos a ***analizar*** más detenidamente esta ***expresión***:

- En primer lugar, es fácil ver que esta expresión es válida sólo para un intervalo de tiempo, ***T***, que es el ***tiempo de símbolo***, lo que significa que dicha expresión se va a repetir cada T segundos a la salida del modulador.
- En segundo lugar, podemos observar que la forma de la señal varía, según cambie el valor de la **variable 'i'**, y ésta sólo tiene M estados posibles, coincidiendo cada uno de ellos con los M símbolos del sistema, como también sabemos.
- Ambas expresiones de la señal son propias, como es lógico, de una señal paso de banda. La ***envolvente compleja***[†2] [†4] [†1], ***š(t)***, o sea, su señal paso de baja equivalente, es una señal compleja, cuyas expresiones, tanto en forma polar, como en forma cartesiana, son las siguientes:

➢ $\check{s}(t) = \underbrace{a_i \cdot \exp(j \cdot \theta_i) \cdot g(t)}_{\textit{forma polar}} = \underbrace{A_i \cdot g(t) + j \cdot B_i \cdot g(t)}_{\textit{forma cartesiana}}$

- ***g(t)*** es el ***pulso conformador, pulso transmisor, o pulso básico***, que, como su nombre indica, va dando forma a cada símbolo (representado éste con una delta de Dirac, de amplitud la de dicho símbolo) que entra en dicho Filtro Transmisor, y lo va dotando de características físicas reales, para poder ser transmitido por el canal. En nuestro caso, hemos optado por un ***filtro raíz de coseno alzado***, pero eso lo veremos un poco más adelante[†1].
- En cuanto a su ***expresión vectorial***, ya mencionamos el ***carácter bidimensional*** de este tipo de señales. Así, una forma típica vectorial, que ya mostramos con anterioridad en otros apartados, sería[†1] [†4]:

➢ $s(t) = \underbrace{s_{i1} \cdot \Phi_1(t)}_{\text{Fase}} + \underbrace{s_{i2} \cdot \Phi_2(t)}_{\text{Cuadratura}} \qquad i = 0,1,.....M\text{-}1 \quad 0 \quad ; \quad 0 \le t \le T$

Comparando ambas expresiones de s(t), e identificando componentes, hemos optado por tomar como ***elementos*** de la ***base vectorial***, las siguientes funciones:

➢ $\Phi_1(t) = g(t) \cdot \cos(w_c \cdot t)$
➢ $\Phi_2(t) = -(g(t) \cdot \text{sen}(w_c \cdot t))$

Así, las coordenadas de un punto de mensaje genérico serían:

➢ $s_{i1} = A_i$
➢ $s_{i2} = B_i$

Nos parece oportuno hacer un ***inciso***. Realmente, hemos escogido una ***base ortogonal***, pero ***no ortonormal***, ya que los ***módulos de $\Phi_1(t)$ y $\Phi_2(t)$***, $||\Phi_1(t)||$ y $||\Phi_2(t)||$, son iguales a:

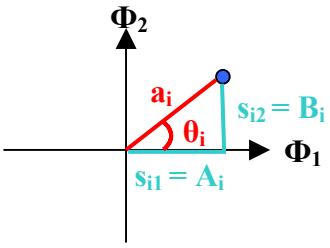
➢ $||\Phi_1(t)|| = ||\Phi_2(t)|| = \sqrt{(E_g / 2)}$

, siendo E_g la energía del pulso g(t). No olvidemos que la ***energía de una función*** es el cuadrado del módulo de dicha función; de tal manera que, si esa función representa algún símbolo en una constelación, la energía de ese punto de mensaje estará relacionada cuadráticamente con la distancia de ese punto al origen de coordenadas. Así, para el caso de g(t) [†1] [†4]:

➢ $||g(t)|| = \sqrt{E_g}$

Así, si quisiéramos usar ***vectores ortonormales***, deberíamos multiplicar los nuestros por el inverso de su módulo, es decir, por $\sqrt{(2/ E_g)}$. De la misma forma, deberíamos multiplicar nuestras coordenadas, s_{i1} y s_{i2} , por $\sqrt{(E_g / 2)}$.

De todas formas el resultado es el mismo, y usando nuestros vectores, escalamos las coordenadas, y resulta todo mucho más simple desde el punto de vista analítico, puesto que no tenemos que ir arrastrando esa constante ($\sqrt{(E_g / 2)}$).
Una vez aclarado este punto sobre la ***no ortonormalidad*** de nuestras funciones vectoriales, nos parece oportuno dibujar un símbolo cualquiera, mostrando las coordenadas que lo definen, y que hemos comentado previamente:



a_i : Amplitud del símbolo
θ_i : Fase del símbolo
$s_{i1} = A_i$: Coordenada en fase
$s_{i2} = B_i$: Coordenada en cuadratura

Fig. 39.- Símbolo APK

- Así, para cada una de los M posibles valores de la ***variable 'i'***, esto es, para cada uno de los ***M posibles símbolos***, tendremos ***una pareja distinta*** de valores ***fase/cuadratura (A_i , B_i)***, o bien, una pareja diferente de valores ***Amplitud/fase (a_i , θ_i)***. Se refleja claramente el tipo de modulación empleado, ya que cada símbolo tendrá una pareja de amplitud y fase distinta.

Por último, hablaremos fugazmente acerca de sus ***constelaciones***. El hecho de tener, para este tipo de representación simbólica, ***dos grados de libertad*** (amplitud y fase), y poder elegir independientemente los niveles de amplitud de las componentes ASK de fase y de cuadratura, explica que exista ***mucha variedad*** en cuanto a la forma de las constelaciones de las señales M-QAM, para un mismo valor de M. Así por ejemplo, para ***M = 16***, se usan ***constelaciones*** con los siguientes aspectos:

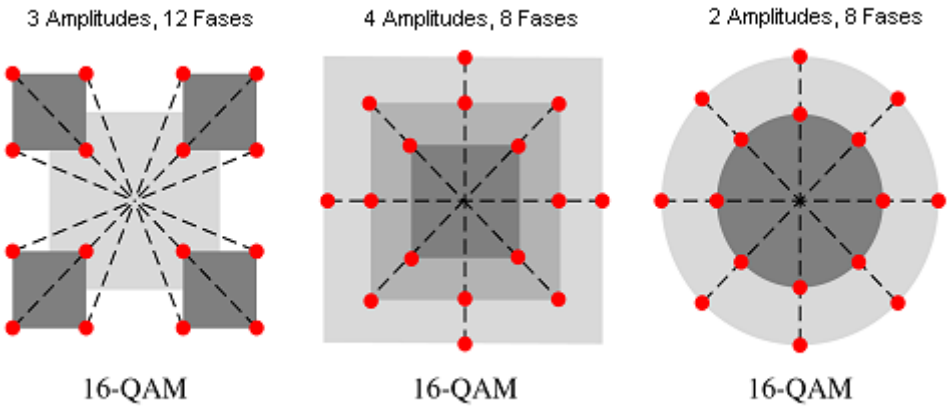


Fig. 40.- Constelaciones 16-QAM

Nosotros hemos optado por ***constelaciones rectangulares***[†1][†4] (o derivadas) principalmente, donde los símbolos están igualmente distanciados unos de otros. A este tipo de constelación pertenece la primera de las tres representaciones simbólicas de las modulaciones 16-QAM mostradas anteriormente.

Una cosa que sí nos gustaría resaltar, es el hecho de que dibujar ***puntos*** en las ***constelaciones*** por comodidad, pero realmente se trata de ***vectores***.

Aquí acabamos nuestra particular introducción a la modulación M-QAM, y ahora sí, sin más dilación, procedamos a mostrar el aspecto que presentará nuestro sistema ***modulador M-QAM o M-APK*** :

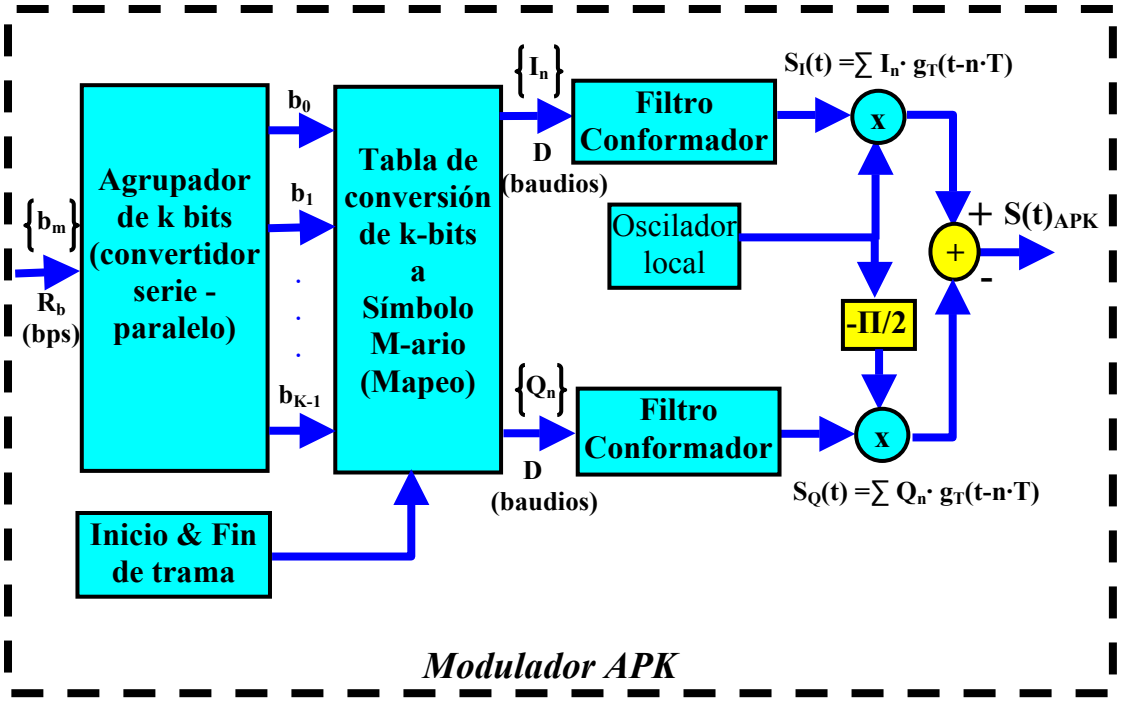


Fig. 41.- Modulador APK

Agrupador de k bits & Tabla de conversión

El ***Agrupador de k bits*** es una especie de convertidor serie a paralelo al que le llegan la secuencia binaria, ***{b_m}***, con una ***tasa de bits R_b***, procedente de la Fuente digital, y los va almacenando en bloques de k bits (b_0,b_1,b2...b_{k-1}), para pasarlos a la ***Tabla de conversión***, que no es más que una tabla de correspondencia M-aria, en la que a un conjunto de k bits se los relaciona con uno y sólo un punto de mensaje o símbolo de la constelación. A esta operación relacional se la suele llamar ***mapeo***[†1][†4]. A la salida de dicha Tabla, tendremos una secuencia de ***coordenadas en fase, {I_n},*** y ***cuadratura, {Q_n},*** de los símbolos correspondientes a la sucesión de bloques de k bits que han ido pasando por el conjunto Agrupador de k-bits / Tabla de conversión. Para realizar esta operación, es decir, para tomar a la entrada un grupo de k-bits y dar a la salida un valor en fase y otro en cuadratura, que defina perfectamente la ubicación del símbolo en su constelación correspondiente, este conjunto tiene un tiempo de símbolo T

= 1/D, siendo ***D*** la ***tasa de baudios*** o número de símbolos por segundo que salen de la tabla de conversión. De este razonamiento, obtenemos la relación entre la tasa de bits, y la tasa de símbolos, que ya tuvimos la oportunidad de ver, cuando hablamos, en su momento, del dimensionamiento y parametrización del sistema[†1][†4]:

- $T = k \cdot T_b \leftrightarrow R_b = k \cdot D$

La variable ***k***, como ya sabemos, hace referencia al número de bits que necesitamos para codificar los M símbolos del sistema digital. En un apartado previo comentamos la relación entre k y M[†1][†4]:

- $k = \log_2 (M)$

Como uno de los objetivos a alcanzar en nuestro módem es la ***versatilidad*** de éste, vamos a diseñarlo de tal manera que admita distintas constelaciones M-arias en nuestro sistema, por lo que ambos parámetros de diseño ***M*** y, consecuentemente, ***k***, serán totalmente ***configurables*** por el usuario. De esta manera, según el valor de k, el ***agrupador de bits*** irá almacenando la secuencia binaria en bloques de un tamaño u otro, y para el ***mapeo***, se usará una de las ***siete posibles tablas de conversión***, según el valor de la variable M. Sin embargo, y como ya hemos explicado, M está limitada por la tasa de errores del sistema. Así los distintos valores de M, y los correspondientes a k, que hemos admitido en nuestro modulador son:

M	4	8	16	32	64	128	256
k	2	3	4	5	6	7	8

Tabla 2.- Valores de M y k

Además, sabemos que la ***tasa de símbolos***, ***D***, viene fijada por la frecuencia de muestreo y el número de muestras / símbolo. Para adquirir las mayores prestaciones en cuanto a velocidad de transmisión, la ***frecuencia de muestreo***, ***Fs***, como también dijimos, la hemos fijado en ***44100 hz***. Sin embargo, el ***número de muestras por símbolo***, ***N***, lo hemos hecho configurable por el usuario. Así, para cada valor de N, tendremos un valor distinto para D, y siete valores distintos para R_b(según el valor que le demos al parámetro M del modulador). Así, la ***gama de velocidades*** que adquiere nuestro sistema de comunicaciones es ***muy variada***. De todos modos, como nos interesa la velocidad, escogeremos un valor para N lo más bajo posible. Ya vimos que el valor mínimo práctico de N, sin que existan errores en el sistema, es N=9. En nuestro caso, seguiremos con el valor que hemos tomado hasta ahora, ***N=10***, por lo que la tasa de símbolos será de ***4410 baudios***, y para este valor en concreto, el ***abanico de velocidades*** en nuestro sistema será:

k	2	3	4	5	6	7	8
R_b (bps)	8820	13230	17640	22050	26460	30870	35280

Tabla 3.- Valores de Rb para D=4410 hz.

Tras haber fijado el rango de valores de ciertos parámetros del sistema, sería oportuno, siguiendo con el análisis del esquema del modulador, comentar la ***nomenclatura*** que usamos en ciertos tramos de éste:

- El ***subíndice 'm'*** que acompaña a la secuencia binaria, ***{b_m}***, es un valor discreto que marca los distintos tiempos de bit, esto es, concebimos esta cadena binaria como un tren de deltas de dirac, de amplitud {bm}. Analíticamente, sería de la siguiente forma:

  - $\{b_m\}=\sum_m b_m \cdot \delta(t-m\cdot T_b)$; $b_m = 0$ ó 1 (dato binario)

- De igual manera, en la secuencia de coordenadas simbólicas de fase y de cuadratura, ***{I_n}*** y ***{Q_n}*** respectivamente, el ***subíndice 'n'*** también tiene la misma indicación temporal, refiriéndose, en este caso, al tiempo de símbolo. Para no provocar confusiones en los conceptos temporales, hemos decidido usar distintos subíndices. La representación analítica sería la siguiente:

  - $\{I_n\}=\sum_n I_n \cdot \delta(t-n\cdot T)$
  - $\{Q_n\}=\sum_n Q_n \cdot \delta(t-n\cdot T)$

  Si observamos estas coordenadas, **I** y **Q**, y las comparamos con las coordenadas en fase y cuadratura de la señal genérica que expusimos en la introducción a la modulación digital que hicimos al principio de este apartado del Modulador APK, nos daremos cuenta que son equivalentes a ***A_i*** y ***B_i*** respectivamente. En este caso, hemos prescindido del ***subíndice 'i'*** (que indicaba uno de los M posibles símbolos, ¡¡y no tiene nada que ver, obviamente, con el subíndice 'n'!!), para las variables I y Q.

Para afianzar conceptos, pongamos un pequeño ***ejemplo***, para ver cómo funciona dicha cadena de etapas. Supongamos que hemos elegido una modulación ***4-QAM***. El dibujo está bastante claro, y no creemos que haya nada que comentar, salvo que las coordenadas cartesianas de los símbolos las escalamos de tal manera que la amplitud de éstos sean la unidad.

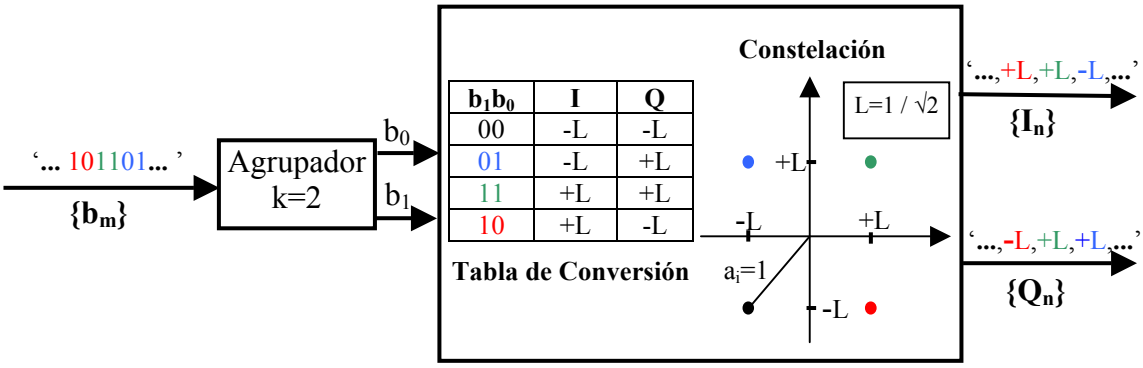


Fig. 42.- Agrupamiento y mapeo

Es interesante comentar que, como podemos deducir en el dibujo, a la vez que formamos ***un símbolo 4-APK***, estamos formando ***dos símbolos 2-ASK***. Esto quiere decir que, hablando de forma general, con la modulación QAM, usamos el mismo ancho de banda que una señal ASK, para transmitir dos señales ASK. Por tanto aprovechamos el doble el ancho de banda[†1][†4].

En este punto del sistema, nos parece propicio a hablar de las ***constelaciones de nuestro sistema***, y cómo hemos plasmado en ellas la ***Tabla de conversión***, esto es, la relación entre los bloques de bits y las características vectoriales de los símbolos.

Constelaciones de nuestro sistema

Ya comentamos que queremos un módem lo más versátil posible. Por ello hicimos el parámetro ***M variable***. El hecho de tener este parámetro de diseño siete estados viables entre M = 4 y M = 256, provoca que tengamos siete tablas de conversión distintas, y con ellas, ***siete*** tipos de ***constelaciones***, cada una, lógicamente con un número distinto de símbolos igual a M.

Para los valores de $\boldsymbol{M = 2^k = 4, 16, 64, 256}$ ***símbolos*** (o sea, aquellas en las que la variable 'k' es par), hemos optado por ***constelaciones cuadradas***, donde los símbolos están igualmente espaciados, formando en su conjunto un cuadrado, y aunque no sean las más óptimas en cuanto a energía media, en relación a otro tipos de constelaciones M-QAM, son muy sencillas de implementar. En este caso, se puede comprobar, que la señal M-QAM es una superposición de dos señales L-ASK, con $L=\sqrt{M}$[†1][†4].

En estas ***constelaciones cuadradas***, la señal de máxima energía tiene ***energía pico unidad***, o sea, en términos de amplitud, la máxima es la unidad. Esto lo hacemos por las ***limitaciones*** que presentaba el rango digital de los convertidores D/A y A/D de la ***tarjeta de sonido***, que ya comentamos en el capítulo sobre el estudio del canal. Así evitamos transmitir muestras que se excedan del rango [1,-1], y que consecuentemente sean truncadas en estos valores extremos.

Para los valores de $\boldsymbol{M = 2^k = 32}$ y***128 símbolos*** (o sea, aquellas en las que la variable 'k' es impar), hemos optado por ***constelaciones casi cuadradas***, ya que nuestra constelación ***32-QAM*** deriva de una constelación cuadrada ***36-QAM***, a la que se le ha eliminado los símbolos de máxima energía, o sea, los símbolos de los picos del cuadrado. De igual manera, la constelación ***128-QAM*** deriva de una constelación cuadrada ***144-QAM***, a la que también se le ha eliminado, en cada cuadrante, los 4 símbolos con mayor energía. En estos dos casos, la ***amplitud máxima*** es un poco ***menor que la unidad***, ya que, por ejemplo, para el caso de 32-QAM, diseñamos una 36-QAM, con amplitud máxima unidad, y luego quitamos esos símbolos de energía pico de la constelación.

La ***constelación*** para $\boldsymbol{M = 8}$ ***símbolos*** que hemos elegido es un tanto peculiar. Se trata de 4 símbolos interiores formando una especie de constelación cuadrada 4-QAM, y 4 símbolos, formando otra constelación 4-QAM, pero desfasada 45º respecto a la anterior. Esta constelación fue escogida por capricho, aunque hay que decir que minimiza la energía media del sistema[†1][†4].

De todos modos, y continuando con la ***capacidad de configuración*** de nuestro sistema, hemos implementado las ***constelaciones*** en el ***código de Matlab***, mediante ***simples tablas*** que relacionan, uno a uno, los M bloques bloques de 'k' bits con las coordenadas en fase y cuadratura de los símbolos de nuestro sistema. De este modo, cuando se quiera ***cambiar el aspecto de la constelación***, porque interese en ese

momento hacerlo, por razones de energía, por ejemplo, pues no se tiene más que cambiar los datos en la tabla del código Matlab, para cambiar la ubicación de los puntos de mensaje.

Y ahora sí, ***veamos*** el ***aspecto de las constelaciones*** de nuestro sistema, que ya mostramos cuando las comparamos con las constelaciones M-PSK. Sin embargo vamos a detallar más los dibujos, mostrando las correspondencias entre los símbolos y los bits, y reflejando la ubicación exacta de los puntos de mensaje en el plano.

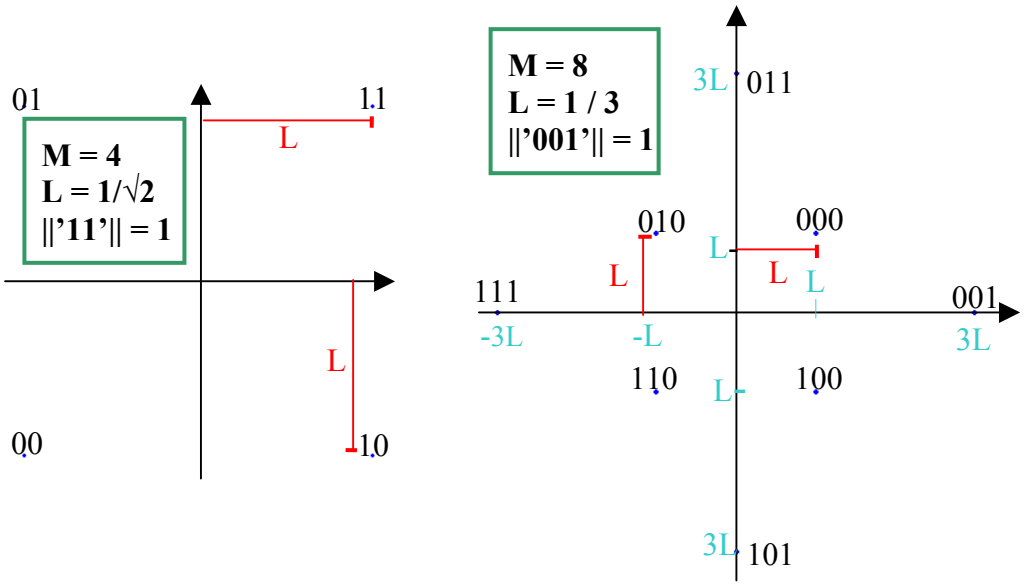


Fig. 43.- 4-APK y 8-APK

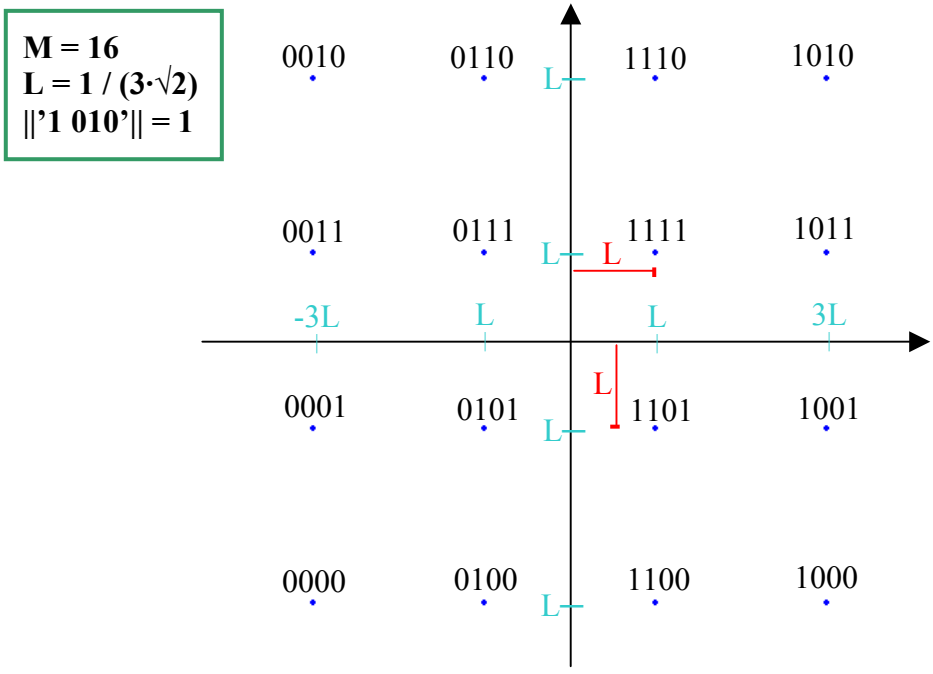


Fig. 44.- 16-APK

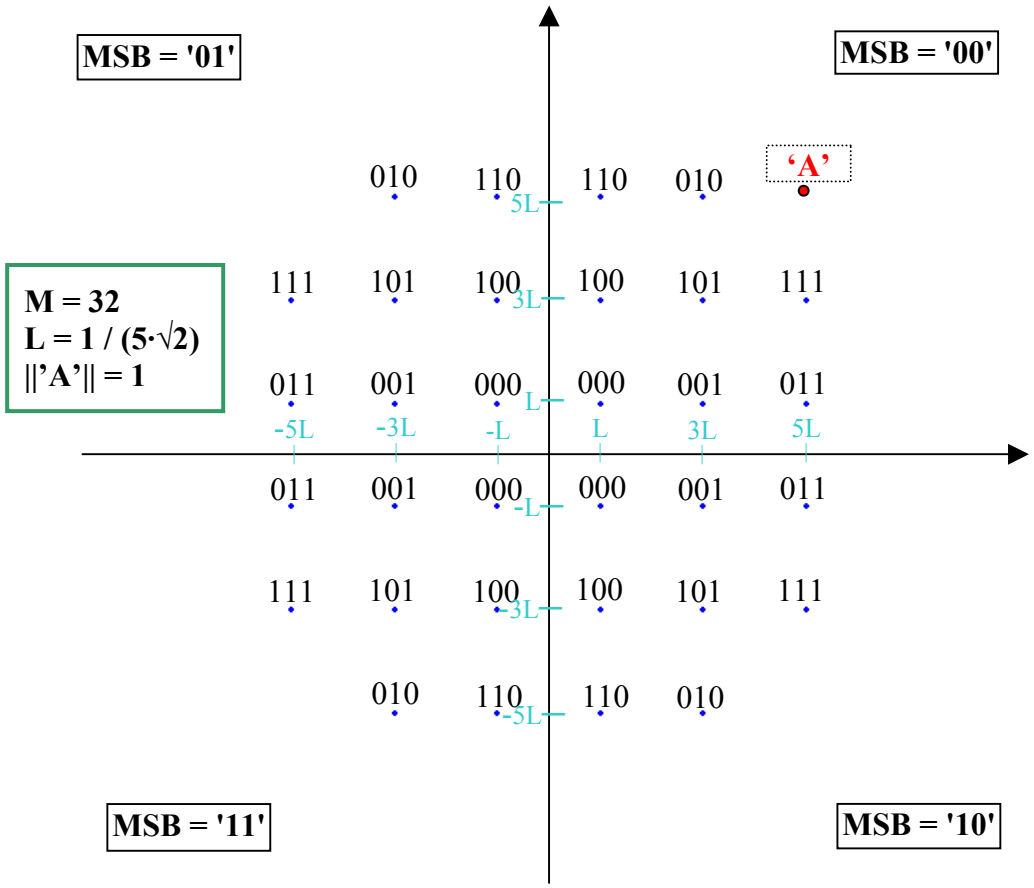


Fig. 45.- 32-APK

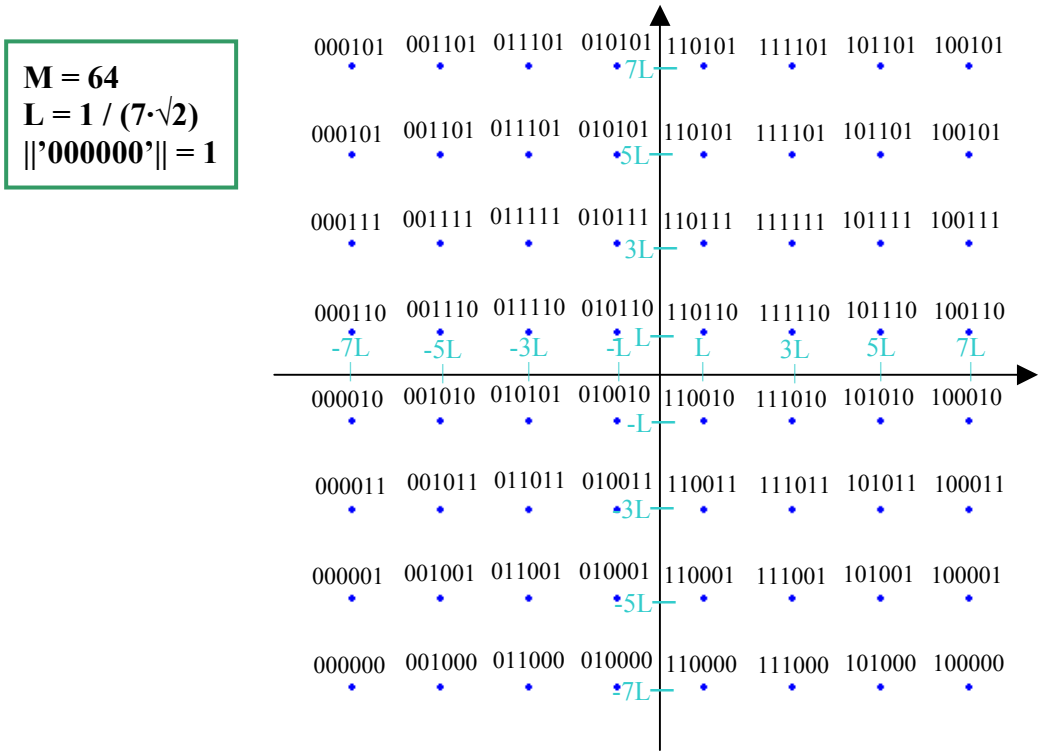


Fig. 46.- 64-APK

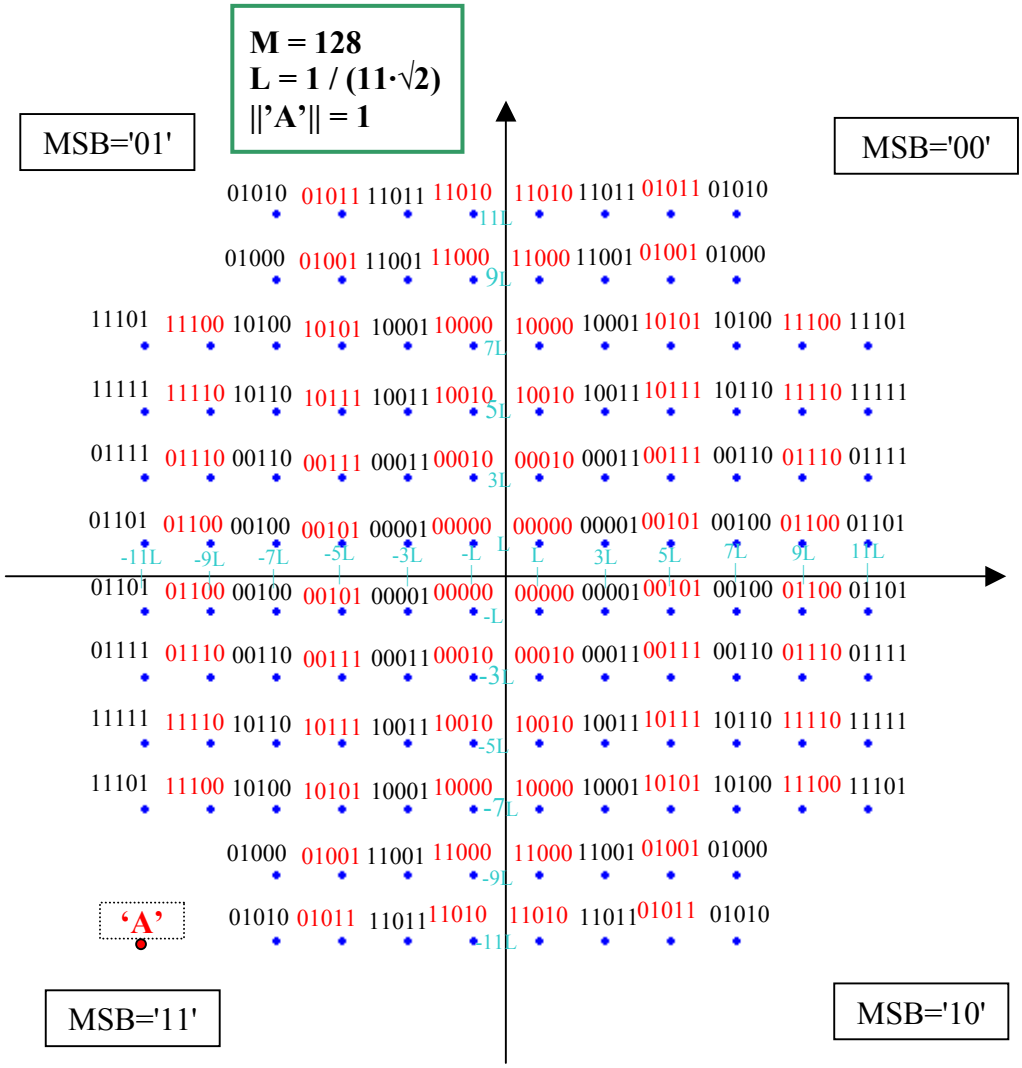


Fig. 47.- 128-APK

M = 256
L = 1 / (15·√2)
||'00000000'|| = 1

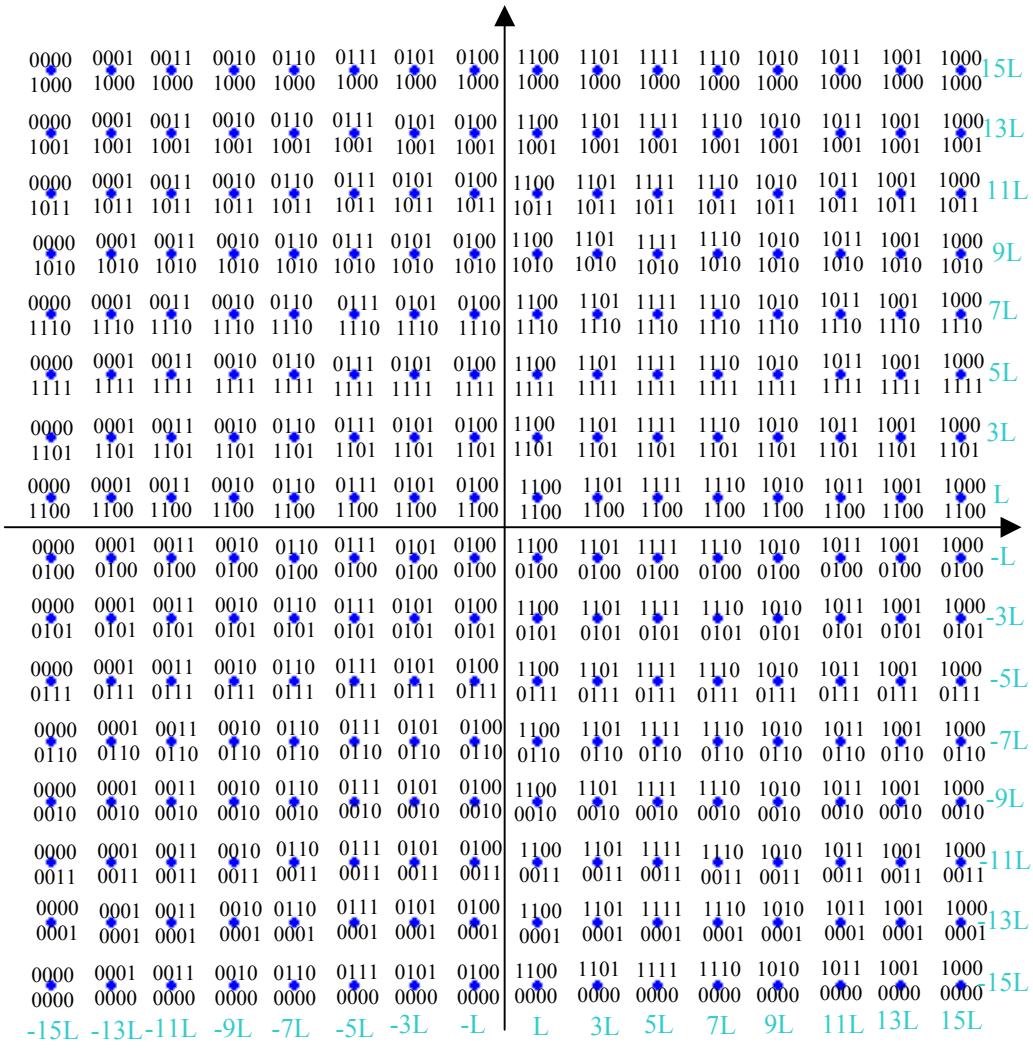


Fig. 48.- 256-APK

Lo más interesante a destacar es el uso que hacemos del ***Código Gray***, para mapear los símbolos. De esta manera procuramos que los símbolos adyacentes sólo se diferencien en un bit, ya que si un símbolo cae fuera de su región de decisión, lo más probable es que esté ubicado en la zona del símbolo adyacente, por lo que el error en la decisión del símbolo desencadenaría un ***único bit erróneo***[†4].

Por otro lado, y como ya comentamos, los ***símbolos*** están ***igualmente espaciados*** en todas nuestras constelaciones (exceptuando el caso especial de M=8), existiendo la misma distancia entre cualquier par de símbolos adyacentes. Esta ***distancia intersimbólica*** es igual a ***2·L***, como podemos apreciar en los dibujos. De igual manera

comentamos que la ***distancia de máxima energía*** también depende del nº de símbolos, M, y que en las ***constelaciones cuadradas*** es igual a la ***unidad***. Sin embargo, la ***distancia de mínima energía***, ***L·√2***, es la misma para todas las constelaciones. Asimismo es obvio, que el valor de la variable L disminuye a medida que aumenta el nº de símbolos, M, pues éstos, para todas las constelaciones (excepto para M=8), están dentro de la misma ***región cuadrada***, centrada en el origen, y de proporciones ***2/√2 x 2/√2***, para que sus diagonales (que serán las distancias de máxima energía) tengan amplitud unidad. De esta manera, si atendemos a la expresión de la señal M-APK, en amplitud y fase, para el símbolo más distanciado posible, sus valores máximo y mínimo serán uno y menos uno respectivamente.

Recordemos por un momento esa expresión analítica:

- $S_i(t) = Re\,[a_i \cdot g(t)\cdot \exp(j\cdot(w_c\cdot t+\theta_i))] = a_i \cdot g(t)\cdot\cos(w_c\cdot t+\theta_i)$
  (con i=0,1...M-1; $0 \le t \le T$)

Si el máximo valor de $a_i=1$, entonces, para ese valor de a_i (que sólo lo alcanzamos para las constelaciones cuadradas y para M = 8), la máxima amplitud que alcanzará el coseno será la de g(t). Al principio pensamos en usar para ***g(t) pulsos rectangulares***, de amplitud unidad. En este caso particular de modulación de portadora mediante pulsos rectangulares, la amplitud máxima sería la unidad, y no saturaría ninguna muestra en nuestra tarjeta de sonido. Seguidamente explicaremos que, debido a la aparición de ***interferencia intersimbólica (ISI)*** en nuestro sistema, decidimos emplear como filtro conformador un pulso ***raíz de coseno alzado***, cuya máxima amplitud ronda, aunque sin alcanzarlo, el valor 0.5, pudiendo usar un valor máximo de a_i=2, y agrandando cuatro veces más la superficie de nuestras constelaciones. Sin embargo, decidimos dejar las constelaciones tal y como las hemos dibujado anteriormente, aunque ***modificar*** el ***tamaño*** de nuestra ***constelación*** es algo tan fácil como ***multiplicar*** la ***variable L*** por el valor que deseemos, así multiplicamos por dicho valor cada lado del cuadrado que encierra la constelación. La idea de usar la variable L en el diseño de nuestras constelaciones, le da a nuestro sistema todavía más ***capacidad de modificación***.

Inicio & Fin de Trama

Justo antes de la salida del mapeo, existe una fase de colocación del ***inicio de la trama***, a nivel de símbolos. Esta operación consiste en introducir una serie de símbolos de amplitud ficticia (puesto que estos símbolos no aparecen en las constelaciones que definimos), que nos servirá para realizar un ***ajuste de niveles*** en el receptor.

Debido a la cantidad de operaciones (convoluciones, etc...) que realizamos, el rango dinámico de nuestra constelación cambiará en el receptor, necesitamos mandar ***símbolos controlados***, para comparar su valor con los símbolos recibidos, y realizar el correspondiente ajuste de niveles tanto en fase como en cuadratura.

Al principio, transmitíamos como inicio de cada trama, una ***serie de 1's y –1's*** alternados, pero, quizás por su oscilación, producían un desajuste mayor en los niveles. Luego probamos con una ***cadena de 1's*** tanto en fase como en cuadratura, y el ajuste de niveles produjo menos errores de decisión en el sistema. Posteriormente, colocamos una ristra de símbolos con amplitud en fase y cuadratura igual a ***0.4***, porque el máximo de cada símbolo, tanto en fase, como en cuadratura, era igual a ***1/√2=0.7071***, y 0.4 es la mitad prácticamente del máximo tanto en fase como en cuadratura. De esta manera intuíamos que la oscilación del último símbolo ficticio al primero de nuestra

constelación (que formará parte del campo LONGITUD de nuestra trama) sería menor, y por tanto habría menos problemas en la detección del campo LONGITUD de la trama.

Por último, mediante mediciones en nuestras pruebas, donde enviábamos solamente una ristra de 0.4's y le aplicábamos la convolución con el Filtro Transmisor, observamos que por muy larga que fuera la cadena de 0.4's que mandáramos a través del canal, siempre aparecía en los ***primeros cuatro valores*** una especie de ***transición***, y los ***últimos cuatro valores*** también variaban, mientras que el resto de valores intermedios quedaban prácticamente similares.

Así que, finalmente, ***decidimos usar 8 símbolos como Inicio de Trama***, delante del campo LONGITUD, de los cuales, empleamos los primeros 4 símbolos para absorber los efectos transitorios, y los 4 siguientes símbolos los usábamos para el ajuste de niveles. Del mismo modo, destinamos como ***Fin de Trama 4 símbolos***, que colocamos detrás del campo CRC, para amortiguar las posibles variaciones que producen las colas transitorias de la convolución con el Filtro Transmisor.

Filtro Conformador

La salida de la tabla de conversión la constituyen dos ramas en paralelo, la ***rama de fase y la de cuadratura***. A ambas se les aplica un Filtro Transmisor o conformador y una traslación en frecuencia. Como el procedimiento para ambas ramas, es el mismo, nos ocuparemos únicamente, en este apartado, de la de fase.

La señal de salida de la tabla de conversión, por la rama de fase, la constituyen un tren de deltas con amplitud la fase de cada símbolo. El ***filtro conformador*** dota a dichos impulsos de características físicas, para poder transmitirlos por el canal. Luego se trasladan en frecuencia, se forma la señal M-APK con la rama de cuadratura, y luego se transmiten. Ya en el receptor, la señal es ecualizada, luego es demodulada, y pasada a través de un Filtro Receptor, que a su vez hace de paso de baja, para recuperar los símbolos de fase.

El ***esquema*** que siguen los símbolos en fase es el siguiente[†1][†4]:

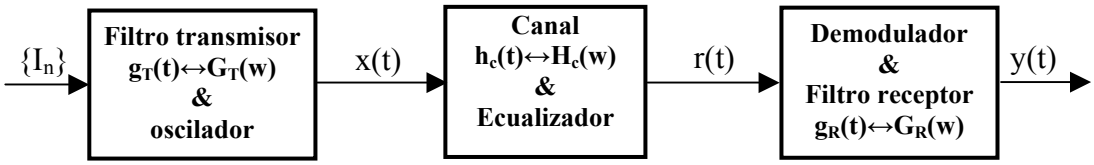


Fig. 49.- Rama de fase

La ***entrada del Filtro Transmisor*** tendrá, como sabemos, la siguiente expresión:

- $\{I_n\}=\sum_n I_n \cdot \delta(t-n\cdot T)$;

La ***salida del Filtro Transmisor***, x'(t), tendrá por tanto la expresión siguiente:

- $x'(t) = \sum_n I_n \cdot g_T(t-n\cdot T)$;

Luego será traslada en frecuencia (señal x(t)), transmitida, demodulada y, a la ***salida del Filtro Receptor***, tendremos la siguiente expresión:

- $y(t) = \mu \cdot \sum_{n} I_n \cdot p(t-n \cdot T) + n(t)$;

, donde la expresión n(t) se corresponde con el ruido del sistema, y ***μ·p(t)*** es la convolución de los bloques transmisor, canal y receptor (***μ*** es un ***factor de escala*** tal que $\mu \cdot p(0) = 1$):

- $\mu \cdot p(t) = g_T(t) * h_c(t) * g_R(t) \leftrightarrow \mu \cdot P(w) = G_T(w) \cdot H_c(w) \cdot G_R(w)$;

Si en el intervalo de símbolo $m \cdot T \leq t < (m+1) \cdot T$, hemos transmitido el símbolo $x(m \cdot T)$, entonces a la salida del Filtro Receptor tendremos:

- $y(mT) = I_m \cdot \mu \cdot p(0) + \mu \cdot \sum_{n,\, n \neq m} I_n \cdot p(mT-n \cdot T) + n(m \cdot t)$;

  - El primer sumando, $I_m \cdot \mu \cdot p(0)$, es algo proporcional al símbolo que transmitimos, y es lo que deseamos obtener.
  - El tercer sumando es la componente del ruido del sistema, que si bien lo podemos caracterizar estadísticamente, lo suponemos despreciable.
  - El segundo sumando, $\mu \cdot \sum I_n \cdot p(mT-n \cdot T)$, es algo desconocido.

Este segundo sumando es la influencia de los demás símbolos transmitidos, anteriores y posteriores a $x(m \cdot T)$, que influyen en la decisión de éste. A este fenómeno lo denominamos ***interferencia intersimbólica (ISI)***, y es uno de los problemas que tenemos que abordar en nuestro diseño[†1][†4].

Nyquist obtuvo ***dos condiciones*** que deben cumplirse para ***transmitir en ausencia de ISI***, ambas equivalentes, una para el dominio del tiempo, y otra, para el dominio de la frecuencia[†1][†4]:

- $\mu \cdot p(n \cdot T) = 1$ para $n = 0$, e igual a cero en los demás instantes de decisión de símbolo. Esta es la ***condición de Nyquist en el dominio del tiempo***, y viene a expresar que un símbolo no influirá en los demás instantes de muestreo.

- En el ***dominio de la frecuencia***, la condición equivalente es la siguiente:

  - $\sum_{n} P(f + n \cdot D) = T = 1/D$ (constante)

  Esto quiere decir, que la suma de los distintos espectros de p(t), separados cada D hz., siendo D la archiconocida tasa de símbolos, debe ser una señal completamente plana.

En principio, íbamos a usar como filtros transmisor y receptor, ***pulsos rectangulares***, pero trataremos de evidenciar que no cumple las condiciones de Nyquist, para transmitir en ausencia de ***ISI***. Para ello, recurriremos al dominio de la frecuencia:

- Si $g_T(t)$ y $g_R(t)$ son ***filtros rectangulares*** de duración T= 1/D. Las transformadas correspondientes, $G_T(w)$ y $G_R(w)$, son ***funciones sampling***, desplazadas por el modulador a la frecuencia de 11025 hz.

- Los ***cruces por cero*** de las funciones sampling estarán en los puntos ***n·D***, para n = 0, ±1, ± 2,...Para una tasa de símbolos D=4410, y una anchura de canal de 20 khz., sólo pasarán el primer lóbulo secundario a cada lado del principal.
- Si suponemos además que el ecualizador elimina la distorsión en amplitud del canal, podemos concluir que $|H_c(w)| = 1$. Por lo que el canal sólo produce un recorte de las funciones $G_T(w)$ y $G_R(w)$.
- Por lo tanto, P(w) tendría el aspecto del producto de dos funciones sampling recortadas por la limitación en banda del canal, y desplazadas por el oscilador a 11025 hz:

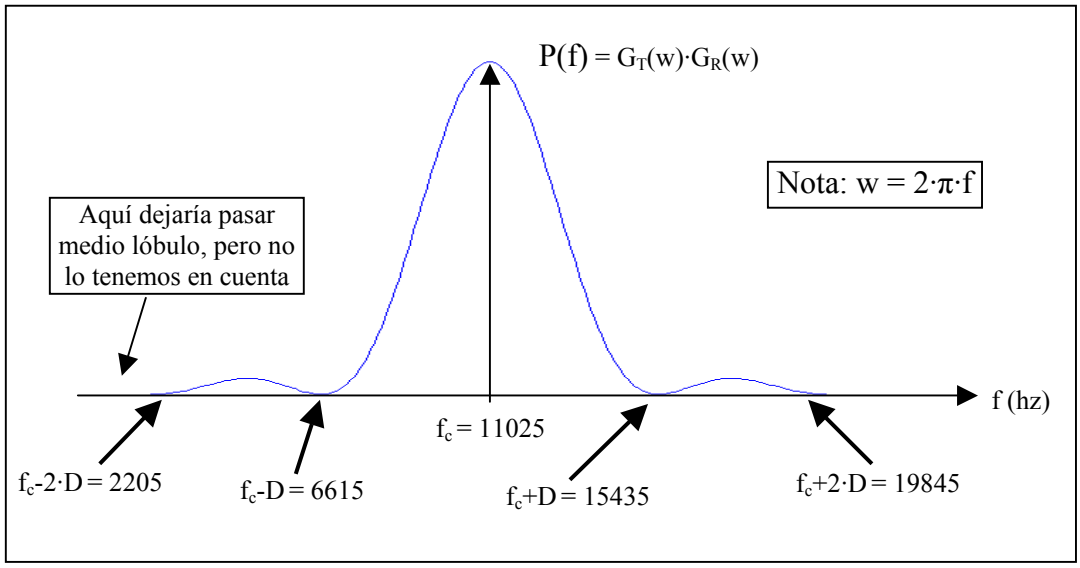


Fig. 50.- Func. Sampling limitada por el canal

- Atendiendo al criterio de Nyquist en la frecuencia, la expresión correspondiente tendrá el siguiente aspecto. Como es obvio, la suma de dichas funciones no dan una constante, por lo que ***aparecerá interferencia entre símbolos***. Así concluimos que ***descartamos el uso de funciones rectangulares, como filtros transmisor y receptor.***

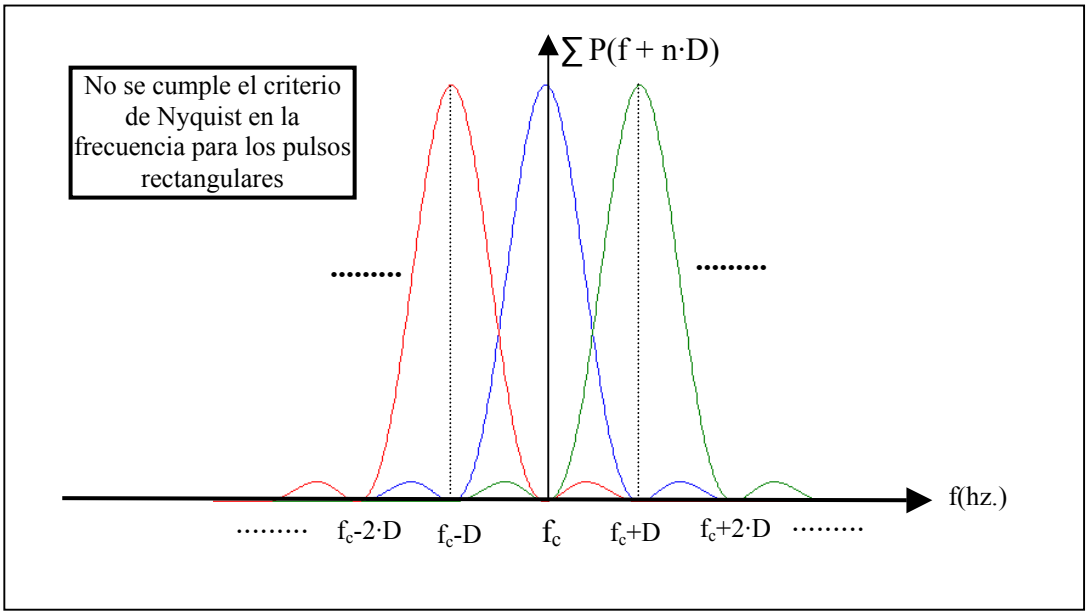


Fig. 51.- Caso de filtros rectangulares

Tendremos que recurrir, por tanto, a pulsos p(t) que cumplan ambos criterios de Nyquist. Durante la carrera hemos visto una familia de pulsos que cumplen dichos criterios: la ***familia de pulsos de coseno alzado***[†1][†4]. Optamos entonces por la idea de usar esta conocida familia de pulsos para modelar el bloque ***P(w)***. Concretamente, si suponemos que el conjunto ***canal & ecualizador*** es un ***bloque ideal***, esto es:

- $|H_c(f)| = 1$;

De este modo, ***P(w)*** queda reducido a la siguiente expresión de antes:

- $P(f) = G_T(f) \cdot G_R(f)$;

Ya veremos que para maximizar la relación señal a ruido en el receptor, usaremos como Filtro Receptor un ***filtro matcheado***, $G_R(f)$ [†1][†4], respecto a $G_T(f)$. En cualquier libro de texto de teoría digital se puede demostrar que la ***relación entre*** $G_T(f)$ ***y*** su correspondiente filtro matcheado, $G_R(f)$, será:

- $G_R(f) = [G_T(f)]^* \cdot e^{-j \cdot 2 \cdot \pi \cdot f \cdot T} \rightarrow |G_R(f)| = |G_T(f)|$; (T: tiempo de símbolo)

Por este motivo, se suele usar la siguiente relación entre P(w), $G_R(f)$ y $G_T(f)$:

- $\sqrt{|P(w)|} = |G_R(f)| = |G_T(f)|$

Así, usaremos como ***filtro conformador***, $G_T(w)$, un ***pulso raíz de coseno alzado***, cuya expresión en la frecuencia es la siguiente[†1] [†4]:

$$G_T(f) = \begin{cases} 1 , & |f| < f_N \cdot (1-\alpha) \\ \{1/2+1/2 \cdot \text{sen}\,[\,(\pi \cdot / (2 \cdot f_N \cdot \alpha)) \cdot (f_N - |f|)\,]\}^{1/2} , & f_N \cdot (1-\alpha) \le |f| \le f_N \cdot (1+\alpha) \\ 0 , & |f| > f_N \cdot (1+\alpha) \end{cases}$$

En recepción tendremos, como hemos comentado, otro filtro en módulo igual al transmisor, por lo que los pulsos, ***P(w)***, que obtenemos finalmente, tendrán forma de ***coseno alzado***, cuya ***respuesta en frecuencia*** será la siguiente[†1] [†4]:

$$P(f) = \begin{cases} 1/(2 \cdot f_N) , & |f| < f_N \cdot (1-\alpha) \\ (1/(4 \cdot f_N)) \cdot \{1 + \cos[(\pi \cdot / (2 \cdot f_N \cdot \alpha)) \cdot (|f| - f_N \cdot (1-\alpha))]\}, & f_N \cdot (1-\alpha) \le |f| \le f_N \cdot (1+\alpha) \\ 0 , & |f| > f_N \cdot (1+\alpha) \end{cases}$$

La correspondiente ***respuesta impulsiva*** temporal del filtro, ***p(t)***, será:

- $p(t) = \text{sinc}(2 \cdot f_N \cdot t) \cdot \cos(2 \cdot \pi \cdot \alpha \cdot f_N \cdot t) / (1 - 16 \cdot \alpha^2 f_N^2 \cdot t^2)$

(***Nota*** : en la fórmula anterior, f_N, es la ***frecuencia de Nyquist***, y es igual a ***D/2***).

La figura siguiente muestra la ***forma temporal*** y ***frecuencial*** del ***pulso en coseno alzado***, para distintos valores de su factor de Roll-off, α:

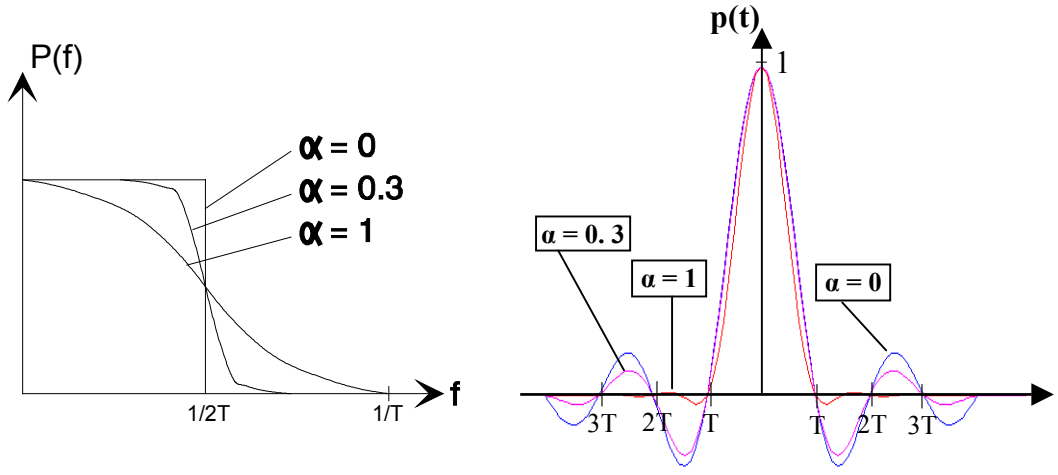


Fig. 52.- Pulso en coseno alzado

Si atendemos a la ***respuesta en frecuencia*** del filtro, ***P(f)***, podemos sacar una serie de conclusiones al respecto:

- El ***ancho de banda*** de este filtro, en ***banda base***, es ***limitado*** e igual a $f_N \cdot (1+\alpha)$, y varía en función del ***factor de Roll-off***, ***α***, cuyo ***rango*** va de ***0 a 1***, y representa el porcentaje (en tantos por 1) respecto a la frecuencia de Nyquist, que se ensancha el filtro.
- Así, el mínimo ancho de banda se da para $\alpha = 0$, y se trata de un filtro rectangular denominado ***pulso de Nyquist***. Este particular filtro tiene la mínima anchura en frecuencia a la que se puede transmitir en ausencia de ISI. Esta frecuencia mínima es denominada ***frecuencia de Nyquist***, f_N, y es igual a la mitad de la tasa de símbolos del sistema. Pero este filtro, como todos intuimos, es ideal, ***irrealizable***.
- Así, aumentando el valor de alfa para buscar ***soluciones prácticas*** o realizables, vamos incrementando el ancho de banda de nuestro sistema.
- El máximo ancho de banda para esta familia de pulsos se da para $\alpha = 1$. Para este valor de α obtenemos el ***pulso de coseno alzado propiamente dicho***. Para este caso, la anchura de nuestro sistema equivalente en paso de baja será igual a la tasa de símbolos (el doble para nuestro sistema paso de banda).

Para dotar a nuestro sistema de ***mayor versatilidad***, ***α*** será ***modificable*** mediante vía software, y por tanto, nuestro sistema poseerá un ***ancho de banda*** con una amplia gama de posibles valores, que abarcarán el siguiente ***rango***, en ***banda base***:

- $D/2 \leq WB \leq D$

Al tratarse de una señal trasladada en frecuencia, el ancho de banda se duplica, por lo que el ***verdadero rango*** de nuestro ancho de banda, centrado en ***11025 hz***.,será:

- $D \leq WB \leq 2 \cdot D$

Concretamente, si nos remitimos a nuestro ***ejemplo*** habitual de ***N=10*** muestras por símbolo y ***D= 4410*** símbolos por segundo, los ***límites*** de nuestro ***ancho de banda*** podrán variar dependiendo del valor del factor de Roll-off, α, de la siguiente manera:

	límite inferior del WB: $f_c - f_N(1+\alpha)$	límite superior del WB: $f_c + f_N(1+\alpha)$
α=0	11025 – 2205 = 8820 hz.	11025 + 2205 = 13230 hz.
α=1	11025 – 4410 = 6615 hz.	11025 + 4410 = 15435 hz.

Cuyos valores quedan dentro del rango de frecuencias del canal.

Habría que aclarar que la ***cota superior*** de nuestro ancho de banda no está realmente en D hz.: podríamos seguir aumentando el ancho de banda, siempre que se cumpliera la condición de Nyquist en la frecuencia. Pero el ancho de banda, como sabemos, es un parámetro que nos interesa disminuir. Además, correríamos el riesgo de salirnos de los límites del canal y perder parte de nuestro espectro de potencia.

Respecto a la ***figura*** de la ***respuesta impulsional*** del pulso coseno alzado, es interesante apreciar que se cumple la condición de Nyquist en el tiempo[†1][†4], ya que su amplitud es **1** en $\mathbf{t = 0}$ y **0** en $\mathbf{t = nT}$, con $\mathbf{n \neq 0}$, evitando así la interferencia intersimbólica.

Para acabar con el tema del pulso conformador, añadiremos la manera en que implementaremos, vía software, dicho proceso de filtrado. Para realizar la convolución

de nuestra señal de entrada (cadena de símbolos tanto en la rama de fase como en la de cuadratura) con nuestro filtro conformador, usaremos la función *'rcosflt'* de ***Matlab***. Los ***parámetros de entrada*** de dicha función nos permitirán elegir, además de la señal de entrada al filtro, el ***factor de Roll-off*** usado por el filtro raíz de coseno alzado, el número de muestras por símbolo (de tal manera que existirá ***sobremuestreo***, esto es, a la entrada del filtro existirá una tasa de ***D símbolos/segundo***, y a la salida una tasa muestreo igual a ***Fs muestras/segundo***, ya que por cada símbolo de entrada, aparecerán N muestras a la salida) y el ***retraso de grupo*** del sistema.

Este último parámetro producirá unas ***colas*** en nuestra señal que tenemos que tener en cuenta a la hora de detectar los símbolos. Hemos optado por el ***retraso*** por defecto de ***3 símbolos***, esto es, existirá una cola de ***3·N muestras*** hasta la muestra correspondiente al instante de decisión del primer símbolo de nuestra trama. De igual forma, existirá una cola de 3·N muestras posterior a la última muestra del último símbolo. El retraso de grupo será un ***valor fijo*** en nuestro sistema.

Para que veamos un ***ejemplo***, vamos a pasar por nuestro filtro conformador un símbolo de amplitud unidad, con un sobremuestreo de 10 muestras/símbolo, y un factor de Roll-off igual a 0.7:

- Proceso de filtrado de un símbolo unidad a través del pulso raíz de coseno alzado:

```
% x: señal de entrada
   x=1;
% y: señal de salida
   y=rcosflt(x,1,10,'sqrt',0.7,3);
   plot(y,'-o');
```

A continuación mostramos la forma que adquiere el símbolo a la salida del filtro, y rellenamos de color rojo las muestras que marcan los intantes de decisión de cada símbolo. Como se puede apreciar, existe una distancia de 10 muestras entre cada par de instantes de símbolos. Además, se puede apreciar con bastante claridad las colas de 30 muestras correspondientes al retraso de 3 intervalos de símbolo, a cada lado del instante de símbolo que hemos generado.

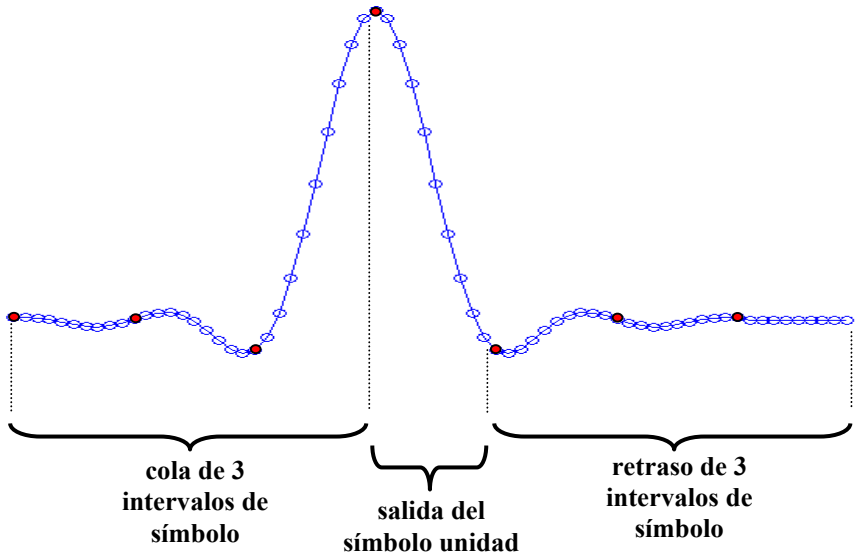


Fig. 53.- Delta a salida del Filtro Transmisor

Oscilador

A la ***salida del Filtro Transmisor***, tras la convolución de nuestra secuencia de deltas de dirac con el pulso conformador, tendremos las siguientes señales, tanto en la ***rama*** de ***fase***, $S_I(t)$, y como en la de ***cuadratura***, $S_Q(t)$:

- rama de fase: $S_I(t) = \sum_n I_n \cdot g_T(t-n\cdot T)$;
- rama de cuadratura: $S_Q(t) = \sum_n Q_n \cdot g_T(t-n\cdot T)$;

Si recordamos por un momento la ***expresión*** de una señal ***M-QAM***, para uno de sus posibles ***símbolos***[†1][†4]:

$$S_i(t) = \underbrace{Ii\cdot g(t)\cdot\cos(w_c\cdot t)}_{\textit{M-ASK en fase}} - \underbrace{Q_i\cdot g(t)\cdot sen(w_c\cdot t)}_{\textit{M-ASK en cuadratura}} \text{ con } i=0,1...M-1;\quad 0 \le t \le T$$

, observaremos que sólo necesitamos multiplicar la rama de fase por un coseno, y la rama de cuadratura por un seno. Dicho de otra forma, sólo nos queda trasladar a la frecuencia de fc=11025 hz. nuestra señal en banda base, y para ello, lógicamente, acudimos a estas funciones sinusoidales, que tienen la propiedad de trasladar en frecuencia a las señales que multiplican.

Nuestro ***oscilador local*** será precisamente un coseno a la frecuencia de 11025 hz. El ***desfasador de π/2*** radianes es precisamente para convertir dicho coseno a un seno a la misma frecuencia y poder aplicarlo para modular la rama de cuadratura.

Finalmente, para completar la expresión de la ***señal M-APK*** en el dominio del tiempo, nos queda restar la rama de cuadratura a la rama de fase. Así, la señal que vamos a transmitir tendrá la siguiente expresión:

- $S_{APK}(t) = S_I(t)\cdot\cos(w_c\cdot t) - S_Q(t)\cdot sen(w_c\cdot t)$

, siendo $S_I(t)$ y $S_Q(t)$ las señales ya mostradas en banda base para las ***ramas*** de ***fase*** y ***cuadratura*** respectivamente a la salida del filtro conformador:

- $S_I(t) = \sum_n I_n \cdot g_T(t-n\cdot T)$;
- $S_Q(t) = \sum_n Q_n \cdot g_T(t-n\cdot T)$;

Visto de otra forma, si rescatamos la ***relación*** entre la señal ***paso de banda*** y la señal ***paso de baja*** equivalente, podemos expresar la ***señal M-APK*** de la siguiente forma:

- $S_{APK}(t) = Re\,[\check{s}(t)\cdot\exp(j\cdot w_c\cdot t)]$;

, es inmediato entrever que la ***envolvente compleja*** de nuestra señal ***M-APK*** se puede descomponer , como es obvio, en:

- $\check{s}(t) = S_I(t) + j \cdot S_Q(t)$;

La ***frecuencia*** de ***portadora***, en la cual centramos el espectro de nuestra señal, la elegimos igual a 11025 hz. por dos razones fundamentales:

- Por que se trata de un ***valor cercano a*** la frecuencia central de nuestro canal, que vienen a ser aproximadamente ***10 khz***.
- Porque las ***frecuencias de muestreo*** más altas de nuestra tarjeta (11025 hz., 22050 hz., 44100 hz.) son ***múltiplos de nuestra frecuencia de portadora***. Lo que significa que nuestro coseno digital es una señal periódica, con un mismo número de muestras en cada período de portadora. Por ejemplo, para el caso Fs = 44100 hz, existirán 4 muestras por cada período de portadora. Véamoslo analíticamente:

  - $f_c = (1/4) \cdot Fs \rightarrow T_c = 4 \cdot Ts$ ($T_c = 1/f_c$: período de portadora)

Por último, cabría comentar que para la implementación vía Matlab de nuestro oscilador, usaremos sus funciones *'cos'* y *'sen'*.

A continuación pasamos a mostrar un ejemplo práctico para afianzar visualmente los conceptos aquí explicados.

Ejemplo práctico

Veamos, a modo de ***ejemplo práctico***, la evolución que seguirían una serie de símbolos al pasar por ambas ramas (fase y cuadratura). Esto es, vamos a tomar la primera columna de nuestra constelación para ***M=256*** símbolos y, obviando los procesos previos de fuente digital, fases de entramado, y mapeo (puesto que no los necesitamos, y además los hemos ya explicado gráficamente), vamos a hacerlos pasar por el ***filtro conformador*** y el ***oscilador***, para contemplar la forma temporal de las señales en cada caso. El ***esquema*** será el que ya conocemos:

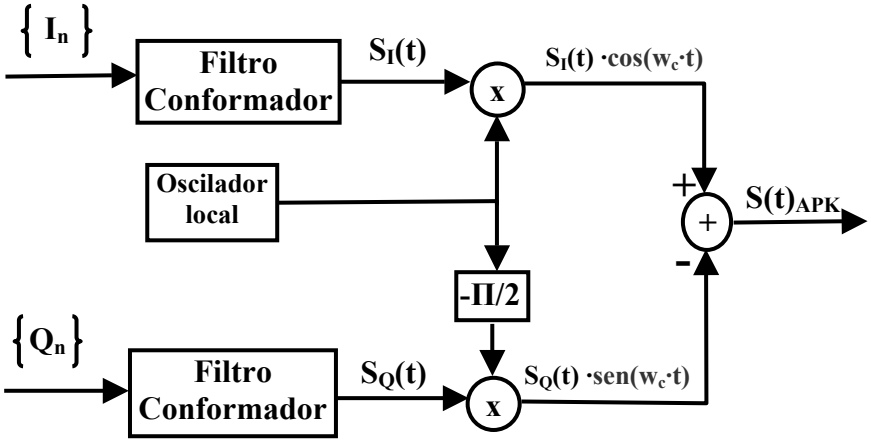


Fig. 54.- Ramas de fase y cuadratura

Los ***símbolos*** que representaremos serán los indicados a continuación:

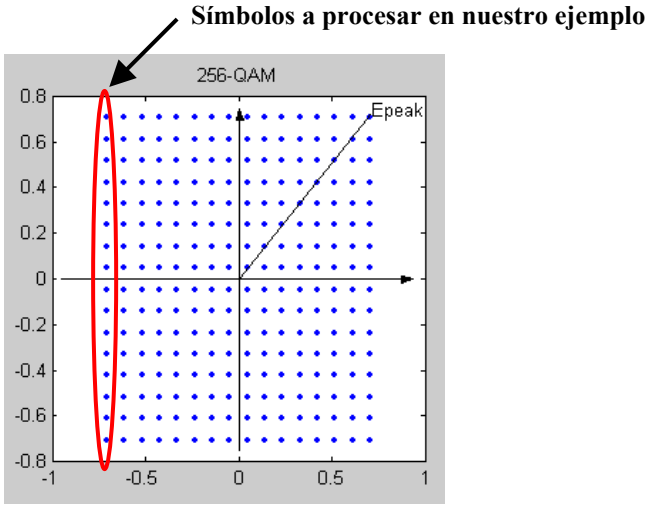


Fig. 55.- Símbolos del Ejemplo

Los ***símbolos*** tendrán la misma ***coordenada de fase***, y la ***coordenada de cuadratura*** irá aumentando para cada símbolo, porque iremos representando los símbolos desde la parte inferior hasta la superior.

A la salida del pulso conformador tendremos las siguientes señales, $S_I(t)$ y $S_Q(t)$:

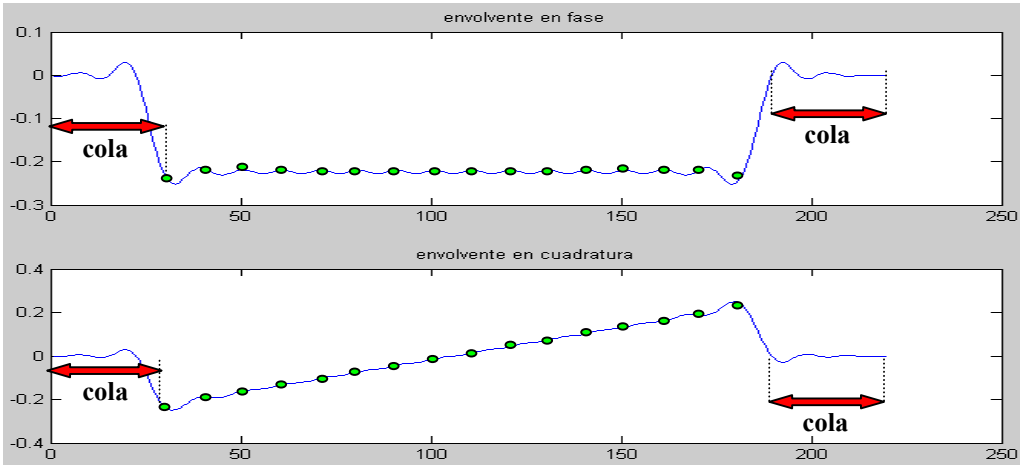


Fig. 56.- Envolventes del Ejemplo

Para la ocasión, hemos optado por tomar ***N=10*** muestras por cada símbolo, un valor de ***Roll-off*** de ***0.7***, y un ***retraso de grupo*** de ***3 intervalos de símbolo*** (que ya mencionamos que se trataba de un valor fijo), es decir, de ***30 muestras***. Dichas ***colas*** se pueden apreciar claramente al principio y al final de la señal. A partir de las colas empiezan a aparecer las muestras correspondientes a los símbolos a procesar en nuestro ejemplo. Indicamos cada instante de decisión de símbolo con puntos verdes.

Se puede apreciar que, aproximadamente y como ya dijimos, los cuatro símbolos más cercanos a cada cola todavía no están estabilizados, y su valor no es fiable. En un caso completo esta especie de ***transición*** afectará sólo a los símbolos de los campos Inicio y Fin de Trama, que por supuesto, no forman parte de nuestra información propiamente dicha y además tampoco los usaremos para ajustar los niveles de nuestro sistema.

En la ***envolvente de fase***, ***$S_I(t)$,*** podemos comprobar, como era de esperar, que tenemos un mismo valor para cada símbolo, precisamente $1/\sqrt{2}$. Por otro lado, en la ***envolvente de cuadratura***, el valor va aumentando para cada símbolo, entre $-1/\sqrt{2}$ y $+1/\sqrt{2}$. Si bien no aparecen estos valores es porque, como ya comentamos, el ***filtro raíz de coseno alzado*** atenúa dichos valores.

Tras pasar por el oscilador, la señal modulada tendrá el aspecto sinusoidal que le mostramos a continuación. Lo único a comentar es que, lógicamente, cada una de estas señales es una señal modulada en amplitud (ASK), a la frecuencia de portadora de 11025 hz., y la envolvente de dichas señales son las representadas inmediatamente antes. Hemos demostrado de nuevo que la señal ***APK resultante*** está formada por ***dos señales ASK en cuadratura***.

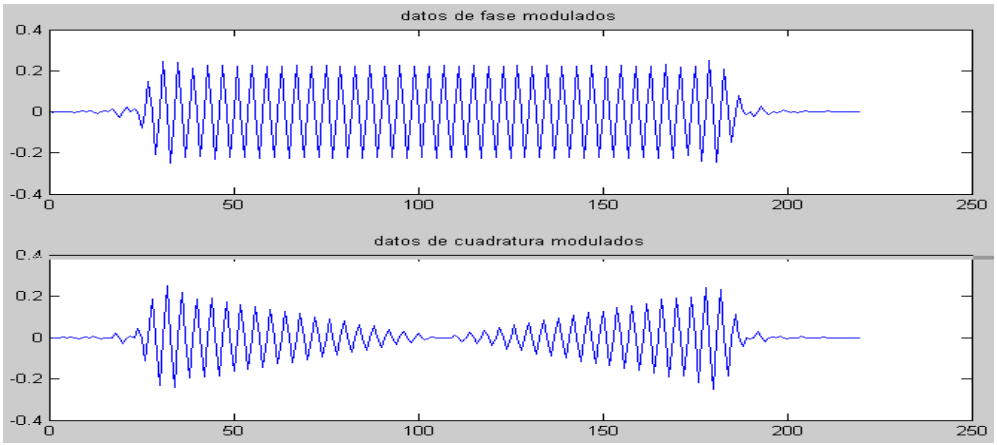


Fig. 57.- Datos modulados del Ejamplo

En último lugar pasamos a mostrarles la ***señal APK*** correspondiente a esta secuencia simbólica, que no es más que la resta entre la señal de fase y la de cuadratura moduladas:

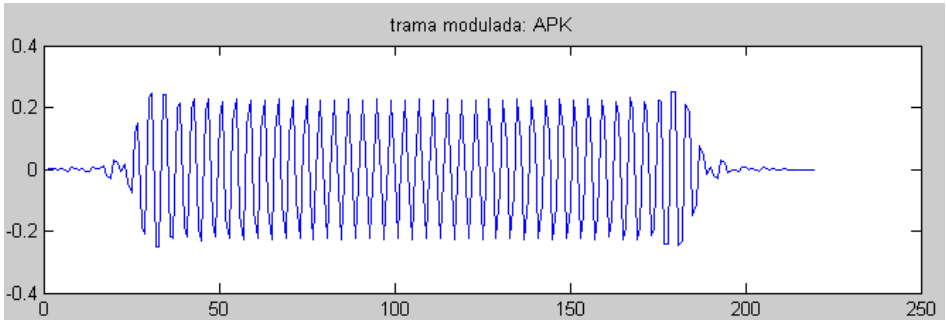


Fig. 58.- Señal APK del Ejemplo

Es interesante comentar que, al escoger como ***frecuencia de portadora*** exactamente la ***cuarta parte*** de la ***frecuencia de muestreo***, cada muestra de la señal digital correspondiente aparecerá en los ***instantes n·π/2***. Por lo que cada muestra de la señal APK, corresponderá a la señal de fase o a la de cuadratura cambiada de signo, pues en dichos instantes, una de las dos señales es igual a cero, ya que no olvidemos que ambas portadoras están desfasadas π/2 radianes.

- Fase de Entramado II

Este apartado es ***consecuencia de la separación en tramas*** de nuestra información para disminuir los efectos de la ***falta de sincronismo***, y deriva, por tanto, de la necesidad de identificar cada una de las tramas en el receptor y poderlas ***ecualizar de manera óptima***.

Además, también es resultado de la ***dificultad para trasmitir y recibir las tramas en tiempo real***, es decir, una a una, conforme se vayan formando. En principio, intentamos esta opción, pero nos resultó misión imposible debido a las limitaciones de nuestro hardware. Las ***razones*** que motivaron el ***descarte*** de la idea de trabajar en ***tiempo real*** para solapar los tiempos de procesamiento de ambos bloques y aumentar así la rapidez de nuestro sistema fueron las siguientes:

- Necesitamos ***abrir dos sesiones de Matlab***:
  - una sesión para procesar las tramas y transmitirlas.
  - otra sesión para recibir y procesar las tramas.

  Es imposible, por tanto, transmitir y recibir, en la misma sesión de Matlab.
- Utilizamos el comando 'clock', para ***sincronizar ambas sesiones*** en un tiempo horario absoluto, y a partir de ahí, ***transmitir y recibir cada mismo intervalo de tiempo***(empezaríamos unas décimas de segundos antes en el receptor, para evitar la pérdida de datos transmitidos).
- Pero no contábamos con que el ordenador no atiende simultáneamente a todos los procesos, y no se transmite en los instantes deseados. Es prácticamente imposible controlar los ***tiempos de CPU*** del ordenador. Para el caso de comunicación entre dos ordenadores, quizás pudiéramos conseguir algunos resultados, pero siempre hay que tener en cuenta que el ordenador no hace todas las tareas simultáneamente, y siempre tendría procesos a los que atender, y por lo tanto, tampoco podría realizar la transmisión justo en los instantes programados por nosotros.
- Además, para asegurarnos de que el transmisor y receptor habían realizado el procesamiento de trama, antes de volver a estar dispuestos a transmitir o recibir otra trama, el intervalo entre cada transmisión o recepción debía ser de más de un segundo, con lo que la media de transmisión era superior a un segundo por trama, evitando así fallos en la comunicación, asegurando que ambos bloques estuvieran preparados para transmitir o recibir datos. De esta manera ***no aprovechábamos*** las ventajas en ***velocidad*** de la señalización ***M_QAM***, pues dicha ventaja la absorvía el ***largo tiempo de espera***.

Por estas razones decidimos ***unir todas las tramas*** en transmisión y ***transmitirlas de una vez***. Pero el hecho de dividir la información en tramas, y usar un ecualizador distinto en cada intervalo de trama, simula la transmisión de cada trama por separado, si bien no conseguimos minimizar el tiempo, simultaneando procesamientos tanto en el receptor como en el transmisor.

La fisonomía que presenta nuestra ***Fase de entramado II*** es la siguiente:

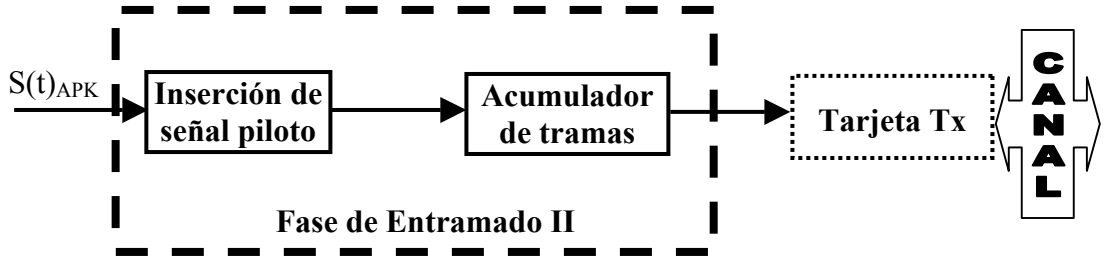


Fig. 59.- Fase de Entramado II

En la primera etapa, se inserta una ***señal piloto*** al comienzo de la trama. Esta señal consiste en una secuencia de 30 muestras nulas, seguida de un pulso de dirac, y otra secuencia de cincuenta ceros. Con esta operación pretendemos conseguir las siguientes ***ventajas***:

- ***Separación entre las diversas tramas***. De esta manera podemos identificar cada trama en el receptor, si conseguimos engancharnos a esta señal piloto. Se trata, por tanto, de un método de ***sincronización de trama***, esto es, cada señal piloto marca la sincronización de la trama que le sucede.
- Esta delta de dirac de la señal piloto la utilizaremos para diseñar nuestro ***ecualizador de cero forzado***. Dicho impulso de dirac, al pasar por el canal, se convertirá a la salida en la respuesta impulsiva de dicho canal, justo unos instantes antes de cada trama. Por tanto, obtendremos un ***ecualizador dinámico***, es decir, éste irá variando en función de la respuesta impulsiva, que será distinta en cada trama, debido a la falta de sincronización. Así obtendremos un ***igualador óptimo*** para cada trama en particular, a partir del impulso que le precede.
  De esta manera, conseguimos un ***canal distinto para cada trama***, y nuestra situación adquiere el mismo efecto que si transmitiéramos las tramas independientemente (con la dificultad que ya hemos comentado que ello hubiera conllevado).

Por otra parte, el ***acumulador de tramas***, es una especie de almacén en el que vamos almacenando las tramas una a una (separadas por su correspondiente señal piloto), para posteriormente lanzarlas a través de la línea de salida de la tarjeta, mediante la orden de Matlab '*wavplay'*.

- Convertidor digital/analógico

Se trata, como ya dijimos de un dispositivo ubicado en el puerto LINE OUT de la ***tarjeta de sonido***, y aunque es un ***componente hardware***, cuyo modo de funcionamiento es totalmente transparente a nuestro proyecto, sí podemos modificar ciertos parámetros de éste, que influyen de manera determinante en la velocidad y resolución de dicho proyecto[†6]:

- Es aquí donde, mediante nuestra ***variable Fs***, indicamos la ***velocidad de muestreo***, esto es, la rapidez a la que vamos a ir convirtiendo las muestras digitales del ordenador a valores analógicos. [†7] [†8]
  Aunque en el programa hemos dejado una puerta abierta a la elección del usuario (pues se pide a dicho usuario final el valor de la frecuencia de muestreo, con lo que lo convertimos en un ***parámetro totalmente configurable***), el valor de nuestra frecuencia de muestreo debe estar siempre dentro del ***rango teórico de Fs***, que establecimos entre los límites ***27563 hz. < Fs < 69750 hz.*** , durante el dimensionamiento que hicimos del sistema en el apartado III.2.2, para un número de muestras por símbolos típico, ***N = 10***.
  Teniendo en cuenta que las frecuencias de muestreo estándares para los dispositivos de audio para PC son ***8000, 11025, 22050 y 44100 hz.*** [†6], no sólo ***estamos usando la Fs máxima posible***, sino que la única viable para que no existan problemas de solapamiento o "aliasing" o de limitación del canal.
- Por otra parte, también vamos a indicar, mediante la ***variable NB***, el número de bits que van representar la ***resolución*** de nuestro sistema[†7] [†8], es decir, cuanto más grande sea dicho número de bits, más niveles de decisión podremos codificar en nuestro ***rango digital*** de entrada (que como sabemos sobradamente va de ***–1 a 1***)[†6], más pequeño, por tanto será el paso de cuantización o escalón o distancia entre dichos niveles de decisión, y más parecido existirá entre la señal analógica reconstruida y la señal digital que tenemos en el ordenador en forma de muestras (pues, cabe recalcar, que las muestras pertenecientes a nuestras señales las representamos en nuestro ordenador mediante cadena de unos y ceros, aunque nos dé la impresión de estar usando valores analógicos). Esta resolución generalmente será de ***16 bits***, aunque para hacer nuestro programa compatible con tarjetas antiguas que no admitan esta resolución, podríamos hacer que se admitiera también el valor de 8 bits por muestra. El ***único inconveniente*** es que, por defecto, la función que utilizamos para mandar las muestras determina, en función del formato de los símbolos, un tipo de resolución determinada. Así, para nuestros datos, que son del ***tipo 'double'***, utilizará ***16 bits por muestra***[†6].
  Para forzar a nuestro sistema que utilice una precisión de 8 bits por muestra, tendríamos que convertir nuestros datos a ***'uint'***, es decir , a entero sin signo, mediante la ***orden homónima 'uint'***[†6]. El cambio de formato iría previo a la función ***wavplay***, y no lo hemos implementado. Sólo dejamos la solución por si se quiere transmitir a ***8 bits/muestras***.

La transmisión de la señal a través de la tarjeta de sonido se implementa mediante la función de Matlab '*wavplay'*, donde le pasamos como parámetros de entrada la frecuencia de muestreo y la resolución de bits, además de las muestras de la señal a transmitir.

De esta manera, cada trama a transmitir, delimitada por las señales piloto, constará de los siguientes bloques bien diferenciados:

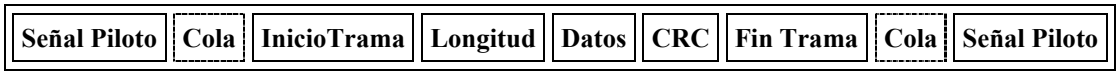


Fig. 60.- Campos de Trama

Hemos punteado el campo denominado COLA, para resaltar que no se trata realmente de un campo colocado intencionadamente por nosotros, sino que es fruto, como sabemos, de la convolución de nuestra señal con el filtro raíz de coseno alzado, pero lo hemos incluido para que lo tengamos en cuenta al procesar la señal en el BLOQUE RECEPTOR.

Ejemplo práctico

Tomemos un fichero llamado 'prueba.txt', y hagámoslo pasar por nuestro ***Bloque Transmisor***, hasta el instante previo a la transmisión de la trama. Con la ayuda de nuestro editor de sonido 'Cool Edit', veamos el verdadero aspecto de la señal que hemos creado, a partir de los datos leídos del fichero. Luego, mediante un *'zoom',* nos centraremos en cualquier ***trama***, para analizar las diversas partes que la componen, y que ya hemos visto de manera teórica en este apartado.

Los parámetros del sistema que hemos usado son los siguientes (aunque algunos son fijos y los conocemos sobradamente):

- F_S = 44100 muestras / sg.
- NB = 16 bits de precisión.
- M = 256 símbolos.
- N = 10 muestras / símbolo.
- Tamaño del fichero = 1000 bytes.
- BDT = 20 bytes de datos / trama. (tendremos 50 tramas).
- Roll-off = 0.7
- retardo de grupo = 3 símbolos (30 muestras).

Hemos ***dividido*** la información en ***pequeñas cantidades*** de bytes con la intención de poder visualizar mejor ciertas partes de la trama, como las colas, ya que, si hubiéramos tomado por ejemplo 100 bytes de datos para cada trama, tendrían un total de 1301 muestras en la trama, con lo que dichos retardos apenas se apreciarían.

Una vez realizados todos los procesos estudiados en el transmisor, el ***aspecto global*** que presentaría la ***señal*** justo antes de ser transmitida sería el que veremos a continuación:

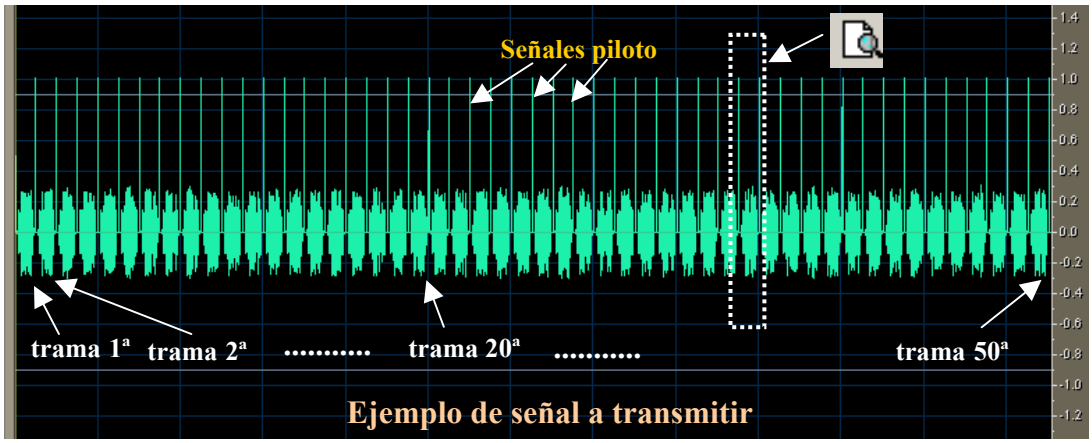


Fig. 61.- Ejemplo de Señal a transmitir

Realicemos un 'zoom' a la zona delimitada por las líneas punteadas, para poder observar mejor las partes de la ***trama*** en cuestión.

Debido al carácter homogéneo de cada trama, podemos centrarnos en una ***trama genérica 'n'***.

En los extremos se puede ver el final de la trama 'n-1' que le precede y el comienzo de la trama 'n+1'que le sucede (con su correspondiente señal piloto).

Seguidamente mostramos un dibujo de dicha trama genérica 'n', para aclarar un poco mejor los distintos campos de la trama ya estudiados:

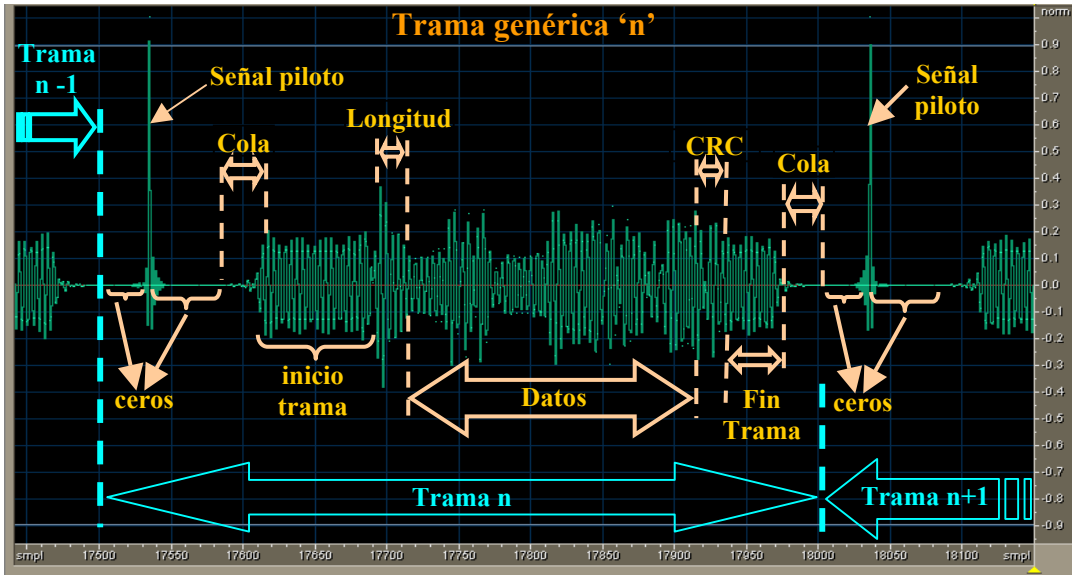


Fig. 62.- Trama genérica

La trama 'n', comienza con una ***secuencia de 30 ceros*** que anuncian la llegada de la señal piloto, seguida de otros ***50 ceros***, con la misión ya conocida de separar ambas tramas. Por la propia interpolación parece que se trata de una delta continua en el tiempo, pero las muestras son todas nulas, exceptuando la muestra de valor unidad de

nuestra señal piloto. A continuación aparece la parte de la señal que es modulada a 11025 hz., esto es:

- ***Inicio de Trama***: 8 símbolos $\xrightarrow{N=10}$ 80 muestras, que nos servirá para el ajuste de niveles en el receptor y para amortiguar el efecto transitorio de la cola sobre el campo longitud.
- ***Longitud***: 2 bytes $\xrightarrow{M=256}$ 2 símbolos $\xrightarrow{N=10}$ 20 muestras. Indica la longitud del campo datos en bytes.
- ***Datos***: Es la información propiamente que recogemos del fichero origen. En nuestro ejemplo ocupará: 20 bytes $\xrightarrow{M=256}$ 20 símbolos $\xrightarrow{N=10}$ 200 muestras.
- ***CRC:*** Se trata de nuestro código redundante cíclico. Al igual que el campo longitud consta de 2 bytes, por lo que ocupará 20 muestras.
- ***Fin de Trama***: 4 símbolos $\xrightarrow{N=10}$ 40 muestras, que nos servirá para amortiguar el efecto transitorio de la cola al paso por el Filtro Transmisor.

Estos ***tramos*** de la señal están ***atenuados*** debido al proceso que sufren en el ***Filtro Transmisor***. De ahí que, por ejemplo, la amplitud de ***Inicio de Trama*** no llegue hasta el valor de 0.4, y se quede en 0.15 aproximadamente.

Además, debido a dicha convolución de nuestros símbolos con el filtro raíz de coseno alzado, aparece unas ***colas*** como consecuencia del ***retraso de grupo*** de la señal. Ya comentamos que este retraso era de tres intervalos de símbolo, con lo que habrá 30 muestras espurias en cada extremo de los datos filtrados en nuestro sistema (no olvidemos que la secuencia de ceros y la señal piloto no sufren ningún proceso de modulación, sino que simplemente se añaden al principio de cada trama).

BLOQUE RECEPTOR

Nuestro Bloque Receptor abarcará el tramo desde el convertidor analógico/digital del puerto LINE IN de la tarjeta receptora (que también aparece con línea discontinua, por tratarse de un componente hardware transparente a nuestro diseño software), hasta el destino digital que supondrá la creación del fichero imagen, que deberá ser idéntico al fichero original.

Al igual que en el transmisor, las únicas subetapas consideradas en los libros de texto como pertenecientes a un Bloque Receptor propiamente dicho, son las correspondientes al detector APK y a la decodificación de canal mediante la autenticación del código CRC recibido.

Vamos a mostrar un ***esquema del Bloque Receptor*** en cuestión, donde también hemos introducido el ***canal***, con su inevitable ***fuente de ruido***:

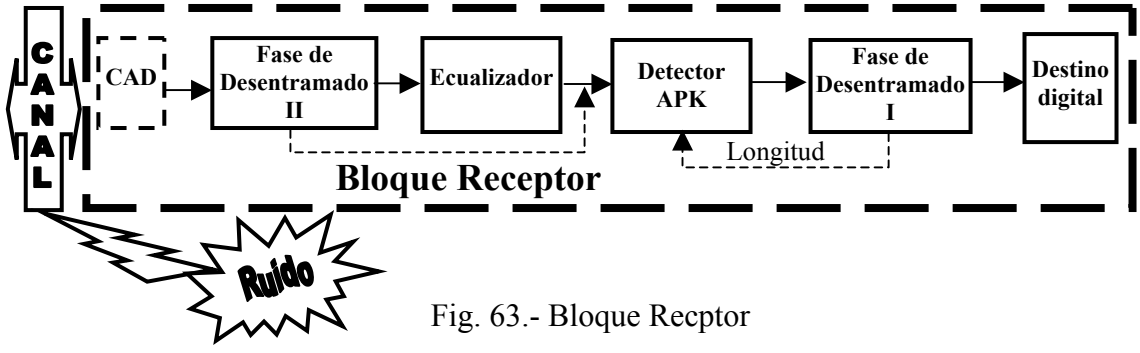


Fig. 63.- Bloque Recptor

Nos gustaría aclarar que con la ***línea discontinua*** que va de la ***Fase de Desentramado II*** hasta la salida del ***Ecualizador***, queremos reflejar que existe una segunda etapa de desentramado a la salida de éste. De igual manera, con la ***otra línea discontinua***, queremos dar a entender que primero se obtienen los bits correspondientes a la ***longitud*** de cada trama, y una vez conocida la cantidad de bytes de ***datos*** que contiene la trama, se obtienen el resto de ésta, es decir, la información propiamente dicha, y el código ***CRC***, desechando los restantes bytes de la trama.

Procedamos ya a entrar, con más detalle, en cada una de las fases de esta etapa receptora. Como el Bloque Receptor está destinado a la captación de la señal modulada y codificada, y a la recuperación del fichero original, las distintas etapas constituyen para nuestra señal el proceso inverso al sufrido por ésta en el transmisor. Por tanto, los conceptos son análogos, por lo que no nos detendremos mucho en ellos, exceptuando el concepto de ***ecualización***[†2], al que sí le dedicaremos más atención, al tratarse de una etapa inexistente en el transmisor, pues de alguna manera se trata de un proceso inverso a la transformación que sufre la señal en el canal de transmisión, el cuál ya analizamos profundamente en nuestro primer capítulo.

- Convertidor analógico/digital

Se trata de un dispositivo ubicado en el puerto LINE IN de la ***tarjeta de receptora,*** con la misión de recoger la señal analógica que le llega del cable de audio y codificar cada muestra (con lo que es obvio que exista una etapa de muestreo y retención antes de este convertidor, aunque no la dibujemos) mediante una ***tabla de conversión inversa*** a la que existía en el convertidor digital/analógico de la tarjeta de sonido, de forma que recuperemos los códigos binarios correspondientes a los valores normalizados entre –1 y 1 que teníamos en la entrada del convertidor digital analógico de la tarjeta transmisora. Naturalmente, somos totalmente ajenos a dichos códigos binarios, pues se trata de una representación interna de cada una de las muestras, aunque estamos limitados a este rango entre –1 y 1.

Tanto la presencia de ruido en el canal, como la misma atenuación no uniforme de éste, modificarán el valor de los datos, con lo que presumiblemente la codificación no coincidirá con la codificación de la correspondiente muestra transmitida.

En esta etapa se recibirá la ***secuencia de tramas completa***. Ni que decir tiene que tendremos que usar los mismos valores de ***Frecuencia de muestreo*** y de ***bits de resolución*** que usamos en el transmisor. Por tanto, ambos son parámetros configurables en esta etapa, pero con la limitación de que sus valores han de ser idénticos a los usados en la etapa transmisora, para procurar una correcta recepción de los datos.

Por tanto, y por lo general, usaremos los valores conocidos de:

- Frecuencia de muestreo: 44100 muestras/segundo.
- Número de bits de resolución: 16 bits/muestra.

La recepción de las muestras se implementará mediante la función de Matlab '*wavrecord'*, donde le pasamos como parámetros de entrada la frecuencia de muestreo, la resolución de bits, además del número total de muestras que vamos a grabar.

Respecto al ***número de muestras a grabar***, realizamos un ***cálculo*** de todas las muestras a transmitir, previo a la ejecución de la función *'wavrecord'*, a cuyo resultado le añadimos ***3 segundos más de muestras***, para tener tiempo de pulsar el botón de

transmisión en la sesión receptora una vez pulsado el botón de recepción, y evitar así tanto la pérdida de señal transmitida como transitorios indeseados.

Ejemplo práctico

Veamos a modo de ejemplo, la forma que presentan los datos a la salida del convertidor de la tarjeta, cuando todavía no han sufrido ningún proceso en recepción, sino sólo la atenuación no uniforme del canal y la adición de un ruido prácticamente inapreciable, además del desplazamiento de las muestras por la falta de sincronismo.

Para esta ocasión transmitiremos la señal que creamos a partir del fichero 'prueba.txt', de 1 Kbyte, en un ejemplo anterior, con los mismos parámetros, que recordamos a continuación:

- F_S = 44100 muestras / sg.
- NB = 16 bits de precisión.
- M = 256 símbolos.
- N = 10 muestras / símbolo.
- BDT = 20 bytes de datos / trama.
- Roll-off = 0.7
- retardo de grupo = 3 símbolos (30 muestras).

Volvemos a insistir en que usaremos un tamaño de bytes de datos por trama bastante superior al usado en este ejemplo, para reducir los bytes de control y optimizar la velocidad del sistema. Aunque existirá la limitación debido a la ausencia de sincronismo (ya comentamos que el número máximo de muestras por trama teóricamente debe ser tal que la diferencia entre los instantes de muestreo de relojes transmisor y receptor no sea superior al 1 %, siendo el 100 % la distancia entre un instante de muestreo y otro, o sea, cuando se gana una muestra en recepción).

El aspecto general que tiene la señal recibida lo mostramos a continuación:

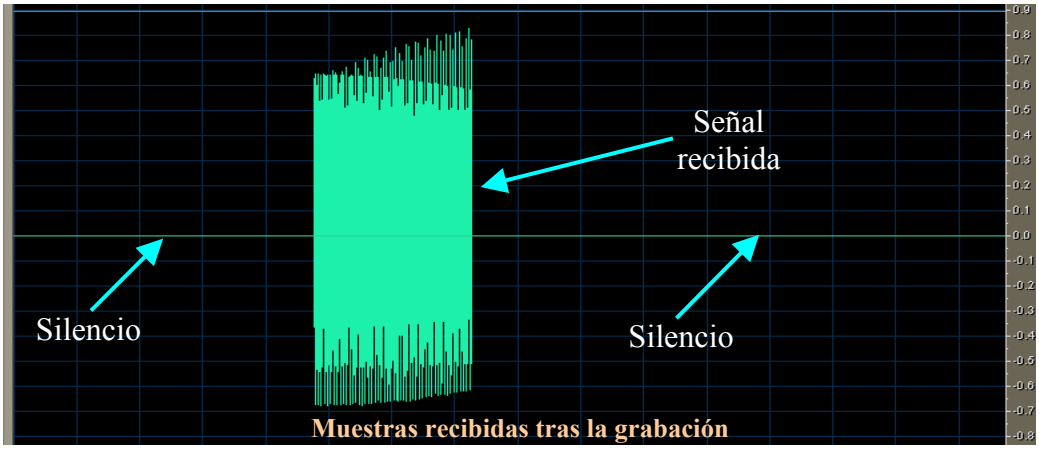


Fig. 64.- Muestras recibidas

La larga secuencia de ceros que existe a ambos lados de la trama es debido a los 3 segundos de muestras que añadimos al número total de muestras a grabar. Se puede

apreciar en esa transmisión en vacío, que el ruido aditivo es prácticamente inexistente en el canal.

Si realizamos un pequeño *'zoom'*, podremos ver mejor el aspecto de la señal completa, formado por secuencia de tramas y de señales piloto:

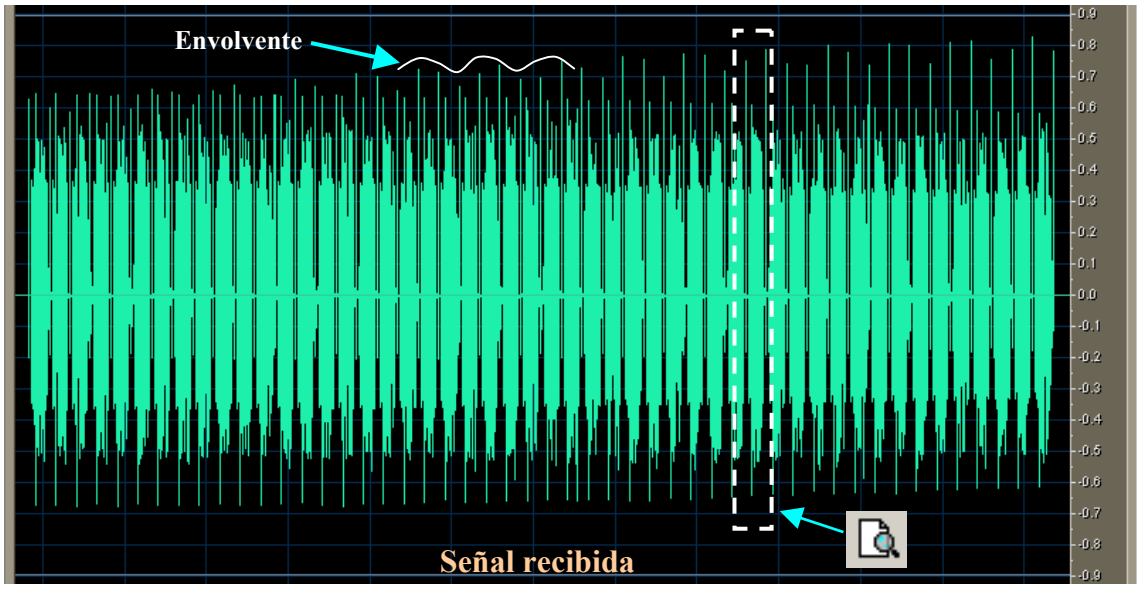


Fig. 65.- Señal Recibida

Se puede observar que las señales pilotos tienen amplitud menor a la unidad, debido a la atenuación del canal. Además, se puede ver que la ***atenuación no es uniforme***, a juzgar por las distintas amplitudes de las señales piloto. Se aprecia también una especie de ***envolvente***, debido a la ***falta de sincronización*** entre el transmisor y el receptor, aunque de dicho fenómeno nos ocuparemos más adelante.

Si realizamos un 'zoom' a la misma trama que ampliamos en el anterior ejemplo donde construimos la señal a transmitir, el resultado es el siguiente:

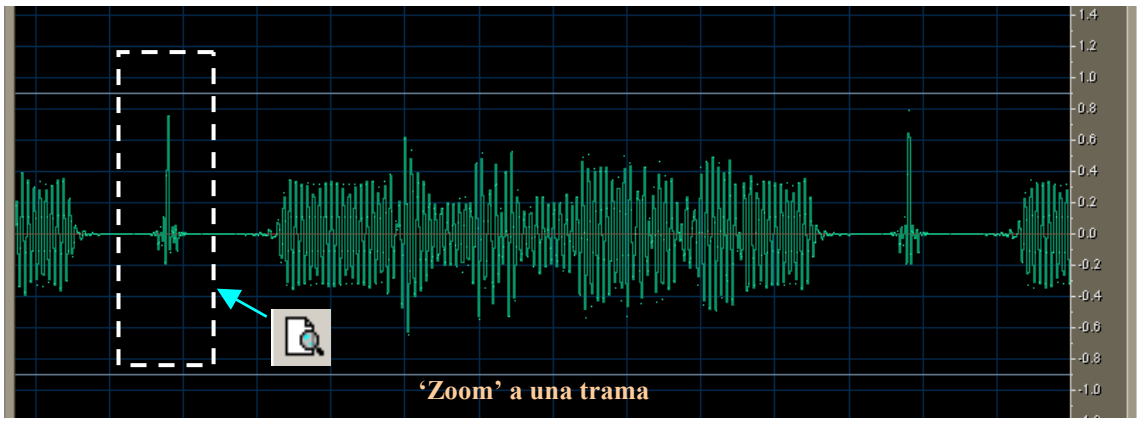


Fig. 66.- Zoom a Trama

Lo más interesante a comentar es la ***transformación*** que sufre nuestra ***señal piloto***, que se ha convertido en la respuesta impulsiva en el instante previo a la trama que le sigue. Aunque parezca que tenga el mismo aspecto que la señal piloto mostrada en la figura del ejemplo anterior, si hacemos un 'zoom', veremos que se trata realmente de la ***respuesta impulsiva del canal***, que tantas veces hemos representado en el capítulo dedicado al estudio del canal:

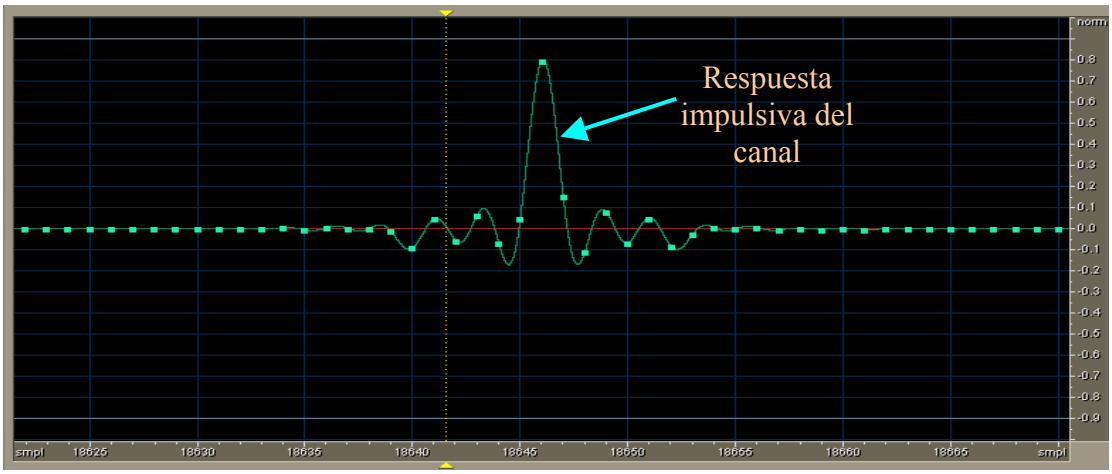


Fig. 67.- Respuesta impulsiva previa a la Trama

A partir de dicha respuesta impulsiva, construiremos, como veremos más adelante, nuestro ecualizador dinámico, adaptado a cada trama.

- Fase de Desentramado II

A la salida de la tarjeta de sonido, nos encontramos con *la* ***señal recibida***, $r(t)_{APK}$, que no es más que la señal transmitida modificada por los efectos del canal, por la adición del ruido de dicho canal y por los efectos del desincronismo.

El primer paso en nuestro procesamiento en recepción, una vez obtenida la señal APK, es separar las tramas para procesarlas de una en una. Por lo tanto, se trata de un procedimiento inverso al realizado en la Fase de Entramado II en transmisión. Así, nuestra ***Fase de Desentramado II*** podría tener el siguiente aspecto:

Fig. 68.- Fase de Desentramado II

En una primera fase se recibe y acumula, en una especie de almacén vectorial, todas las muestras de la señal recibida. Luego nos enganchamos a la primera señal piloto, que marca el comienzo de nuestra señal, y eliminamos todo el silencio que le precede. A partir de aquí, comienza un proceso de ***identificación*** de cada ***trama***, con la ayuda de las ***señales pilotos*** que las preceden, como ya explicamos cuando vimos la Fase de Entramado II durante el Bloque Transmisor.

Por último, mediante un bloque al que hemos llamado ***separador de tramas***, queremos reflejar que vamos tomando las tramas individualmente, para procesarlas de manera independiente, y volverlas a unir en nuestro destino.

Ejemplo práctico

Sigamos con nuestro ejemplo ya habitual de las muestras recibidas procedentes de nuestro fichero *'prueba.txt'*, y veamos el aspecto que tendrá la 'trama 36' (de las 50 que componen la señal transmitida) a la salida de nuestra Fase de Desentramado II.

Queremos hacer notar que tomamos esta trama sin pérdida de generalidad, para ir viendo su aspecto tras los procesos que va sufriendo en cada etapa del Bloque Receptor.

Como podemos apreciar, la trama comprende desde 20 muestras de la ***cola*** que precede a la señal piloto, hasta 20 muestras anteriores a la siguiente ***señal piloto***. Necesitamos estas 20 muestras que preceden a la señal piloto, porque como ya hemos dicho, forman su cola, que junto con otros 30 ceros posteriores al valor máximo de la señal piloto, pues tenemos una ***aproximación muy buena*** de lo que sería la ***respuesta impulsiva del canal***, para la trama que le sucede, y a partir de dicha respuesta, como veremos en el siguiente apartado, construiremos el ***ecualizador de cero forzado***. Esta aproximación es buena, porque hemos comprobado que las muestras alejadas a más de 10 instantes de muestreo del valor máximo de la señal piloto, son prácticamente nulas, y nosotros hemos tomado 20 muestras a un lado y 30 a otro, y así desechamos poca energía de la señal.

La figura que presenta nuestra 'trama 36', a la salida de la Fase de Desentramado II, es la siguiente:

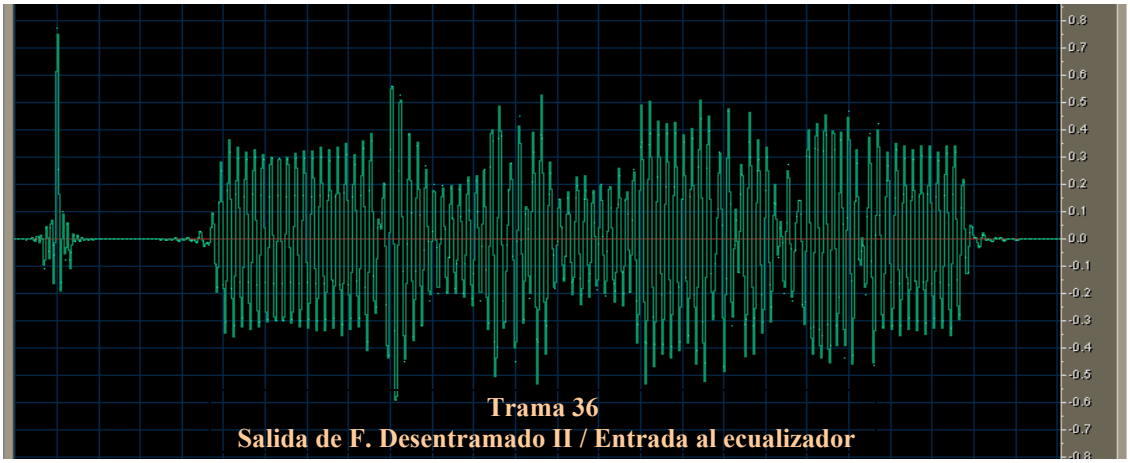


Fig. 69.- Salida de Fase Desentramado II

- Ecualizador

Este es el primer bloque que implementamos vía software en el receptor. Como dijimos, no aparece en el transmisor, porque de alguna manera, supone para la señal el proceso inverso al que es sometida en el canal. Esto es, como ya adelantamos en el capítulo III.1, "Estudio del canal de comunicaciones", el conjunto canal-ecualizador se comportará como un filtro ideal[†2], con una respuesta en frecuencia plana en toda la banda de frecuencias.

El diseño de nuestro ecualizador, dependerá, como sabemos, de dos factores principales:

1) De la relación señal a ruido de nuestro canal.
2) De la varianza temporal del canal.

Respecto a la SNR del sistema, dependiendo de si ésta es elevada o no, recurriríamos a distintos tipos de filtros que ignorarán los efectos del ruido en el caso de una SNR elevada, o lo minimizarán en el caso de una SNR baja.

En nuestro caso, ya tuvimos la oportunidad de comprobar, durante el estudio del cable de audio, en el capítulo III.1, que dicha relación es muy alta, por lo que ***descartamo***s el uso de ***filtros de Wiener***, para minimizar el error cuadrático medio del sistema[†4].

De entre los filtros que se usan en el caso de una SNR elevada, como pueden ser, por citar algunos, el filtro inverso, el tapped-delay-line o el ***igualador de cero forzado***, nosotros nos vamos a decantar por éste último, debido a su simplicidad, tanto de concepto como de implementación mediante un lenguaje de programación como pueda ser Matlab.

De todos modos, si usásemos un filtro de mala calidad, podríamos mantener la relación SNR aumentando la potencia de transmisión, mediante los controles de reproducción de la tarjeta de sonido, y podríamos mantener la efectividad de nuestro igualador de cero forzado.

Respecto al segundo parámetro, dependiendo de si el canal es o no variante con el tiempo, usaremos filtros adaptativos (donde el ecualizador se ajusta continua y automáticamente a las características temporales de la señal de entrada) o no adaptativos respectivamente.

Por citar un ejemplo de ***sistemas LTV*** (lineales y variantes con el tiempo), mencionaremos la ***red telefónica conmutada***, donde su variabilidad temporal de las características del sistema es debido entre otras cosas a los distintos tipos de transmisión de los enlaces usados, que se agruparán posteriormente.

Pero, como veremos más adelante, cuando estudiemos la falta de sincronización del sistema, aunque parezca que nuestro canal varía su respuesta impulsiva con el tiempo, comprobaremos que no es así, y que ese efecto es debido al desplazamiento de las muestras por la falta de sincronización. Por esta razón, ***usaremos un filtro no adaptativo***[†4].

Diseño del Ecualizador

Para ***diseñar*** nuestro ***igualador de cero forzado***, vamos a plantearnos el siguiente sistema genérico, para aplicarlo a nuestro canal:

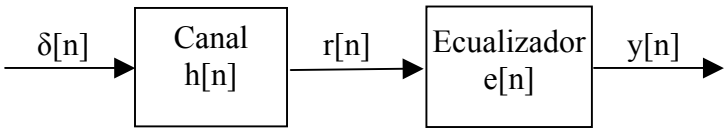


Fig. 70.- Conjuto Canal-Ecualizador

Despreciando la fuente de ruido, si transmitimos una delta, a la salida obtendremos la respuesta impulsiva del canal. Entonces la señal r[n] será idéntica a h[n].

El diseño del ecualizador se basa en la idea de recuperar de nuevo la delta de Dirac a la salida de éste. Es decir, pretendemos que el conjunto de canal más ecualizador se comporte como una delta, con una respuesta totalmente plana en frecuencia (no olvidemos la dualidad "delta en el tiempo-constante en la frecuencia" y viceversa[†2]).

De esta manera, con el ecualizador conseguimos eliminar toda la distorsión en amplitud que introduce el canal en el dominio de la frecuencia. También nos eliminará el poco ISI que aparezca, pues recordemos que estamos usando pulsos raíz de coseno alzado para cumplir el criterio de Nyquist y transmitir así en ausencia de ISI, pues la anchura del canal es mayor que la de nuestros pulsos de coseno alzado, y no los recorta en frecuencia, como ya hemos explicado en reiteradas ocasiones.

Gráficamente, el bloque anterior sería idéntico a éste:

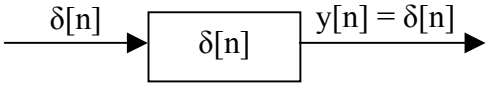


Fig. 71.- Sist. ideal

Así, tenemos la siguiente condición para nuestro diseño:

- $h[n] * e[n] = \delta[n] \quad \leftrightarrow \quad H(f) \cdot E(f) = 1$

  Delta en el tiempo ◄► Respuesta impulsiva plana

Desarrollando la convolución, tenemos la siguiente ecuación:

- $\sum_{k=0}^{M-1} e[k] \cdot h[n-k] = \delta[n]$ (ó $d[n-n_0]$, si admitimos un retraso n_0)

Visto de forma matricial sería de esta forma:

- $\underline{\underline{A}} \cdot \underline{x} = \underline{b}$

Si analizamos cada uno de los vectores y la matriz, tendrían el siguiente aspecto:

h[0]	0	0	0	0	0	0	0	e[0]	=	0
h[1]	h[0]	0	0	0	0	0	0	e[1]		⋯
⋯	⋯	⋯	0	0	0	0	0	⋯		0
h[N-1]	h[N-2]	⋯	h[0]	0	0	0	0	⋯		1
h[N]	h[N-1]	⋯	⋯	h[0]	0	0	0	⋯		0
0	h[N]	h[N-1]	h[N-2]	⋯	h[0]	0	0	⋯		⋯
⋯	0	⋯	h[N-1]	h[N-2]	⋯	h[0]	0	e[M-2]		0
0	⋯	⋯	⋯	h[N-1]	h[N-2]	⋯	h[0]	e[M-1]		0

Fig. 72.- Forma matricial del Ecualizador

Cabe decir que en este caso, ***N*** y ***M*** no tienen nada que ver con el número de muestras por símbolo ni con el número de símbolos de la constelación respectivamente. En esta ocasión, ***M*** hace referencia al ***número de muestras*** que usamos para crear el ***ecualizador***, y ***N*** es el ***número de muestras*** de la ***respuesta impulsiva*** de entrada al ecualizador. La relación óptima sería M=2·N, pero para optimizar aún más los resultados de nuestro ecualizador, usaremos la relación ***M=4·N***, aunque, por otra parte, se ralentizará el proceso de ecualización.

La resolución de este sistema matricial de ecuaciones se consigue con el uso de la pseudoinversa. Es decir, tenemos un sistema cuyas dimensiones son las siguientes:

$\underline{\underline{A}}$	·	$\underline{x}$	=	$\underline{b}$
{ [N+M-1] x [M] }	·	{[M] x [1] }	=	{ [N+M-1] x [1] }

La matriz $\underline{\underline{A}}$ ***no es cuadrada***, por tanto no tiene inversa. Esto implica que no podamos resolver el sistema invirtiendo directamente dicha matriz. Sin embargo, resolvemos el sistema mediante la ***técnica de la pseudoinversa***[†3], que consiste en multiplicar por $\underline{\underline{A}}^t$ a ambos lados de la ecuación. De esta manera, obtenemos la pseudoinversa de $\underline{\underline{A}}$, es decir, $\underline{\underline{A}}^t \cdot \underline{\underline{A}}$, que sí es invertible y podremos así resolver el sistema en cuestión:

$$\underline{x} = (\underline{\underline{A}}^t \cdot A)^{-1} \cdot \underline{\underline{A}}^t \cdot \underline{b}$$

Obteniendo así el ***vector x*** , que son las ***muestras*** correspondientes al ***igualador*** que vamos buscando. Dicho igualador se llama de ***cero forzado*** (***zero-forcing equalizer***), porque impone que el pulso de transmisión sea cero justo en los instantes adyacentes al instante de muestreo en el receptor.

Aunque ***no*** se trate de un ***ecualizador adaptativo***, ***sí*** podemos referirnos a éste como ***ecualizador dinámico*** en el sentido ya comentado de que para cada trama, formaremos un nuevo ecualizador, a partir de las muestras de la respuesta impulsiva que precede a cada trama.

Ejemplo práctico I

Los resultados prácticos que arroja este igualador son óptimos. Para corroborarlo, vamos a mostrar el resultado de ecualizar la respuesta impulsiva del canal en el peor de los casos, es decir, cuando, debido a la falta total de sincronización entre ambos relojes, no cae una muestra en todo lo alto de respuesta impulsiva, sino que se reciben dos muestras en los extremos laterales. La convolución de la respuesta impulsiva del canal y la respuesta impulsiva del ecualizador debe ser lo más parecida posible a una delta de Dirac, lo cual indicaría que el conjunto canal+ecualizador se comportaría como un canal ideal, con su correspondiente respuesta en frecuencia totalmente plana a lo largo del ancho de banda.

Primero vamos a transmitir una un secuencia de deltas bastante separadas, y vamos a escoger el peor caso ya comentado.

La respuesta impulsiva más castigada por la falta de sincronismo la representamos a continuación.

Fig. 73.- Respuesta impulsiva

Podemos ver que no existe ninguna muestra que caiga en exactamente en el pico de máximo de la señal interpolada.

Al igual que en nuestro programa receptor, vamos a usar solamente 50 muestras, para obtener el ecualizador e[n]. La longitud del ecualizador será cuatro veces la longitud de la señal de entrada a éste, con lo que tendremos 200 muestras para nuestro ecualizador.

Mediante la función de matlab que hemos creado, *'z_igualador_ZF'*, obtenemos el ecualizador óptimo para esta delta de dirac, y por tanto, para la trama que le sucede.

Tras la ecualización, la señal obtenida,y[n], tiene el siguiente aspecto (ampliando un poco la imagen):

Fig. 74.- Respuesta impulsiva ecualizada

Es lógico pensar, que si este igualador es capaz de devolver a la salida el impulso que introducimos en el canal, incluso para el caso de máxima desincronización, en los casos normales devolverá una delta perfecta, lo cual corrobora el hecho de que el conjunto canal + ecualizador se comporta como un sistema ideal.

Ejemplo práctico II

Para concluir con este apartado dedicado al ecualizador de canal, vamos a continuar con nuestro especial seguimiento que llevamos haciéndole a la 'trama 36' del fichero 'prueba.txt', para conocer el aspecto que tendrá a la salida del ecualizador, y así compararlo con el que tenía a la entrada de éste, y que representamos anteriormente en el apartado de Fase de Desentramado II:

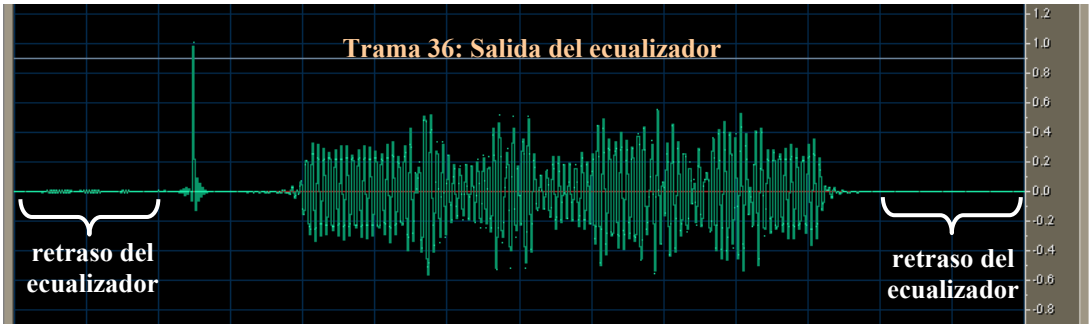


Fig. 75.- Salida del Ecualizador

Se puede apreciar el efecto de la ecualización en nuestra señal piloto, que ha recuperado la amplitud máxima. Si hiciéramos un *'zoom'* a dicha señal piloto, podríamos comprobar que, efectivamente, se trata de una delta de Dirac. El ecualizador, obviamente, se ha aplicado a la trama completa, y su labor principal básicamente es la de reubicar en su sitio las muestras desplazadas.

También hay que hacer notar el ***retraso*** que introduce el ***ecualizador***, que habrá que tenerse en cuenta en nuestro diseño.

- Detector APK

Entramos ya en la ***etapa estrella del receptor*** : el ***detector APK***, ya que en ella es donde se diseña la técnica de demodulación M-QAM.

En primer lugar, expondremos un ***dibujo*** donde se verán reflejadas todas las partes de este ***sistema detector***, y posteriormente trataremos, paso a paso, cada una de ellas, aunque sin volver a reincidir en la explicación de conceptos ya tratados en el Bloque Transmisor.

Como bien podremos apreciar en el dibujo, se trata de un ***proceso inverso al modulador APK***, pues tomamos a la entrada una trama inmersa en una portadora modulada en fase y amplitud, y obtenemos a la salida la secuencia de bits correspondientes a cada uno de los símbolos que componen la trama.

Pasemos a mostrar el esquema del detector APK:

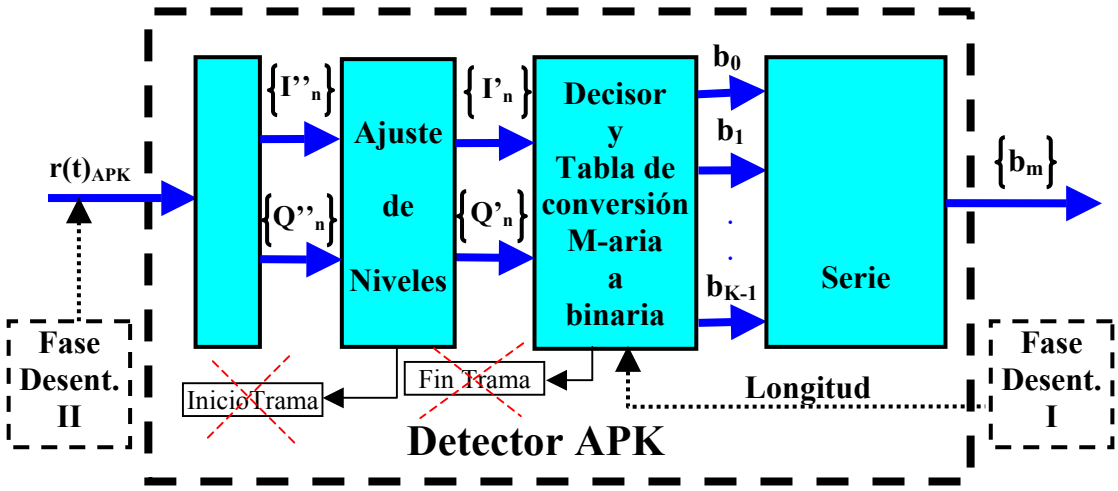


Fig. 76.- Detector APK

Pasemos ya, sin más demora, a analizar cada una de las etapas de este sistema detector:

Fase de Desentramado II

La hemos denominado así porque en realidad es una fase más de esta etapa, en cuanto a que es un proceso inverso al sufrido por la señal en la Fase de Entramado II del Bloque Transmisor.

En este punto previo al detector APK, las tramas van llegando una a una y ecualizadas, con lo que la señal piloto ya ha cumplido todas las funciones para la que fue creada, esto es, la sincronización de trama y la ecualización dinámica de trama. Por tanto, en esta fase, ***desechamos la señal piloto*** de la trama ecualizada *y* las muestras

correspondientes al ***retraso*** que introduce el ***ecualizador*** , y nos quedamos con el resto de campos de la trama en cuestión.

Demodulador

A la entrada de nuestro ***demodulador*** llega una señal cuyo espectro está desplazado a 11025 hz., que es obviamente la frecuencia de portadora que hemos usado.

Nuestro ***demodulador*** tiene la siguiente ***fisonomía interna***:

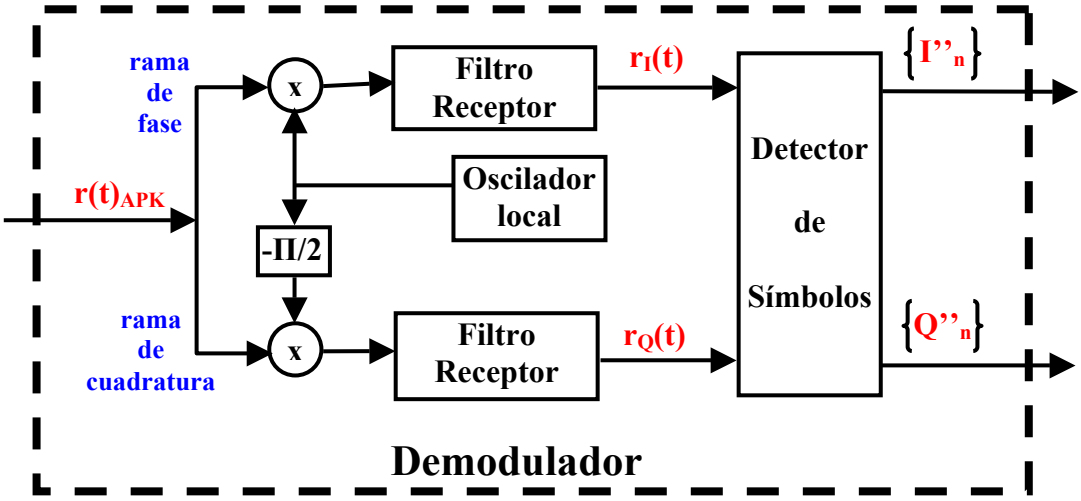


Fig. 77.- Demodulador

La ***misión*** del ***demodulador*** es recuperar la envolvente compleja, para obtener de ella las distancias en fase y en cuadratura de cada símbolo respecto al origen de coordenadas de nuestra constelación, para posteriormente decidir de qué símbolo se trata.

Para ***recuperar*** la ***envolvente***, tenemos que volver a desplazar el espectro de la señal a banda base. Esta es la misión del oscilador, como bien sabemos.

Analíticamente sería de la siguiente forma:

- A la ***entrada del demodulador*** tendremos la trama modulada, $r(t)_{APK}$, que debe ser igual a la transmitida, $S(t)_{APK}$, más un ruido aditivo, n(t), que lo podemos modelar como blanco y gausiano (AWGN) [†1] [†2]:

  - $r(t)_{APK} = S(t)_{APK} + n(t)$;

  Como bien sabemos, el ***ruido n(t)*** es muy ***pequeño***, así que no lo tendremos demasiado en cuenta. Así, la ***señal recibida APK*** será igual a:

  - $r(t)_{APK} = Re\,[\check{r}(t)\cdot\exp(j\cdot w_c\cdot t)] = r_I(t)\cdot\cos(w_c\cdot t) - r_Q(t)\cdot \operatorname{sen}(w_c\cdot t)$;

  donde $r_I(t)$ y $r_Q(t)$, son las ***componentes en fase y cuadratura*** de la envolvente compleja, $\check{r}(t)$, de la señal recibida $r(t)_{APK}$. Es decir:

- $\check{r}(t) = r_I(t) + j \cdot r_Q(t)$;

- Si hacemos pasar nuestra señal por un ***oscilador***, es decir, si multiplicamos nuestra señal en la rama de ***fase*** por ***cos(w_c·t)*** y en la de ***cuadratura*** por un ***sen(w_c·t)***, habremos desplazado la señal f_c = 11025 hz., y por tanto nos la habremos conseguido llevar a banda base que era lo que queríamos.
  Concretamente, para la ***rama de fase*** tendríamos[†1] [†2]:

  - $r(t)_{APK} \cdot \cos(w_c \cdot t) = r_I(t) \cdot \cos^2(w_c \cdot t) - r_Q(t) \cdot \text{sen}(w_c \cdot t) \cdot \cos(w_c \cdot t)$;

    El primer sumando, $r_I(t) \cdot \cos^2(w_c \cdot t)$, se puede descomponer en otros dos, uno que coincide con la componente en fase atenuada, y otro que es la componente en fase atenuada y desplazada $2 \cdot f_c$ hz.:

    - $r_I(t) \cdot \cos^2(w_c \cdot t) = r_I(t)/2 + r_I(t) \cdot \cos(2 \cdot w_c \cdot t)/2$

    El segundo sumando, $-r_Q(t) \cdot \text{sen}(w_c \cdot t) \cdot \cos(w_c \cdot t)$, se puede escribir de otra forma, empleando las conocidas ecuaciones de trigonometría:

    - $-r_Q(t) \cdot \text{sen}(w_c \cdot t) \cdot \cos(w_c \cdot t) = - r_Q(t) \cdot \text{sen}(2 \cdot w_c \cdot t) / 2$;

    donde podemos apreciar que se trata de la componente en cuadratura atenuada, invertida, y desplazada $2 \cdot f_c$ hz.

    Por tanto empleando un ***filtro paso de baja***, podríamos quedarnos, en la rama de fase con la ***componente en fase de la envolvente compleja***, atenuada por dos.

    - $r_I(t)/2$;

    ¡¡ Por esta razón, nuestro oscilador lo multiplicamos por dos !!, y obtenemos exactamente la componente en fase, $r_I(t)$.

- Análogamente, a la ***rama de cuadratura*** le aplicaremos el mismo proceso, pero con el oscilador desfasado $\pi / 2$ radianes respecto al anterior. Aquí ya realizamos los cálculos con el oscilador al doble de amplitud, y signo negativo, para obtener exactamente la componente en cuadratura:

  - $r(t)_{APK} \cdot (-2) \cdot \sin(w_c \cdot t) = r_I(t) \cdot \cos(w_c \cdot t) \cdot \text{sen}(w_c \cdot t) - r_Q(t) \cdot \text{sen}^2(w_c \cdot t)$;

    El segundo sumando, $-r_Q(t) \cdot \text{sen}^2(w_c \cdot t)$, se puede descomponer en otros dos, uno que coincide con la componente en cuadratura, y otro que es la componente en cuadratura, invertida y desplazada $2 \cdot f_c$ hz.:

    - $-r_Q(t) \cdot (-2) \cdot \text{sen}^2(w_c \cdot t) = r_Q(t) - r_Q(t) \cdot \cos(2 \cdot w_c \cdot t)$

    El primer sumando, $r_I(t) \cdot \cos(w_c \cdot t) \cdot \text{sen}(w_c \cdot t)$, se puede escribir de otra forma, empleando las conocidas ecuaciones de trigonometría:

    - $r_I(t) \cdot (-2) \cdot \cos(w_c \cdot t) \cdot \text{sen}(w_c \cdot t) = -r_I(t) \cdot \text{sen}(2 \cdot w_c \cdot t)$;

donde podemos apreciar que se trata de la componente en fase invertida y desplazada $2 \cdot f_c$ hz.

Aquí también podemos intuir que, mediante otro ***filtro paso de baja***, podríamos quedarnos, en la rama de cuadratura, con la ***componente en cuadratura de la envolvente compleja***.

- $r_Q(t)$;

Por último, cabría comentar que para la implementación vía Matlab de nuestro oscilador, volveremos a usar las funciones *'cos'* y *'sen'*.

Nuestro ***Filtro Receptor*** será, como es obvio, un ***pulso raíz de coseno alzado***, con el ***mismo factor de Roll-off*** que el usado en el ***Bloque Transmisor***, para conseguir en ***conjunto*** (no olvidemos que el canal junto con el ecualizador constituyen un filtro ideal con amplitud unidad y constante en el espectro de frecuencias) un ***pulso coseno alzado***, que cumpla la ***condición de Nyquist de transmisión en ausencia de ISI***[†1][†4].

Pero también nos hemos valido de las ***características de limitación en banda*** de este ***pulso raíz de coseno alzado***, para usarlo como el ***filtro paso de baja*** que necesitamos para quedarnos con la componente en fase, $r_I(t)$, y la componente en cuadratura, $r_Q(t)$, en sus respectivas ramas[†2].

De este modo, ambas componentes quedan de la siguiente manera:

- $r_I(t) = \sum_n I''_n \cdot p(t-n \cdot T)$;
- $S_Q(t) = \sum_n Q''_n \cdot p(t-n \cdot T)$;

donde ***p(t)*** es el ***pulso de coseno alzado***, e ***I''$_n$*** y ***Q''$_n$*** son las ***componentes en fase y cuadratura*** respectivamente de los ***símbolos recibidos***, cada n·T segundos (de ahí el subíndice 'n', como bien sabemos), siendo T el intervalo de símbolo. Las dobles comillas es una manera de indicar que no son exactamente de igual valor que las componentes en fase y cuadratura de los símbolos transmitidos.

No olvidemos que el filtro raíz de coseno alzado introduce un ***retraso de grupo***, y que hemos definido ese parámetro como fijo en nuestro sistema, con un valor igual a 3 períodos de símbolo. Por tanto, tendremos que tener en cuenta que, antes de la primera muestra correspondiente al Inicio de trama, exisitirá una cola de ***6·N muestras*** (la suma de los retrasos del Filtro Transmisor y Receptor) siendo 'N' el número de muestras por símbolo), y de igual forma, habrá una cola de 3·N muestras tras la última muestra de nuestro último símbolo (correspondiente al CRC).

También es importante señalar que en este caso no existe sobremuestreo, sino que la velocidad de muestreo a la entrada y salida del Filtro Receptor es la misma.

El último bloque del modulador APK lo hemos llamado ***Detector de símbolos***, y se encarga de tomar una muestra de cada N que le van llegando, siendo 'N' el número de muestras por símbolo del sistema. Es decir, a este bloque le corresponde la labor de quedarse únicamente con los valores de los símbolos del sistema y lo hace precisamente disminuyendo la tasa de muestreo por N, para conseguir así una tasa de símbolos.

Además, en el detector también ***desechamos las 6·N primeras muestras***, que se corresponden con el ***retraso de grupo*** introducido por ambos filtros raíz de coseno alzado.

Ejemplo práctico

Para intentar afianzar visualmente la función que tiene encomendada el Demodulador, procedamos a continuar con nuestro especial seguimiento de la 'trama36'. Vamos a mostrar primeramente, el aspecto que tendrá a la ***salida del Filtro Receptor***, por ejemplo, para la ***rama de fase***:

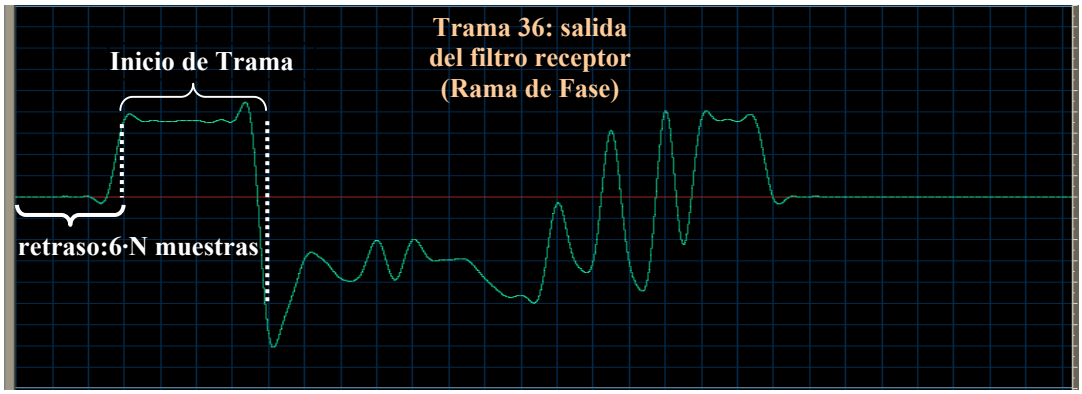


Fig. 78.- Envolvente de Fase

Observamos que ya no existe ningún tipo de modulación, sino que nos hemos quedado con la envolvente compleja (en este ejemplo, al tratarse de la componente en fase, nos hemos quedado con la parte real de la envolvente compleja).

Podemos apreciar el resultado de la fase de desentramado II previa al demodulador APK, donde se eliminó la señal piloto y sus ceros correspondientes.

A continuación mostramos el aspecto de la señal a la **salida del detector de símbolos**, donde previamente se ha eliminado, como ya habíamos comentado, las muestras correspondientes a los retrasos que introducen los filtros raíz de coseno alzado.

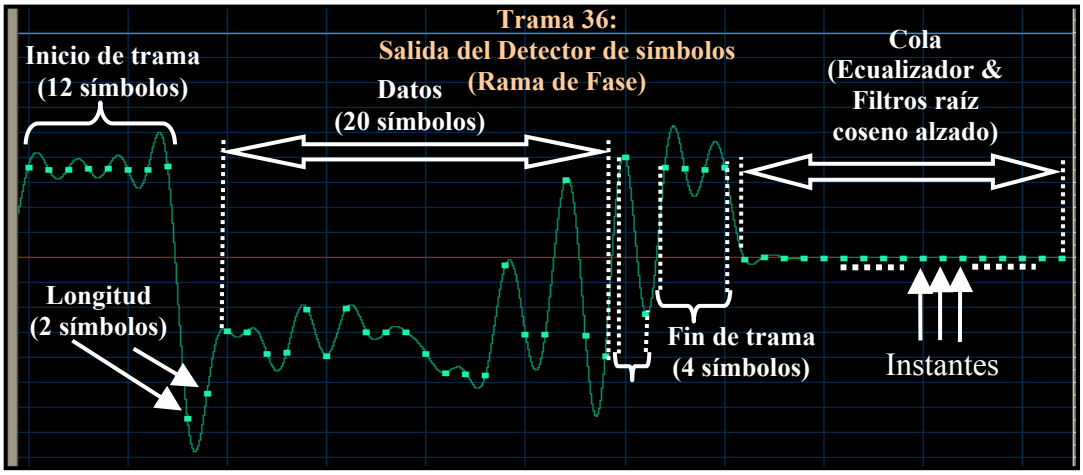


Fig. 79.- Salida del Detector Símbolos

En el recuadro hacemos hincapié de que, a pesar de que la interpolación que realiza el programa para mostrarnos la figura, realmente se trata ya de un tren de deltas de dirac, con distintos pesos correspondientes a los valores en fase de los símbolos, y separados cada T segundos, siendo T el tiempo de símbolo. De esta manera, cada punto representa un símbolo distinto, concretamente su coordenada en fase.

Como estamos en el caso de M=256, cada símbolo equivaldrá a un byte, por lo tanto tendremos:

- Inicio de Trama: 8 símbolos (en realidad, siempre tendremos, como dijimos, 8 símbolos independientemente del valor de M).
- Longitud: 2 símbolos.
- Datos: 20 símbolos.
- CRC: 2 símbolos.
- Fin de Trama: 4 símbolos (independientemente del valor de M).

El resto de símbolos de la señal, como hemos indicado, corresponde a las ***colas*** que generan el ***ecualizador y los filtros raíz de coseno alzado***, al convolucionar con la señal que le llega. En el detector tomamos una muestra de cada diez, y le aplicamos el mismo procedimiento a la cola, con lo que resultan estos ***símbolos espurios***, pero nos desharemos de ellos (y del ***Fin de Trama***) cuando decidamos el valor de los bytes del campo longitud, y sepamos la verdadera duración de la trama.

Ajuste de Niveles

En cada una de las ***dos salidas del Demodulador*** (la de la rama de fase y la de cuadratura) tendremos un tren de deltas de Dirac a la velocidad de D baudios (donde D = 1/T hz.), cuyos pesos serán las distancias en fase y cuadratura respecto al origen de coordenadas de nuestra constelación. Analíticamente podríamos expresarlo de la siguiente forma:

- $\{I''_n\}=\sum_n I''_n \cdot \delta(t-n\cdot T)$
- $\{Q''_n\}=\sum_n Q''_n \cdot \delta(t-n\cdot T)$

Los procesos convolutivos que sufre la señal tanto en los filtros transmisor y receptor como en el ecualizador, y el cambio de amplitud que introduce el canal, entre otras cosas, producen un ***cambio en el rango de niveles de nuestra señal***, a la salida del demodulador.

Por tanto, si intentáramos, con las amplitudes que tenemos, I''_n y Q''_n, decidir qué símbolos hemos recibido, seguramente fracasaríamos en el intento, pues la magnitud del error entre el símbolo recibido y el transmitido sería mayor que la distancia máxima permitida, delimitada por la región de decisión de cada símbolo.

Necesitamos, en consecuencia, implementar un bloque previo al Decisor, que nos proporcione un ***ajuste de niveles***, esto es, que el rango actual nos lo convierta nuevamente al rango [–1,1] con el que partimos en el transmisor.

Para realizar ese ajuste, debemos mandar símbolos conocidos, es decir, que nos valdremos del ***Inicio de Trama*** para ver las amplitudes recibidas y averiguar así la relación entre los símbolos transmitidos y recibidos.

La experiencia nos ha demostrado que la ***relación entre los símbolos transmitidos y los recibidos***, por ejemplo, para la ***rama de fase***, es la siguiente:

- $I''_n / I_n = c$; ($\boldsymbol{I_n}$ representa, obviamente, al ***símbolo ideal***, transmitido)

donde ***'c'*** es un valor se mantenía más o menos ***constante*** para cada par de símbolos relacionados.

Así que tomamos la media de todos los valores de 'c' para los símbolos 5º, 6º, 7º y 8º de cada par de inicio de tramas transmitido y recibido, para evitar los problemas de transitorios con el resto de los 8 símbolos de Inicio de Trama.

Así, si queremos ***desnormalizar*** para quedarnos con el rango con el que transmitimos los símbolos, no tenemos más que dividir los símbolos recibidos, I''_n , por la media de 'c' obtenida previamente. Analíticamente:

- $I_n = I''_n / c$;

Para los ***símbolos de la rama de cuadratura, Q''_n*** , actuaremos con la misma manera de proceder.

Por último, y una vez que el ***Inicio de Trama*** ha cumplido con su cometido, ***lo desechamos*** de la trama y nos quedamos con el resto de campos, siendo el primero de todos ahora, el de longitud de la trama.

Ejemplo práctico

Para concluir con este bloque, mostraremos un ***dibujo*** de la secuencia de símbolos para la ***rama de fase*** a la ***salida del Ajustador de niveles***.

Se puede apreciar claramente que se ha excluido el Inicio de Trama para el resto de procesos del Bloque Receptor.

Además podemos corroborar, que efectivamente, el ***primer campo*** ,correspondiente a la ***longitud de la trama***, es el que efectivamente componen los 2 símbolos que adelantábamos anteriormente.

Sin más dilación, este es el aspecto de la salida de la etapa de ajuste de niveles:

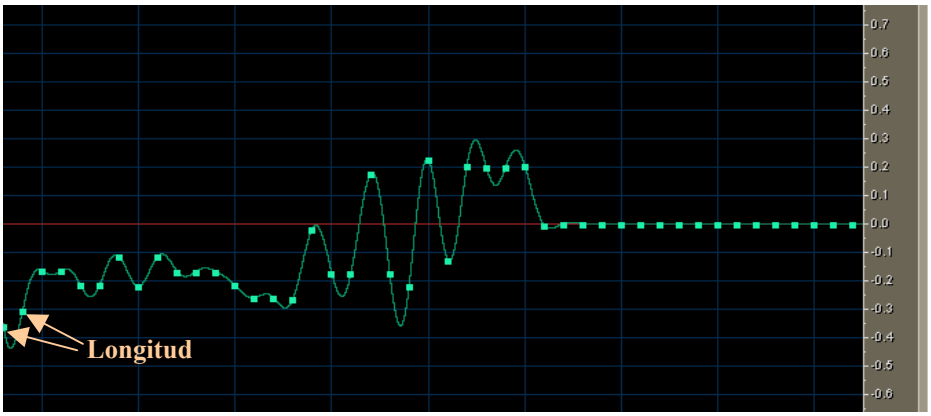


Fig. 80.- Salida del Ajustador Niveles

Decisor y Tabla de conversión M-aria a binaria

A la ***entrada del decisor*** de símbolos, tanto para la rama de fase como para la de cuadratura nos llega una secuencia de amplitudes, que se pueden expresar de la siguiente manera:

- $\{I'_n\}=\sum_n I'_n \cdot \delta(t-n\cdot T)$;
- $\{Q'_n\}=\sum_n Q'_n \cdot \delta(t-n\cdot T)$;

donde la comilla indica que, a pesar del ajuste de niveles, los símbolos que nos llegan al decisor, ***$\{I'_n\}$ y $\{Q'_n\}$***, no van a ser idénticos a los símbolos transmitidos, $\{I_n\}$ y $\{Q_n\}$, debido a que el ecualizador no es perfecto, y no elimina todas las irregularidades en amplitud del canal, y a que el ruido siempre afecta, aunque mínimamente, a la posición de los símbolos en el plano de la constelación.

Para corroborar la anterior afirmación vamos a mostrar un ***dibujo***, con la ayuda del comando *'plot'* de Matlab, de los símbolos recibidos correspondientes a una columna de símbolos transmitidos en un ejemplo anterior. Los símbolos transmitidos fueron los siguientes:

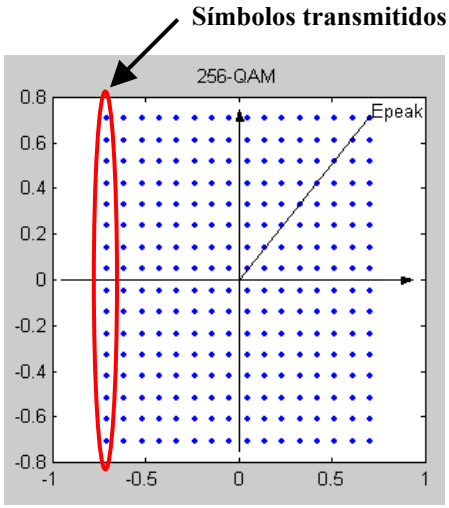


Fig. 81.- Símbolos del Ejemplo

A continuación mostramos la ***constelación*** de todos los posibles símbolos, en ***azul***, y los puntos en ***rojo*** corresponderá a los ***símbolos obtenidos*** durante el procesamiento de la señal en el Bloque Receptor, previo a la decisión de estos símbolos.

Haciendo un pequeño *'zoom'* a la zona punteada, podremos comprobar que realmente las posiciones de los símbolos transmitidos y recibidos no coinciden.

Esta es la imagen resultante y su 'zoom' anexo:

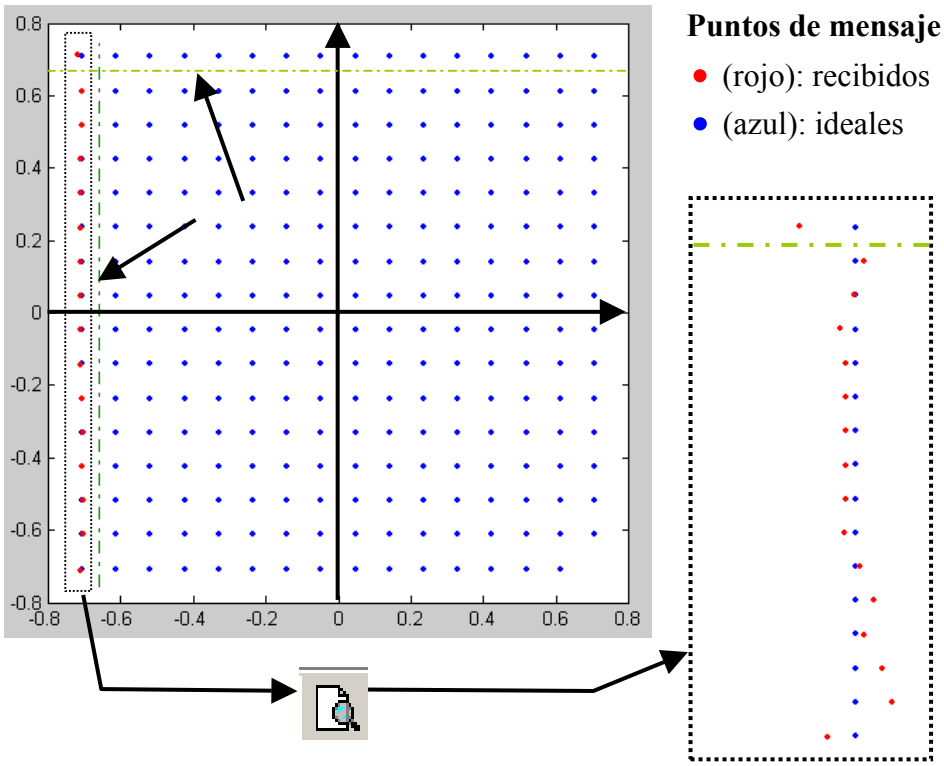


Fig. 82.- Símbolos transmitidos vs. recibidos

Ya comentamos con anterioridad, que nos vamos a basar en el ***detector de máxima verosimilitud***. Este detector consiste en comparar, para cada símbolo recibido, la distancia euclidiana respecto a cualquier posible símbolo de la constelación, y elegir como símbolo transmitido a aquel cuya distancia euclidiana respecto al recibido sea menor[†1][†4].

Esa es la ***labor del Decisor***: decidir para cada símbolo recibido, de qué símbolo transmitido se trata. Por tanto, y como ya sabemos, nos interesa que los símbolos recibidos se parezcan lo más posible a los transmitidos, para que caigan en la ***región de decisión*** de éste y nuestra decisión respecto al símbolo transmitido sea correcta, y no existan errores en los bits[†1][†4].

Ejemplo práctico

De la misma manera que acabamos de hacer en el anterior ejemplo, vamos a mostrar la constelación para M=256, y vamos a ubicar en ella los símbolos obtenidos para nuestra 'trama 36', previo a la entrada en el bloque decisor.

Los ***puntos rojos***, como apuntamos en el texto adjunto, indican la ubicación de los ***símbolos recibidos*** para los campos longitud, datos y CRC de nuestra 'trama 36'.

Podemos apreciar que los símbolos recibidos no coinciden con los ideales:

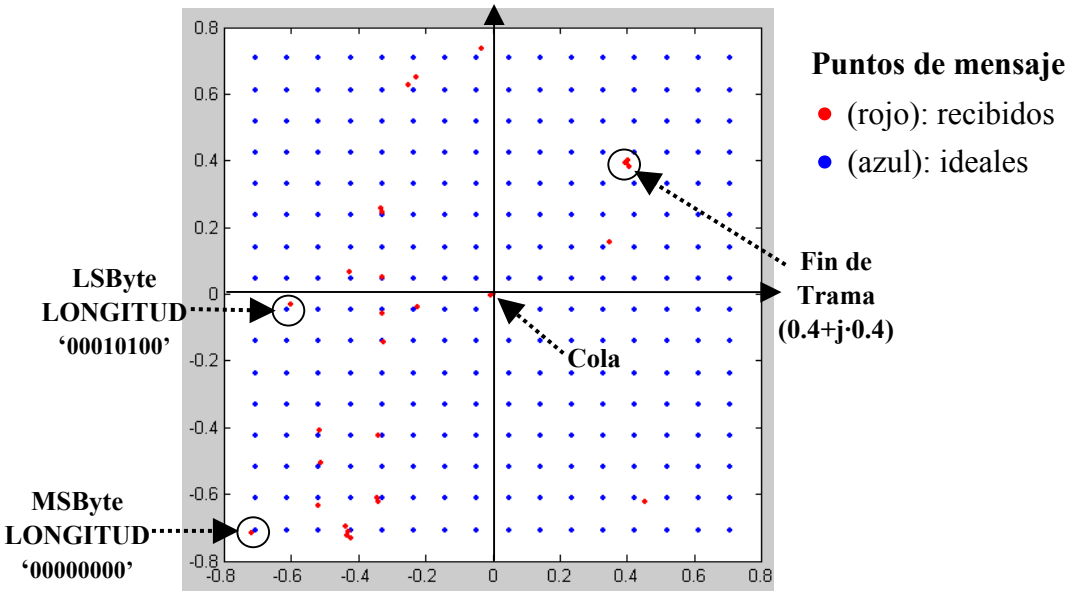


Fig. 83.- Entrada Detector

Aunque posteriormente veremos que los ***símbolos*** correspondientes al ***Fin de Trama*** y a la ***cola*** los desecharemos, una vez decidido el valor del campo longitud de la trama, los hemos pasado por el decisor de símbolos a modo ejemplo, indicando su posición dentro del plano, y comprobando la congruencia de los resultados:

- ***Fin de Trama*** : se trata, como era de esperar, de 4 situados en torno a la posición 0.4+j·0.4 del plano de la constelación.
- ***cola***: la constituyen una serie de valores prácticamente nulos tanto en fase como en cuadratura, por lo que es lógico que se sitúen ***cerca del origen*** de coordenadas.

Otro campo cuyos ***símbolos*** podemos controlar si han caído o no dentro de su región de decisión es el de ***longitud***. Hemos transmitido, en nuestro ejemplo particular, ***20 bytes*** de datos por trama. Por tanto el campo LONGITUD, contendrá el número 20 en binario, usando para ello dos bytes. Su contenido, donde indicamos el byte más significativo y el menos significativo, será el siguiente:

- LONGITUD: 20 = '$\underbrace{00000000}_{\text{MSByte}}\text{-}\underbrace{00010100}_{\text{LSByte}}$'$_b$

Si nos vamos a nuestra Constelación para M = 256, podremos ratificar que, efectivamente, los símbolos recibidos caen dentro de la región de decisión correcta. Hemos indicado con un círculo la ubicación de ambos símbolos.

Para el resto de puntos de mensaje que aparecen en la constelación, hemos de usar el detector de máxima verosimilitud, para decidir de qué símbolo se trata.

Posteriormente, el código CRC se encargará de determinar si han existido errores en las decisiones de los símbolos, y por tanto, de los bits decididos.

Por otra parte, al decidir sobre qué símbolo estamos transmitiendo, lo que hacemos es ***reubicar los símbolos recibidos*** en el plano, ***en la posición del símbolo ideal más cercano***, para poder usar las T***ablas de Conversión Inversas*** a las usadas en el Bloque Transmisor.

Esta ***Tabla de Conversión*** es la misma que la que usamos en el Bloque Transmisor, es decir, la relación entre los símbolos y los bloques de 'k' bits es la misma. La diferencia estriba en que el ***mapeo*** se realiza de forma inversa. En esta etapa, van entrando símbolos (ya reubicados tras el paso por el Decisor), a una velocidad de D baudios, con su correspondiente amplitud en fase y cuadratura, $\{I_n\}$ y $\{Q_n\}$, y vamos obteniendo a la salida el bloque de bits asociado a cada símbolo de entrada.

Convertidor Paralelo – Serie

A la salida de la Tabla de conversión tenemos 'k' bits en paralelo. La ***misión*** del ***convertidor Paralelo – Serie***, como su nombre indica, es transformar ese bloque de 'k' bits en paralelo en una ***secuencia de bits***, $\{b_m\}$, con una tasa de R_b bps, para pasársela a la Fase de Desentramado I.

Ejemplo práctico

Para afianzar conceptos, pongamos un pequeño ***ejemplo***, para ver cómo funciona dicha cadena de etapas. Vamos a representar el mismo ejemplo que el que vimos durante el agrupador de 'k' bits y la tabla de conversión, en el Bloque Transmisor. Lógicamente, al tratarse ambos, de procesos inversos, la fisonomía del esquema es muy parecida.

Aquí mostramos cómo entran 3 símbolos en una tabla de conversión cuaternaria, su correspondencia con un bloque de k=2 bits, y su paso por el conversor paralelo-Serie.

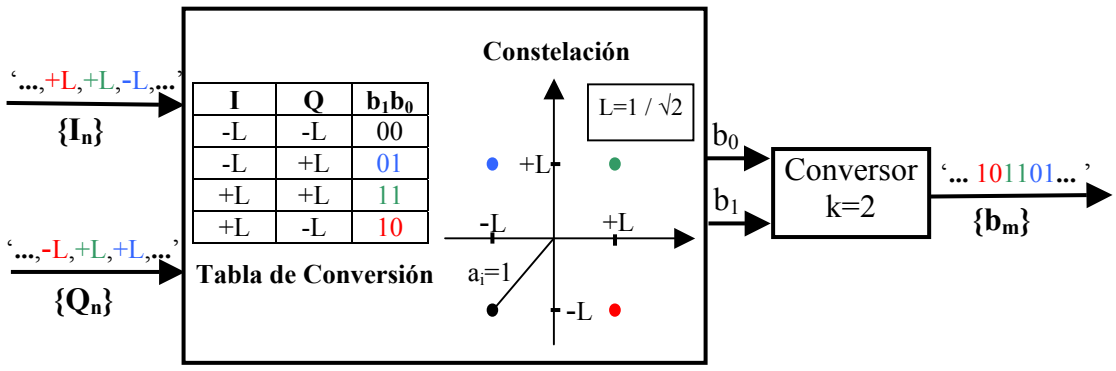


Fig. 84.- Mapeo y Conversión paralelo-serie

Para concluir con el Convertidor Paralelo-Serie y, en consecuencia, con la etapa del Detector APK, vamos a comentar brevemente el significado de esa especie de ***realimentación*** dibujada en el esquema del Detector APK.

Lo que quiero indicar con esa flecha punteada es que la ***decisión*** se realiza en ***dos fases***, pues primero se decide sobre los bits que componen el campo ***LONGITUD***, se procesa éste en la Fase de Desentramado I, y posteriormente se decide sobre el ***resto de bits válidos*** de la trama, ***desechando*** las muestras pertenecientes a la ***cola espuria*** y al ***Fin de Trama***, evitando así el procesamiento innecesario de éstas.

Así, el paso de cada trama por el ***Decisor*** y por el ***Convertidor Paralelo-Serie*** se realiza ***dos veces***: una para el campo ***LONGITUD*** y otra para los campos ***DATOS y CRC***.

- Fase de Desentramado I

Esta etapa la hemos llamado así por tratarse de un proceso contrario a la Fase de Entramado I, ya que en esta etapa vamos a desmontar los campos que agregamos en ese tramo del Bloque Transmisor.

Este bloque lo podríamos ver de la siguiente forma:

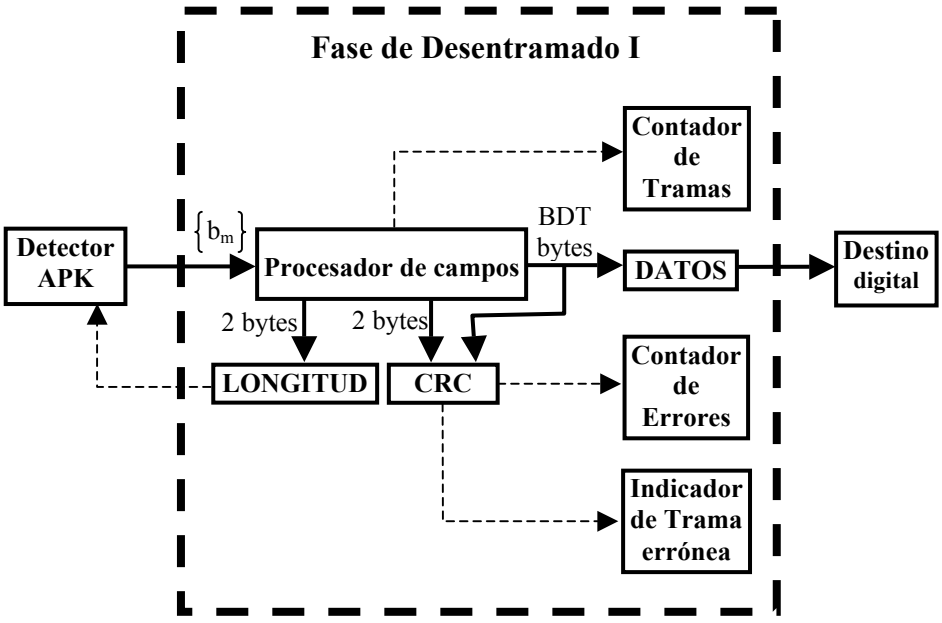


Fig. 85.- Fase de Desentramado I

Mediante este esquema intentamos ***modelar visualmente*** todos los procesos que ocurren en este tramo de nuestro Bloque Receptor, tal y como lo hemos plasmado en nuestro programa en *'Matlab'*.

Vamos a explicar cada una de las partes de este esquema, que aunque a simple vista parece muy enrevesado, es muy fácil de implementar vía software.

Procesador de Campos

El ***Procesador de campos*** es una especie de 'buffer' inteligente que tiene la misión de recibir la secuencia binaria procedente del Detector APK, y distribuir los bits de un determinado campo a sus correspondientes procesadores.

Así, en una primera pasada, le llegan los bits correspondientes al campo LONGITUD de la trama que se está procesando, y el Procesador de campos manda estos dos bytes a su procesador homónimo.

Posteriormente llegan el resto de bits de la trama, esto es, los pertenecientes a los campos DATOS y CRC, y el Procesador de campos los envía a sus procesadores, que los hemos denominado con el mismo nombre.

Por otra parte, el Procesador de campos se encarga de ***contabilizar*** todas las ***tramas*** que le van llegando, incrementando un ***Contador de tramas***.

Procesador de LONGITUD

El procesador del campo LONGITUD recoge los dos bytes y realiza la ***conversión de binario a decimal*** y manda el resultado al bloque Decisor del Detector APK, para que éste conozca el número de símbolos restantes válidos de la trama y elimine los símbolos inservibles de la cola y del Fin de Trama (que ya vimos que sólo servía para amortiguar los efectos del transitorio de las colas), minimizando el número de operaciones inútiles, lo cual desemboca en una mayor rapidez del procesado de la señal en recepción.

La conversión de binario a decimal la llevamos a cabo mediante la orden de Matlab *'bin2dec'*.

Procesador de CRC

Mediante los dos bytes del campo CRC recibido y los BDT (bytes de datos de la trama) bytes del campo DATOS, usando el ***polinomio generador CRC-16***, y mediante una ***división***, como veremos más adelante, comprobamos la existencia o inexistencia de errores en la decisión de los bits del campo de DATOS.

En caso de error(es) en la trama, el Procesador de CRC activará un ***indicador*** que mostrará por pantalla la existencia ***de error en*** esa ***trama***. Además, incrementará un ***Contador de Errores*** que tiene habilitado para el caso.

Procesador de DATOS

El procesador de DATOS actúa como 'buffer' que ***acumula*** todas las secuencias de ***bits*** correspondientes al campo DATOS de todas las tramas.

Luego ***agrupa*** los bits en ***bytes*** y realiza la ***conversión***, ***byte a byte***, de ***binario a decimal***, con el objetivo de mandarlos hacia lo que hemos denominado Destino Digital, para la creación del nuevo fichero.

- Destino Digital

Con los bytes en formato decimal que provienen del procesador de DATOS, procederemos a ***crear el nuevo fichero*** en el ***Destino Digital***. Si no ha habido errores, el fichero de destino y el original han de ser iguales.

Hay que hacer notar que el fichero puede que no esté creado aún, y que al igual que el fichero origen, el fichero de destino puede estar ubicado en cualquier medio de almacenamiento. Para crear el nuevo fichero basta con ***abrirlo*** en ***modo escritura***. Con la apertura del fichero, podemos obtener el ***identificador***, ***fid***, para poder referirnos a él a la hora de escribir los datos de destino. La apertura y la escritura de ficheros la implementaremos en Matlab con las siguientes funciones:

- *fopen*: para abrir el fichero (y crearlo) y obtener el identificador.
- *fwrite*: para leer los bytes del fichero en cuestión.

Al contrario que ocurría con la función '*fread'*, con la función *'fwrite'* leeremos todos los bytes en formato decimal de una sola vez.

Posteriormente procederemos a ***cerrar*** el ***fichero***. Esta acción la implementaremos con la ya conocida función *'fclose'*, con la que damos por concluida nuestra aplicación.

Ejemplo práctico

Durante el análisis de nuestro sistema de comunicaciones hemos expuesto ***numerosos ejemplos ilustrativos***, para mejorar nuestra comprensión sobre el funcionamiento del método de transmisión empleado.

Sin embargo, hasta ahora sólo hemos representado la señal en el dominio temporal. Por este motivo, nos gustaría acabar este extenso capítulo III.2.3, dedicado al diseño de nuestro sistema, mostrando el ***espectro frecuencial*** de la señal que hemos creado en nuestros continuos ejemplos realizados sobre el fichero 'prueba.txt', y contrastar los conceptos expuestos en dicho capítulo con las gráficas espectrales correspondientes.

Conviene volver a recordar las ***características del fichero*** y los ***parámetros*** que usamos para su transmisión:

- F_S = 44100 muestras / sg.
- NB = 16 bits de precisión.
- M = 256 símbolos.
- N = 10 muestras / símbolo.
- BDT = 20 bytes de datos / trama.
- Tamaño del fichero = 1000 bytes (50 tramas).
- Roll-off = 0.7
- retardo de grupo = 3 símbolos (30 muestras).

En primer lugar, vamos a exponer una ***porción de la señal*** en el tiempo justo antes de ser transmitida:

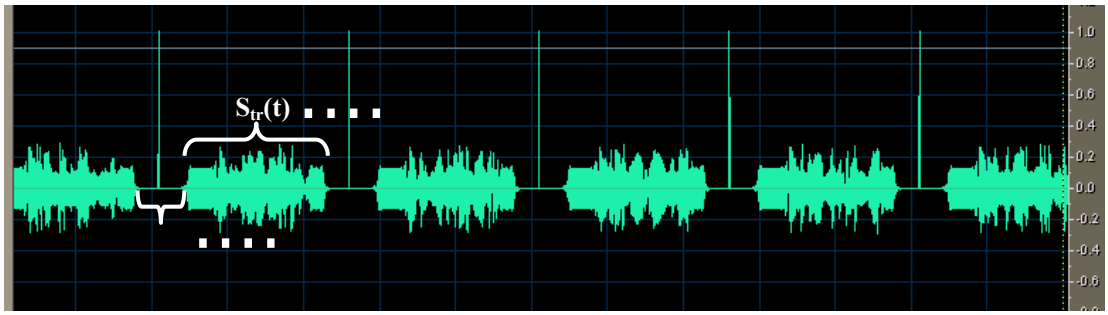


Fig. 86.- Entrada al canal

Cada trama, ***$S_{APK}(t)$***, se puede descomponer en dos tramos bien diferenciados:

- ***$S_{\delta}(t)$***: que corresponde a la señal piloto, sin modular.
- ***$S_{tr}(t)$***: que representa la trama modulada.

Podemos concluir, por tanto, que la ***señal completa***, se puede expresar mediante dos sumandos principales:

- $S_a(t) + S_b(t)$;

Si aplicamos la ***propiedad lineal*** de la ***transformada de Fourier***, podemos descomponer el espectro de nuestra señal a transmitir en las transformadas de estas dos señales temporales[†2].

Así, para ***$S_a(t)$*** tenemos una ***secuencia de deltas de Dirac***, desplazadas entre sí un ***intervalo de trama, T_{tr}***. Visto analíticamente:

- $S_a(t) = \sum \delta(t-k\cdot T_{tr}-t_0)$; (con k=0,1,2...)

donde ***t_0*** es el ***retraso inicial*** y al encontrarse el primer 'δ' en la muestra 31, el valor de t_0 será de 31/44100 segundos, teniendo en cuenta que muestramos a F_s=44100 hz.

Ya dijimos que no nos vamos a explayar con aburridos cálculos matemáticos, por eso vamos a deducir ***cualitativamente*** el ***aspecto*** de nuestra señal ***$S_a(t)$ en la frecuencia***:

- Todos sabemos que un tren de deltas de Dirac infinito en el tiempo, con un período de T_{tr} segundos, se corresponde con otro ***tren de Deltas de Dirac en la frecuencia***, con un ***período*** inverso igual a ***$1/T_{tr}$***. Cuyo valor en hz. es el siguiente[†2]:
  - nº de muestras por trama: N_{tr} = 501
    - señal piloto: 81(piloto)
    - CRC y LONGITUD: 40
    - Inicio & Fin de Trama: 120
    - Datos: 200
    - Retraso: 60
  - T_{tr} = 501 muestras / 44100 (muestras/sg) = 0.0114 sg.
  - f_{tr} = 1 / Ttr = 88.024 hz.

- Pero este ***tren de Deltas*** es ***finito***. Esto equivale a multiplicar nuestro tren infinito por una ***función rectángulo***, cuyo espectro en frecuencia es una función sampling. Así tendremos en frecuencia la convolución de ambos espectros.Lo que resulta en conjunto un ***tren*** de ***funciones parecidas*** a una ***sampling***, separadas ***cada 88 hz***. aproximadamente.
- Por otra parte, el ***retraso t_0*** se traducirá como un ***retraso en la fase*** de la señal, por lo que no afectará a nuestra gráfica, que sólo muestra el módulo de dicha señal a transmitir.

Por otro lado, ***$S_b(t)$*** representa el ***sumatorio*** de todas las ***tramas***, es decir:

- $S_a(t) = \sum S_{tr}(t-k\cdot T_{tr}-t_0)$; (con k=0,1,2... y $t_0 = 81/44100$ sg.) ;

Cualitativamente, el ***espectro*** de esta señal, ***$S_b(t)$***, lo podemos entender, en términos generales, de la siguiente manera:

- La señal ***$S_{tr}(t)$*** se repetirá ***cada intervalo de trama***. Dicha señal, $S_{tr}(t)$, es un ***sumatorio*** de ***pulsos raíz de coseno alzado***, ***$g_T(t)$***, separados ***cada intervalo de símbolo***, ***T***.
- Como ya hemos dicho y sabemos de antemano, el ***retraso temporal*** sólo afecta a la fase, por lo que, en dominio de la frecuencia, podremos reescribir la señal ***$S_b(f)$***, como la ***transformada del Filtro Transmisor***, ***$G_T(f)$***, por dos sumatorios anidados y multiplicado por las amplitudes de los símbolos.
- Como ***sólo nos interesa el ancho de banda*** de la señal, para contrastarlo con los cálculos que hicimos sobre dicho parámetro, obviamos los cálculos matemáticos y concluimos que nuestra señal ***$S_b(t)$ tendrá el mismo ancho espectral que el pulso transmisor***.
- Así, para los parámetros por los que hemos optado, obtenemos los siguientes ***cálculos teóricos*** sobre el ancho de banda de la señal a transmitir:
  - ***Tasa de baudios*** del sistema, ***D***:
    - $D = F_s / N = 4410$ hz.
  - ***Ancho*** de ***nuestra señal*** en ***banda base***:
    - $BW_{LP} = D\cdot(1+\alpha)/2 = 3748.5$ hz.
  - ***Ancho*** de ***nuestra señal modulada*** (***a 11025hz***):
    - $BW_{BP} = 2\cdot BW_{LP} = 7497$ hz.
  - ***Frecuencia de corte superior*** de nuestro espectro:
    - $f_{max} = 14774$ hz.
  - ***Frecuencia de corte inferior*** de nuestro espectro:
    - $f_{min} = 7276.5$ hz.

A continuación, pasamos a mostrar la ***figura*** de dicho ***espectro a transmitir*** por el canal de audio, y comprobaremos que, efectivamente, y de una manera muy aproximada, nuestros cálculos teóricos no difieren mucho de la realidad empírica:

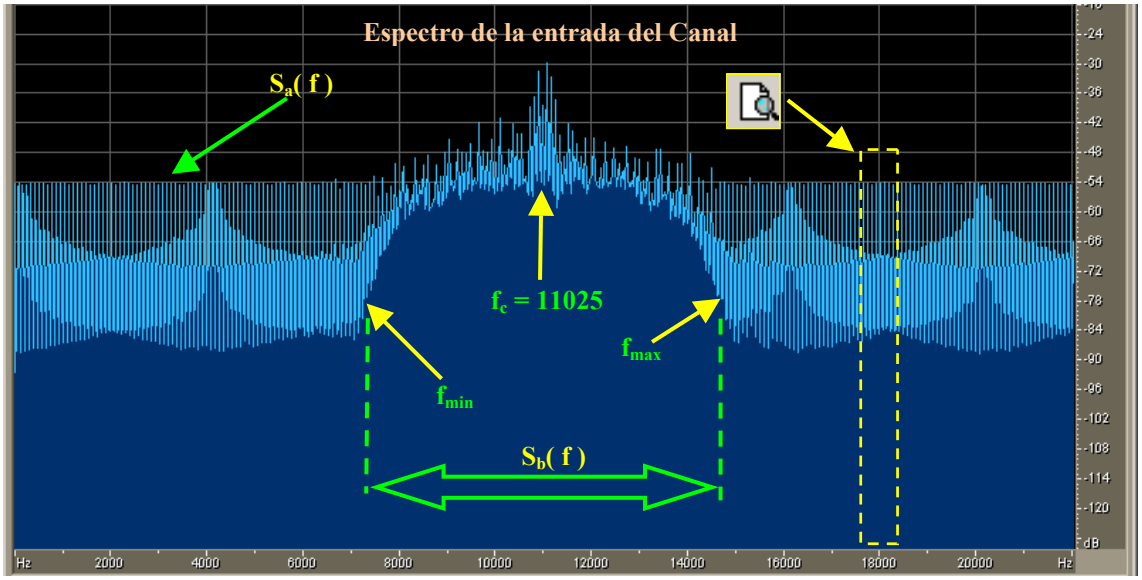


Fig. 87.- Espectro señal a transmitir

Se puede apreciar nuestra ***señal centrada*** en torno a la ***frecuencia de portadora*** de 11025 hz., y la ***figura espectral*** de nuestra señal $S_b(f)$, cuyo espectro básicamente coincide con el del ***pulso raíz de Nyquist***, aunque ***distorsionado***, como adelantamos, por los sumatorios.

Por otra parte, también podemos apreciar el ***tren de samplings distorsionadas*** en todo el espectro, correspondiente al espectro de $S_a(f)$. Si realizamos un *'zoom'*, podremos evidenciar que la frecuencia de repetición de dichas señales, f_{tr}, es una buena aproximación de nuestros cálculos teóricos.

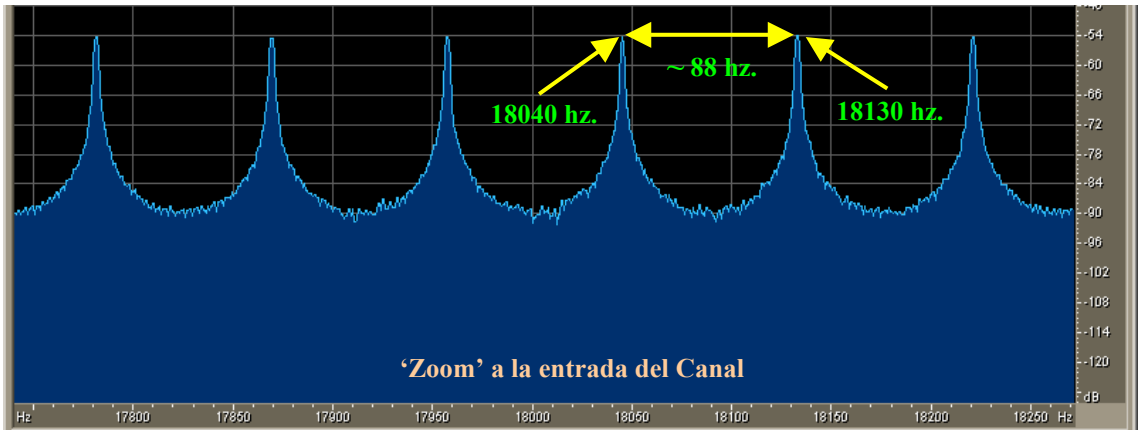


Fig. 88.- zoom al espectro a transmitir

Pasemos a mostrar la ***figura*** de la ***señal recibida***, es decir, justamente a la salida del canal:

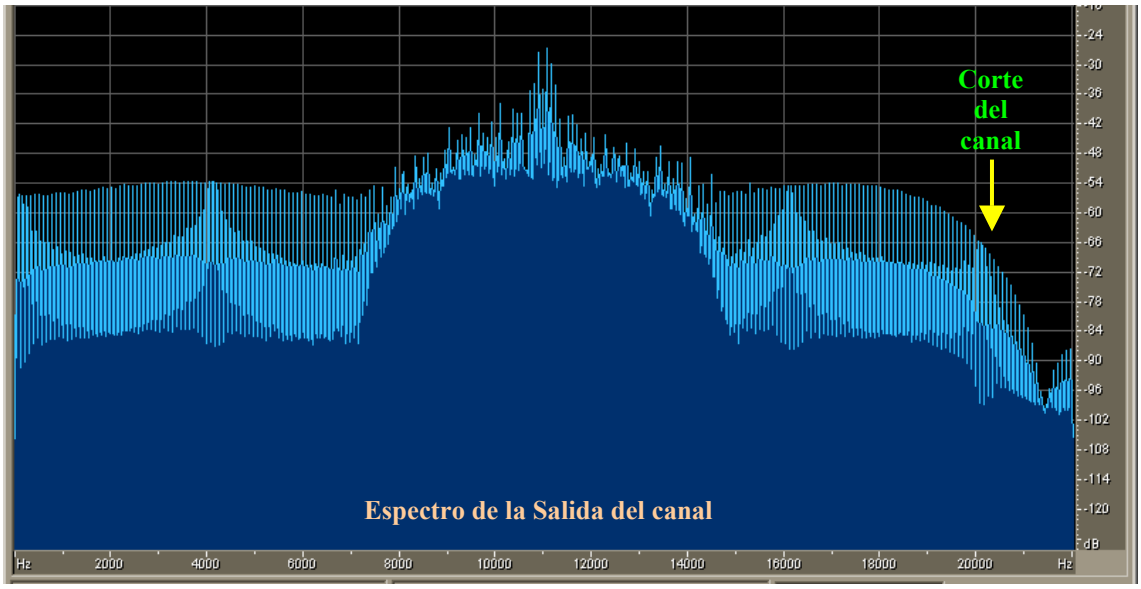


Fig. 89.- Espectro señal recibida

Se puede comprobar perfectamente el ***corte*** que introduce el ***canal***, en torno a los ***20 khz***. Además, tenemos que resaltar que la ***amplitud*** está representada en ***decibelios***, por lo que el ***producto*** de la respuesta en frecuencia del canal por el espectro de la señal de entrada a éste, ***se transforma en una suma*** en la escala de decibelios, por eso se aprecia claramente la figura de la respuesta en frecuencia del canal, que obtuvimos en el capítulo III.1, y encima de ésta, nuestra señal recibida.

También es perceptible la ***distorsión en amplitud*** que introduce el canal, con sus diversas ondulaciones en todo el rango de frecuencias.

Pasemos a mostrar el ***aspecto*** de nuestra entrañable ***'trama 36'***, una vez que se ha producido en el receptor la identificación y separación en tramas:

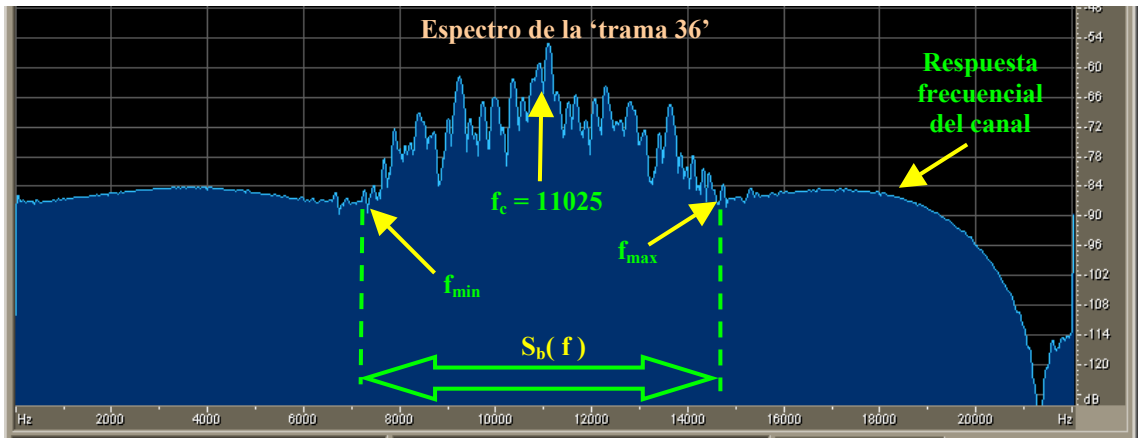


Fig. 90.- Espectro trama 36

Al tratarse del espectro de una sola trama, *no aparece* ya la componente, $S_a(f)$, correspondiente a las señales piloto, pues dicho tren ha sido disuelto a estas alturas de procesamiento en el receptor.

Ahora sí se distinguen más nítidamente la ***respuesta en frecuencia del canal*** y el ***espectro*** correspondiente a la trama modulada, $S_b(t)$.

Se puede apreciar que nuestra señal no se ve recortada por el canal. No tenemos por tanto pérdidas de componentes espectrales en nuestra señal.

A continuación mostramos la misma ***'trama 36'***, a la ***salida del ecualizador***:

Fig. 91.- Espectro trama 36 ecualizada

Podemos ratificar visualmente que el conjunto ***canal + ecualizador*** se comporta como una ***δ(t) ideal***, con una ***respuesta prácticamente plana*** en el dominio de la frecuencia, donde el corte que existía ha 20 khz. ha desaparecido.

Lamentablemente, con esta gráfica no podemos confirmar la ecualización de los datos, aunque es seguro que si el ecualizador ha eliminado la distorsión en amplitud del canal, entonces las componentes en frecuencia han sido menguados por el mismo factor de atenuación en todo el espectro y la ecualización se ha ejecutado con éxito.

Las siguientes dos ***imágenes*** que vamos a mostrar son los de las componentes en fase y cuadratura, a la ***salida del oscilador***:

- ***Componente en fase***:

Fig. 92.- Espectro Salida oscilador. Rama fase

- ***Componente en cuadratura***:

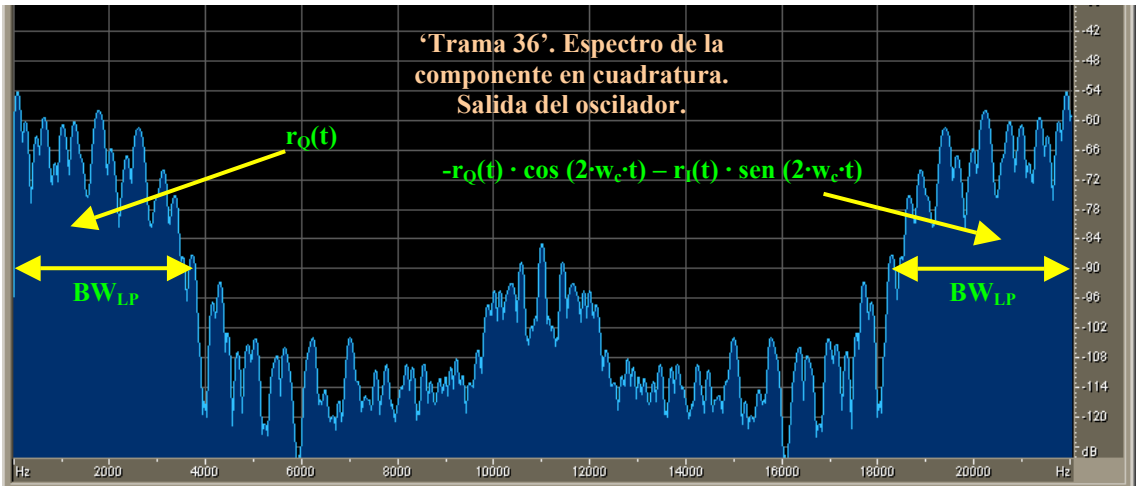


Fig. 93.- Espectro salida oscilador. Rama cuadratura

Se pueden distinguir en ambas gráficas los ***espectros en banda base*** correspondientes a las señales $r_I(t)$ y $r_Q(t)$, respectivamente, que ya dijimos que era lo que nos interesaba recuperar. Por otro lado también aparecen una ***mezcla*** de los ***espectros*** de ambas señales, desplazadas $2 \cdot f_c$ ***hz***., y las señalamos con sus respectivas expresiones temporales.

Por último, mostramos ambas señales, a la ***salida del Filtro Receptor*** paso de baja, donde ya no existen componentes centradas en $2 \cdot f_c$, debido al filtrado del pulso raíz de coseno alzado.

- ***Espectro*** de la ***envolvente de fase***:

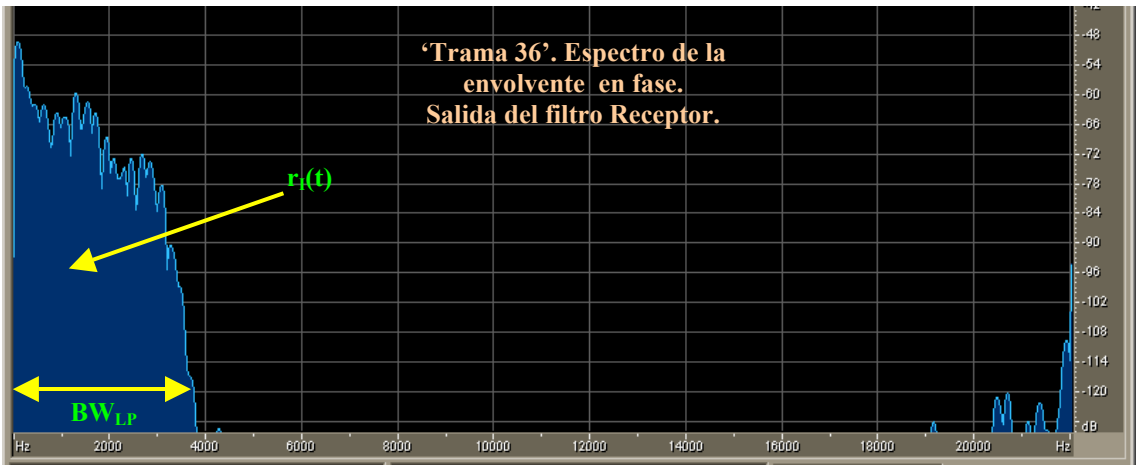


Fig. 94.- Espectro envolvente fase

- ***Espectro*** de la ***envolvente en cuadratura***:

Fig. 95.- Espectro envolvente Cuadratura

Nota: Para evitar confusiones, nos gustaría recalcar que, aunque hallamos indicado los distintos espectros de frecuencia, en estas últimas gráficas relativas a las componentes en fase y en cuadratura, mediante señales temporales, sólo queremos reflejar que las transformadas de dichas expresiones temporales dan lugar a estos espectros en frecuencia, centrados tanto en la banda base como a 2·fc hz. del origen.

III.2.4.- Anomalías en nuestro sistema

En este apartado vamos a enumerar brevemente los ***contratiempos principales*** con los que nos hemos ido encontrando en nuestro proyecto, y las ***soluciones*** que hemos ido adoptando para paliar sus efectos. Finalmente acabaremos centrando nuestro estudio en el problema principal: la ***ausencia de sincronismo***.

- El primer problema con el que nos hemos encontrado es la ***distorsión en amplitud*** que introduce el canal, pues el módulo de su respuesta en frecuencia presenta un pequeño rizado. Este contratiempo lo hemos solucionado introduciendo un ***ecualizador*** a la salida del canal para conseguir, en conjunto con éste, un sistema ideal[†2] [†4].
- Otro problema no menos importante es la ***interferencia entre símbolos (ISI)***, debido a la no idealidad del canal, pues éste es limitado en banda y no podíamos transmitir con pulsos conformadores rectangulares, cuyo espectro lo recortaría el canal en frecuencia. Así, tuvimos que usar ***pulsos raíz de coseno alzado*** para los filtros transmisor y receptor. De todas maneras, el ***ecualizador*** también nos atenúa un poco el efecto de esta posible interferencia entre símbolos[†1].
- El último problema, y el más presente e importante de todos, es la ***ausencia de sincronismo***. Este tema, debido a su trascendencia en los resultados de nuestro proyecto, lo vamos a tratar de forma más detallada en el siguiente subapartado, analizando sus causas, cuantificándolo y aportando distintas vías alternativas para dar solución a este problema.

EL PROBLEMA DEL SINCRONISMO

- Causas y consecuencias

Ya hemos comentado varias veces que el ***formato de transmisión*** que usaremos será el sistema ***PCM***, con las siguientes características:

- Fs = 44100 muestras / sg.
- NB = 16 bits/muestra.
- Monocanal.

El problema del muestreo viene a raíz de que los ***relojes*** del ***transmisor*** y del ***receptor*** son ***ligeramente diferentes***, o sea que ninguno de los dos tiene una frecuencia exacta de 44100 muestras por segundo, sino que tendrán una pequeña variación:

- Reloj Transmisor: $44100 + \delta_T$
- Reloj Receptor: $44100 + \delta_R$

, donde $\boldsymbol{\delta_R}$ y $\boldsymbol{\delta_T}$ son cantidades muy ***pequeñas*** en relación al a la frecuencia base de 44100 hz., y dependen, en todo caso, de la calidad de las tarjetas de sonido empleadas.

La ***principal consecuencia***, y más intuitiva, es el d***esplazamiento*** de las ***muestras*** en el receptor. Eso significa que cada cierto número de muestras, perderemos una de

ellas o ganaremos una muestra espuria, lo que contribuye a que los símbolos (y , como resultado, los bits) transmitidos y recibidos no sean iguales y exista error en la detección de alguno de ellos.

Otra consecuencia de este desplazamiento nos afectó a la hora de ***decidir*** sobre qué ***ecualizador*** usar, ya que ***a simple vista***, ***parecía*** que nuestro canal era lineal y variante con el tiempo (***LTV***) [†4]. Los motivos que nos condujeron a esta hipótesis, fueron los resultados del estudio de la respuesta impulsiva del canal, cuando transmitimos un tren de pulsos y observamos la señal en recepción. Nuestro *'script'* en Matlab fue el siguiente:

- transmisión de un tren de deltas de Dirac

```
% Inicialiazamos la variable;
    s=0;
% Tren de deltas espaciadas cada 7000 muestras
    s(1:7000:3000000) = 1;
% Grabamos la señal en el archivo 's.wav',con la ayuda de 'wavwrite'
% Añadimos ceros para simular silencio y centrar lo que vamos a transmitir.
   wavwrite([zeros(1,70000),s,zeros(1,70000)],44100,16,'s.wav');
% Transmisión a 44100 hz y 16 bits/muestra
% Utilizamos la grabadora de 'Cool Edit'
    sound (s, 44100,16);
```

En primer lugar vamos a mostrar el ***tren de deltas ideal***, antes de ser transmitido por el canal, es decir, el contenido del archivo ***'s.wav'***:

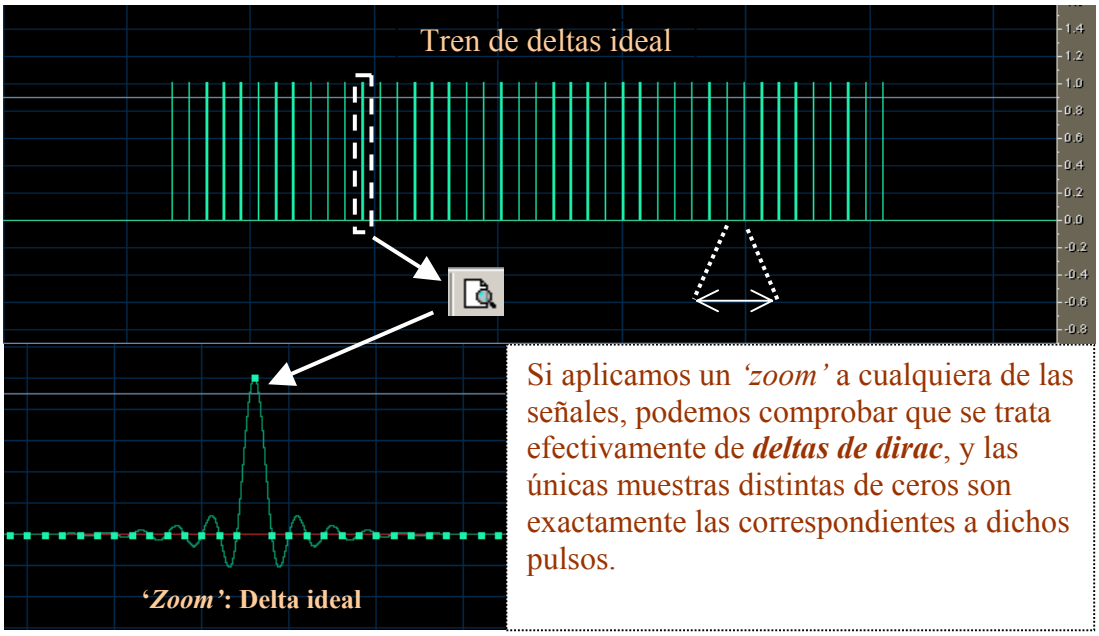


Fig. 96.- Tren de deltas ideal

Al recibir la señal, y editarla con 'Cool Edit', vimos que tenía el siguiente aspecto:

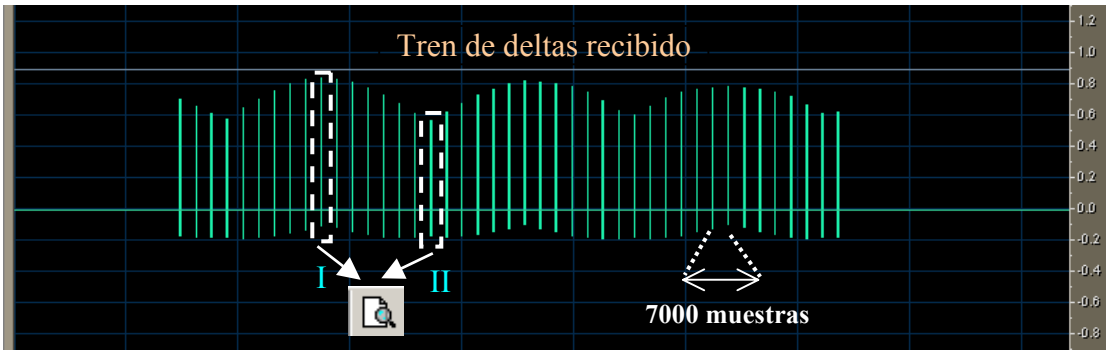


Fig. 97.- Tren de deltas recibido

En un ***primer vistazo*** se puede apreciar la ***atenuación*** que introduce el canal, pues los pulsos recibidos no alcanzan el valor máximo unidad con el que fueron transmitidos.

Otra cosa que salta a la vista es la aparentemente ***distinta forma*** que tiene el ***canal*** de ***tratar*** a los ***pulsos*** que pasaran por éste, ***dependiendo del tiempo*** en que éstos fueran transmitidos.

Sin embargo, si hacemos un *'zoom'* para ver de cerca a la señal de mayor amplitud, y otro para observar la señal de menor valor, observaremos lo siguiente:

- Para el ***pulso de máxima amplitud***:

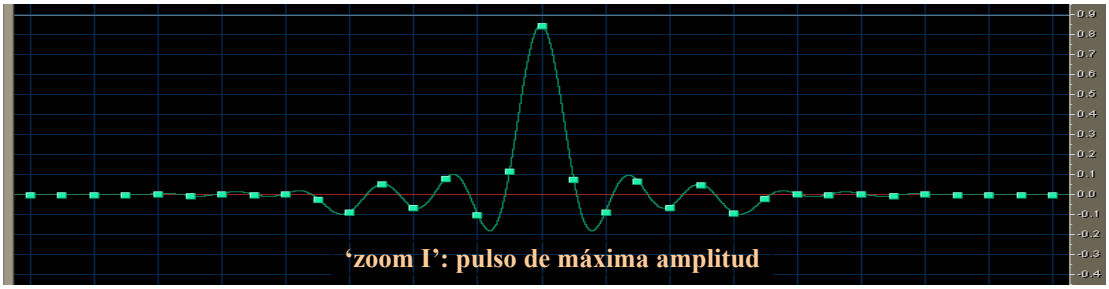


Fig. 98.- Respuesta máx. amplitud

En la anterior gráfica se puede apreciar que la ***envolvente*** es ***idéntica*** a la transmitida, a pesar de que los instantes de muestreo no se realicen exactamente en los cruces por cero de la señal. Por tanto se puede observar un ***desplazamiento*** de las ***muestras***, aunque la muestra correspondiente a la señal máxima está situada en el pico de dicha envolvente.

- Para el ***pulso de mínima amplitud***:

Fig. 99.- Respuesta min. amplitud

En este caso también se trata de la misma envolvente, pero el ***desplazamiento*** de las muestras ***se acentúa*** más, de tal manera que no existe una muestra en el pico de la envolvente.

Por este motivo, nos parecía que las amplitudes eran distintas en función del tiempo en que se transmitieran, por lo que en un principio ***pensamos*** que se trataba de un ***canal LTV***, con las consecuencias que hubiera acarreado tal incidente, pues tendríamos que haber usado ***ecualizadores adaptativos***, mucho más complejos que los que realmente hemos usado[†4].

Sin embargo, el problema no iba más allá de un mero ***desplazamiento de las muestras***, debido a la ya mencionada ***falta de sincronismo***.

Por último, también existe una ***última secuela*** de este ***desplazamiento*** de las muestras en los ***osciladores*** del transmisor y del receptor, pues éstos funcionarán a distinta frecuencia. Esto se reflejará en un ***desplazamiento del espectro*** respecto a los 11025 hz., donde idealmente se encuentra ubicada nuestra señal M-APK de cada trama.

Sin embargo, este desplazamiento en frecuencias es mínimo y no supone problema alguno para el funcionamiento de nuestro sistema.

- Cuantificación del desajuste de la frecuencia de muestreo

Para ***determinar el desajuste*** existente entre dos relojes, podemos ***partir*** de una ***onda sinusoidal*** con una frecuencia de ***11025 hz.***, submúltiplo de nuestra frecuencia de muestreo Fs = 44100hz.

Esta frecuencia de 11025 hz. además de coincidir con nuestra frecuencia de portadora, tiene un ***distribución de muestras*** que como sabemos es muy interesante para nuestro objetivo, pues sólo tiene valores ***1,-1 y 0***.

Para ilustrar esto, veamos un ***ejemplo*** de lo que ocurre en nuestra tarjeta de sonido ***SoundBlaster 128PCI*** en configuración full-duplex[†8] (permite reproducción y grabación simultánea, pero el reloj interno es distinto para cada caso[†7] [†8], y se aprecia la falta de sincronismo).

El *'script'* creado en Matlab para el caso es el siguiente:

- Generación de una ***senoide de 11025 hz***.

```
% Frecuencia de Muestreo
   Fs = 44100;
% Frecuencia de oscilación
   fc=11025;
% Número de períodos a dibujar
   N=5;
% Instantes de muestreo
   t=0:1/Fs:N/fc;
% Senoide
   s = sin(2*pi*fc*t);
% Representación de la señal
   stem(t,s);
```

Veamos, por ejemplo, 5 ciclos de la senoide creada:

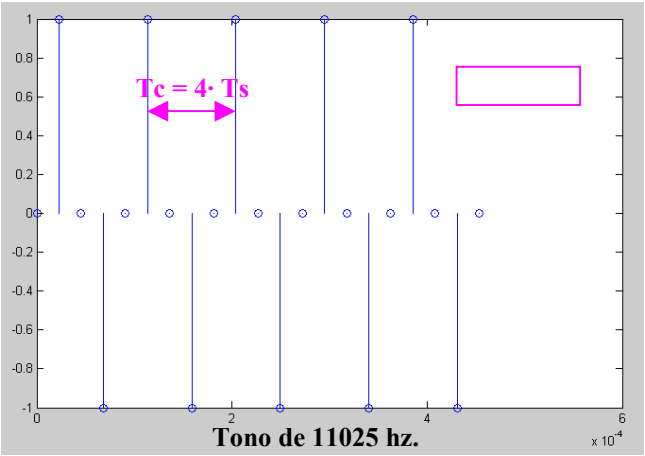


Fig. 100.- Tono de 11025 hz.

Con esta peculiar forma de representación de las muestras (1,-1 y 0), cualquier pequeño desajuste de las muestras se apreciará inmediatamente.

El siguiente paso es ***transmitir dicha señal***, y observar su aspecto en recepción. Pero para observar el efecto de la ausencia de sincronismo, necesitamos ***muchos más ciclos de reloj***, ya que, como hemos anunciado anteriormente (y comprobaremos en este experimento), cada 2 segundos, esto es, cada 44100 muestras/sg · 2 sg = 88200 muestras, se ganará o perderá una muestra.

La ***señal*** que recogemos en ***recepción***, con la grabadora del 'Cool Edit' es la siguiente:

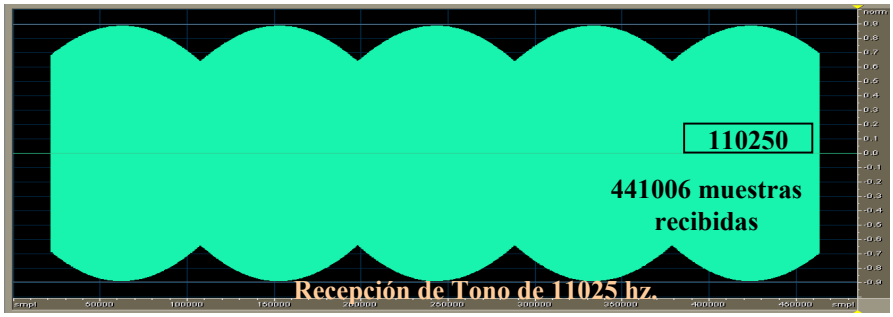


Fig. 101.- Tono de 11025 recibido

En este caso hemos tomado un número de ciclos ***N=110250 ciclos*** de portadora. Esto es, hemos tomado ***10 sg. de señal***, que en términos de muestras, vienen a ser: 44100 muestras/sg ·10sg = ***441001 muestras***.

Esta ***sinusoide de envolvente sinusoidal*** (distinta a la envolvente constante de magnitud unidad que transmitimos) aparece a causa de las ***diferencias*** entre las ***frecuencias de muestreo*** en transmisión y recepción. Las muestras comienzan a moverse y ya no aparecen en las posiciones máximas (1), mínimas (-1) y nulas (0).

Para hallar ese desajuste entre ambos relojes de una forma bastante aproximada, procederemos de la siguiente forma:

- Tomamos frecuencias de muestreo en torno a la frecuencia base, es decir:

  - Fs = ...44099.5, 44099.6, 44099.7,...,44100.1, 44100.2,...,44100.5,...

- Obtenemos, para cada Fs, la señal sinusoidal, en Matlab, por ejemplo.
- Aquella cuyo aspecto coincida con la señal recibida, es la que nos marcará el desfase entre el transmisor y el receptor.

En nuestro caso, conseguimos el máximo parecido para las frecuencias de 44095.5 o 44100.5 hz., por lo que el ***desfase*** entre el reloj del transmisor y el del receptor es de ***0.5 hz.*** Veámoslo:

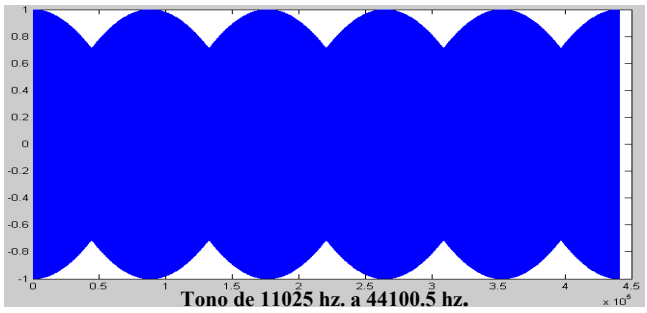


Fig. 102.- Tono de 11025 hz. muestreado a 44100.5 hz.

Para saber si el receptor muestrea 0.5 hz. más rápido o más lento basta con mirar el número de muestras que recibimos en 'Cool Edit', sin contar, claro está, con las muestras correspondientes al silencio previo y posterior a la grabación de la señal:

- Muestras transmitidas: 441001 muestras. (desde t=0 hasta t = 10 sg).
- Muestras recibidas: 441006 muestras recibidas.

Por tanto, hemos demostrado que el ***reloj del receptor*** es ***0.5 hz. más rápido*** que el del transmisor, lo cual concuerda con el hecho de que en 10 sg. hallamos ganado 5 muestras, y ratificamos lo que dijimos con anterioridad: ***cada 2 segundos*** ganamos una ***muestra espuria*** en recepción. Analíticamente:

- Fs' = Fs + δ hz. ; ***δ = 0.5 hz.***
  Fs': frecuencia de muestreo del receptor.
  Fs: frecuencia de muestreo del transmisor.

Además, si contamos el número de muestras de la sinusoide generada en matlab a la frecuencia de muestreo de 44100.5 hz., veremos que obtenemos 441006 muestras, como era de esperar.

Vamos a tratar de ir un poco más lejos en este estudio. ***Tratemos de averiguar la expresión de la envolvente*** a partir de este desfase de frecuencias:

El ***tono recibido***, $S_{rx}(t)$, en su ***forma continua***, tendrá la siguiente expresión analítica, obviando cualquier distorsión del canal:

- $S_{rx}(t) = \text{sen}(2\cdot\pi\cdot f_c\cdot t) \rightarrow (\text{para } t = n\cdot Ts')$;

Si ***muestreamos*** dicha señal recibida en recepción, ***cada Ts'*** (donde Ts' es el tiempo de muestreo en recepción), obtendremos la siguiente ***versión muestreada***:

- $S_{rx}(n\cdot Ts') = \text{sen}(2\cdot\pi\cdot f_c\cdot n\cdot Ts')$;

Vamos a tratar de ***relacionar Ts'*** con ***Ts*** (tiempo de muestreo en transmisión), basándonos para ello en la relación conocida de sus inversas:

- Fs' = Fs + δ hz. ; siendo δ = 0.5 hz.

- $Ts' = 1 / Fs' = 1 / (Fs + \delta) = 1 / Fs \cdot (1+ \Delta)$;

siendo la constante de proporcionalidad: $\Delta = \delta/Fs = 0.5/44100 = 1/88200 \rightarrow 0$

Como Δ tiende a cero, podemos usar la conocida ***aproximación lineal de Taylor***[†3] para los polinomios:

- $(1+\Delta)^{-1} = (1-\Delta)$

De esta manera, obtenemos la ***relación entre Ts' y Ts***:

- $Ts' = Ts\ (1-\Delta)$

Lo cual es lógico, pues si Fs'>Fs → Ts'<Ts, esto es, si el receptor muestrea a mayor frecuencia que el transmisor, su tiempo de muestreo, obviamente, será menor. Es más, cada múltiplo de n = 88200 muestras, esto es, cada 2 sg. , se ve analíticamente que se gana una muestra. Por ejemplo, usando la ecuación anterior:

- para n = 88200 → 88200· Ts'= 88199·Ts (hemos ganado una muestra en recepción).

Si retomamos la expresión del ***tono recibido***, y la escribimos en ***función*** de ***Ts***, obtendremos lo siguiente:

- $S_{rx}(n\cdot Ts') = \text{sen}\,[2\cdot\pi\cdot f_c\cdot n\cdot Ts'] = \text{sen}\,[\,2\cdot\pi\cdot f_c\cdot n\cdot Ts\,(1-\Delta)\,]$;

Si tenemos en cuenta la expresión de la ***frecuencia digital***:

- $\omega_c = 2\cdot\pi\cdot f_c\cdot Ts = \pi/2$

lo cual también es razonable, ya que el rango digital está normalizado por la frecuencia de muestreo, y la ***señal digital*** es ***periódica de período 2·π***, de tal manera que ***2·π se corresponde con Fs***. De este modo, $f_c = Fs/4$, ***se corresponderá*** con la cuarta parte de 2·π, esto es, con ***π/2***, que es precisamente el valor de ω_c.

Si seguimos desarrollando:

- $S_{rx}(n\cdot Ts') = \text{sen}\,[\ (\pi/2)\cdot n\,(1-\Delta)\,] = \text{sen}[n\cdot\pi/2 - n\cdot\pi\cdot\Delta/2]$

Utilizando ***relaciones trigonométricas***[†3], logramos llegar a que el ***tono recibido***, muestreado δ hz. más rápido en recepción, adquiere la siguiente expresión:

- $S_{rx}(n\cdot Ts') = \underbrace{\cos[n\cdot\pi\cdot\Delta/2]\cdot\text{sen}[n\cdot\pi/2]}_{\textbf{I}} - \underbrace{\text{sen}[n\cdot\pi\cdot\Delta/2]\cdot\cos[n\cdot\pi/2]}_{\textbf{II}}$;

I — **muestras impares (n=1,3,5,...)**

II — **muestras pares (n=0,2,4,...)**

La ***señal transmitida*** es un tono muestreado en recepción, ***cada Ts*** sg.:

- $S_{tx}(n\cdot Ts) = \text{sen}\,[2\cdot\pi\cdot f_c\cdot n\cdot Ts] = \text{sen}\,[n\cdot\pi/2]$;

se trata de un ***seno de amplitud constante***, que como podemos apreciar, sólo tiene ***valores 1,-1 y 0***.

Sin embargo, la ***señal que recibimos***, $S_{rx}(n\cdot Ts')$ tiene ***dos sumandos***, que hemos denominado ***I*** y ***II***. Ambas constan de una ***señal sinusoidal modulada por otra señal sinusoidal***[†2].

Las ***envolventes*** de las ***señales I*** y ***II*** serán, respectivamente:

- $\cos[n\cdot\pi\cdot\Delta/2]$ y $\text{sen}[n\cdot\pi\cdot\Delta/2]$

Las ***señales portadoras*** serán , respectivemnte, en ***I*** y ***II***:

- $\text{sen}[n\cdot\pi/2]$ y $\cos[n\cdot\pi/2]$

Dado el carácter discreto del ángulo de ambas señales portadoras, en los instantes en que la señal I sea distinta de cero, la señal II se anulará, y viceversa. Por tanto:

- La ***señal I*** aparecerá en los ***instantes impares*** de muestreo.
- La ***señal II*** aparecerá en los ***instantes pares*** de muestreo.

Dibujemos el aspecto de ambas ***envolventes***, para ver si coincide con los resultados empíricos obtenidos anteriormente. El 'script' empleado será el siguiente:

- Obtención de las ***envolventes del tono recibido***:

```
% δ:desplazamiento de frecuencia
   delta=0.5;
% Frecuencia de muestreo
   Fs = 44100;
% Δ: Desplazamiento temporal
   Delta=delta/Fs;
% frecuencia de portadora
   fc=11025;
% Número de ciclos
   N=110250;
% Instantes de muestreo
   t=0:1/Fs:N/fc;
% Envolventes
   s1 = sin(2*pi*fc*t*Delta);
   s3 = -s1;
   s2 = cos(2*pi*fc*t*Delta);
   s4 = - s2;
% Representación de las envolventes:
% Rojo: envolvente I, Azul:envolvente II
   plot(t,s1,'bo',t,s2,'ro',t,s3,'bo',t,s4,'ro');
```

El resultado del 'script' es el siguiente:

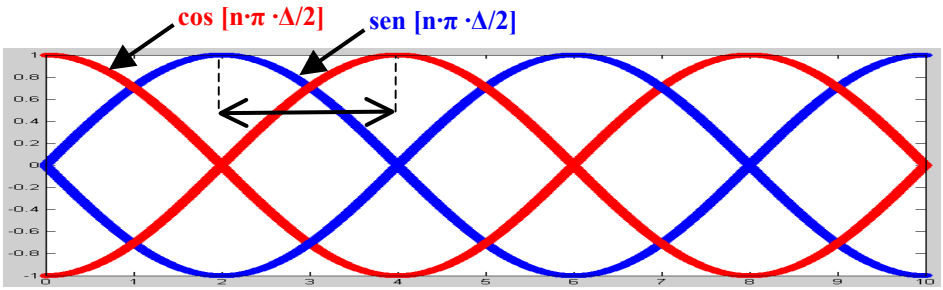


Fig. 103.- Envolventes tono 11025 hz. recibido

Si modulamos ambas señales, a una frecuencia mucho mayor, podremos observar, que el aspecto es exactamente igual al de nuestro tono recibido, $S_{rx}(n\cdot Ts')$.

Antes de acabar con el ejemplo, y a modo de curiosidad, nos gustaría comentar, que la ***distancia entre cada máximo***, es exactamente igual a un ***cuarto de ciclo*** de la envolvente, esto se produce para la siguiente condición:

- $n \cdot \pi \cdot \Delta/2 = \pi/2 \rightarrow n = 1/\Delta = 88200$ muestras (cálculo aproximado)

Entonces, los ***máximos*** en $S_{rx}(n \cdot Ts')$ se producen aproximadamente ***cada 2 sg***., o bien, cada ***88200 muestras***, lo cual es congruente si nos fijamos en el eje temporal de la figura del tono obtenido mediante 'Cool Edit'. Esto da bastante fiabilidad a nuestros cálculos.

- Solución al problema del sincronismo

Dar una ***solución*** a este importante problema sería fácil si los relojes fueran modificables vía ***hardware*** con algún tipo de ***potenciómetro*** o algo similar, de forma que los ajustes serían definitivos. Sin embargo, esta ***posibilidad*** queda totalmente ***descartada***, pues, actualmente, todas las ***tarjetas*** tienen un ***alto grado de integración***, lo cual nos impide cualquier tipo de intervención externa[†7].

Ello nos obliga a recurrir a otras opciones. Entre ellas, podemos destacar la ***posibilidad*** que tienen muchas ***tarjetas*** de ***funcionar*** con ***distintas frecuencias***. Por ejemplo, supongamos que tenemos una ***tarjeta Soundblaster bien ajustada*** y una ***tarjeta de mala calidad*** que no dispone realmente de 44100 hz. como frecuencia base, sino de 44075, 19 hz. En estos casos tan desfavorables, una buena solución es aprovechar la capacidad de la tarjeta Soundblaster para modificar su frecuencia base: puede transmitir o recibir (según sea transmisora o receptora) a una frecuencia de 44075 hz., pues su resolución es de 1 hz., es decir, que puede funcionar perfectamente a 44099 hz., 44098 hz.,44097 hz., etc, hasta llegar a lo que nos interese.

Cuando ***nos hemos adaptado a la tarjeta de mala calidad***, ya tenemos la seguridad de que el desajuste existente no excede de 1 hz., aunque, claro está, ***habremos perdido*** prestaciones en cuanto a ***velocidad***, ya que la frecuencia de muestreo usada ya es menor que los 44100 hz. iniciales.

Por lo tanto, ***la situación es la siguiente***: el transmisor y el receptor tienen ***relojes*** cuyas frecuencias de muestreo son ***parecidas***, aunque ***no*** están ***totalmente sincronizados***, dado que existe una pequeña diferencia de algo menos de 1 hz. Es con esto con lo que el proyecto va a tener que funcionar y no lo podemos modificar.

Sin embargo, aún nos queda una opción. Con un desajuste menos que 1 hz., podemos observar que durante intervalos de tiempo algo menores que 1 segundo, la sincronización no es perfecta, pero sí muy buena. A saber, podemos aprovechar este hecho para transmitir nuestros datos en los intervalos de tiempo en donde existe sincronización, olvidándonos de las colas de tiempo en las que el desajuste se hace visible e importante. Por lo tanto, ***disminuiremos*** el ***tamaño*** de las ***tramas*** que tenemos que transmitir, para que el ***problema del sincronismo nos afecte lo menos posible***.

Sin embargo, ***nuestro propósito*** será formar ***tramas de longitud máxima***, dentro de lo permisible, de forma que optimicemos el diseño del transmisor y del receptor con el ***menor número de bucles*** posible, ya que los bucles, se ejecutarían más veces, cuanto menor fuera el tamaño de las tramas, haciendo mucho más lento el sistema (en Matlab, las ***llamadas a funciones*** son un factor clave en la velocidad de las aplicaciones) en cuanto a procesamiento de la señal, debido también a que aumentaría el número de

campos de control, y las tramas serían menos eficientes en cuanto a información transmitida por cada una de ellas.

Nota: Podría ocurrir que este ***desajuste*** fuera ***realmente pequeño*** y no fuera necesario intervenir para encontrar una solución. Sería el caso de tener ***2 tarjetas de sonido de muy buena calidad***. No es habitual, y ***no es nuestro caso***, pero si lo fuera, el diseño y la realización del sistema de transmisión de datos se simplificaría muchísimo, pues desaparecerían muchos inconvenientes, y podría aumentar la longitud de las tramas, el número de símbolos usados, etc...

Elección de un BDT óptimo

Por último, una vez que conocemos todos la dimensión de todos los campos de nuestras tramas, nos gustaría comentar un parámetro de diseño que hemos almacenado en la variable ***BDT*** de nuestro programa: el ***número de bytes del campo DATOS*** de cada trama. Aunque se trata de un ***parámetro totalmente configurable*** en nuestro sistema, ***nos interesa*** aumentar su valor al ***máximo permisible***, para reducir así el número de tramas, y con ello el número de bucles de procesamientos y de información de control de dichas tramas. Ya hemos reiterado que es precisamente la ***falta de sincronismo*** la que nos obliga a reducir el tamaño de las muestras a enviar por cada trama. Para ello establecemos como ***desplazamiento máximo admisible*** el ***1 %*** del desplazamiento completo de una muestra. Esto, como bien sabemos, nos lleva a la conclusión de que podemos transmitir ***882 muestras por cada trama***. Si consideramos un valor crítico de ***M=256 símbolos***(en el sentido de mayor probabilidad de error en la decisión de cada símbolo), y un valor típico de ***N=10 muestras/símbolos***, podemos a estas alturas averiguar el número de muestras de control necesarios para cada trama, y así poder intuir un ***valor óptimo para la variable BDT***:

- Los datos de control serán , como bien intuimos:
    - ***retraso***: 60 muestras (variable en función de N).
    - ***Inicio & Fin de trama***: 120 muestras (variable en función de N).
    - ***Longitud***: 20 muestras (variable en función de N y M).
    - ***CRC***: 20 muestras (variable en función de N y M).
    - ***Señal piloto y ceros***: 80 muestras (independientemente).

- Así, los datos de control suman un total de 301 muestras, con lo cual, las muestras referentes al campo DATOS constituyen un total de 581 muestras.
- Esto quiere decir que podemos rellenar el campo DATOS hasta un total de 58’1 símbolos, es decir, para M=256, un total de 60 bytes aproximadamente. Ese es un valor óptimo para nuestro sistema sin que se vea amenazado por la ausencia de sincronismo.
- Estos cálculos son aproximados y muy pesimistas, porque consideramos aceptable valores por debajo del 1 % de desviación de las muestras. Así no descartamos que podamos usar mayor número de bytes, y sobretodo para valores de M bajos, con una constelación más desahogada.
- Así, concluimos que un valor entre ***60*** y ***70 bytes*** es razonablemente óptimo para nuestra variable ***BDT***, y para cualquier valor de M.

III.2.5.- Estructuración de las Tramas

La ***división de los datos*** en porciones más pequeñas es debido a dos ***razones fundamentales***:

- ***Facilitar*** la ***computación*** de los datos tanto en el transmisor como en el receptor, liberando a éstos de cálculos excesivos.
- ***Evitar*** los ***problemas*** de falta de ***sincronización***.

La ***primera razón*** es bastante obvia. Supongamos que tenemos un fichero a transmitir con un tamaño de 100 kb, por ejemplo. Si nos hubiésemos planteado mandar dicho fichero en una sola trama, el número de muestras a procesar sería tal, que la CPU de nuestro PC se "agotaría", pues no está preparada para procesar tantos cálculos matemáticos a velocidad de vértigo.

El ***segundo motivo*** lo hemos explicado ya detenidamente, y es la principal razón por la que se divide la información en paquetes más pequeños, para que el transmisor y el receptor no desincronicen durante el envío de la trama.

La necesidad de ***controlar*** esas ***porciones de información***, estructurándolas en ***tramas***, es vital. Por consiguiente, debemos adoptar un ***formato de tramas específico***, para que tanto el receptor como el transmisor no tengan ningún problema en comunicarse. Además, dicha normalización es fundamental en una red, caso de que se produzca una ampliación de nuestro proyecto, y tracemos dicha red uniendo, mediante cables de audio, todos los Pc's que pertenezcan a ella. Esas normas, obviamente, incluirán aspectos para poder identificar la trama en recepción, tales como la longitud de los datos de cada trama, códigos para detección de errores, etc...

Para su estructuración, partiremos (como ya dijimos durante la explicación de la Fase de Entramado I del Bloque Transmisor) de la ***recomendación IEEE 802.3***, más conocida popularmente como ***Ethernet***[†10].

A continuación mostraremos un ***dibujo de una trama*** original, correspondiente a la ***recomendación IEEE 802.3***, detallando el significado de sus campos:

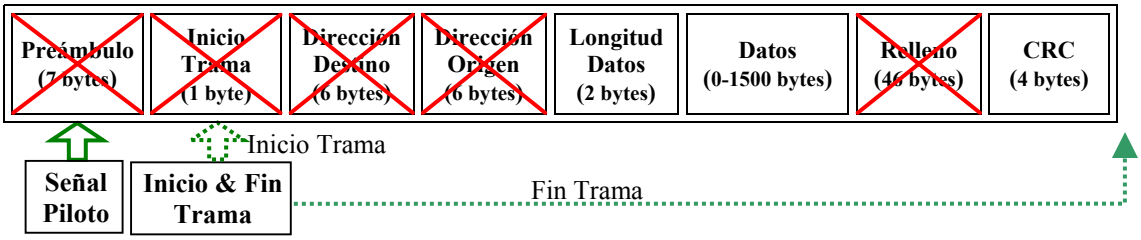


Fig. 104.- Trama Ethernet modificada

Cabe comentar que hemos tachado los campos que no vamos a usar para nuestro proyecto, y hemos indicado que, aunque vamos a usar un ***Inicio de Trama*** (para el ***ajuste de niveles***), éste no se corresponde con el de la normalización en cuestión. También recalcamos que nuestra ***señal piloto*** viene a realizar, en cierto modo, las labores de sincronización del preámbulo.

El significado de los campos es el siguiente[†10]:

- ***Preámbulo***: son 7 bytes iguales (10101010) que se utilizan para el sincronismo de trama: con ellos, se generan en el receptor una onda cuadrada de 10 Mhz., con el objetivo de que éste se sincronice con el reloj transmisor.
  Como para nosotros es imposible manipular las frecuencias de los relojes de las tarjetas con un ajuste muy fino vía software, y además suponemos conocidas de antemano dichas frecuencias de muestreo, hemos prescindido de este campo, en favor de nuestra señal piloto, que como ya sabemos, además de engancharnos a la trama, nos sirve para construir el igualador de cero forzado.
- ***Inicio de Trama***: se trata de un byte (10101011), cuyo único fin es el de marcar el comienzo de la trama.
  Al principio conservamos este campo, por mantener el parecido con la trama original, pero finalmente decidimos eliminarlo, pues teníamos otro campo de Inicio de Trama, y realmente este byte no nos aportaba nada, sino todo lo contrario, ya que suponía un exceso de muestras a transmitir, con lo vital que resultaba acortar el contenido de las tramas, para soslayar el problema de la falta de sincronismo.
- ***Dirección de Destino y de origen***: Ambos campos se usan para saber a quién va dirigido el mensaje y quién lo envía. Además, existe un carácter especial que puede indicar que el mensaje va dirigido a un grupo de usuarios o a todos los usuarios.
  Para nuestro proyecto no necesitamos estos campos, pues la comunicación será punto a punto. Sin embargo, en el caso de una posible ampliación del proyecto, por ejemplo, usando nuestra técnica de transmisión mediante cables de audio en una red, sí serían necesarios dichos campos.
- ***Longitud del campo de datos***: Como ya sabemos, estos dos bytes indican al receptor el número de bytes del campo DATOS, con lo que la longitud de la trama queda totalmente definida.
- ***Datos***: Este campo corresponde, como también adelantamos en su momento, a los bytes del fichero origen, que hemos troceado, enviado, y que queremos volver a montar en recepción, para reconstruir el fichero en destino.
  Por supuesto, este campo nunca llegará a ocupar 1500 bytes, ya que para garantizar un error de sincronización máximo de un 1 %, hemos explicado durante la Fase de entramado I, que sólo podemos enviar en total unas 882 muestras por trama aproximadamente, y evitar así problemas de sincronización.
  Además, si la trama tuviera 1500 bytes, calcular su código de redundancia, ecualizarla y demodularla vía software dotaría al sistema, como ya hemos dicho, de una lentitud desesperante, llegando a 'agotar' la CPU.
- ***Relleno***: Estos bytes se emplean para garantizar que la trama total tenga una longitud mínima de 64 bytes (sin contar con el preámbulo ni el Inicio de Trama). Esto se hace con el fin de desechar las tramas muy cortas (menores de 64 bytes) que puedan aparecer, como consecuencia de transmisiones abortadas por colisiones[†10]. Lógicamente, este campo no lo necesitamos para nada, pues no se nos van a dar casos de colisiones en una comunición unilateral como la nuestra.
- ***Código de redundancia cíclica o CRC***: sirve para hacer una detección de errores. Si algunos bits de datos llegan al receptor erróneamente (por causa

del ruido o de una anómala ecualización), es casi seguro que el código de redundancia será incorrecto y, por lo tanto, el error será detectado.
En nuestras tramas hemos prescindido del código CRC-32, de 4 bytes, y hemos optado por un CRC-16 (usado en HDLC), que solamente nos ocupe 2 bytes, para disminuir el proceso computacional, como veremos en el siguiente apartado.

A continuación vamos a dedicar un apartado especial al código CRC, para detallar su modo de funcionamiento, y las ventajas de usar dicho código de detección de errores.

CÓDIGO DE REDUNDANCIA CÍCLICA (CRC)

Básicamente, y procurando no adentrarnos mucho en el tema de ***codificación de canal*** (pues su explicación se sale del objetivo de nuestro proyecto), el ***CRC*** se calcula realizando una ***división módulo 2*** del campo DATOS entre un ***polinonomio generador***.

Cabría mencionar también que en cualquier libro de texto de redes de ordenadores puede verse que, con ***puertas XOR*** y algunos ***registros de desplazamiento***, podríamos realizar la división mencionada.

Los 3 ***polinomios generadores más usados*** en la actualidad son los siguientes[†10]:

- ***CRC-16***: $x^{16} + x^{15} + x^2 + 1$
- ***CRC-CCITT***: $x^{16} + x^{12} + x^5 + 1$
- ***CRC-32***: $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$

- Generación del código CRC

La ***generación del código CRC*** se realiza en el ***Bloque Transmisor***, y su ***procedimiento*** es el siguiente[†10] [†1]:

- Usaremos un ***polinomio generador*** de ***grado m=16*** (m+1=***17 elementos***).
- Dicho ***polinomio generador*** lo representaremos mediante una cadena de unos y ceros correspondientes a los (m+1) = ***17 coeficientes binarios*** del polinomio.
- ***Generación*** del ***Código CRC***:
  - Se añaden tantos ceros a los datos como indique el grado del polinomio generador.
  - Se realiza la división módulo 2 de los datos entre el polinomio generador.
  - El resto resultante es el ***CRC***, que obviamente ocupará ***dos bytes***.

La ***división*** la implementaremos ***vía software***, a pesar de que su realización hardware es mucho más potente y simple. Dicha división, para hallar el código CRC, tiene los siguientes ***componentes***:

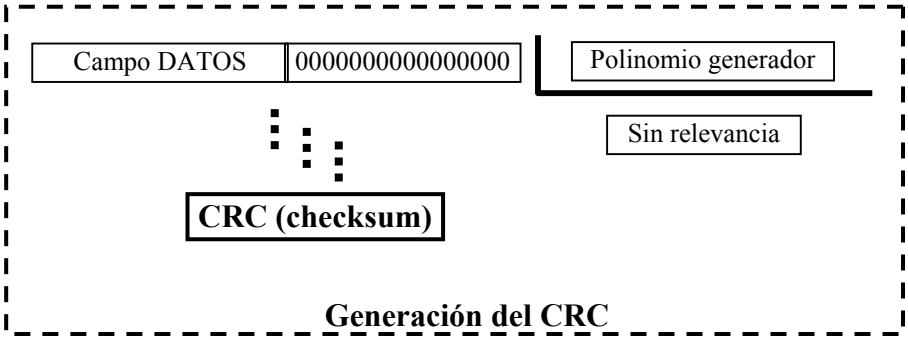


Fig. 105.- Generación CRC-16

- Detección de errores mediante CRC

Por otra parte, en el ***Bloque Receptor***, procederemos a realizar ***otra división*** para ***detectar*** la posible existencia de ***errores***[†10] [†1].

- Usaremos un ***polinomio generador*** de ***grado m=16*** (17 elementos).
- Dicho ***polinomio generador*** lo representaremos de la misma forma que durante el procedimiento de generación del Código CRC.
- Se añade, a los datos, el CRC obtenido en el receptor (2 btyes).
- Se realiza la ***división*** del ***conjunto datos+CRC*** entre el ***polinomio generador***.
- El ***Resto*** de dividir los datos junto con su CRC entre el mismo polinomio generador ***determinará*** la existencia o ausencia de ***errores***.
    - Si dicho ***Resto*** es ***cero***, ***no*** existen ***errores***.
    - Si por el contrario, dicho Resto contuviera algún bit distinto de cero, significaría que habría habido errores en la decisión de los bits.

Dicha ***división*** también la implementaremos vía software y tendrá el siguiente ***aspecto***:

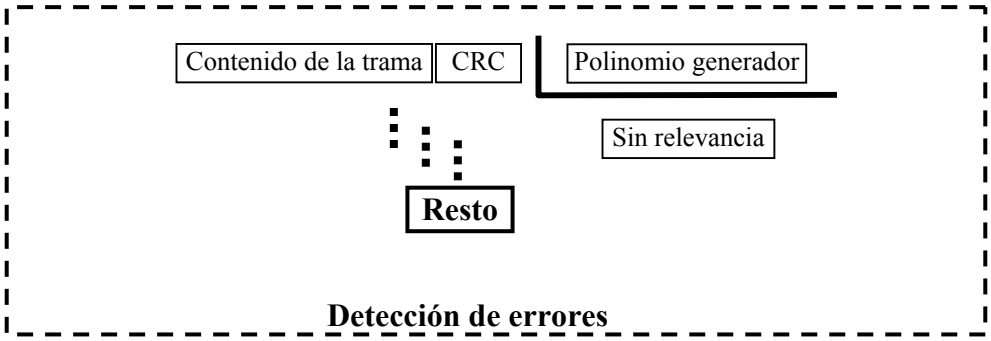


Fig. 106.- Detección errores

Antes de ver un ejemplo aclarativo de dicha división, nos gustaría precisar cómo se realizan los dos tipos de ***operaciones binarias*** que aparecen en dicha ***división***:

- La ***resta*** se reduce a realizar una operación ***XOR***.
- El ***cociente*** es 1 si la cifra más significativa del dividendo es 1, si no, es cero.

Ejemplo práctico

Para afianzar estos conceptos, y facilitar la labor de comprensión de cara a la posterior implementación, mediante Matlab, del ***cálculo*** del ***CRC*** y corrección de errores, veamos este sencillo ejemplo:

- Datos: 110101
- Polinomio generador: $x^3 + 1 \rightarrow 1\ 0\ 0\ 1$ (grado m=3→m+1=4 elementos)
- Cálculo del CRC:

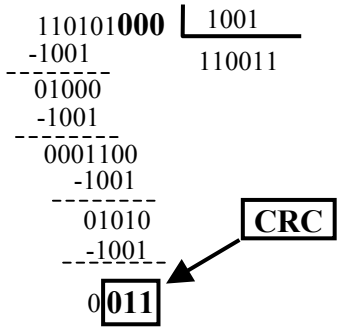


Fig. 107.- Ejemplo cálculo CRC

- Para ***determinar*** la existencia o ausencia de ***errores*** tomaríamos el campo de datos recibidos y le añadiríamos el CRC, para formar el dividendo. El divisor seguiría siendo el polinomio generador. El aspecto del divisor, sería el siguiente:

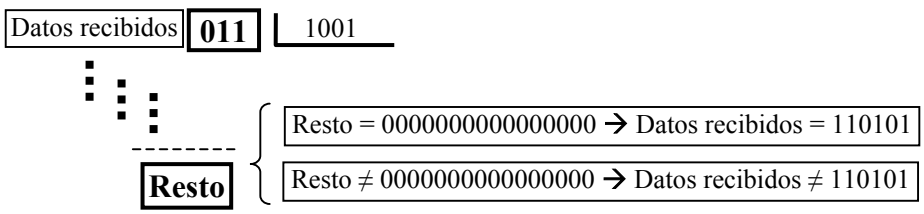


Fig. 108.- Ejemplo detección de errores

- Ventajas del CRC-16

Es importante dotar de ***celeridad computacional*** al sistema que estamos diseñando. El cálculo del ***CRC*** vía software supone un ***obstáculo*** a este propósito. A partir de las siguientes afirmaciones, vamos a ***justificar*** el empleo del ***CRC-16***, en detrimento del CRC-32[†10] [†4]:

- Si tenemos en cuenta que el rango típico de nuestra variable de diseño BDT, ***número de bytes del campo DATOS por trama***, para un caso crítico de probabilidad de error, es decir, para ***M = 256***, dichos valores oscilan entre ***60*** y ***70 bytes*** aproximadamente. Aunque intentaremos aumentar esa cifra, es obvio que nunca llegaremos a ocupar íntegramente los 1500 bytes del campo DATOS. Así, no tendremos que usar un código de detección tan potente como el ***CRC-32***.
- El ***CRC-32*** es computacionalmente más ***complejo*** y ***lento*** que el CRC-16.
- El ***CRC-16*** tiene importantes ***características***. Es capaz de detectar, en una ráfaga de 16 bits como máximo:

  - Errores simples.
  - Errores dobles.
  - Todos los números impares de errores (1,3,5,7,9...)
  - Además, el 99.9984 % de otros patrones de error podrán ser detectados.

Por otra parte, los protocolos en el nivel de red y superiores (modelo OSI-ISO para la interconexión de sistemas abiertos), usualmente utilizan un simple ***checksum*** para verificar que los datos transportados no se han corrompido debido al procesamiento en los distintos nodos. En nuestro caso, necesitamos algo mejor, pues la transmisión del fichero en su totalidad debe hacerse sin errores. Un simple checksum no es tan potente como un código de redundancia cíclica[†10].

III.3.- IMPLEMENTACIÓN EN MATLAB

En este apartado vamos a implementar, como ya adelantamos en varias ocasiones, todo el diseño llevado a cabo en el capítulo III.2, mediante la ***programación en Matlab*** de funciones que simulen la labor de cada uno de los bloques que intervienen en nuestro sistema.

Como ya hicimos en el anterior apartado, vamos a dividir este capítulo en ***dos grandes bloques***:

- Programación en Matlab del Bloque Transmisor.
- Programación en Matlab del Bloque Receptor.

La descripción, tanto conceptual como funcional, de nuestro sistema, fue explicado ya en el anterior capítulo. Por tanto, aquí nos limitaremos mayormente a mostrar los *'archivos.m'* que contienen las *funciones* que hemos generado, y describir un poco lo que hace cada uno de los comandos de Matlab integrados en dichos archivos.

El ***orden a seguir*** será el mismo que llevamos en el capítulo III.2, es decir, iremos desde la ***fuente*** hasta el ***destino***, vinculando cada bloque de nuestro sistema con la *función* del '*archivo.m'* en cuestión, comentando sus parámetros de entrada y de salida, y relacionando dicha *función* con la labor del bloque al que implementa vía software.

Es conveniente leer este ***capítulo*** de una forma paralela al anterior, el ***III.2***, pues éste contiene ***ejemplos muy ilustrativos*** que nos ayudarán a comprender mejor el desempeño de cada uno de los tramos de ambos bloques.

Antes de comenzar, nos gustaría añadir que, para distinguir claramente las *funciones*, de nuestros comentarios externos a dichas *funciones* (no olvidemos que dentro de las mismas funciones, también existirán ***comentarios***, precedidos del *'%'*), escribiremos en cursiva cada una de ellas, consiguiendo así una mejor legibilidad.

Además la definición completa de cada *función*, estará escrita en distintos colores tal y como aparece en Matlab, para localizar más rápidamente el inicio del comentario sobre cada una de dichas *funciones*:

- Los comentarios se realizarán en color verde.
- Las sentencias condicionales y referentes a los bucles estarán de color azul:
  - Condicionales: if, else, elseif, end.
  - Bucles: while, for, end.
- Las constantes de cadena, es decir, los caracteres que van entre 'comillas'.
- El resto de palabras irán en color negro.

Nota: Los *'archivos.m'* los hemos llamado igual a las funciones que contienen. Dichas funciones y, por tanto, dichos *'archivos.m'*, van precedidos de un *'z_'*. Dicha *'z_'* inicial no tiene ninguna trascendencia: es una forma de ordenar nuestros *'archivos.m'* (los implementados finalmente en nuestro proyecto), situándolos al final del directorio temporal donde se alojaban durante la fase de prueba, antes de introducirlas en un directorio final, una vez acabada la implementación en Matlab. Posteriormente, y por ser un poco conservadores, hemos dejado el nombre de las funciones tal cual, con dicha 'z_', para hacer una distinción entre nuestras funciones y las funciones propias de Matlab y, por qué no decirlo, como una especie de homenaje al 'ceceo' de nuestra tierra.

III.3.1.- Programación en Matlab del Bloque Transmisor

Adelantaremos primero la ***función principal***, para ir desgranándola poco a poco, en cada tramo del correspondiente bloque transmisor.

El ***programa principal*** se encuentra en el archivo *'z_main.m'*. La función *'z_main'* contenida en dicho archivo es la siguiente:

- **z_main:**

```
function z_main

 clc
 disp(' ');
 disp('                    PROYECTO FIN DE CARRERA');
 disp(' ');
 disp('      DISEÑO E IMPLEMENTACION DE UN MODEM APK MEDIANTE SOUNDBLASTER');
 disp(' ');
 disp(' ');
 disp('      AUTOR: Jose Miguel Moreno Perez');
 disp('      DIRECTOR: José Ramón Cerquides Bueno');
 disp(' ');
 disp(' ');
 disp('      ************************************************************');
 disp('      *                  TRANSMISOR                              *');
 disp('      ************************************************************');
 disp(' ');
 disp(' ');
%SE PREGUNTA SI SE DESEA METER LOS DATOS
 defecto=[];
 while (isempty(defecto)|(~isequal(defecto,'Y') & ~isequal(defecto,'y') &...
     ~isequal(defecto,'N') & ~isequal(defecto,'n')))
    defecto=input('      desea meter usted los datos?[Y/N]: ','s');
 end
%SE USAN LOS SIGUIENTES PARAMETROS POR DEFECTO
 if isequal(defecto,'n')|isequal(defecto,'N')
    fichero='prueba.txt';M=256;N=10;BDT=20;Fs=44100;NB=16;alfa=0.7;
%SE INTRODUCEN LOS NUEVOS PARAMETROS DEL SISTEMA
 else
    disp(' ');
    fichero=input('      Introduzca el fichero que desea transmitir: ','s');
    M=input('      Número de niveles de la modulación M-QAM (M=4/8/16/32/64/128/256): ');
    N=input('      Número de muestras por símbolo: ');
    BDT=input('      Longitud (bytes) del campo de datos de las tramas Ethernet:');
    Fs=input('      Frecuencia de muestreo (Hz): ');
    NB=input('      Resolución (8 ó 16 bits): ');
    alfa=input('      Factor de Roll-off (entre 0 y 1): ');
 end
 disp(' ');
 disp(' ');
 dat=z_transmite_fich(Fs,M,N,BDT,alfa,fichero);
 disp(' ');
 disp('      PULSE CUALQUIER TECLA PARA TRANSMITIR EL FICHERO...');
 pause;
 wavplay(dat,Fs,'async'); %en ms-dos hay que transmitir de manera sincrona,quito 'async'
 disp(' ');
 disp('      **********FICHERO TRANSMITIDO**********');
```

Lo primero que realiza en la función principal es la ***presentación por pantalla*** de nuestro proyecto. Para ello nos basamos en las siguientes funciones de Matlab:

- *'clc'*: limpia la pantalla y devuelve el cursor a la posición superior izquierda.
- *'disp'*: muestra en pantalla la cadena de caracteres entrecomillada.

Así, dicha presentación nos muestra los siguientes aspectos del proyecto:

- Nombre del proyecto.
- Autor del proyecto.
- Director del proyecto.
- Bloque transmisor

Posteriormente, el programa nos ofrece la posibilidad de introducir los ***parámetros variables*** de nuestro sistema, o bien utilizar los que ya tiene ***por defecto***. El hecho de introducir parámetros por defecto agiliza enormemente la fase de pruebas, pues no hay necesidad de volver a meter dichos datos una y otra vez, cada vez que se ejecute la aplicación.

En esta ocasión, los parámetros que el sistema muestra por defecto son los usados en los ejemplos que hicimos con el fichero *'prueba.txt'*. Enumeraremos dichos parámetros con sus valores por defecto (que naturalmente pueden ser modificados), y las variables usadas para su almacenamiento:

- Nombre del fichero (fichero): 'prueba.txt'
- Número de símbolos del sistema (M): 256
- Número de muestras / símbolo (N): 10
- Número de Bytes de datos / trama (BDT): 20
- Frecuencia de muestreo (Fs): 44100
- Número de bits de resolución (NB): 16
- Factor de Roll-off (alfa): 0.7

La sentencia:

- *dat=z_transmite_fich(Fs,M,N,BDT,alfa,fichero);*

es una llamada a la función '*z_transmite_fich'* , y abarca la implementación de todo el ***Bloque Transmisor***. Los parámetros de entrada de esta función son, obviamente, los parámetros de diseño definidos anteriormente, y devuelve como parámetro de salida el vector *dat*, que almacena todas las muestras de la señal completa, justo antes de ser emitida a través del cable de audio.

Luego se informa por pantalla que se pulse una tecla para enviar la señal. Esta acción la implementamos mediante la orden de Matlab:

- *'pause'*: pausa la ejecución en este punto hasta pulsar una tecla.

Tras pulsar dicha tecla (naturalmente, antes de haber pulsado en otra sesión de Matlab abierta para el receptor, como veremos, para grabar los datos que lleguen del canal), se ejecutará la siguiente línea de comando:

- *wavplay(dat,Fs,'async');*

La anterior sentencia, como ya vimos en el apartado III.2, se encarga de mandar las muestras del vector *dat* a través del canal de audio. Los parámetros de entrada son la frecuencia de muestreo y un parámetro denominado *'async'*, que sirve para devolver el control al programa principal una vez iniciada la transferencia, sin esperar a que se termine de completar ésta.

Finalmente, con ayuda de la orden *'disp'*, indicamos en la pantalla que el fichero ha sido transmitido.

Pasemos a explicar la función *'z_transmite_fich'*.

- **z_transmite_fich:**

```
function out=z_transmite_fich(Fs,M,N,BDT,alfa,fichero)

%M: MODULACION M_ARIA.
%N: NUMERO DE MUESTRAS / SIMBOLO.
%BDT: BYTES DE DATOS / TRAMA.
%fichero: NOMBRE DEL FICHERO A TRANSMITIR.
%Fs:frecuencia de muestreo.
%alfa:Factor de Roll-off.

 disp('PROCESANDO FICHERO. POR FAVOR, ESPERE...');
%APERTURA DEL FICHERO.
 fid=fopen(fichero,'r');
%B: BYTES DE DATOS DEL FICHERO QUE SERAN LEIDOS EN CADA ITERACION.
 B=BDT;
%LECTURA DEL FICHERO
%count: Nº DE BYTES LEIDOS DEL FICHERO.
%a:VECTOR COLUMNA QUE CONTIENE LOS BYTES LEIDOS DEL FICHERO.
 [a,count]=fread(fid,B);
%INICIALIZACION DE LA SEÑAL PILOTO Y DEL CONTADOR DE TRAMAS.
 y=[1,zeros(1,50)];
 n_trama=0;
%PARA CADA TRAMA...(MIENTRAS EXISTAN BYTES DE DATOS QUE LEER)
 while count>0
%b: MATRIZ QUE CONTIENE LOS VALORES DE a EN BYTES.
    b=dec2bin(a,8);
%FORMACION DE LA CADENA DE BITS DEL CAMPO DATOS.
    a=reshape(b',[1,count*8]);
%FASE DE ENTRAMADO I.
    trama=z_ethernet(dec2bin(count,16),a);
%RELLENO DE CEROS.
    while rem(length(trama),log2(M))
      trama=[trama,'0'];
    end
%MODULADOR APK.
    [trama_mod]=z_bloq_mod ( trama,M,N,alfa,Fs);
%FASE DE ENTRAMADO II.
    y=[y,trama_mod,zeros(1,30),1,zeros(1,50)];
%MUESTRO TRAMA NUMERO...
    n_trama=n_trama+1;
    aux=sprintf('Trama: %d',n_trama);
    disp(aux);
%LECTURA DE LA SIGUIENTE TRAMA Y ACTUALIZACION DEL BUCLE.
    [a,count]=fread(fid,B);
```

```
 end
%CLAUSURA DEL FICHERO.
 fclose(fid);
%out: PARAMETRO DE SALIDA. SEÑAL A TRANSMITIR.
 out=y;
```

La función comienza con las oportunas líneas de comentarios sobre las descripciones de los distintos parámetros de entrada.

Posteriormente nos muestra por pantalla un mensaje para indicarnos que el fichero se está procesando.

Luego procederemos a la ***apertura del fichero***, mediante la orden:

- *fid=fopen(fichero,'r');*

cuyos parámetros de entrada y salida de *fopen* son los siguientes:

- *fichero*: cadena que contiene el nombre del fichero que vamos a abrir.
- *'r'*: indica que el fichero lo vamos a abrir en modo lectura('read').
- *fid*: es el identificador del fichero abierto. Con él nos podemos referir al fichero en cuestión, para operar sobre él (en este caso, para leerlo). Si el nombre del fichero se introdujera mal, el identificador adquiriría el valor –1 y se produciría un error en la apertura.

Seguidamente, y como ya adelantamos en el capítulo III.2, generamos un ***bucle*** para procesar ***una trama en cada iteración***. La condición que debe cumplirse para salir del bucle es, obviamente, que se hayan leído todas las tramas, esto es, que el número de bytes leídos del fichero, en una de las iteraciones, sea cero.

La ***lectura del fichero*** se hará, como sabemos, mediante la orden:

- *[a,count]=fread(fid,B);*

donde el parámetro *B* nos permite controlar el número de bytes a leer, en cada llamada a la función *'fread'*. Lógicamente, la variable *B* será igual a *BDT*, que es el número de bytes de datos de cada trama.

En el parámetro de salida *a* se almacenarán los bytes leídos. La variable *count* es un contador de los bytes leídos "de una tacada", esto es, nos marca el número de bytes del campo DATOS que contiene la trama que estamos construyendo en cada iteración. Dicha variable es la que usamos en la condición de salida del bucle, pues cuando *count* sea igual a cero, es que ya no quedan más datos que leer. Por consiguiente, todas las tramas tendrán un número de bytes *count=B*, excepto la última trama, que puede tener un número menor de bytes.

Ya explicamos con un ejemplo gráfico, durante la Fase de ***Fuente Digital*** del Bloque Transmisor, la labor de las siguientes líneas de función:

- *b=dec2bin(a,8);*
- *%FORMACION DE LA CADENA DE BITS DEL CAMPO DATOS.*
- *a=reshape(b',[1,count*8]);*

Con la función '*dec2bin'* convertimos cada elemento de la columna de datos en decimal, *a*, en una cadena binaria de *8* bits. El resultado es el vector *b*, que tiene una dimensión de valor *count x 8*.

Por último, con la función '*reshape'*, transformamos *b* en una cadena binaria y almacenamos el resultado en la variable *a*.

Esta variable, *a,*contiene los datos binarios listos para entrar en la ***Fase de Entramado I***. Dicho tramo del bloque transmisor lo implementaremos mediante la llamada a la función z_ethernet:

- *trama=z_ethernet(dec2bin(count,16),a);*

Con el primer parámetro de entrada, *dec2bin(count,16)*, obtengo la longitud del campo DATOS de cada trama, que es precisamente el valor de *count*, obtenido a la salida de '*fread'*, y lo transformamos en un número binario de *16* bits, que son los dos bytes que ocupa dicho campo.

El parámetro de salida, *trama*, constituye la secuencia de datos a la salida de la ***Fase de Entramado I***, justo antes de entrar en el ***Modulador APK*** del Bloque Transmisor.

El procesado que sufre la secuencia de bits a su paso por el ***Modulador APK*** del Bloque Transmisor, hasta convertirse, a su salida, en la trama modulada $S(t)_{APK}$, lo implementamos mediante la llamada a la función:

- *[trama_mod]=z_bloq_mod(trama,M,N,alfa,Fs);*

, donde le pasamos como entrada varios de los parámetros de diseño fijados en la función principal, *'z_main2'*, además de la *trama* binaria a procesar.

Dentro de la función, veremos como se agrupan la secuencia binaria en bloques de k bits, siendo $k = \log_2 (M)$. Así, se rellena con ceros la secuencia binaria, hasta conseguir un número de bits múltiplo de k. Esto se logra mediante las siguientes líneas de código:

```
while rem(length(trama),log2(M))
    trama=[trama,'0'];
end
```

, donde programamos la ejecución de este 'minibucle', esto es, rellenamos con un *'0'* el final de la *trama* en cada iteración, hasta conseguir que el resto, '*rem'*, de la longitud de la *trama*, *length(trama)*, entre k = *log2(M)*, sea cero.

Como ya hemos dicho, el resultado de la llamada a '*z_bloq_mod'* es la obtención del parámetro de salida *trama_mod*, que se corresponde con la trama modulada $S(t)_{APK}$.

El procesamiento en el Bloque Transmisor se completa con la siguiente sentencia, donde se simula el paso por la ***Fase de Entramado II***:

- *y=[y,trama_mod,zeros(1,30),1,zeros(1,50)];*

A la trama modulada, *trama_mod*, se le inserta la ***señal piloto***, con sus correspondientes ceros, esto es, *zeros(1,30),1,zeros(1,50)*.

Además, el vector *y* hace de ***Acumulador*** de todas las tramas que van llegando procedentes de la ***Fuente Digital*** en cada iteración del bucle.

Además, al final de la ejecución de cada iteración del ***bucle de procesamiento de tramas***, nos hemos permitido imprimir en pantalla un mensaje, con el número de trama

procesada, si bien, esto no lo hemos modelado mediante ningún bloque en la fase de diseño III.2. Estas líneas de código son las siguientes:

- *n_trama=n_trama+1;*
- *aux=sprintf('Trama: %d',n_trama);*
- *disp(aux);*

La variable *n_trama* indica el número de trama procesada hasta el momento, y la sentencia *n_trama=n_trama+1;* no es más que una manera de incrementar su valor en cada iteración del bucle de trama.

La sentencia *aux=sprintf('Trama: %d',n_trama);* a grandes rasgos, lo que hace es almacenar, en la variable auxiliar *aux*, lo que encierra entre comillas de manera literal, excepto la expresión *%d*, que indica que en su lugar incluya el valor de la variable *n_trama*. Por ejemplo, si *n_trama*=5, al ejecutarse dicha línea de comando, la variable *aux* almacenaría la siguiente cadena de caracteres:

- *Trama: 5*

Cuando se ha terminado de leer todo el fichero, y de procesar la totalidad de las tramas, salimos del bucle y nos disponemos a proceder con la ***clausura*** de dicho ***fichero***, mediante la orden:

- *fclose(fid);*

Por último, devolvemos las muestras, listas para ser transmitidas, en el programa principal, mediante el parámetro de salida *out*.

Una vez visto la función '*z_transmite_fich'*, y a modo de esquema general, como hicimos en el capítulo III.2, ***analizaremos*** más detenidamente las distintas ***llamadas realizadas*** dentro de dicha función, para profundizar más en la implementación software del ***Bloque Transmisor***.

- **z_ethernet**

```
function[t]=z_ethernet(long,datos)

%Función que forma una trama Ethernet IEEE 802.3
%datos: bits del campo DATOS de la trama.
%long: bits del campo LONGITUD de la trama.

%Generacion del CRC-16
 crc=z_calc_crc(datos,0);
%t: trama a la salida de Fase de Entramado I.
 t=[long,datos,crc];
```

Como se puede apreciar en la última línea:

- *t=[long ,datos , crc];*

, el parámetro de salida, *t*, de nuestra ***Fase de Entramado I***, contiene, como era de suponer, los siguientes campos:

- LONGITUD
- DATOS
- CRC-16

Tanto el campo LONGITUD, *long*, como el campo DATOS, fueron introducidos como parámetros de entrada, y fueron generados ya en la función '*z_transmite_fich'*. Sin embargo, el CRC-16 lo calculamos mediante la llamada a la función:

- *crc=z_calc_crc(datos,16);*

Esta función la usamos tanto en el ***Bloque Transmisor***, para ***calcular*** el valor del ***código de redundancia cíclica***, como en el ***Bloque Receptor***, para ***detectar*** la posible existencia de ***errores***, ya que el procedimiento empleado es el mismo y sólo difiere la entrada de datos empleada en esta función.

En el caso del Bloque Transmisor, el parámetro de entrada de dicha función va a ser la variable vectorial *datos*, que como ya hemos comentado, comprende toda la secuencia binaria del campo DATOS. Su código es el siguiente:

- **z_calc_crc:**

```
function[y]=z_calc_crc(datos,bloque)

% Esta funcion se usa indistintamente tanto en el transmisor, para calcular...
% ...el CRC-16, como en el receptor, para detectar la existencia de errores.
%En bloque transmisor:
%datos= DATOS
%bloque=0;
%En bloque receceptor:
%datos= DATOS+CRC
%bloque~=0;

%x: Polinomio Generador o divisor.
 x=[1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1];
 lx=length(x);
%añado tantos ceros como indique el grado del polinomio (lx-1)
%solo si estamos en el bloque transmisor
 if bloque==0
   data=[datos,dec2bin(0,lx-1)];
 else
   data=datos;
 end
%Convierto la cadena de caracteres a matriz fila
%d: polinomio dividendo
 d=z_cad2mat(data);
 ld=length(d);
%comienza la division
 i=1;
 daux=d(i:i+lx-1);
 while i+lx<=ld
    if daux(1)==1
      aux=xor(daux,x);
```

```
    daux=[aux(2:lx),d(i+lx)];
  else
   daux=[daux(2:lx),d(i+lx)];
  end
  i=i+1;
end
if daux(1)==1
  aux=xor(daux,x);
else aux=daux;
end
%y: codigo CRC ya transformado en cadena binaria.
y=z_mat2cad(aux(2:length(aux)));
```

El parámetro de entrada *datos*, como ya hemos dicho, corresponde a los bits del campo DATOS. Por otro lado, el parámetro *bloque* lo usamos para indicar si estamos en el Bloque Transmisor o en el Receptor. Es decir, en la llamada a '*z_calc_crc'*, ponemos el segundo parámetro a cero si nos encontramos en el Bloque Transmisor. Esto nos sirve en las siguientes líneas de código:

- *if bloque==0*
- *data=[datos,dec2bin(0,lx-1)];*
- *end*

, donde indicamos que si *bloque* tiene valor nulo, *if bloque==0*, esto es, si estamos en el ***Bloque Transmisor***, entonces tenemos la seguridad de que la variable de entrada *datos* no contendrá ningún código CRC, y tenemos que añadirle un número de ceros igual al grado del polinomio generador, es decir, *data=[datos,dec2bin(0,lx-1)];*. En caso contrario, si estuviéramos en el ***Bloque Receptor***, *bloque* sería distinto de cero, y la variable local *data* sería igual a la variable de entrada *datos*, que representaría en este caso a los bits recibidos junto con el código CRC obtenido en esta misma función en el Bloque Transmisor, como ya explicamos en la Fase de Diseño III.2.

Mediante las siguientes líneas creamos el ***polinomio generador*** *x*, que ya vimos que tenía grado 16 (17 elementos), y hallamos su longitud, *lx* (cuyo valor, claro está, será 17), mediante la función *'length'*:

- *x=[1 1 0 0 0 0 0 0 0 0 0 0 0 1 0 1];*
- *lx=length(x);*

El vector *data* (que como ya hemos dicho estará formado por la secuencia binaria del campo DATOS y 2 bytes de ceros adicionales), constituirá nuestro ***polinomio dividendo***, como ya explicamos durante el subapartado CÓDIGO DE REDUNDANCIA CÍCLICA del apartado III.2.5.

Pero, para realizar la división del vector *data* entre el polinomio generador, necesitamos convertir esa cadena de caracteres en un vector donde cada elemento tenga un valor numérico, y no tenga un valor ASCII. Para ello nos valemos de las siguientes líneas de comando:

- *d=z_cad2mat(data);*
- *ld=length(d);*

La primera línea realiza la llamada a la función '*z_cad2mat'*, que hemos creado para el caso (a partir de la función propia de Matlab *str2num*, como veremos un poco más

adelante), donde obtenemos a partir de la cadena de caracteres binarios, *data*, un vector de números binarios *d*. La diferencia estriba en que el valor numérico de un carácter ASCII no coincide con su valor numérico real, y no podemos realizar operaciones aritméticas con caracteres de una manera cómoda. Así, por ejemplo, el carácter ASCII '0', tiene un valor igual a 48.

La siguiente línea, como suponemos, halla la longitud, *ld*, del vector numérico *d*.

Ya tenemos ambos operandos:

- *d*: el polinomio de datos o ***dividendo***, de longitud *ld*.
- *x*: el polinomio generador o ***divisor***, de longitud *lx*.

A partir de aquí implementamos la ***división***:

```
i=1;
daux=d(i:i+lx-1);
while i+lx<=ld
  if daux(1)==1
    aux=xor(daux,x);
    daux=[aux(2:lx),d(i+lx)];
  else
    daux=[daux(2:lx),d(i+lx)];
  end
  i=i+1;
end
%if bin2dec(z_mat2cad(daux))~=0
if daux(1)==1
    aux=xor(daux,x);
else
   aux=daux;
end
```

En cada iteración, tomaremos un bloque de 17 elementos, del dividendo *d* y lo almacenaremos en *daux*, esto es, *daux=d(i:i+lx-1);*.

Luego, si el primer elemento del bloque *daux* es *1*, es decir, *if daux(1)==1*, obviaremos el valor igual a 1 del cociente (pues no nos interesa y no lo almacenamos en ninguna variable), y realizamos la pertinente resta entre el bloque *daux* del dividendo y el polinomio generador, *x*, (como procederíamos en cualquier división). La ***resta***, como adelantamos en el Apartado de Diseño, la implementaremos con funciones *'xor'* de las librerías de Matlab, esto es, *aux=xor(daux,x);*. El resultado lo almacenamos en la variable auxiliar *aux*, y actualizamos el bloque *daux*, tomando después un nuevo elemento del dividendo y despreciando el primer elemento de *daux* (que, obviamente será cero), es decir, *daux=[daux(2:lx),d(i+lx)];*.

En cambio, si el primer elemento del bloque daux es *0*, el cociente será cero (que ya hemos dicho que no nos interesa), y la diferencia entre daux y x sería igual a daux, por lo que no es necesario ejecutar esa línea, y simplemente nos limitamos a eliminar el primer elemento de daux (que será un valor nulo), y a bajar un nuevo elemento del dividendo, *d*.

El bucle se repetirá hasta que no existan más elementos del divisor que tomar, es decir, *while i+lx<=ld* ,con lo que realizamos la última resta y lo que obtengamos será el ***resto***, cuyos 16 elementos menos significativos (recordemos que el primer elemento será nulo) constituirán el ***código CRC***, cuyo carácter vectorial lo volvemos a

transformar en cadena de caracteres mediante la función '*mat2str'*, y lo almacenamos posteriormente en el parámetro de salida *y*, es decir, *y=z_mat2cad(aux(2:length(aux)));*.

A continuación explicaremos las dos únicas llamadas que se producen dentro de la función *z_calc_crc*. Se trata, como sabremos de las funciones:

- z_cad2mat
- z_mat2cad

Ambas funciones realizan procesos inversos, como veremos en el siguiente análisis.

- **z_cad2mat:**

```
function[y]=z_cad2mat(x)
y=[];
i=1;
while i<=length(x)
  y=[y,str2num(x(i))];
      i=i+1;
end
```

Mediante el índice *i* recorremos el cadena binaria de entrada *x*, mediante un bucle, carácter a carácter, y, mediante la función *'str2num'*, vamos transformando dichos caracteres, *x(i)*, a valores numéricos, y los vamos almacenando en el vector numérico *y* de salida.

- **z_mat2cad:**

```
function[y]=z_mat2cad(x)
y='';
for i=1:length(x)
  y=[y,int2str(x(i))];
end
```

Mediante el índice *i* recorremos el vector numérico de entrada *x*, mediante un bucle, elemento a elemento, y, mediante la función *'int2str'*, vamos transformando dichos elementos numéricos, *x(i)*, a caracteres, y los vamos almacenando en cadena binaria de salida, *y*.

Analizemos seguidamente, la llamada:

- *[trama_mod]=z_bloq_mod (trama,M,N,alfa,Fs);*

, que se produce dentro de la función '*z_transmite_fich'*, y que implementa el ***Modulador APK*** que mostramos en el Capítulo de Diseño III.2. , donde procesamos la entrada binaria de cada trama y obtenemos a la salida el vector *trama_mod*, es decir, la señal $S(t)_{APK}$, correspondiente a dicha trama. Volvemos a recomendar que se le eche

conjuntamente un vistazo al Bloque ***Modulador APK***, tanto a sus dibujos como al texto explicativo, para comprender mejor el código en Matlab desarrollado al respecto.

- **z_bloq_mod:**

```
function [ dat_mod] = z_bloq_mod (tramaIEE,M,N,alfa,Fs)
%este es el bloque modulador APK.
%usa pulsos en coseno alzado.
%trama IEEE: secuencia binaria de la trama, a la entrada del modulador.

%AGRUPADOR DE K BITS & TABLA CONVERSION
%vect_c_fas: secuencia de componentes en fase de los símbolos.
%vect_c_quad: secuencia de componentes en cuadratura de los símbolos.
 [vect_c_fas,vect_c_quad]=z_form_simb(tramaIEE,M);

%INICIO & FIN DE TRAMA
% 0.4: niveles ficticios tanto para la fase como para la cuadratura de los símbolos.
 [vect_c_fas]=[0.4 0.4 0.4 0.4 0.4 0.4 0.4 0.4 vect_c_fas 0.4 0.4 0.4 0.4];
 [vect_c_quad]=[0.4 0.4 0.4 0.4 0.4 0.4 0.4 0.4 vect_c_quad 0.4 0.4 0.4 0.4];

%dibujamos los puntos transmitidos en la constelacion
% z_plot_constelac(vect_c_fas,vect_c_quad,M);

%FILTRO CONFORMADOR & OSCILADOR LOCAL
% dat_mod: señal S(t)APK de la trama correspondiente, a la salida del oscilador.
 [dat_mod]=z_oscillator(vect_c_fas,vect_c_quad,N,alfa,Fs);
```

La primera línea:

- *[vect_c_fas,vect_c_quad]=z_form_simb(tramaIEE,M);*

implementa el conjunto de bloques en cascada ***Agrupador de k bits & Tabla de Conversión de k bits a símbolo M-ario***, analizados en el Capítulo de Diseño III.2. A partir de la secuencia binaria de la trama, *tramaIEE*, obtenemos a la salida los símbolos M-arios correspondientes, en dos vectores, uno que almacena la distancia en cuadratura de cada símbolo, *vect_c_quad*, y otro que almacena la distancia en fase, *vect_c_fas*.

Seguidamente, le añadimos el ***Inicio de Trama***, como también comentamos en la Fase de Diseño, introduciéndole ***8 símbolos*** de valor: *0.4*+j·*0.4*, que se corresponderá, como ya sabemos, con símbolos ***ficticios*** en nuestra constelación, que nos servirán para el posterior ***Ajuste de Niveles*** en el ***Bloque Receptor***, y para compensar el efecto inestables de los transitorios sobre las muestras.También añadimos 4 símbolos ficticios con el mismo valor, y que constituirán el ***Fin de Trama***, con la finalidad de que el transitorio de la cola no afecte las muestras del campo CRC. El resultado lo volvemos a almacenar en las mismas variables *vect_c_quad* y *vect_c_fas*, que ahora contendrán también estos dos nuevos campos:

- *[vect_c_fas]=[0.4 0.4 0.4 0.4 0.4 0.4 0.4 0.4 vect_c_fas 0.4 0.4 0.4 0.4];*
- *[vect_c_quad]=[0.4 0.4 0.4 0.4 0.4 0.4 0.4 0.4 vect_c_quad 0.4 0.4 0.4 0.4];*

Las siguientes líneas las hemos omitido para la ejecución de nuestro programa, pero nos sirvió de gran ayuda durante la fase de pruebas, puesto que nos dibujaba la ***Constelación*** de los símbolos obtenidos en la función anterior:

- *%dibujamos los puntos transmitidos en la constelacion*
- *% z_plot_constelac(vect_c_fas,vect_c_quad,M);*

Ni que decir tiene que, para volver a tener disponibles dichas sentencias, basta con quitarle el carácter *'%'*, que convierte a ambas líneas en comentarios.

Por último, la sentencia:

- *[dat_mod]=z_oscillator(vect_c_fas,vect_c_quad,N,alfa,Fs);*

, simula la labor del ***Filtro Conformador***, y del ***Oscilador*** en ambas ramas, la de fase y la de cuadratura, y la ***resta final*** entre la señal $S_I(t) \cdot \cos(w_c \cdot t)$, de la rama de fase, y la señal $S_Q(t) \cdot \text{sen}(w_c \cdot t)$, de la rama de cuadratura, para formar la trama modulada $S(t)_{APK}$, que almacenamos en la variable vectorial *dat_mod*, y cuyo valor devolvemos a la función *'z_transmite_fich'*, como parámetro de salida.

Veamos a continuación, cada una de las llamadas hechas en esta función *'z_bloq_mod'*.

- **z_form_simb:**

```
function [v_c_f,v_c_q]=z_form_simb(tram_bin,M);

%funcion que recibe una cadena binaria de entrada y devuelve...
%...los componentes en fase y cuadratura de los simbolos

%k: longitud del simbolo.
 k=log2(M);
 j=1;
 i=1;
 while i<=length(tram_bin)
%AGRUPADOR DE K BITS & Tabla de Conversion a simbolo M-ario
   [v_c_f(j),v_c_q(j)]=z_tab_constelac(tram_bin(i:i+k-1));
   i=i+k;
   j=j+1;
 end
```

La secuencia de bits de entrada, *tram_bin*, es recorrida en el bucle mediante la variable *i*, y en la sentencia:

- *[v_c_f(j),v_c_q(j)]=z_tab_constelac(tram_bin(i:i+k-1));*

llamamos a la función *'z_tab_constelac'*, para realizar el ***mapeo*** y obtener, a partir de cada bloque de k bits *tram_bin(i:i+k-1)*, su correspondiente símbolo M-ario. La variable *k* indica la longitud del bloque binario correspondiente a cada símbolo M-ario, y su valor dependerá, como sabemos ya sobradamente, del número de símbolos de la constelación, *M*, mediante la siguiente relación:

- *k=log2(M);*

La variable *j* recorre los vectores de salida de la función *'z_tab_constelac'*, *v_c_f* y *v_c_q*, que almacenan los valores de las coordenadas en fase y en cuadratura de los símbolos respectivamente.

Veamos el procedimiento seguido en la ***Tabla de Conversión*** mencionada.

- **z_tab_constelac:**

```
function [v_c_f,v_c_q]=z_tab_constelac(simb_bin)

%funcion que acepta un bloque de k bits
%devuelve el vector de componentes de
%fase y cuadratura del simbolo correspondiente.


%M=4....2 bits por simbolo
 if length(simb_bin)==2
    [v_c_f,v_c_q] = z_tab_M_4( simb_bin );

%M=8....3 bits por simbolo
 elseif length(simb_bin)==3
     [v_c_f,v_c_q] = z_tab_M_8( simb_bin );

%M=16....4 bits por simbolo
 elseif length(simb_bin)==4
     [v_c_f,v_c_q] = z_tab_M_16( simb_bin );

%M=32....5 bits por simbolo
 elseif length(simb_bin)==5
    [v_c_f,v_c_q] = z_tab_M_32( simb_bin );

%M=64....6 bits por simbolo
 elseif length(simb_bin)==6
    [v_c_f,v_c_q] = z_tab_M_64( simb_bin );

%M=128....7 bits por simbolo
 elseif length(simb_bin)==7
    [v_c_f,v_c_q] = z_tab_M_128( simb_bin );

%M=256....8 bits por simbolo
 elseif length(simb_bin)==8
    [v_c_f,v_c_q] = z_tab_M_256( simb_bin );

 end
```

En esta función, dependiendo de la longitud del bloque de k bits, *length(simb_bin)*, utilizamos una de las ***7 posibles Tablas de Conversión***, como ya explicamos en la etapa de diseño, es decir:

- Si k = 2 → M = 4 → llamamos a la función *'z_tab_M_4'*.

- Si k = 3 → M = 8 → llamamos a la función *'z_tab_M_8*.
- Si k = 4 → M = 16 → llamamos a la función *'z_tab_M_16*.
- Si k = 5 → M = 32 → llamamos a la función *'z_tab_M_32'*.
- Si k = 6 → M = 64 → llamamos a la función *'z_tab_M_64'*.
- Si k = 7 → M = 128 → llamamos a la función *'z_tab_M_128'*.
- Si k = 8 → M = 256 → llamamos a la función *'z_tab_M_256'*.

Los vectores *v_c_f* y *v_c_q*, serán, respectivamente, las componentes en fase y en cuadratura de los símbolos resultantes de la conversión simbólica.

A continuación mostramos las ***7 Tablas de mapeo*** existentes en nuestro sistema.

- **z_tab_M_4:**

```
function [x,y] = z_tab_M_4( simb_bin )

%normalizamos, para que la mayor amplitud
%de simbolo este entre -1 y 1.
 L=1/sqrt(2);

 if simb_bin=='00'
   x=-L;
   y=-L;
 elseif simb_bin=='01'
   x=-L;
   y=L;
 elseif simb_bin=='11'
   x=L;
   y=L;
 elseif simb_bin=='10'
   x=L;
   y=-L;
 end
```

- **z_tab_M_8:**

```
function [x,y] = z_tab_M_8( simb_bin )

%normalizamos, para que la mayor amplitud
%de simbolo este entre -1 y 1.
  L=1/3;

  if simb_bin=='000' %0
    x=L;
    y=L;
  elseif simb_bin=='001'   %1
    x=3*L;
    y=0;
  elseif simb_bin=='011' %2
    x=0;
    y=3*L;
  elseif simb_bin=='010'  %3
    x=-L;
```

```
    y=L;
  elseif simb_bin=='110'  %4
    x=-L;
    y=-L;
  elseif simb_bin=='111'  %5
    x=-3*L;
    y=0;
  elseif simb_bin=='101'  %6
    x=0;
    y=-3*L;
  elseif simb_bin=='100'  %7
    x=L;
    y=-L;
  end
```

- **z_tab_M_16:**

```
function [x,y] = z_tab_M_16( simb_bin )

%normalizamos, para que la mayor amplitud
%de simbolo este entre -1 y 1.
  L=1/(3*sqrt(2)); %distancia cartesiana de minima energia

  if simb_bin=='0000' %0
    x=-3*L;
    y=-3*L;
  elseif simb_bin=='0001'  %1
    x=-3*L;
    y=-L;
  elseif simb_bin=='0011' %2
    x=-3*L;
    y=L;
  elseif simb_bin=='0010'  %3
    x=-3*L;
    y=3*L;
  elseif simb_bin=='0110'  %4
    x=-L;
    y=3*L;
  elseif simb_bin=='0111'  %5
    x=-L;
    y=L;
  elseif simb_bin=='0101'  %6
    x=-L;
    y=-L;
  elseif simb_bin=='0100'  %7
    x=-L;
    y=-3*L;
  elseif simb_bin=='1100'  %8
    x=L;
    y=-3*L;
  elseif simb_bin=='1101'  %9
    x=L;
    y=-L;
  elseif simb_bin=='1111'  %10
    x=L;
    y=L;
  elseif simb_bin=='1110'  %11
    x=L;
    y=3*L;
  elseif simb_bin=='1010'  %12
```

```
    x=3*L;
    y=3*L;
  elseif simb_bin=='1011'  %13
    x=3*L;
    y=L;
  elseif simb_bin=='1001'  %14
    x=3*L;
    y=-L;
  elseif simb_bin=='1000'  %15
    x=3*L;
    y=-3*L;
  end
```

- **z_tab_M_32:**

```
function [x,y] = z_tab_M_32( simb_bin )

%normalizamos, para que la mayor amplitud
%de simbolo este entre -1 y 1.
  L=1/(5*sqrt(2));

  if simb_bin=='00000' %0
    x=L;
    y=L;
  elseif simb_bin=='00001' %1
    x=3*L;
    y=L;
  elseif simb_bin=='00011'  %2
    x=5*L;
    y=L;
  elseif simb_bin=='00010'  %3
    x=3*L;
    y=5*L;
  elseif simb_bin=='00110'  %4
    x=L;
    y=5*L;
  elseif simb_bin=='00111'  %5
    x=5*L;
    y=3*L;
  elseif simb_bin=='00101'  %6
    x=3*L;
    y=3*L;
  elseif simb_bin=='00100'  %7
    x=L;
    y=3*L;
  elseif simb_bin=='01100'  %8
    x=-L;
    y=3*L;
  elseif simb_bin=='01101'  %9
    x=-3*L;
    y=3*L;
  elseif simb_bin=='01111'  %10
    x=-5*L;
    y=3*L;
  elseif simb_bin=='01110'  %11
    x=-L;
    y=5*L;
  elseif simb_bin=='01010'   %12
```

```
    x=-3*L;
    y=5*L;
  elseif simb_bin=='01011'  %13
    x=-5*L;
    y=L;
  elseif simb_bin=='01001'  %14
    x=-3*L;
    y=L;
  elseif simb_bin=='01000'  %15
    x=-L;
    y=L;
  elseif simb_bin=='11000' %16
    x=-L;
    y=-L;
  elseif simb_bin=='11001' %17
    x=-3*L;
    y=-L;
  elseif simb_bin=='11011'   %18
    x=-5*L;
    y=-L;
  elseif simb_bin=='11010'   %19
    x=-3*L;
    y=-5*L;
  elseif simb_bin=='11110'  %20
    x=-L;
    y=-5*L;
  elseif simb_bin=='11111'  %21
    x=-5*L;
    y=-3*L;
  elseif simb_bin=='11101'  %22
    x=-3*L;
    y=-3*L;
  elseif simb_bin=='11100'  %23
    x=-L;
    y=-3*L;
  elseif simb_bin=='10100'   %24
    x=L;
    y=-3*L;
  elseif simb_bin=='10101' %25
    x=3*L;
    y=-3*L;
  elseif simb_bin=='10111' %26
    x=5*L;
    y=-3*L;
  elseif simb_bin=='10110'   %27
    x=L;
    y=-5*L;
  elseif simb_bin=='10010'    %28
    x=3*L;
    y=-5*L;
  elseif simb_bin=='10011'  %29
    x=5*L;
    y=-L;
  elseif simb_bin=='10001'  %30
    x=3*L;
    y=-L;
  elseif simb_bin=='10000'  %31
    x=L;
    y=-L;
  end
```

- **z_tab_M_64:**

```
function [x,y] = z_tab_M_64( simb_bin )

%normalizamos, para que la mayor amplitud
%de simbolo este entre -1 y 1.
  L=1/(7*sqrt(2));

if simb_bin=='000000'  %0
    x=-7*L;
    y=-7*L;
  elseif simb_bin=='000001' %1
    x=-7*L;
    y=-5*L;
  elseif simb_bin=='000011'   %2
    x=-7*L;
    y=-3*L;
  elseif simb_bin=='000010'   %3
    x=-7*L;
    y=-L;
  elseif simb_bin=='000110'  %4
    x=-7*L;
    y=L;
  elseif simb_bin=='000111'  %5
    x=-7*L;
    y=3*L;
  elseif simb_bin=='000101'  %6
    x=-7*L;
    y=5*L;
  elseif simb_bin=='000100'  %7
    x=-7*L;
    y=7*L;
  elseif simb_bin=='001100'   %8
    x=-5*L;
    y=7*L;
  elseif simb_bin=='001101' %9
    x=-5*L;
    y=5*L;
  elseif simb_bin=='001111' %10
    x=-5*L;
    y=3*L;
  elseif simb_bin=='001110'   %11
    x=-5*L;
    y=L;
  elseif simb_bin=='001010'    %12
    x=-5*L;
    y=-L;
  elseif simb_bin=='001011'  %13
    x=-5*L;
    y=-3*L;
  elseif simb_bin=='001001'  %14
    x=-5*L;
    y=-5*L;
  elseif simb_bin=='001000'  %15
    x=-5*L;
    y=-7*L;
  elseif simb_bin=='011000' %16
```

```
    x=-3*L;
    y=-7*L;
  elseif simb_bin=='011001' %17
    x=-3*L;
    y=-5*L;
  elseif simb_bin=='011011'  %18
    x=-3*L;
    y=-3*L;
  elseif simb_bin=='011010'  %19
    x=-3*L;
    y=-L;
  elseif simb_bin=='011110'  %20
    x=-3*L;
    y=L;
  elseif simb_bin=='011111'  %21
    x=-3*L;
    y=3*L;
  elseif simb_bin=='011101'  %22
    x=-3*L;
    y=5*L;
  elseif simb_bin=='011100'  %23
    x=-3*L;
    y=7*L;
  elseif simb_bin=='010100'  %24
    x=-L;
    y=7*L;
  elseif simb_bin=='010101' %25
    x=-L;
    y=5*L;
  elseif simb_bin=='010111' %26
    x=-L;
    y=3*L;
  elseif simb_bin=='010110'  %27
    x=-L;
    y=L;
  elseif simb_bin=='010010'   %28
    x=-L;
    y=-L;
  elseif simb_bin=='010011'  %29
    x=-L;
    y=-3*L;
  elseif simb_bin=='010001'  %30
    x=-L;
    y=-5*L;
  elseif simb_bin=='010000'  %31
    x=-L;
    y=-7*L;
  elseif simb_bin=='110000' %32
    x=L;
    y=-7*L;
  elseif simb_bin=='110001' %33
    x=L;
    y=-5*L;
  elseif simb_bin=='110011'  %34
    x=L;
    y=-3*L;
  elseif simb_bin=='110010'  %35
    x=L;
    y=-L;
  elseif simb_bin=='110110'  %36
```

```
    x=L;
    y=L;
  elseif simb_bin=='110111'  %37
    x=L;
    y=3*L;
  elseif simb_bin=='110101'  %38
    x=L;
    y=5*L;
  elseif simb_bin=='110100'  %39
    x=L;
    y=7*L;
  elseif simb_bin=='111100'   %40
    x=3*L;
    y=7*L;
  elseif simb_bin=='111101' %41
    x=3*L;
    y=5*L;
  elseif simb_bin=='111111' %42
    x=3*L;
    y=3*L;
  elseif simb_bin=='111110'   %43
    x=3*L;
    y=L;
  elseif simb_bin=='111010'    %44
    x=3*L;
    y=-L;
  elseif simb_bin=='111011'  %45
    x=3*L;
    y=-3*L;
  elseif simb_bin=='111001'  %46
    x=3*L;
    y=-5*L;
  elseif simb_bin=='111000'  %47
    x=3*L;
    y=-7*L;
  elseif simb_bin=='101000' %48
    x=5*L;
    y=-7*L;
  elseif simb_bin=='101001' %49
    x=5*L;
    y=-5*L;
  elseif simb_bin=='101011'   %50
    x=5*L;
    y=-3*L;
  elseif simb_bin=='101010'   %51
    x=5*L;
    y=-L;
  elseif simb_bin=='101110'  %52
    x=5*L;
    y=L;
  elseif simb_bin=='101111'  %53
    x=5*L;
    y=3*L;
  elseif simb_bin=='101101'  %54
    x=5*L;
    y=5*L;
  elseif simb_bin=='101100'  %55
    x=5*L;
    y=7*L;
  elseif simb_bin=='100100'   %56
```

```
    x=7*L;
    y=7*L;
  elseif simb_bin=='100101' %57
    x=7*L;
    y=5*L;
  elseif simb_bin=='100111' %58
    x=7*L;
    y=3*L;
  elseif simb_bin=='100110'   %59
    x=7*L;
    y=L;
  elseif simb_bin=='100010'    %60
    x=7*L;
    y=-L;
  elseif simb_bin=='100011'  %61
    x=7*L;
    y=-3*L;
  elseif simb_bin=='100001'  %62
    x=7*L;
    y=-5*L;
  elseif simb_bin=='100000'  %63
    x=7*L;
    y=-7*L;
  end
```

- **z_tab_M_128:**

```
function [x,y] = z_tab_M_128( simb_bin )

%normalizamos, para que la mayor amplitud
%de simbolo este entre -1 y 1.
  L=1/(11*sqrt(2));

  if simb_bin=='0000000' %0
    x=L;
    y=L;
  elseif simb_bin=='0000001' %1
    x=3*L;
    y=L;
  elseif simb_bin=='0000011'   %2
    x=3*L;
    y=3*L;
  elseif simb_bin=='0000010'   %3
    x=L;
    y=3*L;
  elseif simb_bin=='0000110'  %4
    x=7*L;
    y=3*L;
  elseif simb_bin=='0000111'  %5
    x=5*L;
    y=3*L;
  elseif simb_bin=='0000101'  %6
    x=5*L;
    y=L;
  elseif simb_bin=='0000100'  %7
    x=7*L;
    y=L;
  elseif simb_bin=='0001100'   %8
```

```
    x=9*L;
    y=L;
  elseif simb_bin=='0001101' %9
    x=11*L;
    y=L;
  elseif simb_bin=='0001111' %10
    x=11*L;
    y=3*L;
  elseif simb_bin=='0001110'   %11
    x=9*L;
    y=3*L;
  elseif simb_bin=='0001010'    %12
    x=7*L;
    y=11*L;
  elseif simb_bin=='0001011'  %13
    x=5*L;
    y=11*L;
  elseif simb_bin=='0001001'  %14
    x=5*L;
    y=9*L;
  elseif simb_bin=='0001000'  %15
    x=7*L;
    y=9*L;
  elseif simb_bin=='0011000' %16
    x=L;
    y=9*L;
  elseif simb_bin=='0011001' %17
    x=3*L;
    y=9*L;
  elseif simb_bin=='0011011'   %18
    x=3*L;
    y=11*L;
  elseif simb_bin=='0011010'   %19
    x=L;
    y=11*L;
  elseif simb_bin=='0011110'  %20
    x=9*L;
    y=5*L;
  elseif simb_bin=='0011111'  %21
    x=11*L;
    y=5*L;
  elseif simb_bin=='0011101'  %22
    x=11*L;
    y=7*L;
  elseif simb_bin=='0011100'  %23
    x=9*L;
    y=7*L;
  elseif simb_bin=='0010100'   %24
    x=7*L;;
    y=7*L;;
  elseif simb_bin=='0010101' %25
    x=5*L;;
    y=7*L;;
  elseif simb_bin=='0010111' %26
    x=5*L;;
    y=5*L;;
  elseif simb_bin=='0010110'   %27
    x=7*L;;
    y=5*L;
  elseif simb_bin=='0010010'    %28
```

```
    x=L;;
    y=5*L;
  elseif simb_bin=='0010011'  %29
    x=3*L;
    y=5*L;
  elseif simb_bin=='0010001'  %30
    x=3*L;
    y=7*L;
  elseif simb_bin=='0010000'  %31
    x=L;
    y=7*L;
  elseif simb_bin=='0110000' %32
    x=-L;
    y=7*L;
  elseif simb_bin=='0110001' %33
    x=-3*L;
    y=7*L;
  elseif simb_bin=='0110011'   %34
    x=-3*L;
    y=5*L;
  elseif simb_bin=='0110010'   %35
    x=-L;
    y=5*L;
  elseif simb_bin=='0110110'  %36
    x=-7*L;
    y=5*L;
  elseif simb_bin=='0110111'  %37
    x=-5*L;
    y=5*L;
  elseif simb_bin=='0110101'  %38
    x=-5*L;
    y=7*L;
  elseif simb_bin=='0110100'  %39
    x=-7*L;
    y=7*L;
  elseif simb_bin=='0111100'   %40
    x=-9*L;
    y=7*L;
  elseif simb_bin=='0111101' %41
    x=-11*L;
    y=7*L;
  elseif simb_bin=='0111111' %42
    x=-11*L;
    y=5*L;
  elseif simb_bin=='0111110'   %43
    x=-9*L;
    y=5*L;
  elseif simb_bin=='0111010'    %44
    x=-L;
    y=11*L;
  elseif simb_bin=='0111011'  %45
    x=-3*L;
    y=11*L;
  elseif simb_bin=='0111001'  %46
    x=-3*L;
    y=9*L;
  elseif simb_bin=='0111000'  %47
    x=-L;
    y=9*L;
  elseif simb_bin=='0101000' %48
```

```
    x=-7*L;
    y=9*L;
  elseif simb_bin=='0101001' %49
    x=-5*L;
    y=9*L;
  elseif simb_bin=='0101011'  %50
    x=-5*L;
    y=11*L;
  elseif simb_bin=='0101010'  %51
    x=-7*L;
    y=11*L;
  elseif simb_bin=='0101110'  %52
    x=-9*L;
    y=3*L;
  elseif simb_bin=='0101111'  %53
    x=-11*L;
    y=3*L;
  elseif simb_bin=='0101101'  %54
    x=-11*L;
    y=L;
  elseif simb_bin=='0101100'  %55
    x=-9*L;
    y=L;
  elseif simb_bin=='0100100'  %56
    x=-7*L;
    y=L;
  elseif simb_bin=='0100101' %57
    x=-5*L;
    y=L;
  elseif simb_bin=='0100111' %58
    x=-5*L;
    y=3*L;
  elseif simb_bin=='0100110'  %59
    x=-7*L;
    y=3*L;
  elseif simb_bin=='0100010'   %60
    x=-L;
    y=3*L;
  elseif simb_bin=='0100011'  %61
    x=-3*L;
    y=3*L;
  elseif simb_bin=='0100001'  %62
    x=-3*L;
    y=L;
  elseif simb_bin=='0100000'  %63
    x=-L;
    y=L;
  elseif simb_bin=='1100000' %64
    x=-L;
    y=-L;
  elseif simb_bin=='1100001' %65
    x=-3*L;
    y=-L;
  elseif simb_bin=='1100011'  %66
    x=-3*L;
    y=-3*L;
  elseif simb_bin=='1100010'  %67
    x=-L;
    y=-3*L;
  elseif simb_bin=='1100110'  %68
```

```
    x=-7*L;
    y=-3*L;
  elseif simb_bin=='1100111'  %69
    x=-5*L;
    y=-3*L;
  elseif simb_bin=='1100101'  %70
    x=-5*L;
    y=-L;
  elseif simb_bin=='1100100'  %71
    x=-7*L;
    y=-L;
  elseif simb_bin=='1101100'   %72
    x=-9*L;
    y=-L;
  elseif simb_bin=='1101101' %73
    x=-11*L;
    y=-L;
  elseif simb_bin=='1101111' %74
    x=-11*L;
    y=-3*L;
  elseif simb_bin=='1101110'   %75
    x=-9*L;
    y=-3*L;
  elseif simb_bin=='1101010'    %76
    x=-7*L;
    y=-11*L;
  elseif simb_bin=='1101011'  %77
    x=-5*L;
    y=-11*L;
  elseif simb_bin=='1101001'  %78
    x=-5*L;
    y=-9*L;
  elseif simb_bin=='1101000'  %79
    x=-7*L;
    y=-9*L;
  elseif simb_bin=='1111000' %80
    x=-L;
    y=-9*L;
  elseif simb_bin=='1111001' %81
    x=-3*L;
    y=-9*L;
  elseif simb_bin=='1111011'   %82
    x=-3*L;
    y=-11*L;
  elseif simb_bin=='1111010'   %83
    x=-L;
    y=-11*L;
  elseif simb_bin=='1111110'  %84
    x=-9*L;
    y=-5*L;
  elseif simb_bin=='1111111'  %85
    x=-11*L;
    y=-5*L;
  elseif simb_bin=='1111101'  %86
    x=-11*L;
    y=-7*L;
  elseif simb_bin=='1111100'  %87
    x=-9*L;
    y=-7*L;
  elseif simb_bin=='1110100'   %88
```

```
    x=-7*L;
    y=-7*L;
  elseif simb_bin=='1110101' %89
    x=-5*L;
    y=-7*L;
  elseif simb_bin=='1110111' %90
    x=-5*L;
    y=-5*L;
  elseif simb_bin=='1110110'  %91
    x=-7*L;
    y=-5*L;
  elseif simb_bin=='1110010'   %92
    x=-L;
    y=-5*L;
  elseif simb_bin=='1110011'  %93
    x=-3*L;
    y=-5*L;
  elseif simb_bin=='1110001'  %94
    x=-3*L;
    y=-7*L;
  elseif simb_bin=='1110000'  %95
    x=-L;
    y=-7*L;
  elseif simb_bin=='1010000' %96
    x=L;
    y=-7*L;
  elseif simb_bin=='1010001' %97
    x=3*L;
    y=-7*L;
  elseif simb_bin=='1010011'  %98
    x=3*L;
    y=-5*L;
  elseif simb_bin=='1010010'  %99
    x=L;
    y=-5*L;
  elseif simb_bin=='1010110'  %100
    x=7*L;
    y=-5*L;
  elseif simb_bin=='1010111'  %101
    x=5*L;
    y=-5*L;
  elseif simb_bin=='1010101'  %102
    x=5*L;
    y=-7*L;
  elseif simb_bin=='1010100'  %103
    x=7*L;
    y=-7*L;
  elseif simb_bin=='1011100'  %104
    x=9*L;
    y=-7*L;
  elseif simb_bin=='1011101' %105
    x=11*L;
    y=-7*L;
  elseif simb_bin=='1011111' %106
    x=11*L;
    y=-5*L;
  elseif simb_bin=='1011110'  %107
    x=9*L;
    y=-5*L;
  elseif simb_bin=='1011010'   %108
```

```
    x=L;
    y=-11*L;
  elseif simb_bin=='1011011'  %109
    x=3*L;
    y=-11*L;
  elseif simb_bin=='1011001'  %110
    x=3*L;
    y=-9*L;
  elseif simb_bin=='1011000'  %111
    x=L;
    y=-9*L;
  elseif simb_bin=='1001000' %112
    x=7*L;
    y=-9*L;
  elseif simb_bin=='1001001' %113
    x=5*L;
    y=-9*L;
  elseif simb_bin=='1001011'   %114
    x=5*L;
    y=-11*L;
  elseif simb_bin=='1001010'   %115
    x=7*L;
    y=-11*L;
  elseif simb_bin=='1001110'  %116
    x=9*L;
    y=-3*L;
  elseif simb_bin=='1001111'  %117
    x=11*L;
    y=-3*L;
  elseif simb_bin=='1001101'  %118
    x=11*L;
    y=-L;
  elseif simb_bin=='1001100'  %119
    x=9*L;
    y=-L;
  elseif simb_bin=='1000100'   %120
    x=7*L;
    y=-L;
  elseif simb_bin=='1000101' %121
    x=5*L;
    y=-L;
  elseif simb_bin=='1000111' %122
    x=5*L;
    y=-3*L;
  elseif simb_bin=='1000110'   %123
    x=7*L;
    y=-3*L;
  elseif simb_bin=='1000010'    %124
    x=L;
    y=-3*L;
  elseif simb_bin=='1000011'  %125
    x=3*L;
    y=-3*L;
elseif simb_bin=='1000001'  %126
    x=3*L;
    y=-L;
elseif simb_bin=='1000000'  %127
    x=L;
    y=-L;
end
```

- **z_tab_M_256:**

```
function [x,y] = z_tab_M_256( simb_bin )

%normalizamos, para que la mayor amplitud
%de simbolo este entre -1 y 1.
 L=1/(15*sqrt(2));

 if simb_bin=='00000000' %0
   x=-15*L;
   y=-15*L;
 elseif simb_bin=='00000001' %1
   x=-15*L;
   y=-13*L;
 elseif simb_bin=='00000011'  %2
   x=-15*L;
   y=-11*L;
 elseif simb_bin=='00000010'  %3
   x=-15*L;
   y=-9*L;
 elseif simb_bin=='00000110'  %4
   x=-15*L;
   y=-7*L;
 elseif simb_bin=='00000111'  %5
   x=-15*L;
   y=-5*L;
 elseif simb_bin=='00000101'  %6
   x=-15*L;
   y=-3*L;
 elseif simb_bin=='00000100'  %7
   x=-15*L;
   y=-L;
 elseif simb_bin=='00001100'  %8
   x=-15*L;
   y=L;
 elseif simb_bin=='00001101' %9
   x=-15*L;
   y=3*L;
 elseif simb_bin=='00001111' %10
   x=-15*L;
   y=5*L;
 elseif simb_bin=='00001110'  %11
   x=-15*L;
   y=7*L;
 elseif simb_bin=='00001010'   %12
   x=-15*L;
   y=9*L;
 elseif simb_bin=='00001011'  %13
   x=-15*L;
   y=11*L;
 elseif simb_bin=='00001001'  %14
   x=-15*L;
   y=13*L;
 elseif simb_bin=='00001000'  %15
   x=-15*L;
   y=15*L;
 elseif simb_bin=='00011000' %16
```

```
    x=-13*L;
    y=15*L;
  elseif simb_bin=='00011001' %17
    x=-13*L;
    y=13*L;
  elseif simb_bin=='00011011'  %18
    x=-13*L;
    y=11*L;
  elseif simb_bin=='00011010'  %19
    x=-13*L;
    y=9*L;
  elseif simb_bin=='00011110'  %20
    x=-13*L;
    y=7*L;
  elseif simb_bin=='00011111'  %21
    x=-13*L;
    y=5*L;
  elseif simb_bin=='00011101'  %22
    x=-13*L;
    y=3*L;
  elseif simb_bin=='00011100'  %23
    x=-13*L;
    y=L;
  elseif simb_bin=='00010100'  %24
    x=-13*L;
    y=-L;
  elseif simb_bin=='00010101' %25
    x=-13*L;
    y=-3*L;
  elseif simb_bin=='00010111' %26
    x=-13*L;
    y=-5*L;
  elseif simb_bin=='00010110'  %27
    x=-13*L;
    y=-7*L;
  elseif simb_bin=='00010010'   %28
    x=-13*L;
    y=-9*L;
  elseif simb_bin=='00010011'  %29
    x=-13*L;
    y=-11*L;
  elseif simb_bin=='00010001'  %30
    x=-13*L;
    y=-13*L;
  elseif simb_bin=='00010000'  %31
    x=-13*L;
    y=-15*L;
  elseif simb_bin=='00110000' %32
    x=-11*L;
    y=-15*L;
  elseif simb_bin=='00110001' %33
    x=-11*L;
    y=-13*L;
  elseif simb_bin=='00110011'  %34
    x=-11*L;
    y=-11*L;
  elseif simb_bin=='00110010'  %35
    x=-11*L;
    y=-9*L;
  elseif simb_bin=='00110110'  %36
```

```
    x=-11*L;
    y=-7*L;
  elseif simb_bin=='00110111'  %37
    x=-11*L;
    y=-5*L;
  elseif simb_bin=='00110101'  %38
    x=-11*L;
    y=-3*L;
  elseif simb_bin=='00110100'  %39
    x=-11*L;
    y=-L;
  elseif simb_bin=='00111100'   %40
    x=-11*L;
    y=L;
  elseif simb_bin=='00111101' %41
    x=-11*L;
    y=3*L;
  elseif simb_bin=='00111111' %42
    x=-11*L;
    y=5*L;
  elseif simb_bin=='00111110'   %43
    x=-11*L;
    y=7*L;
  elseif simb_bin=='00111010'    %44
    x=-11*L;
    y=9*L;
  elseif simb_bin=='00111011'  %45
    x=-11*L;
    y=11*L;
  elseif simb_bin=='00111001'  %46
    x=-11*L;
    y=13*L;
  elseif simb_bin=='00111000'  %47
    x=-11*L;
    y=15*L;
  elseif simb_bin=='00101000' %48
    x=-9*L;
    y=15*L;
  elseif simb_bin=='00101001' %49
    x=-9*L;
    y=13*L;
  elseif simb_bin=='00101011'   %50
    x=-9*L;
    y=11*L;
  elseif simb_bin=='00101010'   %51
    x=-9*L;
    y=9*L;
  elseif simb_bin=='00101110'  %52
    x=-9*L;
    y=7*L;
  elseif simb_bin=='00101111'  %53
    x=-9*L;
    y=5*L;
  elseif simb_bin=='00101101'  %54
    x=-9*L;
    y=3*L;
  elseif simb_bin=='00101100'  %55
    x=-9*L;
    y=L;
  elseif simb_bin=='00100100'   %56
```

```
    x=-9*L;
    y=-L;
  elseif simb_bin=='00100101' %57
    x=-9*L;
    y=-3*L;
  elseif simb_bin=='00100111' %58
    x=-9*L;
    y=-5*L;
  elseif simb_bin=='00100110'  %59
    x=-9*L;
    y=-7*L;
  elseif simb_bin=='00100010'   %60
    x=-9*L;
    y=-9*L;
  elseif simb_bin=='00100011'  %61
    x=-9*L;
    y=-11*L;
  elseif simb_bin=='00100001'  %62
    x=-9*L;
    y=-13*L;
  elseif simb_bin=='00100000'  %63
    x=-9*L;
    y=-15*L;
  elseif simb_bin=='01100000' %64
    x=-7*L;
    y=-15*L;
  elseif simb_bin=='01100001' %65
    x=-7*L;
    y=-13*L;
  elseif simb_bin=='01100011'  %66
    x=-7*L;
    y=-11*L;
  elseif simb_bin=='01100010'  %67
    x=-7*L;
    y=-9*L;
  elseif simb_bin=='01100110'  %68
    x=-7*L;
    y=-7*L;
  elseif simb_bin=='01100111'  %69
    x=-7*L;
    y=-5*L;
  elseif simb_bin=='01100101'  %70
    x=-7*L;
    y=-3*L;
  elseif simb_bin=='01100100'  %71
    x=-7*L;
    y=-L;
  elseif simb_bin=='01101100'  %72
    x=-7*L;
    y=L;
  elseif simb_bin=='01101101' %73
    x=-7*L;
    y=3*L;
  elseif simb_bin=='01101111' %74
    x=-7*L;
    y=5*L;
  elseif simb_bin=='01101110'  %75
    x=-7*L;
    y=7*L;
  elseif simb_bin=='01101010'   %76
```

```
    x=-7*L;
    y=9*L;
  elseif simb_bin=='01101011'  %77
    x=-7*L;
    y=11*L;
  elseif simb_bin=='01101001'  %78
    x=-7*L;
    y=13*L;
  elseif simb_bin=='01101000'  %79
    x=-7*L;
    y=15*L;
  elseif simb_bin=='01111000' %80
    x=-5*L;
    y=15*L;
  elseif simb_bin=='01111001' %81
    x=-5*L;
    y=13*L;
  elseif simb_bin=='01111011'   %82
    x=-5*L;
    y=11*L;
  elseif simb_bin=='01111010'   %83
    x=-5*L;
    y=9*L;
  elseif simb_bin=='01111110'  %84
    x=-5*L;
    y=7*L;
  elseif simb_bin=='01111111'  %85
    x=-5*L;
    y=5*L;
  elseif simb_bin=='01111101'  %86
    x=-5*L;
    y=3*L;
  elseif simb_bin=='01111100'  %87
    x=-5*L;
    y=L;
  elseif simb_bin=='01110100'   %88
    x=-5*L;
    y=-L;
  elseif simb_bin=='01110101' %89
    x=-5*L;
    y=-3*L;
  elseif simb_bin=='01110111' %90
    x=-5*L;
    y=-5*L;
  elseif simb_bin=='01110110'   %91
    x=-5*L;
    y=-7*L;
  elseif simb_bin=='01110010'    %92
    x=-5*L;
    y=-9*L;
  elseif simb_bin=='01110011'  %93
    x=-5*L;
    y=-11*L;
  elseif simb_bin=='01110001'  %94
    x=-5*L;
    y=-13*L;
  elseif simb_bin=='01110000'  %95
    x=-5*L;
    y=-15*L;
  elseif simb_bin=='01010000' %96
```

```
    x=-3*L;
    y=-15*L;
  elseif simb_bin=='01010001' %97
    x=-3*L;
    y=-13*L;
  elseif simb_bin=='01010011'  %98
    x=-3*L;
    y=-11*L;
  elseif simb_bin=='01010010'  %99
    x=-3*L;
    y=-9*L;
  elseif simb_bin=='01010110' %100
    x=-3*L;
    y=-7*L;
  elseif simb_bin=='01010111' %101
    x=-3*L;
    y=-5*L;
  elseif simb_bin=='01010101' %102
    x=-3*L;
    y=-3*L;
  elseif simb_bin=='01010100' %103
    x=-3*L;
    y=-L;
  elseif simb_bin=='01011100'  %104
    x=-3*L;
    y=L;
  elseif simb_bin=='01011101' %105
    x=-3*L;
    y=3*L;
  elseif simb_bin=='01011111' %106
    x=-3*L;
    y=5*L;
  elseif simb_bin=='01011110'  %107
    x=-3*L;
    y=7*L;
  elseif simb_bin=='01011010'   %108
    x=-3*L;
    y=9*L;
  elseif simb_bin=='01011011' %109
    x=-3*L;
    y=11*L;
  elseif simb_bin=='01011001' %110
    x=-3*L;
    y=13*L;
  elseif simb_bin=='01011000' %111
    x=-3*L;
    y=15*L;
  elseif simb_bin=='01001000' %112
    x=-L;
    y=15*L;
  elseif simb_bin=='01001001' %113
    x=-L;
    y=13*L;
  elseif simb_bin=='01001011'  %114
    x=-L;
    y=11*L;
  elseif simb_bin=='01001010'  %115
    x=-L;
    y=9*L;
  elseif simb_bin=='01001110' %116
```

```
    x=-L;
    y=7*L;
  elseif simb_bin=='01001111' %117
    x=-L;
    y=5*L;
  elseif simb_bin=='01001101' %118
    x=-L;
    y=3*L;
  elseif simb_bin=='01001100' %119
    x=-L;
    y=L;
  elseif simb_bin=='01000100' %120
    x=-L;
    y=-L;
  elseif simb_bin=='01000101'%121
    x=-L;
    y=-3*L;
  elseif simb_bin=='01000111'%122
    x=-L;
    y=-5*L;
  elseif simb_bin=='01000110' %123
    x=-L;
    y=-7*L;
  elseif simb_bin=='01000010' %124
    x=-L;
    y=-9*L;
  elseif simb_bin=='01000011' %125
    x=-L;
    y=-11*L;
  elseif simb_bin=='01000001' %126
    x=-L;
    y=-13*L;
  elseif simb_bin=='01000000' %127
    x=-L;
    y=-15*L;
  elseif simb_bin=='11000000'%128
    x=L;
    y=-15*L;
  elseif simb_bin=='11000001'%129
    x=L;
    y=-13*L;
  elseif simb_bin=='11000011' %130
    x=L;
    y=-11*L;
  elseif simb_bin=='11000010' %131
    x=L;
    y=-9*L;
  elseif simb_bin=='11000110' %132
    x=L;
    y=-7*L;
  elseif simb_bin=='11000111' %133
    x=L;
    y=-5*L;
  elseif simb_bin=='11000101' %134
    x=L;
    y=-3*L;
  elseif simb_bin=='11000100' %135
    x=L;
    y=-L;
  elseif simb_bin=='11001100' %136
```

```
    x=L;
    y=L;
  elseif simb_bin=='11001101' %137
    x=L;
    y=3*L;
  elseif simb_bin=='11001111' %138
    x=L;
    y=5*L;
  elseif simb_bin=='11001110'  %139
    x=L;
    y=7*L;
  elseif simb_bin=='11001010'   %140
    x=L;
    y=9*L;
  elseif simb_bin=='11001011'  %141
    x=L;
    y=11*L;
  elseif simb_bin=='11001001'  %142
    x=L;
    y=13*L;
  elseif simb_bin=='11001000'  %143
    x=L;
    y=15*L;
  elseif simb_bin=='11011000' %144
    x=3*L;
    y=15*L;
  elseif simb_bin=='11011001' %145
    x=3*L;
    y=13*L;
  elseif simb_bin=='11011011'  %146
    x=3*L;
    y=11*L;
  elseif simb_bin=='11011010'  %147
    x=3*L;
    y=9*L;
  elseif simb_bin=='11011110'  %148
    x=3*L;
    y=7*L;
  elseif simb_bin=='11011111'  %149
    x=3*L;
    y=5*L;
  elseif simb_bin=='11011101'  %150
    x=3*L;
    y=3*L;
  elseif simb_bin=='11011100'  %151
    x=3*L;
    y=L;
  elseif simb_bin=='11010100'  %152
    x=3*L;
    y=-L;
  elseif simb_bin=='11010101' %153
    x=3*L;
    y=-3*L;
  elseif simb_bin=='11010111' %154
    x=3*L;
    y=-5*L;
  elseif simb_bin=='11010110'  %155
    x=3*L;
    y=-7*L;
  elseif simb_bin=='11010010'   %156
```

```
    x=3*L;
    y=-9*L;
  elseif simb_bin=='11010011' %157
    x=3*L;
    y=-11*L;
  elseif simb_bin=='11010001' %158
    x=3*L;
    y=-13*L;
  elseif simb_bin=='11010000' %159
    x=3*L;
    y=-15*L;
  elseif simb_bin=='11110000' %160
    x=5*L;
    y=-15*L;
  elseif simb_bin=='11110001' %161
    x=5*L;
    y=-13*L;
  elseif simb_bin=='11110011'  %162
    x=5*L;
    y=-11*L;
  elseif simb_bin=='11110010'  %163
    x=5*L;
    y=-9*L;
  elseif simb_bin=='11110110' %164
    x=5*L;
    y=-7*L;
  elseif simb_bin=='11110111' %165
    x=5*L;
    y=-5*L;
  elseif simb_bin=='11110101' %166
    x=5*L;
    y=-3*L;
  elseif simb_bin=='11110100' %167
    x=5*L;
    y=-L;
  elseif simb_bin=='11111100'  %168
    x=5*L;
    y=L;
  elseif simb_bin=='11111101' %169
    x=5*L;
    y=3*L;
  elseif simb_bin=='11111111' %170
    x=5*L;
    y=5*L;
  elseif simb_bin=='11111110'  %171
    x=5*L;
    y=7*L;
  elseif simb_bin=='11111010'   %172
    x=5*L;
    y=9*L;
  elseif simb_bin=='11111011' %173
    x=5*L;
    y=11*L;
  elseif simb_bin=='11111001' %174
    x=5*L;
    y=13*L;
  elseif simb_bin=='11111000' %175
    x=5*L;
    y=15*L;
  elseif simb_bin=='11101000' %176
```

```
    x=7*L;
    y=15*L;
  elseif simb_bin=='11101001' %177
    x=7*L;
    y=13*L;
  elseif simb_bin=='11101011'  %178
    x=7*L;
    y=11*L;
  elseif simb_bin=='11101010'  %179
    x=7*L;
    y=9*L;
  elseif simb_bin=='11101110'  %180
    x=7*L;
    y=7*L;
  elseif simb_bin=='11101111'  %181
    x=7*L;
    y=5*L;
  elseif simb_bin=='11101101'  %182
    x=7*L;
    y=3*L;
  elseif simb_bin=='11101100'  %183
    x=7*L;
    y=L;
  elseif simb_bin=='11100100'  %184
    x=7*L;
    y=-L;
  elseif simb_bin=='11100101' %185
    x=7*L;
    y=-3*L;
  elseif simb_bin=='11100111' %186
    x=7*L;
    y=-5*L;
  elseif simb_bin=='11100110'  %187
    x=7*L;
    y=-7*L;
  elseif simb_bin=='11100010'   %188
    x=7*L;
    y=-9*L;
  elseif simb_bin=='11100011'  %189
    x=7*L;
    y=-11*L;
  elseif simb_bin=='11100001'  %190
    x=7*L;
    y=-13*L;
  elseif simb_bin=='11100000'  %191
    x=7*L;
    y=-15*L;
  elseif simb_bin=='10100000' %192
    x=9*L;
    y=-15*L;
  elseif simb_bin=='10100001' %193
    x=9*L;
    y=-13*L;
  elseif simb_bin=='10100011'  %194
    x=9*L;
    y=-11*L;
  elseif simb_bin=='10100010'  %195
    x=9*L;
    y=-9*L;
  elseif simb_bin=='10100110'  %196
```

```
    x=9*L;
    y=-7*L;
  elseif simb_bin=='10100111' %197
    x=9*L;
    y=-5*L;
  elseif simb_bin=='10100101' %198
    x=9*L;
    y=-3*L;
  elseif simb_bin=='10100100' %199
    x=9*L;
    y=-L;
  elseif simb_bin=='10101100' %200
    x=9*L;
    y=L;
  elseif simb_bin=='10101101' %201
    x=9*L;
    y=3*L;
  elseif simb_bin=='10101111' %202
    x=9*L;
    y=5*L;
  elseif simb_bin=='10101110' %203
    x=9*L;
    y=7*L;
  elseif simb_bin=='10101010' %204
    x=9*L;
    y=9*L;
  elseif simb_bin=='10101011' %205
    x=9*L;
    y=11*L;
  elseif simb_bin=='10101001' %206
    x=9*L;
    y=13*L;
  elseif simb_bin=='10101000' %207
    x=9*L;
    y=15*L;
  elseif simb_bin=='10111000' %208
    x=11*L;
    y=15*L;
  elseif simb_bin=='10111001' %209
    x=11*L;
    y=13*L;
  elseif simb_bin=='10111011' %210
    x=11*L;
    y=11*L;
  elseif simb_bin=='10111010' %211
    x=11*L;
    y=9*L;
  elseif simb_bin=='10111110' %212
    x=11*L;
    y=7*L;
  elseif simb_bin=='10111111' %213
    x=11*L;
    y=5*L;
  elseif simb_bin=='10111101' %214
    x=11*L;
    y=3*L;
  elseif simb_bin=='10111100' %215
    x=11*L;
    y=L;
  elseif simb_bin=='10110100' %216
```

```
    x=11*L;
    y=-L;
  elseif simb_bin=='10110101' %217
    x=11*L;
    y=-3*L;
  elseif simb_bin=='10110111' %218
    x=11*L;
    y=-5*L;
  elseif simb_bin=='10110110'  %219
    x=11*L;
    y=-7*L;
  elseif simb_bin=='10110010'   %220
    x=11*L;
    y=-9*L;
  elseif simb_bin=='10110011'  %221
    x=11*L;
    y=-11*L;
  elseif simb_bin=='10110001'  %222
    x=11*L;
    y=-13*L;
  elseif simb_bin=='10110000'  %223
    x=11*L;
    y=-15*L;
  elseif simb_bin=='10010000' %224
    x=13*L;
    y=-15*L;
  elseif simb_bin=='10010001' %225
    x=13*L;
    y=-13*L;
  elseif simb_bin=='10010011'  %226
    x=13*L;
    y=-11*L;
  elseif simb_bin=='10010010'  %227
    x=13*L;
    y=-9*L;
  elseif simb_bin=='10010110'  %228
    x=13*L;
    y=-7*L;
  elseif simb_bin=='10010111'  %229
    x=13*L;
    y=-5*L;
  elseif simb_bin=='10010101'  %230
    x=13*L;
    y=-3*L;
  elseif simb_bin=='10010100'  %231
    x=13*L;
    y=-L;
  elseif simb_bin=='10011100'  %232
    x=13*L;
    y=L;
  elseif simb_bin=='10011101' %233
    x=13*L;
    y=3*L;
  elseif simb_bin=='10011111' %234
    x=13*L;
    y=5*L;
  elseif simb_bin=='10011110'  %235
    x=13*L;
    y=7*L;
  elseif simb_bin=='10011010'   %236
```

```
    x=13*L;
    y=9*L;
  elseif simb_bin=='10011011' %237
    x=13*L;
    y=11*L;
  elseif simb_bin=='10011001' %238
    x=13*L;
    y=13*L;
  elseif simb_bin=='10011000' %239
    x=13*L;
    y=15*L;
  elseif simb_bin=='10001000'%240
    x=15*L;
    y=15*L;
  elseif simb_bin=='10001001'%241
    x=15*L;
    y=13*L;
  elseif simb_bin=='10001011'  %242
    x=15*L;
    y=11*L;
  elseif simb_bin=='10001010'  %243
    x=15*L;
    y=9*L;
  elseif simb_bin=='10001110' %244
    x=15*L;
    y=7*L;
  elseif simb_bin=='10001111' %245
    x=15*L;
    y=5*L;
  elseif simb_bin=='10001101' %246
    x=15*L;
    y=3*L;
  elseif simb_bin=='10001100' %247
    x=15*L;
    y=L;
  elseif simb_bin=='10000100'  %248
    x=15*L;
    y=-L;
  elseif simb_bin=='10000101'%249
    x=15*L;
    y=-3*L;
  elseif simb_bin=='10000111'%250
    x=15*L;
    y=-5*L;
  elseif simb_bin=='10000110'  %251
    x=15*L;
    y=-7*L;
  elseif simb_bin=='10000010'   %252
    x=15*L;
    y=-9*L;
  elseif simb_bin=='10000011' %253
    x=15*L;
    y=-11*L;
  elseif simb_bin=='10000001' %254
    x=15*L;
    y=-13*L;
  elseif simb_bin=='10000000' %255
    x=15*L;
    y=-15*L;
   end
```

Para acabar con el análisis sobre la implementación de las ***Tablas de Conversión***, nos gustaría reincidir, como ya hicimos en la Etapa de Diseño, que ***para modificar*** el ***tamaño*** de las constelaciones, basta con redefinir la variable *L* en la primera línea de cada una de estas funciones, *'z_tab_M_4'*, *'z_tab_M_8'*, etc...

Por otra parte, nos gustaría volver a destacar, como también lo hiciéramos en la etapa de diseño, la ***facilidad para cambiar la fisonomía de la constelación***, sin más que cambiar, para cada conjunto de bits, la distancia en fase, *x*, y en cuadratura, *y*, respecto del origen, de su símbolo correspondiente.

- **z_oscillator:**

```
function [trama_mod] = z_oscillator( v_c_f,v_c_q,N,alfa,Fs)

%fc:frecuencia de oscilacion
 fc=11025;

% FILRO CONFORMADOR O TRANSMISOR
%para las componentes en fase.Su envolvente sera:
 tram_env_fas=rcosflt(v_c_f.',1,N,'sqrt',alfa);
%para las componentes en quadratura. Su envolvente sera:
 tram_env_quad=rcosflt(v_c_q.',1,N,'sqrt',alfa);

%dibujo las envolventes
% subplot(2,1,1);plot(tram_env_fas);title('envolvente en fase');
% subplot(2,1,2);plot(tram_env_quad);title('envolvente en cuadratura');
% disp('pulse una tecla para continuar');
% pause;

% OSCILADOR:
 m=0:length(tram_env_fas)-1;
%rama de fase
 trama_fase=(tram_env_fas.').*cos(2*pi*(fc/Fs)*m);
%rama de cuadratura
 trama_quad=(tram_env_quad.').*sin(2*pi*(fc/Fs)*m);

%dibujo las señales ya moduladas
% subplot(2,1,1);plot(trama_fase);title('datos de fase modulados');
% subplot(2,1,2);plot(trama_quad);title('datos de cuadratura modulados');
% disp('pulse una tecla para continuar');
% pause;

%formo la señal APK completa:
 trama_mod=trama_fase-trama_quad;

%dibujo la señal APK de la trama
% plot(trama_mod);title('trama modulada: APK');
% disp('pulse una tecla para continuar');
% pause;
```

Esta función acepta como entrada las componentes en fase y en cuadratura de los distintos símbolos de la trama, y se encarga de simular las dos ramas, de cuadratura y de fase.

Antes de nada, lo primero que hacemos es definir la ***frecuencia de oscilación***, f_c:

- *fc=11025*;

Acto seguido, implementamos el ***filtrado*** de la secuencia de símbolos en fase, *v_c_f*, y en cuadratura, *v_c_q*, con los ***pulsos raíz de coseno alzado***:

- *tram_env_fas=rcosflt(v_c_f.',1,N,'sqrt',alfa);*
- *tram_env_quad=rcosflt(v_c_q.',1,N,'sqrt',alfa);*

Ya dijimos que para dicho filtrado nos valdríamos de la función propia de Matlab *'rcosflt'*, cuyos parámetros de entrada , además de ambas secuencia de símbolos de entrada, serán:

- *1, N*: indica que la secuencia de muestras a la salida será N veces mayor que el tren de símbolos a la entrada. Es decir, que tendremos N muestras / símbolo, que, al fin y al cabo, es la definición de nuestra variable de diseño N.
- *'sqrt'*: indicamos que el filtro será raíz de coseno alzado.
- *alfa*: es nuestro factor de roll-off, cuyo valor fijamos en la función *'z_main'*.
- *retraso*: no aparece como parámetro de entrada, por eso la función toma, por defecto, 3 símbolos que preceden y otros tres que suceden,al filtrado. Son las $3 \cdot N$ muestras de cola a cada lado de la envolvente resultante.

Los vectores con las muestras en la rama de fase y de cuadratura, a la salida de ambos filtros transmisores, *tram_env_fas* y *tram_env_quad* respectivamente, son las señales $S_I(t)$ y $S_Q(t)$, que hemos reflejado en el ***Modulador APK***, durante la Fase de Diseño.

Dichas variables vectoriales pasarán a continuación por sendos ***Osciladores*** de fase y cuadratura. Primeramente generamos el índice temporal discreto, que almacenaremos en la variable *m*:

- *m=0:length(tram_env_fas)-1;*
- *trama_fase=(tram_env_fas.').*cos(2*pi*(fc/Fs)*m);*
- *trama_quad=(tram_env_quad.').*sin(2*pi*(fc/Fs)*m);*

Donde *Fs* es la ***frecuencia de muestreo***, que vamos arrastrando como parámetro de entrada, en las distintas llamadas a funciones donde la necesitemos, pues su valor, que de seguro será 44100, ya se definió durante la función *'z_main'*.

Vemos cómo usamos la función interna de Matlab, *'cos'*, para la rama de fase, y la función también propia de Matlab, *'sin'*, para la rama de cuadratura.

Por último, sólo nos falta restar ambas ramas, para obtener la trama modulada, es decir, la señal $S(t)_{APK}$ deseada, tal y como desarrollamos analíticamente en la fase de diseño III.2:

- $S_{APK}(t) = S_I(t) \cdot \cos(w_c \cdot t) - S_Q(t) \cdot \text{sen}(w_c \cdot t)$;

La secuencia de muestras de dicha señal la almacenamos en la variable de salida *trama_mod*, de la siguiente manera:

- *trama_mod=trama_fase-trama_quad;*

Dicho valor lo devolveremos a la función 'z_bloq_mod', que a su vez lo devolverá a la función 'z_transmite_fich', para que pase a la ***Fase de Entramado II***, como ya explicamos anteriormente.

También cabría añadir que hemos omitido sentencias donde dibujamos, mediante las funciones *'plot'* y *'subplot'*, las envolventes en fase y en cuadratura, sus correspondientes señales moduladas $S_I(t)\cdot\cos(w_c\cdot t)$ y $S_Q(t)\cdot\text{sen}(w_c\cdot t)$, y la señal $S_{APK}(t)$ final. Dichas gráficas las usamos durante la fase de pruebas, y ahora aparecen comentadas, es decir, precedidas del carácter *'%'*. Por consiguiente, si se quisiera volver a habilitar dichas líneas de comando y ver las gráficas de las distintas tramas, no tenemos más que suprimir dichos porcentajes.

Hasta aquí llega la fase de procesamiento en el ***Bloque Transmisor***.

III.3.2.- Programación en Matlab del Bloque Receptor

Adelantaremos primero la ***función principal***, como hicimos en el Bloque Transmisor, para ir desgranándola poco a poco, en cada tramo del correspondiente ***Bloque Receptor***.

El ***programa principal*** se encuentra en el archivo 'z_main.m'. La función *'z_main'* contenida en dicho archivo es la siguiente:

- **z_main:**

```
function z_main

 clc
 disp(' ');
 disp('                    PROYECTO FIN DE CARRERA');
 disp(' ');
 disp(' DISEÑO E IMPLEMENTACION DE UN MODEM APK MEDIANTE SOUNDBLASTER');
 disp(' ');
 disp(' ');
 disp('      AUTOR: Jose Miguel Moreno Perez');
 disp('      DIRECTOR: José Ramón Cerquides Bueno');
 disp(' ');
 disp(' ');
 disp('      ************************************************************');
 disp('      *                  Receptor                    *');
 disp('      ************************************************************');
 disp(' ');
 disp(' ');


%Preguntamos si deseamos meter parametros
 defecto=[];
 while (isempty(defecto)| (~isequal(defecto,'Y') & ~isequal(defecto,'y') &...
       ~isequal(defecto,'N') & ~isequal(defecto,'n')))
   defecto=input('      desea meter usted los datos?[Y/N]: ','s');
 end
%Parametros por defecto
 if isequal(defecto,'n')|isequal(defecto,'N')
   fichero='pruebarx.txt';M=256;N=10;BDT=20;Fs=44100;NB=16;alfa=0.7;N_B_fich=1000;
 else
%Introducir parametros
   fichero=input('      Introduzca el nombre del fichero que desea crear: ','s');
   M=input('      Número de niveles de la modulación APK (M=4/8/16/32/64/128/256): ');
   N=input('      Número de muestras por símbolo: ');
%BDT: bytes de datos de la trama(sin cabecera).
   BDT=input('      Longitud (bytes) del campo de datos de las tramas Ethernet: ');
   Fs=input('      Frecuencia de muestreo (Hz): ');
   NB=input('      Resolución (8 ó 16 bits): ');
%nº de bytes del fichero.
   N_B_fich=input('      Longitud (bytes) del fichero: ');
   alfa=input('      Factor de Rol-off (entre 0 y 1): ');
 end

%CALCULO APROXIMADO DE MUESTRAS A GRABAR
%n_m_piloto: son los 30 ceros, la señal piloto y los 50 ceros.
 n_m_piloto=81; %muestras
```

```
%retraso
 ret=6; %simbolos
%cab_trama: cabecera de la trama. Son 4 bytes(long(2), CRC(2)).
 cab_trama=4; %bytes.
%simb_fict: simbolos ficticios
 simb_fict=12;
%y añadimos 3 sg extra.
%nmuestras: nº de muestras que vamos a grabar con la funcion wavrecord.
nmuestras= round(N_B_fich*8*N/log2(M)) + ceil(N_B_fich/BDT)*( n_m_piloto + ret*N...
 + simb_fict*N + round(cab_trama*8*N/log2(M)))+3*44100; %3 sg extras
 disp(' ');
 disp(' ');
 disp(' ');
 disp('      PULSE CUALQUIER TECLA PARA COMENZAR A GRABAR...');
 pause;
 rx=wavrecord(nmuestras,Fs,1);
 disp(' ');
 disp('      PULSE CUALQUIER TECLA PARA PROCESAR LOS DATOS REGISTRADOS...');
 pause;
 z_recibe_fich(Fs,M,N,BDT,fichero,rx.',alfa);
 disp(' ');
 disp(' ');
 disp('      **********************FICHERO TRANSMITIDO**********************');
```

Al igual que en el programa principal del Bloque Transmisor, mostramos la misma ***presentación en la pantalla***, con la ayuda de las funciones *'clc'* y *'disp'*, cuyo funcionamiento ya lo explicamos en la función *'z_main'* del Bloque Transmisor.

Posteriormente, el programa nos ofrece la posibilidad de introducir los ***parámetros variables*** de nuestro sistema, o bien utilizar los que ya tiene por defecto. Obviamente, los parámetros de diseño, tanto los nuevos, como los que vienen ***por defecto***, deben coincidir con los que hemos fijado en el Bloque Transmisor, para un correcto funcionamiento de nuestro sistema.

Por lo tanto, al igual que en el transmisor, los parámetros que el sistema muestra por defecto son los usados en los ejemplos que hicimos con el fichero *'prueba.txt'*.

Ulteriormente, calculamos el ***número total de muestras a grabar*** mediante la función *'wavrecord'*. Este valor lo almacenamos en la variable *nmuestras*. Será un valor aproximado al número de muestras a transmitir (31 muestras menos concretamente, correspondientes a una señal piloto y 30 de sus ceros que no incluimos), más 3·44100 muestras, o sea, incluyo 3 segundos extras de muestras, para que nos dé tiempo a pulsar para transmitir en la sesión transmisora, después de pulsar para recibir en la sesión receptora. Para calcular dicho número de muestras, creamos varias variables:

- *n_m_piloto*: 81 muestras de la señal piloto y sus ceros.
- *retraso*: 6 símbolos del retraso.
- *cab_trama*: 4 bytes de los campos LONGITUD y CRC.
- *simb_fict:* 12 símbolos ficticios (de valor 0.4).

Luego, con la ayuda de las funciones ya comentadas, *'disp'* y *'pause'*, se nos pide que pulsemos una tecla para empezar a grabar los datos, y acto seguido se realiza la ***grabación de las muestras*** y los ***3 segundos extras*** mediante la llamada a la función:

- *rx=wavrecord(nmuestras,Fs,1);*

donde le indicamos el número total de muestras a grabar, *nmuestras*, la ***Frecuencia de muestreo*** usada en la grabación, *Fs*, y el parámetro de valor *1* indica simplemente que usamos ***un solo canal***.

Las muestras las almacenamos en el vector *rx*, que representará la señal recibida a la salida del canal.

Por último realizamos la llamada al grueso del ***Bloque Receptor***, la función:

- *z_recibe_fich(Fs,M,N,BDT,fichero,rx.',alfa);*

donde crearemos el ***Fichero De Destino***, *fichero*, a partir de la muestras recibidas, *rx*. Para ello, introduciremos como parámetros de entrada, todos los parámetros de diseño fijados en esta función principal.

Una vez terminado este procesamiento, la función principal culmina mostrando un mensaje por pantalla donde indica que el fichero ha sido transmitido.

Vayamos, por tanto, con el ***núcleo del bloque receptor***:

- **z_recibe_fich:**

```
function z_recibe_fich(Fs,M,N,BDT,fichero,rx,alfa)
%BLOQUE RECEPTOR
%M: modulación M-ARIA
%N: número de muestras por símbolo
%BDT: bytes de datos por trama
%fichero: el fichero destino que vamos a crear
%rx: datos registrados por la tarjeta de sonido
%alfa: factor de roll-off.

 disp(' ');
 disp('      PROCESANDO MUESTRAS. POR FAVOR, ESPERE...');
 disp(' ');

%BUSCO LA DELTA DE DIRAC PREVIA A TODO
 i=1;
  while abs(rx(i))<0.3
    i=i+1;
 end

%BUCLE DE IDENTIFICACION DE TRAMAS
%Elimino el silencio inicial
 rx=rx(i-30:length(rx));
%Voy A Recorrer Los Datos Recibidos Para Poder Separar Las Tramas Recibidas
%busco la delta previa a la trama
%indices: vector que contiene la posición de comienzo de cada trama
 indices=[];
 k=1;          %recorre los datos desde el ppio hasta el final
 while (k<length(rx))
   y=find(abs(rx(k:k+14)<0.01));
   if length(y)==15
     while (abs(rx(k))<0.3)&(k<length(rx))
       k=k+1;
     end
     indices=[indices,k-20];
     k=k+100;
   else
```

```
      k=k+1;
    end
  end
%FIN DE BUCLE DE IDENTIFICACION DE TRAMAS

%ASIGNACION PREVIA AL TRATAMIENTO DE LA PRIMERA TRAMA
%datos: variable donde vamos a almacenar el campo DATOS recibido
 datos=[];
 n=1;
%inicio:principio de cada trama
 inicio=indices(n);
%final:final de cada trama
 final=indices(n+1);
%tr: trama a procesar
 tr=rx(inicio:final);
%cont_tramas: contador de tramas
 cont_tramas=0;
%cont_err: contador de errores
 cont_err=0;

%BUCLE DE PROCESAMIENTO DE TRAMAS
 while final<indices(length(indices))
%err: flag que se pone a uno si existe trama erronea
    err=0;

%ECUALIZACION DE LA SEÑAL
%h: respuesta impulsiva
    h=tr(1:50);
%x:igualador de cero forzado
    x=z_igualadorZF(h);
%tr_eq:trama ecualizada
    tr_eq=conv(x,tr);

%ELIMINACION DE LA SEÑAL PILOTO Y DE SUS CEROS
    [muest_max,pos_max]=max(abs(tr_eq));
    tr_eq=tr_eq(pos_max+50+1:length(tr_eq));

%BLOQUE DEMODULADOR.
%Filtro Receptor y oscilador local
    [trama_env_fase, trama_env_quad]=z_bloq_dem(tr_eq,N,alfa);
%Detector de símbolos
%retraso de dos convoluciones: 2*3*N
%desecho las 6*N muestras de la cola del principio
    v_c_fase=trama_env_fase(2*N*3+1:N:length(trama_env_fase));
    v_c_quad=trama_env_quad(2*N*3+1:N:length(trama_env_quad));

%AJUSTE DE NIVELES
%INICIO DE TRAMA: 8 simbolos ficticios de valor 0.4+j0.4
%v_c_f_n:vector de componentes de fase 'normalizados' entre –1 y 1.
%v_c_q_n:vector de componentes de quadratura 'normalizados' entre –1 y 1.
    [v_c_f_n,v_c_q_n]=z_ajuste_niveles(v_c_fase,v_c_quad);

%ELIMINACIÓN DEL INICIO DE TRAMA
   v_c_f_n=v_c_f_n(9:length(v_c_f_n));
   v_c_q_n=v_c_q_n(9:length(v_c_q_n));

%EXTRACCION DEL CAMPO LONGITUD
%n_simb_long: nº de simbolos del campo LONGITUD de la trama
%son 2  bytes= 16  bits.
    n_simb_long=ceil(16/log2(M));
```

```
%DECISION DE LOS SIMBOLOS del campo LONGITUD
    [v_simb_dec]=z_calc_simb_dec(v_c_f_n(1:n_simb_long),v_c_q_n(1:n_simb_long),M);
%OBTENCION DE LOS BITS del campo LONGITUD
    bits_dec=z_form_bits(v_simb_dec,M);

%PROCESADOR DEL CAMPO LONGITUD
%Campo LONGITUD=2 bytes. Desde el bit 1 hasta el 16
%n_bytes:numero de bytes del campo DATOS
    n_bytes=bin2dec(bits_dec(1:16));
%Nota:
%si M=8, 32, 64 ó 128. Los bits a partir del 17 pertenecen al campo DATOS
%aux: vector auxiliar que contiene los primeros bits del campo DATOS
    aux=bits_dec(17:length(bits_dec));

%EXTRACCION DE LOS CAMPOS DATOS Y CRC
%extraemos el resto de símbolos de los campos DATOS Y CRC
%n_simb_rest: número de símbolos restantes que hay en DATOS Y CRC
    n_simb_rest=ceil( ( (n_bytes+2)*8-length(aux) )/log2(M) );
%DECISION DE LOS SIMBOLOS deDATOS y CRC-16.
    [v_simb_dec]=z_calc_simb_dec(v_c_f_n(n_simb_long+1:n_simb_long+n_simb_rest),...
    v_c_q_n(n_simb_long+1:n_simb_long+n_simb_rest),M);
%OBTENCION DE LOS BITS de DATOS y CRC-16.
    bits_dec=z_form_bits(v_simb_dec,M);

%Inserto el principio del campo DATOS: aux
%bits_dec: contiene DATOS+CRC
    bits_dec=[aux bits_dec];

%Eliminamos los bits últimos que ya no pertenezcan al campo CRC-16
    while mod(length(bits_dec),8)
       bits_dec(length(bits_dec))=[];
    end

%PROCESADOR DEL CAMPO DATOS
%los ultimos 16 bits corresponden al CRC-16
    for i=1:8:length(bits_dec)-16
       dat_aux=bin2dec(bits_dec(i:i+7));
       datos=[datos ; dat_aux];
    end

%PROCESADOR DEL CAMPO CRC
%usamos DATOS+CRC-16 para comprobar la existencia de errores
%resto: es el resto de la división. Si es nulo no existen errores.
    resto=z_calc_crc(bits_dec,1);
%si existe error en la trama, muestro un mensaje y pongo su flag a 1.
%es decir, si resto no es una cadena de ceros...
    if strcmp(resto,'0000000000000000')==0
         err=1;
    end

%CONTADOR DE TRAMAS Y DE ERRORES.
    cont_tramas=cont_tramas+1;
%si existe error indicamos trama erronea
    if err==1
       cont_err=cont_err+1;
       aux=sprintf('Trama: %d erronea', cont_tramas);
    else
       aux=sprintf('Trama: %d',cont_tramas);
    end
    disp(aux);
```

```
%ACTUALIZACION  DEL BUCLE
%ASIGNACION DE VALORES DE LA NUEVA TRAMA
    n=n+1;
    inicio=indices(n);
    final=indices(n+1);
    tr=rx(inicio:final);
 end
%FIN DE BUCLE DE PROCESAMIENTO DE TRAMAS

%PRESENTACION DE RESULTADOS EN PANTALLA
    aux=sprintf('Numero de Tramas transmitidas: %d', cont_tramas);
    disp(' ');
    disp(aux);
    aux=sprintf('Numero de Tramas erroneas: %d', cont_err);
    disp(' ');
    disp(aux);

%DESTINO DIGITAL.CREACION DEL FICHERO.
    disp(' ');
    disp(' ');
    disp('      creando el nuevo fichero...');
    fid=fopen(fichero,'w');
    count=fwrite(fid,datos);
    fclose(fid);
```

Tras un mensaje mostrado por pantalla para indicarnos que las muestras se están procesando, el primer bloque con el que nos encontramos, es el que se corresponde con la ***Fase de Desentramado II***. El vector que ejerce de ***Acumulador de Tramas*** es la variable *rx*, donde almacenamos las muestras de todas las tramas de la señal recibida, además de los 3 segundos de silencio.

Seguidamente, nos enganchamos a la primera de todas las 'deltas' recibidas (que ya decimos que no se trata realmente de deltas, si no de respuestas impulsivas del canal, de ahí que entrecomillemos dicha palabra), para marcar el ***inicio de la señal***. Lo hacemos de la siguiente manera:

```
i=1;
while abs(rx(i))<0.3
    i=i+1;
end
```

Este bucle simplemente recorre la variable *rx* (lógicamente empezará recorriendo parte de los 3 segundos de silencio), hasta que se encuentra con un valor mayor a 0.3.

Al salir del bucle, tenemos, en la variable *i*, la posición dentro de *rx*, de la primera 'delta' recibida. Esto es así, porque si nos fijamos mejor en la imagen de una de las 'deltas' mostradas durante la Fase de Diseño, veremos que la muestra correspondiente al máximo recibido de la señal piloto, tendrá siempre valor mayor que 0.3. En el mejor de los casos, donde el desplazamiento de las tramas es mínimo, dicha muestra puede alcanzar un valor de 0.8. En el peor de los casos, donde exista mucho desincronismo, el pico de la 'delta' recibida puede repartirse en dos muestras, alcanzando siempre cada una de ellas un valor mínimo de 0.5 ó 0.6. Todos estos datos los obtenemos a partir de pruebas empíricas.

Veamos, por ejemplo, para nuestro fichero *'prueba.txt'*, y en las condiciones que usamos durante los ejemplos desarrollados en la Fase de Diseño III.2, la primera de esas 'deltas' recibidas (sin pérdida de generalidad):

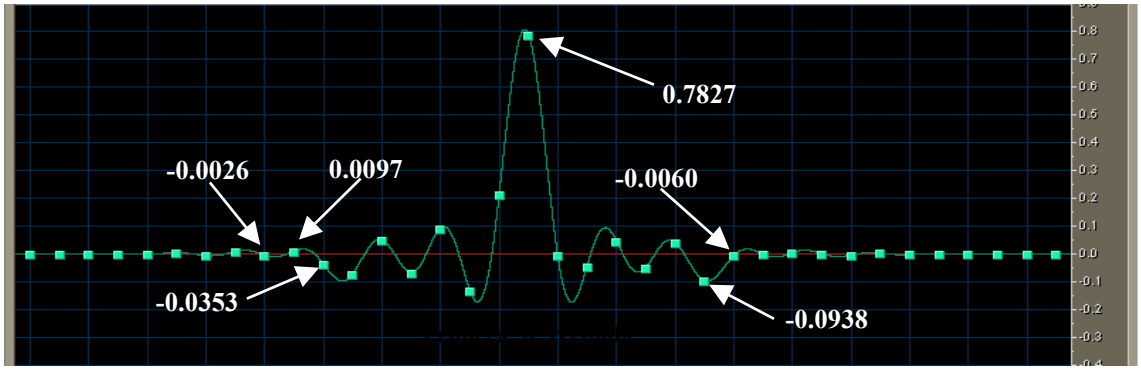


Fig. 109.- Detalle respuesta impulsiva

Comprobamos visualmente que, en efecto, el máximo será de 0.78, mayor que 0.3. Podemos apreciar también que, a partir de la 8ª o 9ª muestra a cada lado del máximo de dicha señal piloto, los valores son menores a 0.01. Este hecho lo usaremos en el siguiente bucle de identificación de tramas como un umbral a partir del cual consideramos si existe silencio o no silencio.

El bucle que le sigue dentro de la función *'z_recibe_fich'*, por tanto, implementa el ***Identificador de Tramas***, subbloque de la ***Fase de Desentramado II*** que ya mostramos en la Fase de Diseño III.2:

- *BUCLE DE IDENTIFICACION DE TRAMAS*

```
%Elimino el silencio inicial
 rx=rx(i-30:length(rx));
%Voy A Recorrer Los Datos Recibidos Para Poder Separar Las Tramas Recibidas
%busco la delta previa a la trama
%indices: vector que contiene la posición de comienzo de cada trama
 indices=[];
 k=1;        %recorre los datos desde el ppio hasta el final
 while (k<length(rx))
   y=find(abs(rx(k:k+14)<0.01));
   if length(y)==15
     while (abs(rx(k))<0.3)&(k<length(rx))
       k=k+1;
     end
     indices=[indices,k-20];
     k=k+100;
   else
     k=k+1;
   end
 end
%FIN DE BUCLE DE IDENTIFICACION DE TRAMAS
```

Como dijimos, tenemos el comienzo de la primera trama, marcado por el máximo de la señal piloto que la antecede, en la muestra número *i* del ***Acumulador de Tramas*** *rx*. Eliminamos todas las muestras anteriores (correspondientes al silencio),

quedándonos con las muestras a partir de 30 muestras anteriores a la posición almacenada en *i*.

Redefinimos así el vector rx:

- *rx=rx(i-30:length(rx));*

Ya anticipamos, que a partir de la 8ª o 9ª muestra previa al máximo de cada señal piloto, los valores son menores que 0.01. Por consiguiente, las primeras muestras del nuevo vector *rx* tendrá un valor prácticamente nulo.

Posteriormente, y siguiendo con el proceso de ***Identificación de Tramas***, nos encontramos con dos bucles anidados. El ***bucle externo*** sirve para recorrer completamente el vector *rx* mediante el índice *k*:

- *while (k<length(rx))*
- ….
- *end*

Dentro de dicho bucle, vamos examinando, de 15 en 15 muestras, si cada una de esas 15 muestras son prácticamente nulas:

- *y=find(abs(rx(k:k+14)<0.01));*

Para ello nos valemos de la función propia de Matlab, *'find'*, que evalúa una expresión lógica para cada miembro de un vector de entrada , y nos devuelve otro vector a su salida con los números de posiciones del vector de entrada, que cumplen dichas condiciones lógicas. En nuestro caso evaluamos si cada una de las 15 muestras del vector *rx*, escogidas en cada iteración, son menores en valor absoluto que 0.01. El vector *y* almacena las posiciones de las muestras que cumplen la condición anterior.

Si las 15 muestras cumplen dicha condición, tenemos garantizado que estamos en una zona de ceros. Como las primeras muestras del vector rx son prácticamente nulas, al empezar el bucle externo (k=1), tenemos asegurado que las primersa 15 muestras cumplen dicha condición. Esto equivale a decir que el vector *y* tiene una longitud igual a 15, porque almacena las quince posiciones relativas a cada muestra. Es decir:

- *if length(y)==15*

Si cumplimos dicha condición, veremos, más adelante, que siempre nos encontraremos al comienzo de la zona de 30 ceros previos al máximo de la señal piloto (no olvidemos que la señal piloto estaba arropada a su izquierda por 30 ceros, y a su derecha por cincienta ceros antes del comienzo de la cola). Por tanto, en el ***bucle interno***, volveremos a repetir el proceso de búsqueda de la siguiente 'delta' recibida:

- *while (abs(rx(k))<0.3) & (k<length(rx))*
- *k=k+1;*
- *end*

En este caso también imponemos la condición *k<length(rx)*, por si se trata de la última 'delta' (detrás de la cual solo existen cincuenta muestras prácticamente nulas y silencio, cuyas muestras, *rx(k),* no van a cumplir la condición *abs(rx(k))<0.3*), se salga entonces del bucle al terminar el vector *rx*.

Al salir de este bucle interno, *k* contendrá la posición de *rx* donde existe un máximo de la señal piloto, execepto en la última iteración externa, donde contendrá la posición de la última muestra del vector *rx*. Las posiciones de la muestra vigésima a la izquierda de cada máximo, *k-20*, (y la 20ª antes de la última muestra de *rx*), las incluiremos en el vector *índices*:

- *indices=[indices,k-20];*

Por último incrementamos el valor de *k* en 100 posiciones, *k=k+100*, teniendo en cuenta que desde el máximo de cada señal piloto hasta el comienzo de la cola existen 50 ceros, y que dicho retraso (cuyas muestras pueden estar por debajo de 0.01), hasta el Inicio de Trama, ocupa 30 muestras. Así, adelantando en 100 posiciones el valor de *k*, correspondiente al máximo de la señal piloto, podemos asegurar estar en una zona de la trama donde no existe silencio.

La próxima vez que entremos en el bucle externo, estaremos en una zona de la trama, como dijimos, y no cumpliremos la condición *if length(y)==15*, por lo que no entraremos en el bucle interno, y simplemente adelantaremos una posición k, *k=k+1*, adelantando de igual modo el nuevo bloque de 15 muestras a evaluar en la siguiente iteración externa. Llegará un momento en que se vuelva a cumplir de nuevo que las 15 muestras elegidas sean de menor valor que 0.01, es decir, que volvamos a estar al comienzo de la zona de 30 ceros previos a la 'delta', como preveíamos, y se vuelva a entrar en el bucle interno para almacenar una nueva posición en el vector *indices*, y así hasta llegar al final del vector *rx*.

Veamos un ***ejemplo*** aclaratorio de este ***Identificador de Tramas***. Supongamos que tenemos el siguiente vector *rx*, formado por cuatro tramas. En el dibujo indicamos las posiciones que almacenaría el vector *índices*, y el salto de 100 muestras que se produce cada vez que se encuentra un máximo, y cómo esta nueva posición cae dentro de una trama. Hay que hacer notar no hemos dibujado el largo silencio a ambos lados de dicha señal *rx*.

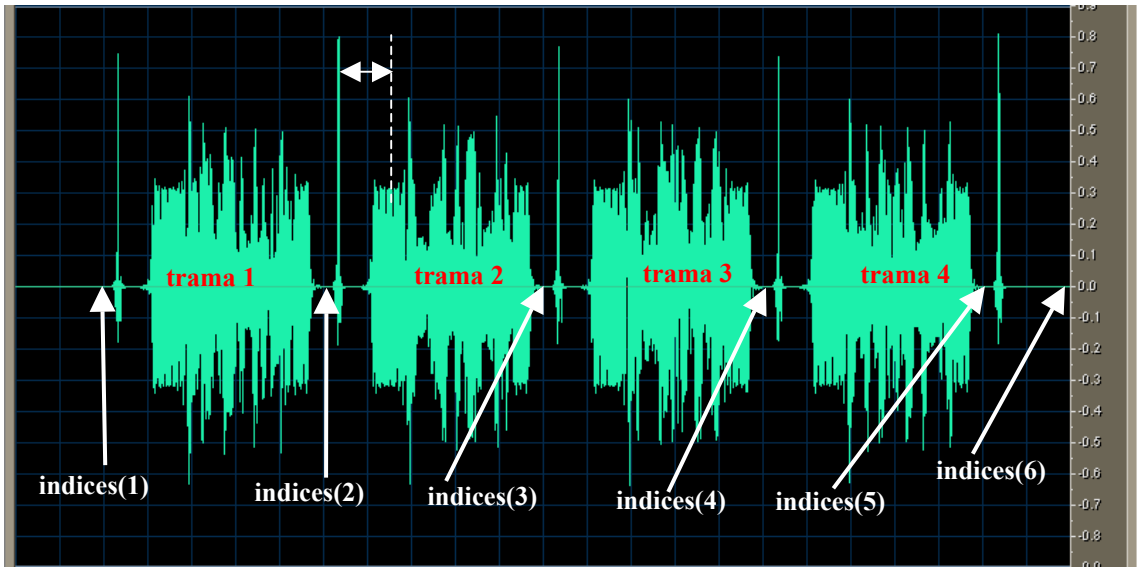


Fig. 110.- Detalle de identificación de tramas

El vector *indices* contiene las posiciones, dentro de *rx*, del comienzo de cada trama. Esto es, como ya anunciamos durante la Fase de Diseño III.2, nuestra trama va a comenzar 20 muestras antes del máximo de la señal piloto, y va a terminar 20 muestras antes del máximo de la siguiente señal piloto. Así, el primer elemento de *índices*, *indices(1)*, marca la posición, dentro de *rx*, de la primera trama. Por otra parte, el segundo elemento, *indices(2)*, marcará la posición, dentro de *rx*, del final de la primera trama, que a su vez, como veremos, será el comienzo de la segunda trama. Así, los elementos consecutivos de *indices* marcarán la longitud de cada trama.

Antes de entrar en el bucle que procesa cada trama, realizamos la ***asignación de valores a la primera trama***:

- *ASIGNACION PREVIA AL TRATAMIENTO DE LA PRIMERA TRAMA*

  - *%datos: variable donde vamos a almacenar el campo DATOS recibido*
  - *datos=[];*
  - *n=1;*
  - *%inicio:principio de cada trama*
  - *inicio=indices(n);*
  - *%final:final de cada trama*
  - *final=indices(n+1);*
  - *%tr: trama a procesar*
  - *tr=rx(inicio:final);*
  - *%cont_tramas: contador de tramas*
  - *cont_tramas=0;*
  - *%cont_err: contador de errores*
  - *cont_err=0;*

La variable *n* recorrerá el vector *indices*. La variable *inicio* se corresponderá con la posición, dentro del vector *rx*, del *inicio* de la trama a procesar, y la variable *final*, marcará la posición, dentro de *rx*, de la última muestra de la trama a procesar. Por tanto, *inicio* será siempre igual a *índices(n)*, y *final* equivaldrá a *indices(n+1)*. Así, la primera trama, estará contenida, dentro de *rx*, entre los valores *indices(1)* e *indices(2)*.

La variable *tr* será un vector que contendrá únicamente las muestras referentes a la trama a procesar. Así, *tr(1)* se corresponderá con *rx(inicio)*, y la última posición de tr, coincidirá con *rx(final)*. De este modo, la asignación a *tr* es la siguiente:

- *tr=rx(inicio:final);*

Por último, inicializamos la variable *cont_tramas*, que implementa el ***Contador de Tramas*** mostrado en el dibujo de la ***Fase de Desentramado I***, analizada en la Fase de Diseño III.2. Por otro lado, también inicializamos el ***Contador de Errores***, *cont_err*, también presente en dicha Fase de Desentramado I.

Una vez tenemos el valor para la primera trama, entramos en el ***bucle de procesamiento tramas***, donde éstas se procesarán,como bien sabemos, individualmente.

Es recomendable que se eche un vistazo a las distintas gráficas de la *'trama 36'*, expuestas durante los numerosos ejemplos que llevamos a cabo con el fichero 'prueba.txt', durante el transcurso de la Fase de Diseño:

El esqueleto de tan extenso bucle es el siguiente:

- *%BUCLE DE PROCESAMIENTO DE TRAMAS*
- *while final<indices(length(indices))*

```
    .................
    %ECUALIZACION DE LA SEÑAL
    .................
    %ELIMINACIÓN DE LA SEÑAL PILOTO Y SUS CINCUENTA CEROS.
    .................
    %BLOQUE DEMODULADOR
    .................
    %AJUSTE DE NIVELES
    .................
    %ELIMINACIÓN DEL INICIO DE TRAMA
    ...............
    %CAMPO LONGITUD
    ..............
    %CAMPO DE DATOS
    ..............
    %CAMPO CRC-16
    ..............
    %CONTADOR DE TRAMAS Y DE ERRORES.
    ...............
    %ACTUALIZACION DE VALORES DE LA NUEVA TRAMA
      n=n+1;
      inicio=indices(n);
      final=indices(n+1);
      tr=rx(inicio:final);
end
%FIN DE BUCLE DE PROCESAMIENTO DE TRAMAS
```

Antes de comenzar con el estudio de cada bloque del bucle, vamos a ver la forma que tiene éste de actualizarse, al final de cada iteración, y la condición que ha de cumplirse para que éste deje de ejecutarse.

La forma de actualizarse no tiene más misterio que incrementar la variable que recorre el vector *indices*. De esta forma, si estamos en el procesamiento, para el ejemplo anterior, de la primera trama (*n=1*), durante el bucle, *inicio* tendrá valor *indices(1)* y *final* será igual a *indices(2)*. Al ejecutarse el ***tramo de actualización de la trama***, *inicio* pasará a valer *indices(2)* y *final* adquirirá un valor igual a *indices(3)*. Esto significa que al ejecutarse seguidamente *tr=rx(inicio:final);*, la variable *tr* representará ahora la segunda trama. Cuando se procese la cuarta y última trama (en nuestro ejemplo), inicio adquirirá el valor *indices(5)* y *final* tomará el valor *indices(6)*. Si vemos el dibujo, ahí ya no existe trama, pero si miramos la condición del bucle, *final<indices(length(indices))*, dichos datos contenidos en la nueva *tr* (silencio mayormente) no se procesarán, pues *final* será igual en este caso a *indices(length(indices))*, y nos saldremos del bucle, de un modo congruente, puesto que no existen más tramas que procesar.

Ahora sí, vamos a ver cada tramo de este amplio bucle de procesamiento de tramas. Para tener una visión global de la función *'z_recibe_fich'*, y especialmente del anterior 'megabucle', las llamadas a funciones dentro de dicho bucle las comentaremos por encima, enumerando sus entradas y sus salidas (como si de cajas negras se trataran) para analizarlas una vez acabada la explicación de esta función.

- *ECUALIZACION DE LA SEÑAL*

```
%h: respuesta impulsiva
 h=tr(1:50);
%x:igualador de cero forzado
```

- *x=z_igualadorZF(h);*
- *%tr_eq:trama ecualizada*
- *tr_eq=conv(x,tr);*

Sea *tr* la trama a procesar, lo primero que hacemos es almacenar, en la variable vectorial *h*, las primeras cincuenta muestras de *tr*, que se corresponden con la ***respuesta impulsiva*** previa a dicha trama (recordemos las características dinámicas y óptimas de nuestro ***Ecualizador***):

- *h=tr(1:50);*

Luego llamamos a la función *'z_igualadorZF'*, para, a partir de dicha ***respuesta impulsiva***, *h*, obtener el vector *x*, que contiene las muestras del ***Igualador*** creado en esta función:

- *x=z_igualadorZF(h);*

Ya no tenemos más que ecualizar la trama completa. Es decir, vamos a realizar la convolución, con la ayuda de la función *'conv'* propia de Matlab, de la trama *tr* con el ecualizador óptimo para esa trama, *x*.

- *tr_eq=conv(x,tr);*

Las muestras de la trama ecualizada se almacenarán en la variable *tr_eq*.

- *ELIMINACIÓN DE LA SEÑAL PILOTO Y SUS CINCUENTA CEROS*

  - *[muest_max,pos_max]=max(abs(tr_eq));*
  - *tr_eq=tr_eq(pos_max+50+1:length(tr_eq));*

Una vez ecualizada la trama, a la ***Salida del Ecualizador***, completaremos la ***Fase de Desentramado II***, con la ***eliminación de la señal piloto***, que ya ha cumplido el cometido para la que fue creada, es decir, alineamiento y ecualización dinámica.

Para ello, primeramente buscamos la delta ecualizada, que coincidirá con la muestra de mayor amplitud. Posteriormente, redefinimos el contenido de la trama ecualizada, *tr_eq*, eliminando de ésta la ***cola*** que introduce el ***Ecualizador***, los 20 ceros anteriores a la delta, la propia delta, y los cincuenta ceros posteriores a la misma. Por tanto, la primera muestra para *tr_eq* se corresponderá con el comienzo del ***retraso de grupo*** que introduce el ***Pulso Transimisor raíz de coseno alzado***.

- *BLOQUE DEMODULADOR.*

  - *%Filtro receptor y oscilador local*
  - *[trama_env_fase, trama_env_quad]=z_bloq_dem(tr_eq,N,alfa);*
  - *%Detector de símbolos*
  - *%retraso de dos convoluciones: 2*3*N*
  - *%desecho las 6*N muestras de la cola del principio*
  - *v_c_fase=trama_env_fase(2*N*3+1:N:length(trama_env_fase));*
  - *v_c_quad=trama_env_quad(2*N*3+1:N:length(trama_env_quad));*

Llegamos al ***Bloque Demodulador***, que lo dibujamos interno al ***Detector APK***. Como podemos ver en los esquemas de la Fase de Diseño III.2, dicho demodulador recibirá a la entrada la trama recibida y ecualizada, que denominamos en su momento $r(t)_{APK}$, y que está representada por la variable *tr_eq*.

La llamada a la función:

- *[trama_env_fase, trama_env_quad]=z_bloq_dem(tr_eq,N,alfa);*

implementará el paso de la señal $r(t)_{APK}$ a través del Oscilador y del Filtro Rececptor, tanto para la rama de fase como para la rama de cuadratura. Para diseñar el ***Filtro receptor raíz de coseno alzado***, introducimos como parámetros de entrada, además de *tr_eq*, el número de muestras por símbolo, *N*, y el factor de Roll-off, *alfa*.

Las salidas de esta función, *trama_env_fase* y *trama_env_quad*, como es de esperar, serán la envolvente en fase y la envolvente en cuadratura de la señal recibida, esto es, las señales $r_I(t)$ y $r_Q(t)$, tal y como la denominamos en el esquema del Demodulador, visto durante la Fase de Diseño.

Las siguientes dos líneas de programa implementan la labor del ***Detector de Símbolos***, para las ramas de fase y cuadratura respectivamente:

- *v_c_fase=trama_env_fase(2*N*3+1:N:length(trama_env_fase));*
- *v_c_quad=trama_env_quad(2*N*3+1:N:length(trama_env_quad));*

A partir de la muestra número $6 \cdot N+1$ (no olvidemos que *N* es el número de muestras por símbolo), y hasta el final de la trama de envolvente de fase (y de cuadratura), vamos quedándonos con una muestra de cada *N* de ellas. Esto es, estamos detectando los símbolos recibidos, tal y como explicamos durante la Fase de Diseño.
Además, eliminamos, de paso, el ***retraso inicial*** de $6 \cdot N$ muestras que introducen los dos ***Filtros Raíz de coseno alzado***.

De esta manera, las variables de salida *v_c_fase* y *v_c_quad*, se corresponden con las secuencias de símbolos $\{I''_n\}$ y $\{Q''_n\}$, a la salida del detector de símbolos, tal y como las denominamos en la Fase de Diseño.

- *AJUSTE DE NIVELES*

  - *%INICIO DE TRAMA: 8 simbolos ficticios de valor 0.4+j0.4*
  - *%v_c_f_n:vector de componentes de fase 'normalizados' entre –1 y 1.*
  - *%v_c_q_n:vector de componentes de quadratura 'normalizados' entre –1 y 1.*
  - *[v_c_f_n,v_c_q_n]=z_ajuste_niveles(v_c_fase,v_c_quad);*

El bloque que sigue al ***Demodulador***, dentro del ***Detector APK***, es el ***Ajustador de Niveles***. Su misión es la de devolver a los símbolos contenidos en *v_c_fase* y *v_c_quad* al rango (-1, 1) del que partimos en el receptor, para una decisión adecuada del bloque de bits correspondiente en la posterior ***Tabla de Decisión***. Para ello, vimos que nos valíamos de ***símbolos ficticios y controlados***, que están en las 8 primeras posiciones de *v_c_fase* y *v_c_quad*. A la salida de este Ajustador de Niveles, obtendremos los símbolos normalizados entre –1 y 1, esto es, *v_c_f_n*, para la rama de fase, y *v_c_q_n*, para la rama

de cuadratura, que se corresponderán con los trenes de símbolos $\{I'_n\}$ y $\{Q'_n\}$ respectivamente, vistos en el Detector APK durante la Fase de Diseño.

- *ELIMINACIÓN DEL INICIO DE TRAMA*

  - *v_c_f_n=v_c_f_n(9:length(v_c_f_n));*
  - *v_c_q_n=v_c_q_n(9:length(v_c_q_n));*

Una vez realizado dicho Ajuste de Niveles, procedemos a la ***eliminación del inicio de trama***, en ambas ramas, volviendo a almacenar el resultado en *v_c_f_n* y *v_c_q_n*.

- *EXTRACCION DEL CAMPO LONGITUD*

  - *%n_simb_long: nº de simbolos del campo LONGITUD de la trama*
  - *%son 2 bytes= 16 bits.*
  - *n_simb_long=ceil(16/log2(M));*
  - *%DECISION DE LOS SIMBOLOS del campo LONGITUD*
  - *[v_simb_dec]=z_calc_simb_dec(v_c_f_n(1:n_simb_long),v_c_q_n(1:n_simb_long),M);*
  - *%OBTENCION DE LOS BITS del campo LONGITUD*
  - *bits_dec=z_form_bits(v_simb_dec,M);*

Ahora nos quedamos con los símbolos correspondientes a los dos primeros bytes de la trama, que forman el campo LONGITUD. Para el caso de *M*=4, cada símbolo lo representa un bloque de dos bits, y por tanto, con 8 símbolos obtenemos justamente los 16 bits que necesitamos, y no tendríamos más que dividir el número de bits del campo Longitud, es decir, 16, entre el número de bits que componen cada símbolo, que para *M*=4 ya hemos dicho que sería 2. Sin embargo para otros valores de M, como por ejemplo *M*=128, cada símbolo lo compone un bloque de 7 bits, y si volviéramos a realizar la misma división, es decir, 16 bits entre 7 bits/símbolo, obtendríamos 2.29 símbolos, esto significa que necesitaríamos al menos 3 símbolos para tomar los 16 bits del campo LONGITUD. Sin embargo, de los 21 bits correspondientes a estos 3 símbolos, 16 pertenecerán al campo LONGITUD y el resto, o sea, 5, pertenecerán al campo DATOS. Por esta razón usamos el comando *'ceil'* de Matlab, porque toma un valor real a la entrada y lo aproxima al entero inmediatamente superior, es decir, *ceil*(16/7)=*ceil*(2.29)=3:

- *n_simb_long=ceil(16/log2(M));*

La variable *n_simb_long* almacena el número de símbolos que necesitamos para tomar el campo LONGITUD.

Con los primeros símbolos, pertenecientes al campo LONGITUD, de los trenes de símbolos en fase y cuadratura, *v_c_f_n(1:n_simb_long)* y *v_c_q_n(1:n_simb_long)* entramos en el ***Decisor del Detector APK***, implementado mediante la función:

- *[v_simb_dec]=z_calc_simb_dec(v_c_f_n(1:n_simb_long),v_c_q_n(1:n_simb_long),M);*

Dicha función realizará la decisión sobre qué símbolo se ha transmitido, y reubicará cada uno de los símbolos en la posición del punto de mensaje ideal que se haya decidido como símbolo transmitido para dicho símbolo recibido. El vector de salida, *v_simb_dec*, es un vector de números complejos, donde la parte real representa las distancias en fase, y la parte imaginaria encarna las coordenadas en cuadratura de los símbolos reubicados en posiciones ideales dentro de la constelación, tras la debida decisión.

Una vez que tenemos dichas posiciones de puntos de mensaje ideales, podemos usar la conveniente ***Tabla de Conversión M-aria a Binaria***, cuya correspondencia entre bloques de bits y símbolos es idéntica a la vista en el Bloque Transmisor, excepto que en estas tablas entramos con un tren de símbolos complejos, *v_simb_dec*, y salimos con un bloque de bits por cada símbolo de entrada. Lógicamente, y como vimos durante la Fase de Diseño, tiene que existir un ***Convertidor Paralelo a Serie***, para convertir los distintos bloques de bits a la salida de la tabla, en una cadena binaria. Toda esta conversión de símbolos a bits y de paralelo a serie, se realiza mediante la llamada a la función:

- *bits_dec=z_form_bits(v_simb_dec,M);*

En la variable *bits_dec*, tendremos ya la cadena de bits ($\{b_m\}$, tal y como lo denominamos durante la Fase de Diseño), relativos a los símbolos introducidos a la entrada de la Tabla, *v_simb_dec*.

- *PROCESADOR DEL CAMPO LONGITUD*

  - *%Campo LONGITUD=2 bytes. Desde el bit 1 hasta el 16*
  - *%n_bytes:numero de bytes del campo DATOS*
  - *n_bytes=bin2dec(bits_dec(1:16));*
  - *%Nota:*
  - *%si M=8, 32, 64 ó 128. Los bits a partir del 17 pertenecen al campo DATOS*
  - *%aux: vector auxiliar que contiene los primeros bits del campo DATOS*
  - *aux=bits_dec(17:length(bits_dec));*

Estos bits pasarán al ***Procesador de Campos***, según nos muestran los esquemas de la Fase de Diseño, donde la variable *bits_dec* constituiría el 'bufer' de dicho campo. Aquí se dividirá la secuencia de bits en dos bloques:

- bits 1 al 16: corresponderán al campo LONGITUD.
- bits 17 en adelante: corresponderán al campo DATOS.

Así, los primeros 16 bits, *bits_dec(1:16)*, se mandarán al ***Procesador de Longitud***, donde se calculará el valor en decimal correspondiente al número binario formado por estos dos bytes.

- *n_bytes=bin2dec(bits_dec(1:16));*

Dicho valor en decimal, que se almacenará en la variable *n_bytes*, contendrá el número de bytes del campo DATOS que suceden al campo LONGITUD. Dicha variable numérica, *n_bytes*, se remitirá al ***Decisor del Detector APK***, para que éste se quede con

ese número de bytes, más los 2 siguientes correspondientes al campo CRC, y desprecie todos los demás, evitando así cálculos inservibles.

Por otro lado, los siguientes bits hasta el final de *bits_dec*, constituirán los primeros bits del campo DATOS, y se almacenarán en otra variable auxiliar, *aux*, a la espera de ser insertados, una vez que se hayan extraído todos los bits restantes de este campo DATOS:

- *bits_dec(17:length(bits_dec));*

Por ejemplo, para *M*=128, necesitaríamos, como ya mencionamos, 3 símbolos para obtener los 16 bits del campo LONGITUD. Así, de los 21 bits extraídos, 16 pertenecerían al campo LONGITUD, y los bits del 17 al 21 corresponderían al campo DATOS.

- *EXTRACCION DE LOS CAMPOS DATOS Y CRC*

```
%extraemos el resto de símbolos de los campos DATOS Y CRC
%n_simb_rest: número de símbolos restantes que hay en DATOS Y CRC
 n_simb_rest=ceil( ( (n_bytes+2)*8-length(aux) )/log2(M) );
%DECISION DE LOS SIMBOLOS deDATOS y CRC-16.
 [v_simb_dec]=z_calc_simb_dec(v_c_f_n(n_simb_long+1:n_simb_long+...
n_simb_rest),v_c_q_n(n_simb_long+1:n_simb_long+n_simb_rest),M);
%OBTENCION DE LOS BITS de DATOS y CRC-16.
 bits_dec=z_form_bits(v_simb_dec,M);

%Inserto el principio del campo DATOS: aux
%bits_dec: contiene DATOS+CRC
 bits_dec=[aux bits_dec];

%Eliminamos los bits últimos que ya no pertenezcan al campo CRC-16
 while mod(length(bits_dec),8)
    bits_dec(length(bits_dec))=[];
 end
```

De nuevo en el ***Decisor***, y una vez obtenido el valor en decimal del número de bytes del campo DATOS, ya podremos seleccionar los símbolos que restan de los campos DATOS y CRC, y despreciar el resto, que ya comentamos en la Fase de Diseño que eran resultado de las ***colas*** de convolución con los ***pulsos raíz de coseno alzado*** y del ***Ecualizador***.

Lo primero es calcular el número de símbolos del campo DATOS y CRC, que restan por decidir, y transformarlos en bits. Este número lo almacenaremos en una variable que llamaremos *n_simb_rest*, o sea, ***número de símbolos restantes***:

- *n_simb_rest=ceil(((n_bytes+2)*8-length(aux))/log2(M));*

Analizaremos el cálculo realizado mediante un ***ejemplo numérico***:

- Supongamos que nuestras tramas son de 20 bytes de DATOS, por lo que nuestra variable *n_bytes* tendría un valor 20 en cada iteración.
- El número de bytes que forman, por tanto, los campos DATOS y CRC sería igual a n_bytes+2 = 22, ya que el CRC ocupa dos bytes.

- Consecuentemente, el número de bits que forman estos campos será igual a (*n_bytes*+2) bytes * *8* bits/byte = 176 bits.
- A estos bits les restamos los del principio del campo DATOS, *length(aux)*, almacenados en la variable auxiliar *aux*, y que ya fueron extraídos de los trenes de símbolos en fase y cuadratura a la entrada del Decisor, *v_c_f_n* y *v_c_q_n*, durante la extracción de los bits del campo LONGITUD. Si suponemos encontrarnos en un sistema 128-ario, tendríamos que restar de los bits 17 al 21, es decir, *length(aux)* = 5 bits.
- Por tanto, el número de bits que restan por decidir será igual a:

  - *(n_bytes+2)*8-length(aux)* = 176 – 5= 171 bits restantes.

- El número de símbolos restantes será, por consiguiente:

  - *(n_bytes+2)*8-length(aux))* bits */ log2(M)* bits/simbolo = 171/7= 24.43

- Necesitamos un mínimo 25 símbolos para abarcar todos los bits restantes, así que volvemos a hacer uso del comando *'ceil'*, para obtener esa cifra:

  - *ceil (* 24.43*)* = 25 símbolos

- Así, tendremos que extraer más bits de los necesarios, concretamente extraeremos 4 bits más de los 171 que necesitamos:

  - 25 símbolos * 7 bits/símbolo = 175 bits extraídos

Ahora sí, decidimos sobre cada uno de los símbolos restantes, y reubicamos su posición, situándolo en la localización del punto de mensaje ideal más cercano donde aquél haya caído, siguiendo el ***Criterio de Decisión de Máxima Verosimilitud***:

```
[v_simb_dec]=z_calc_simb_dec(v_c_f_n(n_simb_long+1:n_simb_long+n_simb_rest),...
v_c_q_n(n_simb_long+1:n_simb_long+n_simb_rest),M);
```

Nota: los puntos suspensivos al final de la primera de ambas líneas significa que ésta continúa en la siguiente línea.

Como vemos, volvemos a llamar a la función *'z_calc_simb_dec'*. En este caso decidiremos sobre el tramo de símbolos de *v_c_f_n* y *v_c_q_n* que va desde el símbolo inmediatamente posterior al último extraído para el campo LONGITUD, *n_simb_long+1* (no olvidemos que la variable *n_simb_long* almacenaba el valor del número de símbolos que necesitamos para tomar el campo LONGITUD) , hasta el último símbolo restante, *n_simb_long+n_simb_rest*.

Ahora, la variable de salida del ***Decisor***, *[v_simb_dec]*, almacenará el tren símbolos reubicados, correspondientes al resto de bytes de los campos DATOS y CRC que quedaban por decidir. Esta variable temporal, al tratarse, como sabemos, de un vector complejo, almacena la información de fase en la parte real parte real, y la de cuadratura en la parte imaginaria.

El siguiente bloque a implementar nuevamente es la ***Tabla de Conversión M-aria a Binaria***. Aquí, como sabemos sobradamente, entrarán los símbolos denominados restantes, y obtendremos, tras un ***mapeo*** de símbolos a bloques de *k* bits, y una ulterior

conversión paralelo a serie (al igual que ocurría durante la extracción del campo LONGITUD), la cadena de bits $\{b_m\}$ correspondiente:

- *bits_dec=[aux bits_dec];*

Volvemos a almacenar la cadena de bits en la variable *bits_dec*, y ya de nuevo en el ***Procesador de Campos*** de la ***Fase de Desentramado I***, insertamos los bits iniciales del campo DATOS, extraídos junto con los bits del campo longitud, y que almacenamos, en su momento, en la variable auxiliar *aux*:

- *bits_dec=[aux bits_dec];*

A continuación ***eliminamos*** los ***últimos bits espurios***, que no pertenecen ni a CRC ni a DATOS, como vimos en el ejemplo anterior. Para ello, nos inventamos un bucle donde vamos eliminando el último bit de *bits_dec*, hasta que la longitud de *bits_dec* sea múltiplo de 8, consiguiendo así un número exacto de bytes:

```
while mod(length(bits_dec),8)
   bits_dec(length(bits_dec))=[];
end
```

Ahora sí, la variable *bits_dec* contiene todos los bits correspondientes a los campos DATOS y CRC, que el ***Procesador de Campos*** de la ***Fase de Desentramado I*** se encargará de distribuir a sus respectivos Procesadores homónimos.

- *PROCESADOR DEL CAMPO DATOS*

```
%los ultimos 16 bits corresponden al CRC-16
 for i=1:8:length(bits_dec)-16
    dat_aux=bin2dec(bits_dec(i:i+7));
    datos=[datos ; dat_aux];
 end
```

Las anteriores líneas implementan el subbloque perteneciente a la ***Fase de Desentramado I***, y al cual hemos denominado ***Procesador de Datos***:

Mediante la variable *i*, recorremos, de *8* en *8*,todos los bits del campo DATOS (recordemos que los últimos 16 bits de la variable *bits_dec* constituyen el campo CRC):

- *for i=1:8:length(bits_dec)-16*

De esta manera, vamos formando bytes, y convirtiendo cada uno de esos grupos de *8* bits a formato decimal, con la ayuda de la función de Matlab *'bin2dec'*, que precisamente realiza la ***conversión de formato binario a decimal*** de un número:

- *dat_aux=bin2dec(bits_dec(i:i+7));*

El número con formato decimal para cada byte se almacena en una variable temporal, *dat_aux*, para utilizarla en la siguiente línea, donde incrementaremos la longitud de la variable datos en un número decimal, para cada iteración de este bucle,

hasta almacenar todos los números en base diez, referentes a todos los bytes del campo DATOS de la trama:

- *datos=[datos ; dat_aux];*

Esta variable, *datos*, actúa como 'bufer' donde se acumularán todos los números en formato decimal de los bytes procedentes del campo DATOS de cada una de las tramas, es decir, en cada iteración del bucle de procesamiento de tramas, se irán volcando todos los datos decimales del campo DATOS de cada trama al final del vector *datos*.

Nota: Aunque al vector *datos* le hemos dado forma de vector columna, para asemejar la estructura de los datos obtenidos en el ***Bloque Transmisor***, a la salida de la lectura de los datos del Fichero Fuente, realmente es indiferente que dichos datos constituyan un vector fila o columna: la función *'fwrite'* acepta los datos de igual manera.

- *PROCESADOR DEL CAMPO CRC*

```
%usamos DATOS+CRC-16 para comprobar la existencia de errores
%resto: es el resto de la división. Si es nulo no existen errores.
 resto=z_calc_crc(bits_dec,1);
%si existe error en la trama, muestro un mensaje y pongo su flag a 1.
%es decir, si resto no es una cadena de ceros...
 if strcmp(resto,'0000000000000000')==0
    err=1;
 end
```

En el ***Procesador del campo CRC***, volvemos a llamar a la función *'z_calc_crc'*, para determinar la existencia de posibles errores en la decisión de los símbolos.

- *resto=z_calc_crc(bits_dec,1);*

Esta función no la vamos a analizar en un apartado posterior (como ya hemos comentado que haremos con el resto de funciones que se llaman, dentro del bucle de procesamiento de tramas de la función *'z_recibe_fich'*), puesto que ya está analizada en el Bloque Transmisor. Ya dijimos que la única diferencia es que, en este caso, la secuencia de bits de entrada corresponde no sólo al campo DATOS, sino también al código CRC almacenado en el campo homónimo de la Trama, estando contenidos ambos campos en el vector *bits_dec*. El otro parámetro de entrada, en este caso, es un *1*, para indicar que estamos en el ***Bloque Rececptor*** y no hay que añadir ningún bloque de ceros, dentro de la función, al final del ***Dividendo***. Por último, el *resto* de la división aquí no nos determinará ningún código CRC, como en el Bloque Transmisor, sino que dicho *resto* nos confirmará la ***ausencia o existencia de errores***. Por todo lo demás, el procedimiento usado para la división es el mismo, el ***Divisor*** es el ***mismo polinomio generador*** y, como ya hemos comentado, lo que cambia es el ***Dividendo***, que lo forman los bits incluidos en los campos DATOS+CRC.

En caso de ***existencia de errores***:

- *if strcmp(resto,'0000000000000000')==0*

, esto es, si el *resto* es distinto de cero, se activará un flag a *1*, *err=1*, para mostrarnos posteriormente por pantalla la existencia de errores en dicha trama, incrementándose en uno un ***Contador de Errores***, *cont_err*, que lleva la cuenta de ese número de tramas erróneas.

- *CONTADOR DE TRAMAS Y DE ERRORES.*

```
cont_tramas=cont_tramas+1;
%si existe error indicamos trama erronea
if err==1
   cont_err=cont_err+1;
   aux=sprintf('Trama: %d erronea', cont_tramas);
else
   aux=sprintf('Trama: %d',cont_tramas);
end
disp(aux);
```

Mediante la primera línea, *cont_tramas=cont_tramas+1*, el ***Procesador de Campos*** incrementa el ***Contador de Tramas***, *cont_tramas*, que cuenta el número de tramas procesadas hasta el momento.

Las líneas restantes sirven para mostrar por pantalla el número de trama que se está ejecutando, y que aparece precisamente en la variable *cont_tramas*.

Si el ***indicador de errores*** está activado, *if err==1*, se indicará además por pantalla que dicha trama es errónea.

Descartamos la necesidad de volver a explicar cómo se usan los comandos *'sprintf'* y *'disp'* para mostrar un texto y valores de variables por pantalla.

Por último, la ejecución del bucle termina con la ***actualización de la nueva trama*** a procesar (cuyas líneas de código ya hemos examinado), antes de volver a comenzar otra iteración en el 'megabucle' de procesamiento de tramas.

Al concluir dicho bucle, presentamos una serie de ***resultados por pantalla***, con la ayuda de las funciones de Matlab, *'sprintf'* y *'disp'*, donde mostramos:

- El ***número de tramas transmitidas***, almacenado en la variable *cont_tramas*.
- El ***número de tramas erróneas***, almacenado en la variable *cont_err*.

Las líneas de código a las que hacemos referencia son las siguientes:

```
aux=sprintf('Numero de Tramas transmitidas: %d', cont_tramas);
disp(' ');
disp(aux);
aux=sprintf('Numero de Tramas erroneas: %d', cont_err);
disp(' ');
disp(aux);
```

Para concluir la función *'z_recibe_fich'*, implementamos lo que hemos denominado como ***Destino Digital***, que es el bloque donde creamos el ***Fichero de Destino***.

Para ello ***abrimos el fichero*** de nombre *'fichero'* (que es uno de los parámetros de entrada, como sabemos, en la función *'z_recibe_fich'*), mediante la función conocida de Matlab, *'fopen'*.

La apertura se hará en ***modo escritura***, *'w'*:

- *fid=fopen(fichero,'w');*

Posteriormente llevamos a cabo la ***escritura de bytes*** en formato decimal almacenados en la variable *datos*, en el fichero referido por el identificador *fid*:

- *count=fwrite(fid,datos);*

Por último, ***cerramos el fichero*** en cuestión:

- *fclose(fid);*

y terminamos la función *'z_recibe_fich'*, devolviendo el control del programa, al programa principal, *'z_main'*.

A continuación, procederemos a analizar cada una de las ***funciones generadas*** por nosotros, y cuyas llamadas se producen ***dentro del 'megabucle'*** de procesamiento de tramas de la función *'z_recibe_fich'*.

- **z_igualadorZF:**

```
function vect_eq=z_igualadorZF(resp_imp)

%Ecualizador de cero forzado. Zero-Forcing Equalizer Design.
%funcion que devuelve a la salida el vector de ecualizacion
%vect_eq: vector de ecualizacion. Parametro de salida
%resp_imp: respuesta impulsiva del canal.Parametro de entrada.

%l_h: longitud en muestras de la respuesta impulsiva(N+1)
 l_h=length(resp_imp);
%l_eq:Longitud del igualador(M)
%lo normal el M=2N. Escojo 4N que es mejor.
 l_eq=4*l_h;
%columna de la matriz A
 col_mat=[resp_imp,zeros(1,l_eq-1)];  %Longitud=(N+1)+(M)-1 = N+M
 A=toeplitz(col_mat,[resp_imp(1) zeros(1,l_eq-1)]);
 l_col_A=l_h+l_eq-1;
%N+M impar
 if mod(l_col_A,2)~=0
    b=[zeros(1,floor(l_col_A/2)),1,zeros(1,floor(l_col_A/2))]';
%N+M par
 else
    b=[zeros(1,l_col_A/2-1),1,zeros(1,l_col_A/2)]';
 end
%vect_eq: muestras del ecualizador. Vector columna.
vect_eq=inv(A'*A)*A'*b;        %Ecualizador; long=M
%vect_eq: muestras del ecualizador. Vector fila.
 vect_eq=vect_eq.';
```

Esta función genera, a partir del parámetro de entrada, *resp_imp*, que representa las muestras correspondientes a la ***respuesta impulsiva del canal***, el vector *vect_eq*, que contiene las muestras del ***Ecualizador*** de dicho canal.

Para entender de una manera más cómoda la explicación de esta función, es imprescindible echar una ojeada al ***Bloque Ecualizador***, analizado durante la Fase de Diseño, y principalmente a las ***estructuras matriciales*** montadas para la generación de este igualador.

Con esta función intentamos resolver la ***ecuación matricial*** vista durante el estudio del ***Ecualizador***:

- $\underline{\underline{A}} \cdot \underline{x} = \underline{b}$

Para empezar, hallamos la longitud de la respuesta impulsiva, de la cual ya dijimos que ibamos a tomar 50 muestras. Su valor lo almacenamos en la variable *l_h*:

- *l_h=length(resp_imp);*

A partir de *l_h*, hallamos la longitud del Ecualizador, que ya dijimos durante la Fase de Diseño, cuando explicamos este bloque, que ibamos a tomar cuatro veces el número de muestras de la respuesta impulsiva. Así, tomaremos 200 muestras para nuestro igualador, y almacenaremos ese valor en la variable *l_eq*:

- *l_eq=4*l_h;*

Luego, creamos la estructura de datos de la matriz $\underline{\underline{A}}$. Si nos fijamos, se trata de una ***matriz toeplitz asimétrica***. Existe una función en Matlab, llamada precisamente *'toeplitz'*, para crear este tipo de Matrices homónimas.

Para ello, sólo tenemos que darle como parámetros de entrada, la primera columna y la primera fila, y dicha función genera a la salida la matriz toeplitz asimétrica para dicha primera fila y dicha primera columna.

Como ejemplo, veamos el siguiente 'script':

- Generación de una matriz Toeplitz

```
%creo una fila
 fil=[1 2 3];
%creo una columna
 col=[1 5 6];
%genero la matriz toeplitz correspondiente
 mat_toep=toeplitz(col,fil);
```

Tras la ejecución del 'script', la variable matricial *mat_toep*, tendrá el siguiente aspecto:

1	2	3
5	1	2
6	5	1

Para nuestra matriz A, la primera columna será igual a un vector con la respuesta impulsiva, seguida de un número de ceros igual al número de muestras del ecualizador, *l_eq*, menos una unidad:

- *col_mat=[resp_imp,zeros(1,l_eq-1)];*

Por otra parte, la primera fila de la matriz A, que aparece directamente como segundo parámetro de la función 'toeplitz', consta del primer elemento de la respuesta impulsiva, y una longitud de ceros de longitud *l_eq*-1:

- *[resp_imp(1) zeros(1,l_eq-1)]*

Así, la línea que genera la matriz de nuestro sistema a resolver es:

- *A=toeplitz(col_mat,[resp_imp(1) zeros(1,l_eq-1)]);*

Seguidamente hallamos la longitud de las columnas de A, *l_col_A*, que coinciden con la del vector b.

- *l_col_A=l_h+l_eq-1;*

Luego, en función de que esta longitud de b sea par o impar:

- *if mod(l_col_A,2)~=0*

Se centrará el uno correspondiente al delta, de una manera u otra.

Nota: Al tomar 50 muestras de la respuesta impulsiva y 200 para el ecualizador, la longitud será de 249, siempre impar, pero contemplamos esta posibilidad, por si modificásemos el número de muestras de la respuesta impulsiva o del ecualizador.

A continuación obtenemos el valor del vector x, y lo almacenamos en la variable vectorial *vect_eq*:

- *vect_eq=inv(A'*A)*A'*b;*

Para terminar, convertimos, el vector columna, *vect_eq*, en vector fila, ya que, mayormente, estamos trabajando con vectores filas en la función *'z_recibe_fich'*, y esta variable es el parámetro devuelto a esta función.

- **z_bloq_dem:**

```
function [trama_env_fase, trama_env_quad]=z_bloq_dem(tr,N,alfa);
%bloque de demodulacion tanto para rama de fase como para quadratura.
%parametros de entrada:
%tr: trama de simbolos ya ecualizados que van a ser demodulados.
%N:numero de muestras por simbolo.
%parametros de salida:
%trama_env_fase: muestras de la rama de fase.
%trama_env_quad: muestras de la rama de quadratura.
%alfa: factor de Roll-off.
```

```
%frecuencia de oscilacion
fc=11025;
%variable temporal discreta:
 m=0:length(tr)-1;

%RAMA DE FASE
%oscilador
 tr_fase= tr .* (2*cos(2*pi*(fc/Fs)*m));

%filtro receptor
 trama_env_fase=rcosflt(tr_fase,1,N,'sqrt/Fs',alfa);

%RAMA DE CUADRATURA
%oscilador
 tr_quad= tr.* (-2*sin(2*π*(fc/FS)*m));
%filtro receptor
 trama_env_quad=rcosflt(tr_quad,1,N,'sqrt/Fs',alfa);
```

Ya comentamos, durante el análisis de la función *'z_recib_fich'*, que la función *'z_bloq_dem'* implementará las labores del ***Oscilador*** y del ***Filtro Receptor***, para ambas ramas de fase y cuadratura.

Lo primero que hacemos es generar la frecuencias de oscilación, *fc*, además de la variable *m*, que contiene los instantes discretos de muestreo, *m·Ts*:

- *fc=11025;*
- *m=0:length(tr)-1;*

La variable de entrada *tr*, que representa la trama ecualizada a la entrada del ***Demodulador***, es multiplicada por el ***Oscilador***, para desplazarla f_c hz. hasta la banda base. Aquí vuelve a desdoblarse la señal en ***dos tramos: fase y cuadratura***.

- *tr_fase= tr .* (2*cos(2*pi*(fc/Fs)*m));*
- *tr_quad= tr.* (-2*sin(2*pi*(fc/Fs)*m));*

Las señales resultantes, *tr_fase* y *tr_quad*, son filtradas con los ***Pulsos raíz de coseno alzado***, para ***eliminar las componentes de alta frecuencia*** y darles a dichos espectros de entrada la forma definitiva de pulsos en coseno alzado. Para ello volvemos a hacer uso de la función de Matlab, 'rcosflt', con la única diferencia que en este caso ***no existirá sobremuestreo***, lo cual se refleja en el cuarto parámetro, donde la cadena 'Fs', indica este hecho:

- *trama_env_fase=rcosflt(tr_fase,1,N,'sqrt/Fs',alfa);*
- *trama_env_quad=rcosflt(tr_quad,1,N,'sqrt/Fs',alfa);*

Los vectores devueltos a la función *'z_recibe_fich'*, *trama_env_fase* y *trama_env_quad*, constituye la salida de los ***Filtros Receptores*** en ambas ramas, y forman las envolventes en fase, $r_I(t)$, y cuadratura, $r_Q(t)$, de la señal $r(t)_{APK}$ a la entrada del ***Demodulador***, tal y como los nombramos durante la Fase de Diseño.

- **z_ajuste_niveles:**

```
function [v_c_f_n,v_c_q_n]=z_ajuste_niveles(v_c_f,v_c_q);

%funcion que toma los simbolos ficticios de comienzo de trama.
%para realizar ajuste de niveles.
%para evitar transitorios, nos quedamos con los bits 5º,6º,7º y 8º.

%cf: coeficiente de fase.
 cf=(1/0.4)*mean(v_c_f(5:8));
%cf: coeficiente de quadratura.
 cq=(1/0.4)*mean(v_c_q(5:8));
%nuevos vectores de fase y quadratura 'desnormalizados'.
 v_c_f_n=v_c_f/cf;
 v_c_q_n=v_c_q/cq;
```

Ya hemos comentado en varias ocasiones, que hay que volver a ***normalizar*** la secuencia de símbolos a la salida del Detector APK, {I''$_n$} y {Q''$_n$}, para volver a tener los valores de dichos símbolos en el ***rango de transmisión [-1,1]***. En esta función, como su nombre indica, realizaremos este ***Ajuste de Niveles***.

Para ello, incluimos un ***Inicio de Trama*** , formado, como sabemos, por ***8 símbolos*** recibidos , para controlar su valor (igual a *0.4*, por los motivos ya explicados en la Fase de Diseño), para ver la relación con sus valores en Recepción.

Pero para evitar transitorios, sólo vamos a usar los 4 símbolos finales de los 8 que componen el Inicio de Trama.

Lo primero que hacemos es tomar de los 8 símbolos recibidos del campo Inicio de Trama (que son los 8 primeros dentro de la variable de entrada, *v_c_f*), los almacenados en las 4 posiciones finales de este campo, es decir, *v_c_f(5:8)*, entre sus correspondientes símbolos transmitidos, que tendrán como sabemos, un valor igual a *0.4*, y hallamos su media, almacenando la relación en la variable *cf*. Actuamos de igual manera con los símbolos en cuadratura ficticios recibidos, *v_c_q(5:8)*, guardando su relación con los símbolos transmitidos en la variable *cq*.

- *cf=(1/0.4)*mean(v_c_f(5:8));*
- *cq=(1/0.4)*mean(v_c_q(5:8));*

Si suponemos que la relación de proporcionalidad entre los símbolos transmitidos y sus correspondientes símbolos recibidos, es la misma para el Inicio de Trama que para el resto de símbolos de la trama, tan sólo tenemos que dividir los valores de los símbolos recibidos para toda la secuencia de símbolos recibidos, *v_c_f* y *v_c_q*, entre dichos coeficientes, *cf* y *cq*, respectivamente, obteniendo así los vectores de ***símbolos ya normalizados***, *v_c_f_n* y *v_c_q_n*, que devolvemos a la función *'z_recibe_fich'*:

- *v_c_f_n=v_c_f/cf;*
- *v_c_q_n=v_c_q/cq;*

- **z_calc_simb_dec:**

```
function [v_dec]=z_calc_simb_dec(vcf_rx,vcq_rx,M);

%funcion que devuelve el vector de decision de los simbolos transmitidos
%vcf_rx: vector de fase de los simbolos que recibo.
%vcq_rx: vector de quadratura de los simbolos que recibo.
```

```
%calculo la posicion de todos los posibles simbolos txdos.
 [vcf_tx,vcq_tx]=z_all_simb(M);
%realmente solo necesitare la componente en fase y en cuadratura.
%no necesito para nada la amplitud y la fase.

%el vector de simbolos deseados o transmitidos son:
 v_simb_tx=vcf_tx+j*vcq_tx;
%mi vector de simbolos recibidos sera:
 v_simb_rx=vcf_rx+j*vcq_rx;
%vector de decision (vector de simbolos que he decidido)
 v_dec=[];

%para cada simbolo recibido hallo la minima distancia respecto de todos...
%...los posibles simbolos transmitidos o deseados.
 for i=1:length(v_simb_rx);
%v_d:vector de distancias de mi simbolo a todos los posibles simbolos
    v_d=abs(v_simb_tx-v_simb_rx(i));
%d:distancia minima
%i_d: indice del vector v_d donde la distancia a mi simb rx es minima
    [d_min,i_d]=min(v_d);
    v_dec(i)=v_simb_tx(i_d);
 end
```

Esta función implementa, como ya dijimos en el análisis de la función 'z_recibe_fich', el ***Bloque Decisor*** que aparece en el ***Detector APK***, a la salida del Ajustador de Niveles.

La tarea que tiene encomendada la función '*z_calc_simb_dec'*, por lo tanto, es realizar, para cada uno de los símbolos de entrada, la decisión sobre qué símbolo se ha sido transmitido.

El criterio que vamos a tomar es el de ***Máxima Verosimilitud***: vamos a medir la distancia de cada punto de mensaje recibido, hasta cada uno de los *M* símbolos ideales de la constelación en cuestión, y vamos a decidir que el símbolo transmitido es el que más cerca se encuentre del símbolo recibido.

Por tanto, lo primero que hacemos es crear un vector que contenga cada uno de las *M* posiciones ideales de la constelación, mediante su distancia en fase y su distancia en cuadratura.

- *[vcf_tx,vcq_tx]=z_all_simb(M);*

La función *'z_all_simb'* calcula precisamente todos los símbolos de la constelación de *M* elementos, devolviendo la distancia en fase, *vcf_tx* y la distancia en cuadratura, *vcq_tx*, de cada uno de los símbolos ideales del sistema.

A partir de las coordenadas en fase y en cuadratura de los símbolos posiblemente transmitidos o ideales, generamos un vector complejo, *v_simb_tx*, cuya parte real representa las componentes en fase de dichos símbolos, y cuya parte imaginaria pertenece a las componentes en cuadratura de dichos símbolos:

- *v_simb_tx=vcf_tx+j*vcq_tx;*

Con las componentes en fase y en cuadratura de los símbolos recibidos de entrada, hacemos exactamente lo mismo, dando lugar otro vector complejo, *v_simb_rx*:

- *v_simb_rx=vcf_rx+j*vcq_rx;*

Tras inicializar un vector llamado *v_dec*, ejecutamos un bucle donde recorremos, a través de la variable *i*, cada símbolo recibido, para decidir, a partir de su posición en la constelación, qué símbolo hemos transmitido, y reubicar su posición a la posición ideal correspondiente. El bucle concluirá una vez hayamos decidido, sobre todos los símbolos recibidos, su posición ideal en la constelación:

- *for i=1:length(v_simb_rx);*

Así, para cada símbolo recibido, *v_simb_rx(i)*, formamos un vector, *v_d*, que contenga la distancia desde dicho símbolo a cada uno de los símbolos posiblemente transmitidos de la constelación, *v_simb_tx-v_simb_rx(i)*. Dicho vector de distancias, *v_d*, será un vector de números complejos, por lo que las distancias coincidirán con su valor absoluto. De ahí el uso de la función *'abs'* de Matlab, que genera el valor absoluto de la variable que tiene a la entrada:

- *v_d=abs(v_simb_tx-v_simb_rx(i));*

Mediante la función *'min'* de Matlab, obtenemos dos variables a la salida:

- *d_min* : el mínimo valor del vector de entrada.
- *i_d:* el índice para ese mínimo.

Entonces mediante la sentencia:

- *[d_min,i_d]=min(v_d);*

obtenemos la posición *i_d*, del vector *v_d*, donde el valor es mínimo, *d_min*. Pero dicha posición *i_d* coincide con la posición, dentro de *v_simb_tx*, del posible símbolo transmitido, para el cual la distancia es mínima, *v_simb_tx(i_d)*. De esta manera, para el símbolo recibido en esa iteración, *v_simb_rx(i)*, decidimos que se ha transmitido el símbolo ideal *v_simb_tx(i_d)*, pues, atendiendo al ***Criterio de Máxima Verosimilitud***, el símbolo recibido *v_simb_rx(i)* está en la región de decisión de *v_simb_tx(i_d)*, puesto que la distancia a éste es mínima, en comparación con el resto de distancias referentes al resto de posibles puntos del vector de transmisión.

- *v_dec(i)=v_simb_tx(i_d);*

Antes analizar '*z_form_bits'*, que es la última por analizar, de las funciones llamadas dentro de la función *'z_recibe_fich'*, nos gustaría mirar primero la llamada a la función *'z_all_simb'*, para acabar completamente con la función *'z_calc_simb_dec'*.

- **z_all_simb:**

```
function [v_c_f,v_c_q]=z_all_simb(M)

%funcion que halla las componentes en fase y cuadratura...
%...de los posibles simbolos de la constelación.
% v_c_f: vector de componentes de fase de todos los simbolos.
% v_c_q: vector de componentes de quadratura de todos los simbolos.
```

```
%a: vector de numeros binarios con todos los posibles simbolos
%L_simb: longitud del simbolo m_ario
 L_simb=log2(M);
 a=[];
 for i=0:M-1
    aux=z_cod_gray(i,L_simb);
    a=[a aux];
 end
%cad_bits: cadena de caracteres binarios con todos los posibles simbolos.
 cad_bits=z_mat2cad(a);

% hallamos las componentes en fase y cuadratura de los distintos simbolos.
 [v_c_f,v_c_q]=z_form_simb(cad_bits,M);

%dibujo la constelacion correspondiente
%z_plot_constelac(v_c_f,v_c_q,M);
```

Esta función, como ya adelantamos, halla la distancia en fase y la distancia en cuadratura de todos los símbolos ideales de la constelación de *M* elementos.

Mediante la variable *M*, hallamos la longitud, *L_simb*, de cada bloque de k bits que formen un símbolos M-ario:

- *L_simb=log2(M);*

Luego tendrá lugar un bucle donde se recorrerán, consecutivamente, todos los posibles valores, en formato decimal, de la constelación de *M* elementos. Estos valores estarán guardados en la variable *i*:

- *for i=0:M-1*

Cada valor en decimal, será codificado a binario, usando la ***Codificación Gray***, y almacenando dicho código, de *L_simb* bits, en la variable auxiliar *aux*:

- *aux=z_cod_gray(i,L_simb);*

La función *'z_cod_gray'* es precisamente quien realiza la conversión de decimal a código gray, y la analizaremos posteriormente.

La variable aux, en cada iteración, se va acumulando en la variable *a*, previamente inicializada antes de comenzar el bucle, *a=[]* :

- *a=[a aux];*

Así, al acabar el bucle, tendremos en la variable *a* , un vector binario con todos los bits asociados a los *M* elementos de la constelación, ordenados consecutiva y crecientemente, atendiendo a la codificación gray.

Pero nosotros no estamos trabajando con vector de números binarios, sino con cadena de bits. Por eso convertimos el vector de números binarios, *a*, en una cadena de caracteres binarios, *cad_bits*. Para realizar esta conversión, nos valemos de la función *'z_mat2cad'*, usada en la función *'z_calc_crc'*, y que ya explicamos en su momento.

- *cad_bits=z_mat2cad(a);*

Ahora podemos usar la función *'z_form_simb'*, también llamada desde la función *'z_bloq_mod'* del Bloque Transmisor, para implementar los bloques en cascada ***Agrupador de k bits & Tabla de Conversión de k bits a símbolo M-ario***, cuya misión es aceptar una cadena binaria a la entrada, agrupar la secuencia en bloques de k bits, siendo k=log2(M), y generar a la salida las componentes en fase, *v_c_f*, y en cuadratura, *v_c_q*, de los símbolos correspondientes a esta secuencia de bloques de k bits:

- *[v_c_f,v_c_q]=z_form_simb(cad_bits,M);*

Así, a la salida de esta función, *'z_all_simb'*, tendremos las distancias en fase y en cuadratura de todos los símbolos de la constelación, como queríamos.

Comentar, que para la fase de prueba, esta función *'z_all_simb'* llamaba a otra función, *'z_plot_const'*, que, mediante comandos como *'plot'*, representaba todos los símbolos de la constelación en pantalla. Estas líneas están deshabilitadas como comentarios, sin embargo, nos fueron de gran utilidad durante dicha fase de pruebas, para comprobar la ubicación de los puntos en la constelación:

- *%dibujo la constelacion correspondiente*
- *%z_plot_constelac(v_c_f,v_c_q,M);*

Nota: estamos usando la ***Codificación Gray*** porque, como comentamos en la Fase de Diseño, cuando vimos las distintas constelaciones de nuestro sistema, los símbolos están ordenados de esa manera, para minimizar el número de bits erróneos, ante un error en la decisión de un símbolo, puesto que los símbolos consecutivos en la codificación Gray solamente difieren en un bit, y si existiese un error en la decisión de un símbolo, probablemente, dicho símbolo haya caído en la región de decisión vecina al símbolo correcto.

- **z_cod_gray:**

```
function y = graycode(i,L)

 x_cad=dec2bin(i,L);
 x=z_cad2mat(x_cad);
 y(1)=x(1);
 for j=2:L;
   if x(j-1)==1
     y(j)=1-x(j);
   else
     y(j)=x(j);
   end
 end
```

Esta función acepta como entrada el valor de un número decimal, *i*, y forma un vector binario con el ***código Gray***, *y*, del número *i*, con una longitud *L*.

Lo primero que hacemos es generar el ***código binario natural***, perteneciente al número ***i*** introducido, mediante la ya conocida función de Matlab *'dec2bin'*. El resultado será una cadena de caracteres binaria, *x_cad*, con una longitud *L*, dada también como parámetro de entrada.

Seguidamente convertimos esa cadena binaria, *x_cad*, a un vector de números binarios, con ayuda de la función *'z_cad2mat'*, también usada y analizada anteriormente, durante el cálculo del CRC, en el ***Bloque Transmisor***.

El ***bucle*** que sigue, implementa un procedimiento para ***convertir*** un número en ***código binario natural***, en el mismo número, pero ***en código Gray***. Explicaremos dicho proceso paso a paso:

- El primer bit, el más significativo, siempre coincide en ambas codificaciones. De ahí la línea de programa previa al bucle:

  - *y(1)=x(1);*

  donde *y* representa el código gray, y *x* almacena el código binario del número *i*.

- Ya en el bucle, recorremos, mediante el índice *j*, el resto de bits del vector *x*:

  - *for j=2:L;*

- En cada iteración, comprobamos el valor del elemento anterior, *x(j-1)*.
- Si dicho elemento anterior es *1*, almacenamos en la posición actual del número codificado en Gray, *y(j)*, el opuesto del número que haya en la posición actual del mismo número codificado en Binario, *x(j)*:

  - *if x(j-1)==1*
  - *y(j)=1-x(j);*

- En caso contrario, si en la posición del elemento anterior existe un *0*, almacenamos en la posición actual del número codificado en Gray, *y(j)*, el valor existente en la misma posición del número codificado en Binario,*x(j)*.

  - *else*
  - *y(j)=x(j);*

Al terminar este bucle, en el vector numérico *y* tendremos el valor en ***código Gray*** del número *i*.

Ahora sí, analizamos la función que nos queda de las que son llamadas dentro de la función ***'z_recibe_fich'***:

- **z_form_bits:**

```
function [cad_bits]=z_form_bits(v_simb,M)

%funcion que recibe un vector de simbolos de entrada y...
%...devuelve la cadena binaria correspondiente.
 cad_bits=[];
 for i=1:length(v_simb)
%creamos ahora la tabla de constelacion inversa.
%Con ella, pasamos de simbolos a bits.
%aux: cadena de bits correspondiente a cada simbolo.
   [aux]=z_tab_const_inv(v_simb(i),M);
```

```
%cad_bits:cadena binaria de salida
   cad_bits=[cad_bits aux];
 end
```

Esta función, como ya explicamos, acepta un vector de símbolos a la entrada, ya reubicados en su posición presuntamente original (si la detección es correcta) en la que estaban al ser transmitidos, y devuelve, a la salida, cadena binaria correspondiente a todos los símbolos de entrada. Es decir, esta función implementa el ***mapeo de símbolos a bits,*** y su ***Convertidor Paralelo a Serie***, presente en el ***Detector APK***, tal y como vimos durante la fase de Diseño.

Para ello, recorre, mediante el índice *i*, el vector de símbolos, *v_simb*, en un bucle *'for'*:

- *for i=1:length(v_simb)*

En cada iteración, realizamos el ***mapeo de un símbolo***, *v_simb(i)*, por medio de la llamada a la función *'z_tab_const_inv'*, para su correspondiente constelación de *M* puntos, y almacenamos el bloque de caracteres binarios en la variable auxiliar *aux*:

- *[aux]=z_tab_const_inv(v_simb(i),M);*

Acto seguido, ***acumulamos en serie*** ese bloque de caracteres reservados en *aux*, en la cadena de bits, cad_bits:

- *cad_bits=[cad_bits aux];*

Al final del bucle, la variable de salida *cad_bits*, contendrá la cadena binaria correspondiente a los símbolos ideales de entrada.

- **z_tab_const_inv:**

```
function [bloq_bin]=z_tab_const_inv(simb,M);

%funcion que acepta un simbolo y...
%...devuelve bloque binario correspondiente.

%M=4....2 bits por simbolo
 if M==4
   [bloq_bin] = z_tab_inv_M_4( simb );

%M=8....3 bits por simbolo
 elseif M==8
   [bloq_bin] = z_tab_inv_M_8( simb );

%M=16....4 bits por simbolo
 elseif M==16
    [bloq_bin] = z_tab_inv_M_16( simb );

%M=32....5 bits por simbolo
 elseif M==32
   [bloq_bin] = z_tab_inv_M_32( simb );
```

```
%M=64....6 bits por simbolo
 elseif M==64
   [bloq_bin] = z_tab_inv_M_64( simb );

%M=128....7 bits por simbolo
 elseif M==128
   [bloq_bin] = z_tab_inv_M_128( simb );

%M=256....8 bits por simbolo
 elseif M==256
   [bloq_bin] = z_tab_inv_M_256( simb );

end
```

Esta función, llamada desde la anterior función, '*z_form_bits*', implementa la forma de acceder, dependiendo del valor de *M*, a cada una de las ***Tablas de conversión M-aria a Binaria***, y nos devuelve el bloque de caracteres binarios, para cada símbolo de entrada.

Como podemos comprobar, el procedimiento empleado para acceder a una de las ***7 posibles Tablas de Conversión M-aria a Binaria***, es idéntico al empleado en la función gemela del bloque Transmisor, *'z_tab_constelac'*.

Y, para concluir con el análisis de todas la funciones implementadas en ambos bloques, Transmisor y Receptor, vamos a representar las distintas Tablas de Conversión M-aria a Binaria, para cada valor de *M*.

- **z_tab_inv_M_4:**

```
function [cad_bits] = z_tab_inv_M_4( simb )

%Hallo el valor de L.
%L:minima distancia cartesiana a un simbolo...
%...tanto en fase como en quadratura.
%Asi,como dijimos, la mayor distancia
%de simbolo este entre -1 y 1.
 L=1/sqrt(2);

%x:componente en fase del simbolo.
 x=real(simb);
%y:componente en quadratura del simbolo.
 y=imag(simb);

%aplico la tabla y obtengo los bits.
 if (x==-L & y==-L)
   cad_bits='00';
 elseif (x==-L & y==L)
   cad_bits='01';
 elseif (x==L & y==L)
   cad_bits='11';
 elseif (x==L & y==-L)
   cad_bits='10';
 end
```

- **z_tab_inv_M_8:**

```
function [cad_bits] = z_tab_inv_M_8( simb )

%Hallo el valor de L.
%L:minima distancia cartesiana a un simbolo...
%...tanto en fase como en quadratura.
%Asi,como dijimos, la mayor distancia
%de simbolo este entre -1 y 1.
 L=1/3;

%x:componente en fase del simbolo.
 x=real(simb);
%y:componente en quadratura del simbolo.
 y=imag(simb);

%aplico la tabla y obtengo los bits.
 if ( x==L & y==L )
   cad_bits='000' ; %0
 elseif ( x==3*L & y==0 )
   cad_bits='001' ;  %1
 elseif ( x==0 & y==3*L )
   cad_bits='011' ; %2
 elseif ( x==-L &  y==L )
   cad_bits='010' ;  %3
 elseif ( x==-L &  y==-L )
   cad_bits='110' ;  %4
 elseif ( x==-3*L &  y==0 )
   cad_bits='111' ;  %5
 elseif ( x==0 & y==-3*L )
   cad_bits='101' ;  %6
 elseif ( x==L &  y==-L )
   cad_bits='100' ;  %7
 end
```

- **z_tab_inv_M_16:**

```
function [cad_bits] = z_tab_inv_M_16( simb )

%Hallo el valor de L.
%L:minima distancia cartesiana a un simbolo...
%...tanto en fase como en quadratura.
%Asi,como dijimos, la mayor distancia
%de simbolo este entre -1 y 1.
 L=1/(3*sqrt(2));

%x:componente en fase del simbolo.
 x=real(simb);
%y:componente en quadratura del simbolo.
 y=imag(simb);

%aplico la tabla y obtengo los bits.

 if ( x==-3*L & y==-3*L )
   cad_bits='0000' ; %0
 elseif ( x==-3*L & y==-L )
   cad_bits='0001' ;  %1
 elseif ( x==-3*L &  y==L )
```

```
    cad_bits='0011' ; %2
  elseif ( x==-3*L & y==3*L )
    cad_bits='0010' ;  %3
  elseif ( x==-L & y==3*L )
    cad_bits='0110' ;  %4
  elseif ( x==-L & y==L )
    cad_bits='0111' ;  %5
  elseif ( x==-L & y==-L )
    cad_bits='0101' ;  %6
  elseif ( x==-L & y==-3*L )
    cad_bits='0100' ;  %7
  elseif ( x==L & y==-3*L )
    cad_bits='1100' ;  %8
  elseif ( x==L & y==-L )
    cad_bits='1101' ;  %9
  elseif ( x==L & y==L )
    cad_bits='1111' ;  %10
  elseif ( x==L & y==3*L )
    cad_bits='1110' ;  %11
  elseif ( x==3*L & y==3*L )
    cad_bits='1010' ;  %12
  elseif ( x==3*L & y==L )
    cad_bits='1011' ;  %13
  elseif ( x==3*L & y==-L )
    cad_bits='1001' ;  %14
  elseif ( x==3*L & y==-3*L )
    cad_bits='1000' ;  %15
  end
```

- **z_tab_inv_M_32:**

```
function [cad_bits] = z_tab_inv_M_32( simb )

%Hallo el valor de L.
%L:minima distancia cartesiana a un simbolo...
%...tanto en fase como en quadratura.
%Asi,como dijimos, la mayor distancia
%de simbolo este entre -1 y 1.
 L=1/(5*sqrt(2));

%x:componente en fase del simbolo.
 x=real(simb);
%y:componente en quadratura del simbolo.
 y=imag(simb);

%aplico la tabla y obtengo los bits.

 if ( x==L & y==L )
    cad_bits='00000' ; %0
  elseif ( x==3*L & y==L )
    cad_bits='00001' ; %1
  elseif ( x==5*L & y==L )
    cad_bits='00011' ;  %2
  elseif ( x==3*L & y==5*L )
    cad_bits='00010' ;  %3
  elseif( x==L & y==5*L )
    cad_bits='00110' ;  %4
  elseif( x==5*L & y==3*L )
```

```
    cad_bits='00111' ;  %5
  elseif( x==3*L & y==3*L )
    cad_bits='00101' ;  %6
  elseif ( x==L & y==3*L )
    cad_bits='00100' ;  %7
  elseif( x==-L & y==3*L )
    cad_bits='01100' ;   %8
  elseif( x==-3*L & y==3*L )
    cad_bits='01101' ; %9
  elseif( x==-5*L & y==3*L )
    cad_bits='01111' ; %10
  elseif ( x==-L &  y==5*L )
    cad_bits='01110' ;   %11
  elseif( x==-3*L & y==5*L )
    cad_bits='01010' ;    %12
  elseif( x==-5*L &  y==L )
    cad_bits='01011' ;  %13
  elseif ( x==-3*L & y==L )
    cad_bits='01001' ;  %14
  elseif ( x==-L & y==L )
    cad_bits='01000' ;  %15
  elseif ( x==-L & y==-L )
    cad_bits='11000' ; %16
  elseif ( x==-3*L & y==-L )
    cad_bits='11001' ; %17
  elseif ( x==-5*L & y==-L )
    cad_bits='11011' ;   %18
  elseif ( x==-3*L & y==-5*L )
    cad_bits='11010' ;   %19
  elseif ( x==-L & y==-5*L )
    cad_bits='11110' ;  %20
  elseif ( x==-5*L & y==-3*L )
    cad_bits='11111' ;  %21
  elseif  ( x==-3*L & y==-3*L )
    cad_bits='11101' ;  %22
  elseif ( x==-L & y==-3*L )
    cad_bits='11100' ;  %23
  elseif ( x==L & y==-3*L )
    cad_bits='10100' ;   %24
  elseif ( x==3*L & y==-3*L )
    cad_bits='10101' ; %25
  elseif ( x==5*L & y==-3*L )
    cad_bits='10111' ; %26
  elseif ( x==L & y==-5*L )
    cad_bits='10110' ;   %27
  elseif ( x==3*L & y==-5*L )
    cad_bits='10010' ;    %28
  elseif( x==5*L & y==-L )
    cad_bits='10011' ;  %29
  elseif ( x==3*L & y==-L )
    cad_bits='10001' ;  %30
  elseif ( x==L & y==-L )
    cad_bits='10000' ;  %31
  end
```

- **z_tab_inv_M_64:**

```
function [cad_bits] = z_tab_inv_M_64( simb )
```

```
%Hallo el valor de L.
%L:minima distancia cartesiana a un simbolo...
%...tanto en fase como en quadratura.
%Asi,como dijimos, la mayor distancia
%de simbolo este entre -1 y 1.
 L=1/(7*sqrt(2));

%x:componente en fase del simbolo.
 x=real(simb);
%y:componente en quadratura del simbolo.
 y=imag(simb);

%aplico la tabla y obtengo los bits.

 if ( x==-7*L & y==-7*L )
   cad_bits='000000' ; %0
 elseif ( x==-7*L & y==-5*L )
   cad_bits='000001' ; %1
 elseif  ( x==-7*L & y==-3*L )
   cad_bits='000011' ;   %2
 elseif ( x==-7*L & y==-L )
   cad_bits='000010' ;   %3
 elseif ( x==-7*L & y==L )
   cad_bits='000110' ;  %4
 elseif ( x==-7*L & y==3*L )
   cad_bits='000111' ;  %5
 elseif ( x==-7*L & y==5*L )
   cad_bits='000101' ;  %6
 elseif ( x==-7*L & y==7*L )
   cad_bits='000100' ;  %7
 elseif ( x==-5*L & y==7*L )
   cad_bits='001100' ;   %8
 elseif( x==-5*L & y==5*L )
   cad_bits='001101' ; %9
 elseif( x==-5*L & y==3*L )
   cad_bits='001111' ; %10
 elseif ( x==-5*L & y==L )
    cad_bits='001110' ;   %11
 elseif ( x==-5*L & y==-L )
   cad_bits='001010' ;    %12
 elseif( x==-5*L & y==-3*L )
   cad_bits='001011' ;  %13
 elseif ( x==-5*L & y==-5*L )
   cad_bits='001001' ;  %14
 elseif( x==-5*L & y==-7*L )
   cad_bits='001000' ;  %15
 elseif( x==-3*L & y==-7*L )
   cad_bits='011000' ; %16
 elseif  ( x==-3*L & y==-5*L )
   cad_bits='011001' ; %17
 elseif ( x==-3*L & y==-3*L )
   cad_bits='011011' ;   %18
 elseif ( x==-3*L & y==-L )
   cad_bits='011010' ;   %19
 elseif ( x==-3*L & y==L )
   cad_bits='011110' ;  %20
 elseif ( x==-3*L & y==3*L )
   cad_bits='011111' ;  %21
 elseif  ( x==-3*L & y==5*L )
```

```
    cad_bits='011101' ;  %22
elseif ( x==-3*L & y==7*L )
    cad_bits='011100' ;  %23
elseif ( x==-L & y==7*L )
    cad_bits='010100' ;   %24
elseif  ( x==-L & y==5*L )
    cad_bits='010101' ; %25
elseif ( x==-L & y==3*L )
    cad_bits='010111' ; %26
elseif  ( x==-L & y==L )
    cad_bits='010110' ;   %27
elseif ( x==-L & y==-L )
    cad_bits='010010' ;    %28
elseif ( x==-L & y==-3*L )
    cad_bits='010011' ;  %29
elseif  ( x==-L & y==-5*L )
    cad_bits='010001' ;  %30
elseif( x==-L & y==-7*L )
    cad_bits='010000' ;  %31
elseif( x==L & y==-7*L )
    cad_bits='110000' ; %32
elseif( x==L & y==-5*L )
    cad_bits='110001' ; %33
elseif( x==L & y==-3*L )
    cad_bits='110011' ;   %34
elseif ( x==L & y==-L )
    cad_bits='110010' ;   %35
elseif ( x==L & y==L )
    cad_bits='110110' ;  %36
elseif ( x==L & y==3*L )
    cad_bits='110111' ;  %37
elseif ( x==L & y==5*L )
    cad_bits='110101' ;  %38
elseif ( x==L & y==7*L )
    cad_bits='110100' ;  %39
elseif( x==3*L & y==7*L )
    cad_bits='111100' ;   %40
elseif ( x==3*L & y==5*L )
    cad_bits='111101' ; %41
elseif( x==3*L & y==3*L )
    cad_bits='111111' ; %42
elseif ( x==3*L & y==L )
    cad_bits='111110' ;   %43
elseif ( x==3*L & y==-L )
    cad_bits='111010' ;    %44
elseif( x==3*L & y==-3*L )
    cad_bits='111011' ;  %45
elseif ( x==3*L & y==-5*L )
    cad_bits='111001' ;  %46
elseif ( x==3*L & y==-7*L )
    cad_bits='111000' ;  %47
elseif( x==5*L & y==-7*L )
    cad_bits='101000' ; %48
elseif ( x==5*L & y==-5*L )
    cad_bits='101001' ; %49
elseif ( x==5*L & y==-3*L )
    cad_bits='101011' ;   %50
elseif ( x==5*L & y==-L )
    cad_bits='101010' ;   %51
elseif( x==5*L & y==L )
```

```
    cad_bits='101110' ;  %52
  elseif ( x==5*L & y==3*L )
    cad_bits='101111' ;  %53
  elseif ( x==5*L & y==5*L )
    cad_bits='101101' ;  %54
  elseif ( x==5*L & y==7*L )
    cad_bits='101100' ;  %55
  elseif ( x==7*L &  y==7*L )
    cad_bits='100100' ;   %56
  elseif ( x==7*L & y==5*L )
    cad_bits='100101' ; %57
  elseif( x==7*L & y==3*L )
    cad_bits='100111' ; %58
  elseif ( x==7*L & y==L )
    cad_bits='100110' ;   %59
  elseif( x==7*L & y==-L )
    cad_bits='100010' ;    %60
  elseif( x==7*L & y==-3*L )
    cad_bits='100011' ;  %61
  elseif( x==7*L &  y==-5*L )
    cad_bits='100001' ;  %62
  elseif( x==7*L & y==-7*L )
    cad_bits='100000' ;  %63
  end
```

- **z_tab_inv_M_128:**

```
function [cad_bits] = z_tab_inv_M_128( simb )

%Hallo el valor de L.
%L:minima distancia cartesiana a un simbolo...
%...tanto en fase como en quadratura.
%Asi,como dijimos, la mayor distancia
%de simbolo este entre -1 y 1.
 L=1/(11*sqrt(2));

%x:componente en fase del simbolo.
 x=real(simb);
%y:componente en quadratura del simbolo.
 y=imag(simb);

%aplico la tabla y obtengo los bits.

 if (x==L & y==L )
    cad_bits='0000000' ; %0
  elseif (x==3*L & y==L )
    cad_bits='0000001' ; %1
  elseif (x==3*L & y==3*L )
    cad_bits='0000011' ;   %2
  elseif (x==L & y==3*L )
    cad_bits='0000010' ;   %3
  elseif (x==7*L & y==3*L )
    cad_bits='0000110' ;  %4
  elseif (x==5*L & y==3*L )
    cad_bits='0000111' ;  %5
  elseif (x==5*L & y==L )
    cad_bits='0000101' ;  %6
  elseif (x==7*L & y==L )
```

```
    cad_bits='0000100' ;  %7
elseif (x==9*L & y==L )
    cad_bits='0001100' ;   %8
elseif (x==11*L & y==L )
    cad_bits='0001101' ; %9
elseif(x==11*L & y==3*L )
    cad_bits='0001111' ; %10
elseif (x==9*L & y==3*L )
    cad_bits='0001110' ;   %11
elseif(x==7*L & y==11*L )
    cad_bits='0001010' ;    %12
elseif (x==5*L &  y==11*L )
    cad_bits='0001011' ;  %13
elseif (x==5*L & y==9*L )
    cad_bits='0001001' ;  %14
elseif(x==7*L & y==9*L )
    cad_bits='0001000' ;  %15
elseif(x==L & y==9*L )
    cad_bits='0011000' ; %16
elseif (x==3*L & y==9*L )
    cad_bits='0011001' ; %17
elseif (x==3*L &  y==11*L )
    cad_bits='0011011' ;   %18
elseif (x==L & y==11*L )
    cad_bits='0011010' ;   %19
elseif (x==9*L & y==5*L )
    cad_bits='0011110' ;  %20
elseif (x==11*L & y==5*L )
    cad_bits='0011111' ;  %21
elseif (x==11*L & y==7*L )
    cad_bits='0011101' ;  %22
elseif (x==9*L & y==7*L )
    cad_bits='0011100' ;  %23
elseif (x==7*L & y==7*L )
    cad_bits='0010100' ;   %24
elseif (x==5*L & y==7*L )
    cad_bits='0010101' ; %25
elseif (x==5*L & y==5*L )
    cad_bits='0010111' ; %26
elseif  (x==7*L & y==5*L )
    cad_bits='0010110' ;   %27
elseif (x==L & y==5*L )
    cad_bits='0010010' ;    %28
elseif (x==3*L & y==5*L )
    cad_bits='0010011' ;  %29
elseif(x==3*L & y==7*L )
    cad_bits='0010001' ;  %30
elseif (x==L & y==7*L )
    cad_bits='0010000' ;  %31
elseif (x==-L & y==7*L )
    cad_bits='0110000' ; %32
elseif (x==-3*L & y==7*L )
    cad_bits='0110001' ; %33
elseif (x==-3*L & y==5*L )
    cad_bits='0110011' ;   %34
elseif (x==-L & y==5*L )
    cad_bits='0110010' ;   %35
elseif (x==-7*L &  y==5*L )
    cad_bits='0110110' ;  %36
elseif (x==-5*L & y==5*L )
```

```
    cad_bits='0110111' ;  %37
elseif (x==-5*L & y==7*L )
    cad_bits='0110101' ;  %38
elseif (x==-7*L & y==7*L )
    cad_bits='0110100' ;  %39
elseif (x==-9*L & y==7*L )
    cad_bits='0111100' ;   %40
elseif (x==-11*L & y==7*L )
    cad_bits='0111101' ; %41
elseif (x==-11*L & y==5*L )
    cad_bits='0111111' ; %42
elseif  (x==-9*L & y==5*L )
    cad_bits='0111110' ;   %43
elseif  (x==-L & y==11*L )
    cad_bits='0111010' ;    %44
elseif  (x==-3*L & y==11*L )
    cad_bits='0111011' ;  %45
elseif (x==-3*L & y==9*L )
    cad_bits='0111001' ;  %46
elseif (x==-L & y==9*L )
    cad_bits='0111000' ;  %47
elseif  (x==-7*L & y==9*L )
    cad_bits='0101000' ; %48
elseif (x==-5*L & y==9*L )
    cad_bits='0101001' ; %49
elseif (x==-5*L & y==11*L )
    cad_bits='0101011' ;   %50
elseif (x==-7*L & y==11*L )
    cad_bits='0101010' ;   %51
elseif (x==-9*L &  y==3*L )
    cad_bits='0101110' ;  %52
elseif  (x==-11*L & y==3*L )
    cad_bits='0101111' ;  %53
elseif(x==-11*L & y==L )
    cad_bits='0101101' ;  %54
elseif (x==-9*L & y==L )
    cad_bits='0101100' ;  %55
elseif(x==-7*L & y==L )
    cad_bits='0100100' ;   %56
elseif (x==-5*L & y==L )
    cad_bits='0100101' ; %57
elseif(x==-5*L & y==3*L )
    cad_bits='0100111' ; %58
elseif  (x==-7*L & y==3*L )
    cad_bits='0100110' ;   %59
elseif  (x==-L & y==3*L )
    cad_bits='0100010' ;    %60
elseif  (x==-3*L & y==3*L )
    cad_bits='0100011' ;  %61
elseif (x==-3*L & y==L )
    cad_bits='0100001' ;  %62
elseif (x==-L & y==L )
    cad_bits='0100000' ;  %63
elseif (x==-L & y==-L )
    cad_bits='1100000' ; %64
elseif  (x==-3*L & y==-L )
    cad_bits='1100001' ; %65
elseif (x==-3*L & y==-3*L )
    cad_bits='1100011' ;   %66
elseif (x==-L & y==-3*L )
```

```
    cad_bits='1100010' ;   %67
elseif (x==-7*L & y==-3*L )
    cad_bits='1100110' ;  %68
elseif (x==-5*L & y==-3*L )
    cad_bits='1100111' ;  %69
elseif (x==-5*L & y==-L )
    cad_bits='1100101' ;  %70
elseif (x==-7*L & y==-L )
    cad_bits='1100100' ;  %71
elseif (x==-9*L & y==-L )
    cad_bits='1101100' ;   %72
elseif (x==-11*L & y==-L )
    cad_bits='1101101' ; %73
elseif  (x==-11*L & y==-3*L )
    cad_bits='1101111' ; %74
elseif (x==-9*L & y==-3*L )
    cad_bits='1101110' ;   %75
elseif (x==-7*L & y==-11*L )
    cad_bits='1101010' ;    %76
elseif (x==-5*L &  y==-11*L )
    cad_bits='1101011' ;  %77
elseif (x==-5*L & y==-9*L )
    cad_bits='1101001' ;  %78
elseif  (x==-7*L & y==-9*L )
    cad_bits='1101000' ;  %79
elseif (x==-L & y==-9*L )
    cad_bits='1111000' ; %80
elseif(x==-3*L & y==-9*L )
    cad_bits='1111001' ; %81
elseif (x==-3*L &  y==-11*L )
    cad_bits='1111011' ;   %82
elseif (x==-L & y==-11*L )
    cad_bits='1111010' ;   %83
elseif (x==-9*L & y==-5*L )
    cad_bits='1111110' ;  %84
elseif  (x==-11*L & y==-5*L )
    cad_bits='1111111' ;  %85
elseif (x==-11*L & y==-7*L )
    cad_bits='1111101' ;  %86
elseif  (x==-9*L &  y==-7*L )
    cad_bits='1111100' ;  %87
elseif  (x==-7*L &  y==-7*L )
    cad_bits='1110100' ;   %88
elseif(x==-5*L & y==-7*L )
    cad_bits='1110101' ; %89
elseif (x==-5*L & y==-5*L )
    cad_bits='1110111' ; %90
elseif (x==-7*L & y==-5*L )
    cad_bits='1110110' ;   %91
elseif(x==-L & y==-5*L )
    cad_bits='1110010' ;    %92
elseif (x==-3*L & y==-5*L )
    cad_bits='1110011' ;  %93
elseif (x==-3*L & y==-7*L )
    cad_bits='1110001' ;  %94
elseif (x==-L & y==-7*L )
    cad_bits='1110000' ;  %95
elseif  (x==L & y==-7*L )
    cad_bits='1010000' ; %96
elseif  (x==3*L &  y==-7*L )
```

```
    cad_bits='1010001' ; %97
elseif (x==3*L & y==-5*L )
    cad_bits='1010011' ;   %98
elseif  (x==L &  y==-5*L )
    cad_bits='1010010' ;   %99
elseif  (x==7*L &  y==-5*L )
    cad_bits='1010110' ;  %100
elseif  (x==5*L & y==-5*L )
    cad_bits='1010111' ;  %101
elseif (x==5*L & y==-7*L )
    cad_bits='1010101' ;  %102
elseif (x==7*L & y==-7*L )
    cad_bits='1010100' ;  %103
elseif (x==9*L & y==-7*L )
    cad_bits='1011100' ;   %104
elseif (x==11*L & y==-7*L )
    cad_bits='1011101' ; %105
elseif (x==11*L & y==-5*L )
    cad_bits='1011111' ; %106
elseif  (x==9*L &  y==-5*L )
    cad_bits='1011110' ;   %107
elseif (x==L & y==-11*L )
    cad_bits='1011010' ;    %108
elseif(x==3*L & y==-11*L )
    cad_bits='1011011' ;  %109
elseif  (x==3*L & y==-9*L )
    cad_bits='1011001' ;  %110
elseif (x==L & y==-9*L )
    cad_bits='1011000' ;  %111
elseif (x==7*L & y==-9*L )
    cad_bits='1001000' ; %112
elseif (x==5*L & y==-9*L )
    cad_bits='1001001' ; %113
elseif  (x==5*L & y==-11*L )
    cad_bits='1001011' ;   %114
elseif (x==7*L & y==-11*L )
    cad_bits='1001010' ;   %115
elseif (x==9*L & y==-3*L )
    cad_bits='1001110' ;  %116
elseif (x==11*L & y==-3*L )
    cad_bits='1001111' ;  %117
elseif (x==11*L & y==-L )
    cad_bits='1001101' ;  %118
elseif (x==9*L & y==-L )
    cad_bits='1001100' ;  %119
elseif (x==7*L & y==-L )
    cad_bits='1000100' ;   %120
elseif (x==5*L & y==-L )
    cad_bits='1000101' ; %121
elseif (x==5*L & y==-3*L )
    cad_bits='1000111' ; %122
elseif (x==7*L & y==-3*L )
    cad_bits='1000110' ;   %123
elseif (x==L & y==-3*L )
    cad_bits='1000010' ;    %124
elseif (x==3*L & y==-3*L )
    cad_bits='1000011' ;  %125
elseif (x==3*L & y==-L )
    cad_bits='1000001' ;  %126
elseif(x==L & y==-L )
```

```
    cad_bits='1000000' ;  %127
  end
```

- **z_tab_inv_M_256:**

```
function [cad_bits] = z_tab_inv_M_256( simb )

%Hallo el valor de L.
%L:minima distancia cartesiana a un simbolo...
%...tanto en fase como en quadratura.
%Asi,como dijimos, la mayor distancia
%de simbolo este entre -1 y 1.
 L=1/(15*sqrt(2));

%x:componente en fase del simbolo.
 x=real(simb);
%y:componente en quadratura del simbolo.
 y=imag(simb);

%aplico la tabla y obtengo los bits.

 if (x==-15*L & y==-15*L )
    cad_bits='00000000' ; %0
 elseif(x==-15*L & y==-13*L )
    cad_bits='00000001' ; %1
 elseif(x==-15*L &  y==-11*L )
    cad_bits='00000011' ;   %2
 elseif (x==-15*L & y==-9*L )
    cad_bits='00000010' ;   %3
 elseif (x==-15*L & y==-7*L )
    cad_bits='00000110' ;  %4
 elseif (x==-15*L & y==-5*L )
    cad_bits='00000111' ;  %5
 elseif (x==-15*L & y==-3*L )
    cad_bits='00000101' ;  %6
 elseif (x==-15*L & y==-L )
    cad_bits='00000100' ;  %7
 elseif (x==-15*L & y==L )
    cad_bits='00001100' ;   %8
 elseif (x==-15*L & y==3*L )
    cad_bits='00001101' ; %9
 elseif (x==-15*L & y==5*L )
    cad_bits='00001111' ; %10
 elseif (x==-15*L & y==7*L )
    cad_bits='00001110' ;   %11
 elseif (x==-15*L & y==9*L )
    cad_bits='00001010' ;    %12
 elseif (x==-15*L & y==11*L )
    cad_bits='00001011' ;  %13
 elseif (x==-15*L & y==13*L )
    cad_bits='00001001' ;  %14
 elseif (x==-15*L & y==15*L )
    cad_bits='00001000' ;  %15
 elseif (x==-13*L & y==15*L )
    cad_bits='00011000' ; %16
 elseif (x==-13*L & y==13*L )
    cad_bits='00011001' ; %17
 elseif (x==-13*L & y==11*L )
```

```
    cad_bits='00011011' ;   %18
elseif (x==-13*L & y==9*L )
    cad_bits='00011010' ;   %19
elseif (x==-13*L &  y==7*L )
    cad_bits='00011110' ;  %20
elseif (x==-13*L & y==5*L )
    cad_bits='00011111' ;  %21
elseif (x==-13*L & y==3*L )
    cad_bits='00011101' ;  %22
elseif (x==-13*L & y==L )
    cad_bits='00011100' ;  %23
elseif (x==-13*L & y==-L )
    cad_bits='00010100' ;   %24
elseif (x==-13*L & y==-3*L )
    cad_bits='00010101' ; %25
elseif (x==-13*L & y==-5*L )
    cad_bits='00010111' ; %26
elseif (x==-13*L & y==-7*L )
    cad_bits='00010110' ;   %27
elseif (x==-13*L & y==-9*L )
    cad_bits='00010010' ;    %28
elseif (x==-13*L & y==-11*L )
    cad_bits='00010011' ;  %29
elseif (x==-13*L & y==-13*L )
    cad_bits='00010001' ;  %30
elseif (x==-13*L & y==-15*L )
    cad_bits='00010000' ;  %31
elseif (x==-11*L & y==-15*L )
    cad_bits='00110000' ; %32
elseif (x==-11*L & y==-13*L )
    cad_bits='00110001' ; %33
elseif (x==-11*L & y==-11*L )
    cad_bits='00110011' ;   %34
elseif (x==-11*L & y==-9*L )
    cad_bits='00110010' ;   %35
elseif (x==-11*L & y==-7*L )
    cad_bits='00110110' ;  %36
elseif (x==-11*L & y==-5*L )
    cad_bits='00110111' ;  %37
elseif (x==-11*L & y==-3*L )
    cad_bits='00110101' ;  %38
elseif (x==-11*L & y==-L )
    cad_bits='00110100' ;  %39
elseif (x==-11*L & y==L )
    cad_bits='00111100' ;   %40
elseif (x==-11*L & y==3*L )
    cad_bits='00111101' ; %41
elseif (x==-11*L & y==5*L )
    cad_bits='00111111' ; %42
elseif (x==-11*L & y==7*L )
    cad_bits='00111110' ;   %43
elseif (x==-11*L & y==9*L )
    cad_bits='00111010' ;    %44
elseif (x==-11*L & y==11*L )
    cad_bits='00111011' ;  %45
elseif (x==-11*L & y==13*L )
    cad_bits='00111001' ;  %46
elseif (x==-11*L & y==15*L )
    cad_bits='00111000' ;  %47
elseif (x==-9*L & y==15*L )
```

```
    cad_bits='00101000' ; %48
elseif (x==-9*L & y==13*L )
    cad_bits='00101001' ; %49
elseif (x==-9*L & y==11*L )
    cad_bits='00101011' ;  %50
elseif (x==-9*L & y==9*L )
    cad_bits='00101010' ;  %51
elseif (x==-9*L & y==7*L )
    cad_bits='00101110' ; %52
elseif (x==-9*L & y==5*L )
    cad_bits='00101111' ; %53
elseif (x==-9*L & y==3*L )
    cad_bits='00101101' ; %54
elseif (x==-9*L & y==L )
    cad_bits='00101100' ; %55
elseif (x==-9*L & y==-L )
    cad_bits='00100100' ;  %56
elseif (x==-9*L & y==-3*L )
    cad_bits='00100101' ; %57
elseif (x==-9*L & y==-5*L )
    cad_bits='00100111' ; %58
elseif (x==-9*L & y==-7*L )
    cad_bits='00100110' ;  %59
elseif (x==-9*L & y==-9*L )
    cad_bits='00100010' ;   %60
elseif (x==-9*L & y==-11*L )
    cad_bits='00100011' ; %61
elseif (x==-9*L & y==-13*L )
    cad_bits='00100001' ; %62
elseif (x==-9*L & y==-15*L )
    cad_bits='00100000' ; %63
elseif (x==-7*L & y==-15*L )
    cad_bits='01100000' ; %64
elseif (x==-7*L & y==-13*L )
    cad_bits='01100001' ; %65
elseif (x==-7*L & y==-11*L )
    cad_bits='01100011' ;  %66
elseif (x==-7*L & y==-9*L )
    cad_bits='01100010' ;  %67
elseif (x==-7*L & y==-7*L )
    cad_bits='01100110' ; %68
elseif (x==-7*L & y==-5*L )
    cad_bits='01100111' ; %69
elseif (x==-7*L & y==-3*L )
    cad_bits='01100101' ; %70
elseif (x==-7*L & y==-L )
    cad_bits='01100100' ; %71
elseif (x==-7*L & y==L )
    cad_bits='01101100' ;  %72
elseif (x==-7*L & y==3*L )
    cad_bits='01101101' ; %73
elseif (x==-7*L & y==5*L )
    cad_bits='01101111' ; %74
elseif (x==-7*L & y==7*L )
    cad_bits='01101110' ;  %75
elseif (x==-7*L & y==9*L )
    cad_bits='01101010' ;   %76
elseif (x==-7*L & y==11*L )
    cad_bits='01101011' ; %77
elseif (x==-7*L & y==13*L )
```

```
    cad_bits='01101001' ;  %78
elseif (x==-7*L & y==15*L )
    cad_bits='01101000' ;  %79
elseif (x==-5*L & y==15*L )
    cad_bits='01111000' ; %80
elseif (x==-5*L & y==13*L )
    cad_bits='01111001' ; %81
elseif (x==-5*L & y==11*L )
    cad_bits='01111011' ;   %82
elseif (x==-5*L & y==9*L )
    cad_bits='01111010' ;   %83
elseif (x==-5*L & y==7*L )
    cad_bits='01111110' ;  %84
elseif (x==-5*L & y==5*L )
    cad_bits='01111111' ;  %85
elseif (x==-5*L & y==3*L )
    cad_bits='01111101' ;  %86
elseif (x==-5*L & y==L )
    cad_bits='01111100' ;  %87
elseif (x==-5*L & y==-L )
    cad_bits='01110100' ;   %88
elseif (x==-5*L & y==-3*L )
    cad_bits='01110101' ; %89
elseif (x==-5*L & y==-5*L )
    cad_bits='01110111' ; %90
elseif (x==-5*L & y==-7*L )
    cad_bits='01110110' ;   %91
elseif (x==-5*L & y==-9*L )
    cad_bits='01110010' ;    %92
elseif (x==-5*L & y==-11*L )
    cad_bits='01110011' ;  %93
elseif (x==-5*L & y==-13*L )
    cad_bits='01110001' ;  %94
elseif (x==-5*L &  y==-15*L )
    cad_bits='01110000' ;  %95
elseif (x==-3*L & y==-15*L )
    cad_bits='01010000' ; %96
elseif (x==-3*L & y==-13*L )
    cad_bits='01010001' ; %97
elseif (x==-3*L & y==-11*L )
    cad_bits='01010011' ;   %98
elseif (x==-3*L & y==-9*L )
    cad_bits='01010010' ;   %99
elseif (x==-3*L & y==-7*L )
    cad_bits='01010110' ;  %100
elseif (x==-3*L & y==-5*L )
    cad_bits='01010111' ;  %101
elseif (x==-3*L & y==-3*L )
    cad_bits='01010101' ;  %102
elseif (x==-3*L & y==-L )
    cad_bits='01010100' ;  %103
elseif (x==-3*L & y==L )
    cad_bits='01011100' ;   %104
elseif (x==-3*L & y==3*L )
    cad_bits='01011101' ; %105
elseif (x==-3*L & y==5*L )
    cad_bits='01011111' ; %106
elseif (x==-3*L & y==7*L )
    cad_bits='01011110' ;   %107
elseif (x==-3*L & y==9*L )
```

```
    cad_bits='01011010' ;   %108
elseif (x==-3*L & y==11*L )
    cad_bits='01011011' ;  %109
elseif (x==-3*L & y==13*L )
    cad_bits='01011001' ;  %110
elseif (x==-3*L & y==15*L )
    cad_bits='01011000' ;  %111
elseif (x==-L & y==15*L )
    cad_bits='01001000' ; %112
elseif (x==-L & y==13*L )
    cad_bits='01001001' ; %113
elseif (x==-L & y==11*L )
    cad_bits='01001011' ;   %114
elseif (x==-L & y==9*L )
    cad_bits='01001010' ;   %115
elseif (x==-L & y==7*L )
    cad_bits='01001110' ;  %116
elseif (x==-L & y==5*L )
    cad_bits='01001111' ;  %117
elseif (x==-L & y==3*L )
    cad_bits='01001101' ;  %118
elseif (x==-L & y==L )
    cad_bits='01001100' ;  %119
elseif (x==-L & y==-L )
    cad_bits='01000100' ;   %120
elseif (x==-L & y==-3*L )
    cad_bits='01000101' ; %121
elseif (x==-L & y==-5*L )
    cad_bits='01000111' ; %122
elseif (x==-L & y==-7*L )
    cad_bits='01000110' ;   %123
elseif (x==-L & y==-9*L )
    cad_bits='01000010' ;   %124
elseif (x==-L & y==-11*L )
    cad_bits='01000011' ;  %125
elseif (x==-L & y==-13*L )
    cad_bits='01000001' ;  %126
elseif (x==-L & y==-15*L )
    cad_bits='01000000' ;  %127
elseif (x==L & y==-15*L )
    cad_bits='11000000' ; %128
elseif (x==L & y==-13*L )
    cad_bits='11000001' ; %129
elseif (x==L & y==-11*L )
    cad_bits='11000011' ;   %130
elseif  (x==L & y==-9*L )
    cad_bits='11000010' ;   %131
elseif (x==L & y==-7*L )
    cad_bits='11000110' ;  %132
elseif (x==L & y==-5*L )
    cad_bits='11000111' ;  %133
elseif (x==L & y==-3*L )
    cad_bits='11000101' ;  %134
elseif (x==L & y==-L )
    cad_bits='11000100' ;  %135
elseif (x==L & y==L )
    cad_bits='11001100' ;   %136
elseif (x==L & y==3*L )
    cad_bits='11001101' ; %137
elseif (x==L & y==5*L )
```

```
    cad_bits='11001111' ; %138
elseif (x==L & y==7*L )
    cad_bits='11001110' ;  %139
elseif (x==L & y==9*L )
    cad_bits='11001010' ;   %140
elseif (x==L & y==11*L )
    cad_bits='11001011' ; %141
elseif (x==L &  y==13*L )
    cad_bits='11001001' ; %142
elseif (x==L & y==15*L )
     cad_bits='11001000' ; %143
elseif (x==3*L & y==15*L )
     cad_bits='11011000' ; %144
elseif (x==3*L & y==13*L )
     cad_bits='11011001' ; %145
elseif (x==3*L &y==11*L )
     cad_bits='11011011' ;  %146
elseif (x==3*L & y==9*L )
    cad_bits='11011010' ;  %147
elseif(x==3*L & y==7*L )
    cad_bits='11011110' ; %148
elseif(x==3*L & y==5*L )
    cad_bits='11011111' ; %149
elseif(x==3*L & y==3*L )
    cad_bits='11011101' ; %150
elseif (x==3*L & y==L )
    cad_bits='11011100' ; %151
elseif (x==3*L & y==-L )
    cad_bits='11010100' ;  %152
elseif (x==3*L & y==-3*L )
    cad_bits='11010101' ; %153
elseif (x==3*L & y==-5*L )
    cad_bits='11010111' ; %154
elseif (x==3*L & y==-7*L )
    cad_bits='11010110' ;  %155
elseif (x==3*L & y==-9*L )
    cad_bits='11010010' ;   %156
elseif (x==3*L & y==-11*L )
    cad_bits='11010011' ; %157
elseif (x==3*L & y==-13*L )
    cad_bits='11010001' ; %158
elseif (x==3*L & y==-15*L )
    cad_bits='11010000' ; %159
elseif (x==5*L & y==-15*L )
    cad_bits='11110000' ; %160
elseif (x==5*L & y==-13*L )
    cad_bits='11110001' ; %161
elseif (x==5*L & y==-11*L )
    cad_bits='11110011' ;  %162
elseif (x==5*L & y==-9*L )
    cad_bits='11110010' ;  %163
elseif (x==5*L & y==-7*L )
    cad_bits='11110110' ; %164
elseif (x==5*L & y==-5*L )
    cad_bits='11110111' ; %165
elseif (x==5*L & y==-3*L )
    cad_bits='11110101' ; %166
elseif  (x==5*L & y==-L )
    cad_bits='11110100' ; %167
elseif (x==5*L & y==L )
```

```
    cad_bits='11111100' ;   %168
elseif (x==5*L & y==3*L )
    cad_bits='11111101' ; %169
elseif (x==5*L & y==5*L )
    cad_bits='11111111' ; %170
elseif (x==5*L & y==7*L )
    cad_bits='11111110' ;   %171
elseif (x==5*L & y==9*L )
    cad_bits='11111010' ;    %172
elseif (x==5*L & y==11*L )
    cad_bits='11111011' ;  %173
elseif (x==5*L & y==13*L )
    cad_bits='11111001' ;  %174
elseif (x==5*L & y==15*L )
    cad_bits='11111000' ;  %175
elseif (x==7*L & y==15*L )
     cad_bits='11101000' ; %176
elseif (x==7*L & y==13*L )
    cad_bits='11101001' ; %177
elseif (x==7*L & y==11*L )
    cad_bits='11101011' ;   %178
elseif (x==7*L & y==9*L )
    cad_bits='11101010' ;   %179
elseif (x==7*L & y==7*L )
    cad_bits='11101110' ;  %180
elseif (x==7*L & y==5*L )
    cad_bits='11101111' ;  %181
elseif (x==7*L & y==3*L )
    cad_bits='11101101' ;  %182
elseif (x==7*L & y==L )
    cad_bits='11101100' ;  %183
elseif (x==7*L & y==-L )
    cad_bits='11100100' ;   %184
elseif (x==7*L & y==-3*L )
    cad_bits='11100101' ; %185
elseif  (x==7*L & y==-5*L )
    cad_bits='11100111' ; %186
elseif (x==7*L & y==-7*L )
    cad_bits='11100110' ;   %187
elseif (x==7*L & y==-9*L )
    cad_bits='11100010' ;    %188
elseif (x==7*L & y==-11*L )
    cad_bits='11100011' ;  %189
elseif (x==7*L & y==-13*L )
    cad_bits='11100001' ;  %190
elseif (x==7*L & y==-15*L )
    cad_bits='11100000' ;  %191
elseif (x==9*L & y==-15*L )
    cad_bits='10100000' ; %192
elseif (x==9*L & y==-13*L )
    cad_bits='10100001' ; %193
elseif (x==9*L & y==-11*L )
    cad_bits='10100011' ;   %194
elseif (x==9*L & y==-9*L )
    cad_bits='10100010' ;   %195
elseif (x==9*L & y==-7*L )
    cad_bits='10100110' ;  %196
elseif (x==9*L & y==-5*L )
    cad_bits='10100111' ;  %197
elseif (x==9*L & y==-3*L )
```

```
    cad_bits='10100101' ;  %198
elseif (x==9*L & y==-L )
    cad_bits='10100100' ;  %199
elseif (x==9*L & y==L )
    cad_bits='10101100' ;   %200
elseif (x==9*L & y==3*L )
    cad_bits='10101101' ; %201
elseif (x==9*L & y==5*L )
    cad_bits='10101111' ; %202
elseif (x==9*L & y==7*L )
    cad_bits='10101110' ;   %203
elseif (x==9*L &  y==9*L )
    cad_bits='10101010' ;    %204
elseif (x==9*L & y==11*L )
    cad_bits='10101011' ;  %205
elseif (x==9*L & y==13*L )
    cad_bits='10101001' ;  %206
elseif (x==9*L & y==15*L )
    cad_bits='10101000' ;  %207
elseif (x==11*L & y==15*L )
    cad_bits='10111000' ; %208
elseif (x==11*L & y==13*L )
    cad_bits='10111001' ; %209
elseif  (x==11*L & y==11*L )
    cad_bits='10111011' ;   %210
elseif (x==11*L & y==9*L )
    cad_bits='10111010' ;   %211
elseif (x==11*L & y==7*L )
    cad_bits='10111110' ;  %212
elseif (x==11*L & y==5*L )
    cad_bits='10111111' ;  %213
elseif (x==11*L & y==3*L )
    cad_bits='10111101' ;  %214
elseif (x==11*L & y==L )
    cad_bits='10111100' ;  %215
elseif (x==11*L & y==-L )
    cad_bits='10110100' ;   %216
elseif (x==11*L & y==-3*L )
    cad_bits='10110101' ; %217
elseif (x==11*L & y==-5*L )
    cad_bits='10110111' ; %218
elseif (x==11*L & y==-7*L )
    cad_bits='10110110' ;   %219
elseif (x==11*L & y==-9*L )
    cad_bits='10110010' ;    %220
elseif (x==11*L & y==-11*L )
    cad_bits='10110011' ;  %221
elseif (x==11*L & y==-13*L )
    cad_bits='10110001' ;  %222
elseif (x==11*L & y==-15*L )
    cad_bits='10110000' ;  %223
elseif (x==13*L & y==-15*L )
    cad_bits='10010000' ; %224
elseif (x==13*L & y==-13*L )
    cad_bits='10010001' ; %225
elseif (x==13*L & y==-11*L )
     cad_bits='10010011' ;   %226
elseif (x==13*L & y==-9*L )
    cad_bits='10010010' ;   %227
elseif (x==13*L & y==-7*L )
```

```
    cad_bits='10010110' ;  %228
elseif (x==13*L & y==-5*L )
    cad_bits='10010111' ;  %229
elseif (x==13*L & y==-3*L )
    cad_bits='10010101' ;  %230
elseif (x==13*L & y==-L )
    cad_bits='10010100' ;  %231
elseif (x==13*L & y==L )
    cad_bits='10011100' ;   %232
elseif (x==13*L & y==3*L )
    cad_bits='10011101' ; %233
elseif (x==13*L & y==5*L )
    cad_bits='10011111' ; %234
elseif (x==13*L & y==7*L )
    cad_bits='10011110' ;   %235
elseif (x==13*L & y==9*L )
    cad_bits='10011010' ;    %236
elseif (x==13*L & y==11*L )
    cad_bits='10011011' ;  %237
elseif (x==13*L & y==13*L )
    cad_bits='10011001' ;  %238
elseif (x==13*L &  y==15*L )
    cad_bits='10011000' ;  %239
elseif (x==15*L & y==15*L )
    cad_bits='10001000' ; %240
elseif (x==15*L & y==13*L )
    cad_bits='10001001' ; %241
elseif (x==15*L & y==11*L )
    cad_bits='10001011' ;   %242
elseif (x==15*L & y==9*L )
    cad_bits='10001010' ;   %243
elseif (x==15*L & y==7*L )
    cad_bits='10001110' ;  %244
elseif (x==15*L & y==5*L )
    cad_bits='10001111' ;  %245
elseif (x==15*L & y==3*L )
    cad_bits='10001101' ;  %246
elseif (x==15*L & y==L )
    cad_bits='10001100' ;  %247
elseif (x==15*L & y==-L )
    cad_bits='10000100' ;   %248
elseif (x==15*L & y==-3*L )
    cad_bits='10000101' ; %249
elseif (x==15*L & y==-5*L )
    cad_bits='10000111' ; %250
elseif (x==15*L & y==-7*L )
     cad_bits='10000110' ;   %251
elseif (x==15*L & y==-9*L )
    cad_bits='10000010' ;    %252
elseif (x==15*L & y==-11*L )
    cad_bits='10000011' ;  %253
elseif (x==15*L & y==-13*L )
    cad_bits='10000001' ;  %254
elseif (x==15*L & y==-15*L )
    cad_bits='10000000' ;  %255
end
```

Para acabar con el análisis de estas Tablas, nos gustaría recalcar que dichas Tablas son inversas en el sentido de entradas y salidas, a las Tablas de conversión *'z_tab_M_4'*, *'z_tab_M_8'*, etc..., vistas en el Bloque Transmisor.

Pero la relación entre bits y símbolos, para ambos conjuntos de Tablas, es la misma. Con lo cual, si queremos modificar algún parámetro de alguna Tabla, como el valor de la variable L, la posición de cualquier símbolo en la constelación,(x,y), debemos hacerlo de manera análoga en la Tabla inversa correspondiente, para que la decisión sobre los símbolos se correcta.

Aquí concluye exhaustivo estudio sobre la ***implementación en Matlab*** de nuestro proyecto.

III.4.- IMPLEMENTACIÓN EN LENGUAJE C

Esta es la ***última fase de la memoria del proyecto***, y se trata de implementar todo el código desarrollado en Matlab en un lenguaje de alto nivel, concretamente el ***Lenguaje C***. Con esto conseguiremos las siguientes ventajas:

- Crearemos ***dos archivos "stand-alone"*** (uno para el Bloque Transmisor, y otro para el Bloque Receptor), esto es, generaremos dos archivos ejecutables ***'.exe'*** independientes del entorno Matlab.
- Consecuentemente, para la transmisión de ficheros, no necesitaremos ***ejecutar*** el programa ***Matlab*** y, por tanto, no necesitaremos dicho programa para realizar nuestra aplicación práctica.
- Esto supone además la ***portabilidad de dicho código***, si bien es necesario que dichos archivos vayan acompañados de algunas librerías dinámicas de Matlab (archivos de extensión *'.dll'*), como veremos más adelante.
- Además, ***aumentamos la seguridad*** de nuestros programas, al esconder el código fuente, ya que los *'archivos.m'* de Matlab son ficheros de texto ASCII, cuyo código se puede manipular y dañar. Esto no ocurre con los *'ficheros.exe'* ejecutables, ya que se trata de ficheros binarios.
- A priori, la ***velocidad de proceso*** del sistema tendría que verse mejorada por dos razones fundamentales[†6]:

  - El código compilado se ejecuta más rápido que el código interpretado.
  - El lenguaje C puede evitar asiganaciones de memoria innecesarias que el intérprete de Matlab realiza. Consecuentemente, el código compilado trata mejor los bucles que el código interpretado.

  Pero en la práctica esto no es así, pues los archivos principales *'z_main.m'*, tanto en transmisión como en recepción, se ejecutan más rápidamente (dentro del entorno Matlab, obviamente), que los archivos *'z_main.exe'* en el sistema operativo MS-DOS de Windows.
  Las posibles razones pueden deberse a que:

  - Nuestras funciones de Matlab están muy vectorizadas, y el lenguaje C no es óptimo para tratar vectores.
  - Dichas funciones de Matlab trabajan mayormente con funciones de indexamiento de dichos vectores (como el operador ':'), o bien con funciones matemáticas, cuyas traducciones tampoco son óptimas en el lenguaje C.

  Esto da una idea de la potencialidad del lenguaje Matlab, como herramienta de procesamiento matemático.

A continuación vamos a hablar un poco del compilador de Matlab usado para nuestro propósito. Posteriormente vamos a comentar los problemas que hemos tenido con el compilador, por las limitaciones que todavía existen en esta nueva versión 2.1 de dicho compilador.

COMPILADOR DE MATLAB

Para traducir el código de Matlab a lenguaje C, hemos usado un 'toolbox' de Matlab llamado ***Compiler***. Se trata de un compilador que traduce funciones con extensión *'.m'* a funciones propias del lenguaje C, esto es, funciones con extensión *'.c'* y sus cabeceras *'.h'* correspondientes. Además, también tiene la opción de compilar dichos archivos *'.c'*, 'linkarlos' con las cabeceras y debidas librerías de Matlab, y obtener finalmente el archivo ejecutable, en nuestro caso los archivos homónimos *'z_main.exe'*, tanto para el Bloque Transmisor como para el receptor.

Nuestra versión Matlab 6.0 incorpora la versión de compilador ***Compiler 2.1***, y con él no solamente nos ***ahorramos*** el trabajo de ***rescribir el código en C***, con la dificultad que conlleva conocer la compatibilidad de los tipos de datos entre Matlab y el lenguaje C, sino que, además, este compilador soporta las funciones de transmisión y recepción a través de la tarjeta de sonido, y no tenemos necesidad tampoco de adentrarnos en el tedioso estudio y en la compleja utilización de las ***APIs de Windows*** para controlar nuestra tarjeta de sonido[†6].

Las herramientas que necesitamos para compilar nuestros *'archivos.m'* y construir el *'fichero.exe'* ejecutable son[†6]:

- El entorno Matlab, obviamente, para poder usar su compilador.
- El compilador de Matlab.
- El compilador de C.
- La librería MATLAB C/C++ MATH library.

El ***compilador de Matlab*** es ***muy versátil***, en el sentido de que acepta como código fuente a compilar tanto 'ficheros.m' de Matlab, como los archivos externos de Matlab (los llamados MEX-Files) y los *'archivos.c'* y *'archivos.cpp'* de los lenguajes C y C++ respectivamente. El compilador traduce dichos archivos fuentes (que pueden incluso estar mezclados), y obtiene los *'archivos.obj'*, los 'linka' con las *'librerías.h'* y las librerías de Matlab ya mencionadas para obtener el *'archivo.exe'* ejecutable correspondiente. Pero no sólo tiene la opción de crear *'archivos.exe'*, sino que tiene la posibilidad de obtener únicamente el *'archivo.c'*, o bien un archivo externo de Matlab (MEX-File), un *'archivo.cpp'* (código fuente de c++), una librería, etc...

Incluso, mediante la opción ***mbuild –setup***, se puede escoger el compilador de C, entre todos los presentes en nuestro sistema. Nos quedamos con el ***compilador LCC*** que trae Matlab por defecto[†6].

NOTA: MEX-Files: Matlab external files, son archivos externos de Matlab, cuyo código está en C (el intérprete de Matlab también acepta ficheros en fortran), y se pueden ejecutar en el entorno Matlab, y según el sistema operativo tienen una extensión distinta, como los 'archivos.dll' en el sistema Windows).

Todo este abanico de opciones del compilador implica que lo tengamos que configurar para poder utilizarlo. Lo primero es invocarlo, y eso se realiza mediante el comando ***mcc*** de Matlab. La sintaxis es la siguiente:

- mcc [-opciones] fich1.m [fich2.m...fichN.m] [fich1.c/cpp ...fich2.c/cpp]

Ni que decir tiene que si se incluye algún *'fichero.c'* o *'fichero.cpp'* en la línea de comando, junto con uno o más 'ficheros.m', los *'ficheros.c'* pasan directamente al compilador de C, sin pasar por el compilador de Matlab. Todo este proceso es transparente al usuario.

Son precisamente esas ***opciones*** del comando ***mcc*** las que tenemos que estudiar para obtener nuestro *'fichero.exe'* ejecutable, en lugar de cualquier otro código, como un *'fichero.dll'* por ejemplo. Por suerte, existen opciones que son ***macros***[†6], es decir, que constituyen un conjunto de opciones, para obtener un código en cuestión. Así, el macro necesario para compilar un 'archivo.m' , generar el *'fichero.c'* correspondiente y obtener finalmente el fichero ejecutable independiente del entorno Matlab (es decir, un fichero ***'C stand-alone'***), es el ***macro '-m'***.

La ***línea de comando*** en cuestión es la siguiente, tanto para el Bloque Transmisor como para el Receptor (suponemos que nos encontramos en el directorio donde están todos los *'archivos.m'* de nuestro Bloque Transmisor o del Receptor) [†6]:

- *mcc -m -d .\lge_C z_main.m*

La opción *'–d .\lge_C'* simplemente introduce todo el código generado *('archivos.c'*, *'archivos.h'* y *'archivo.exe'*) en la subcarpeta lge_C, del directorio actual.

Cuando el compilador es invocado con esta ***macro '-m'***, traduce los *'archivos.m'* de entrada a archivos homónimos, pero con la extensión propia lenguaje C, esto es, *'archivos.c'*, disponibles para ser compilados y generar cualquiera de los distintos tipos ejecutables que existen. Pero además, también produce un 'fichero.c' llamado ***'wrapper'***, que contiene la interfaz requerida entre el código generado por el compilador y el tipo ejecutable soportado, que este caso será un ***'C stand alone'***. Luego , nuestro compilador de C compilará esos *'archivos.c'*, y los *'archivos.obj'* resultantes serán 'linkados' con los archivos de la ***librería MATLAB C/C++ Math Library***.

Veamos brevemente el ***conjunto de opciones*** que encierra nuestra ***macro '-m'***[†6]:

- –m es equivalente a: -W main -L C -t -T link:exe -h libmmfile.mlib

Analicemos por encima el significado de cada uno de estos parámetros:

- -t: Traduce el código de Matlab a código de C/C++.
- -W main: Produce un fichero *'wrapper'*.El parámetro *'main'* indica que prepara el proceso de compilación para una aplicación de tipo 'C stand-alone'.
- -L C: Genera el código C como 'target language'.
- -T link:exe : Crea un ejecutable como salida.
- -h: Automáticamente, encuentra y compila las funciones que son llamadas dentro de *nuestros 'archivos.m'*.
- libmmfile.mlib: linka con esta librería cada vez que sea necesario.

Una vez ejecutada dicha sentencia, tanto en la carpeta del Bloque Transmisor, como en el directorio del Bloque Receptor, el compilador generará un *'archivo.c'* y un *'archivo.h'* por cada *'archivo.m'* implicado en nuestro código, además del archivo wrapper *'z_main_main.c'*, y del archivo *'z_main.exe'*, que será el que utilicemos para nuestra aplicación fuera del entorno de Matlab. Por último crea una subcarpeta 'bin' que en nuestro caso carece de contenido, pero que parece tener la función de guardar los posibles archivos gráficos que surjan en la compilación.

Para ***distribuir nuestro proyecto***, no es necesario tener la aplicación de Matlab, como ya adelantamos. Sin embargo, sí necesitamos la siguiente información[†6]:

- Los ficheros *'z_main.exe'* tanto del Bloque Receptor como del transmisor.
- El contenido, si existe, de la subcarpeta *\bin* comentada anteriormente.
- Algún fichero externo de Matlab que nuestro programa utilice, como algún *'archivo.dll'* de Windows. Ya veremos posteriormente que necesitaremos los archivos *'playsnd.dll'* y *'rcsnd.dll'* para la transmisión y recepción de las muestras a través de la tarjeta de sonido.
- Todas las librerías dinámicas MATLAB Math run-time.

Estas librerías dinámicas a las que hacemos alusión se encuentran contenidas en un archivo ejecutable, ubicado en la siguiente ruta[†6]:

- *$MATLAB\extern\lib\win32\mglinstaller.exe*

, donde *$MATLAB* simboliza el directorio raíz de Matlab, del que cuelgan el resto de subcarpetas de dicha aplicación. Dicho archivo, ***'mglinstaller.exe'***, al ser ejecutado, desempaqueta y descomprime las librerías dinámicas ***MATLAB Math and Graphics Run-time library***. Previamente, el compilador pedirá el nombre de un directorio, a partir del cual colgarán las subcarpetas *bin\win32*. Las librerías se ubicarán en este último subdirectorio *\win32*.

Tendremos que establecer una ruta hacia este subdirectorio, por ejemplo mediante el comando de MS-DOS *'PATH'*, para que al ejecutarse los archivos *'z_main.exe'*, el sistema encuentre el camino de las librerías especificadas en el código binario de dichos archivos ejecutables de nuestra aplicación. Otra opción sería meter el archivo ejecutable en el mismo directorio que las librerías, o sea, en el subdirectorio *\win32*.

¡¡IMPORTANTE!!: Para evitar problemas en la ejecución de nuestros ficheros ejecutables en PCs carentes de la aplicación Matlab, es necesario tener en cuenta estas indicaciones:

- Incluir los archivos *'wavplay.m'* y *'playsnd.dll'* (necesarios para la transmisión de las muestras), en el mismo directorio que los *'archivos.m'* del Bloque Transmisor, a la hora de la compilación. Con esto evitamos que el compilador se vaya a buscar dichos archivos a las subcarpetas del directorio raíz de Matlab, *'$MATLAB'*.
  Si no hacemos esto, y compilamos el código de Matlab, al intentar ejecutar el archivo 'z_main.exe', dará un error en tiempo de compilación, justo al intentar transmitir, pues intentará buscar la ruta donde se ubican dichos archivos en nuestro PC. Hay que incluir ambos archivos, pues, como veremos, la función *'wavplay'* realiza una llamada al archivo *'playsnd.dll'*.
- Añadir los archivos *'wavrecord.m'* y *'recsnd.dll'* (encargados de la captura de las muestras en recepción), en el mismo directorio que los *'ficheros.m'* del Bloque Receptor, por el mismo motivo que en el punto anterior. Así, evitamos un error en tiempo de ejecución, justo antes de intentar recibir las muestras. En este caso, es la función *'wavrecord'* quien llama al archivo *'recsnd.dll'*.

- Ya en el PC donde queramos instalar nuestra aplicación, el archivo *'playsnd.dll'* deberá estar en el mismo directorio que su ejecutable *'z_main'* correspondiente, para estar localizado en tiempo de ejecución. De la misma manera, el archivo *'recsnd.dll'* deberá anexarse en el mismo directorio donde se sitúe el fichero 'z_main' encargado de la implementación del Bloque Receptor.

Una vez realizadas dichas aclaraciones, y para terminar con el compilador, veamos un ***gráfico*** ilustrativo con los ***pasos*** que sigue la herramienta ***Compiler 2.1*** de Matlab[†6] para la obtención de nuestro *'fichero.exe'*, si bien se trata de un caso genérico, pues algunas de las librerías, como las gráficas, no nos son necesarias:

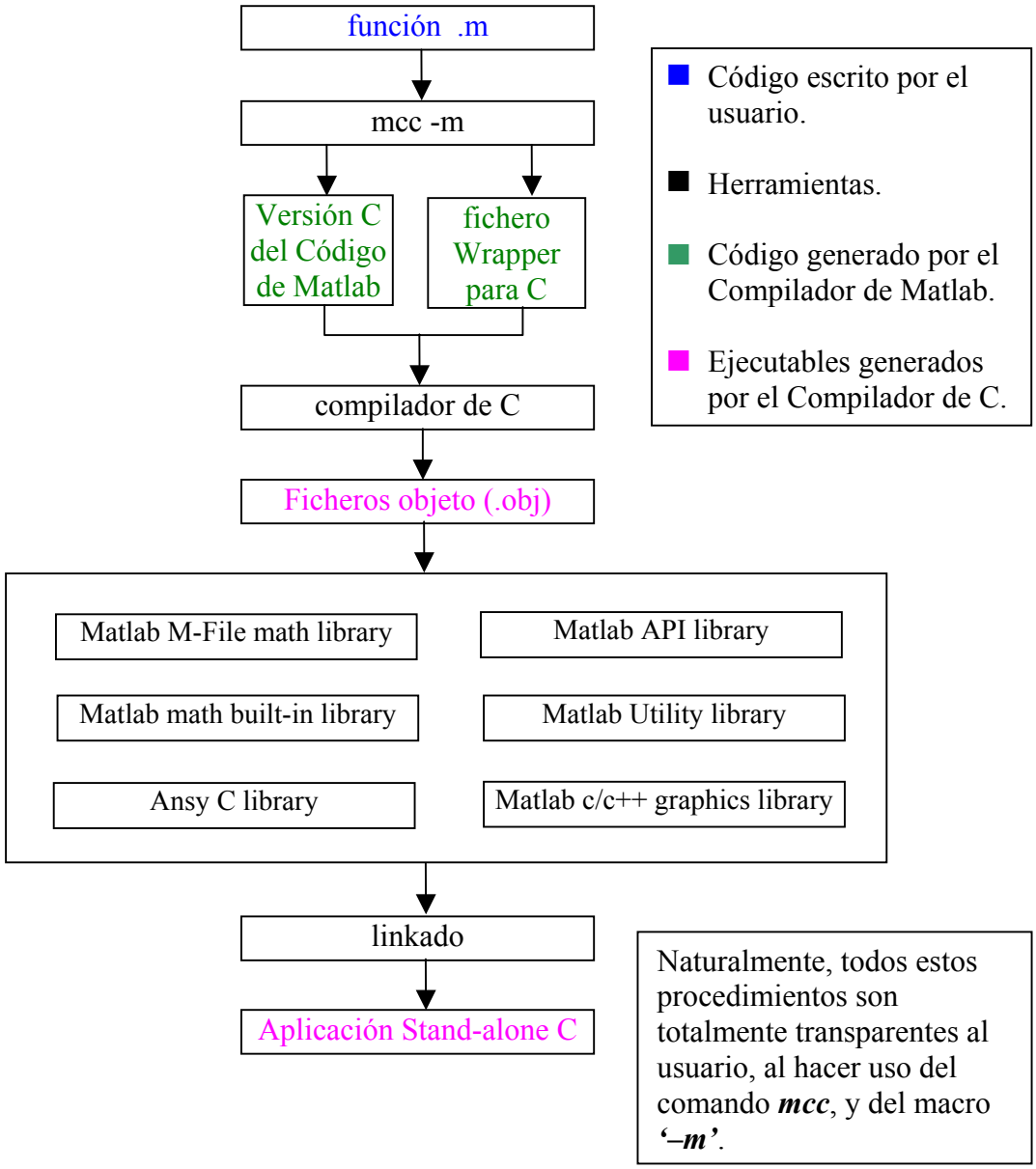


Fig. 111.-Esquema de la herramienta Compiler 2.1

PROBLEMAS EN LA COMPILACIÓN

Seguidamente vamos a comentar los contratiempos con los que nos hemos ido encontrando, y que imposibilitaron que la compilación se hiciera de una primera vez.

Pause

El primer problema lo encontramos en la función *'pause.m'*, que como sabemos, detiene la ejecución del programa hasta que una tecla es pulsada.

Dicha función ha sido incorporada recientemente por la nueva versión del compilador de Matlab. Pero tiene una limitación muy incómoda para la ejecución de nuestro proyecto fuera del entorno de Matlab[†6]:

- 'pause': esta función no la ejecuta correctamente, y el programa no se detiene, así estropeamos nuestra forma de sincronizar el Bloque Receptor y transmisor a la hora de la transmisión/recepción de las muestras.
- 'pause(n)': el parámetro 'n' indica los segundos que se detiene el programa. Una vez transcurrido dicho intervalo de tiempo el programa continuará ejecutándose automáticamente. El compilador sí traduce correctamente esta función cuando introducimos dicho parámetro de entrada, con lo que podríamos usar dicha función de esta manera. El problema reside en la incomodidad de la aplicación, pues tendríamos que calcular bien los tiempos de espera que introduzcamos en el transmisor y en el receptor, y tendríamos que intuir cuando arrancar el receptor, segundos antes de que se ejecute la función *'wavplay'* en el transmisor. Además tendríamos que darnos mucha prisa para meter los parámetros de diseño en receptor antes de que se procesen todas las tramas en transmisión. Finalmente concluimos que sería inviable sincronizar de esta manera el transmisor y el receptor.

Una posible alternativa sería utilizar la función de C:

- getch( );

, que simplemente detiene el programa y recoge un carácter del teclado. Si pulsamos directamente la tecla "enter", el carácter en blanco de teclado se perdería, pues no le asignamos ninguna variable tipo *char* como parámetro de salida para almacenar dicho carácter. Aún así, conseguimos parar la ejecución del programa hasta que no se pulse una tecla. De este modo conseguimos simular la función *'pause'* de Matlab. El único contratiempo es que tendríamos que obtener el fichero 'z_main.exe' en dos fases (tanto para el Bloque Transmisor como para el receptor):

- Obtenemos primeramente el código fuente en lenguaje C para poder manipularlo he introducir dicha función getch(), en lugar de la función que simulara a *'pause'*. Para obtener únicamente dicho código en C, usamos el mismo parámetro *'–m'*, pero añadimos esta vez el modificador *'–c'*, es decir, usamos las opciones *'–mc'*.
- Una vez los ficheros *'z_main.c'* tanto del Bloque Transmisor como del Bloque Receptor (pues son los únicos que incorporan la función *'pause'*) han sido modificados, se realizará la compilación del código C de estas

funciones para obtener el ejecutable. Esta vez tendremos que usar el comando mcc seguida del parámetro *'–m'* y de todos los *'ficheros.c'* implicados. Es decir:

- mcc –m z_main.c z_transmite_fich.c z_bloq_mod.c.....etc...

Otra alternativa mucho más cómoda, y por la cual nos hemos decantado, es usar la función de Matlab[†6]:

- *'input(' ')'*

, que muestra en pantalla lo que está entrecomillado, y además espera un número por teclado. Como existe un espacio en blanco, no mostrará nada en pantalla, y además no hemos asignado ninguna variable de salida para que recoja dicho número de entrada.

Además, cuando pulsemos la tecla *"enter"*, el programa entenderá que el número ha sido introducido. Así, simplemente pulsamos *"enter"* y conseguimos que dicha función tenga el mismo funcionamiento que *'pause'*.

Esta alternativa es más cómoda pues podemos obtener directamente los archivos *'z_main.exe'* ejecutables, ya que la función 'input' es otra de las últimas incorporaciones del compilador Compiler 2.1. El único inconveniente es que no podemos pulsar cualquier tecla, sino simplemente *"enter"*. Por todo ello, y para ser coherentes, hemos tenido que cambiar la frase que aparece en pantalla:

- 'pulse cualquier tecla para....'

por esta otra frase:

- 'pulse la tecla "enter" para...'

cada vez que se producía una pausa controlada en el programa por el motivo que fuese.

Eval

Ya hemos comentado en reiteradas ocasiones que la función *'rcosflt'* es la encargada de construir el pulso raíz de coseno alzado y realizar la convolución de los símbolos de entrada con dicho pulso conformador. Dicha función es propia de Matlab, y en su ejecución realiza una llamada a la función *'rcosfir'*, y ésta a su vez realiza una llamada a la función *'cosfir'*. Tanto la función *'rcosfir'* como *'cosfir'* incluyen en su código de Matlab una función llamada *'eval.m'*, que también es del nuevo repertorio de funciones aceptadas por el compilador, pero que tiene sus limitaciones, y en este caso no la aceptaba, y generaba un error en tiempo de compilación[†6].

Como la función *'rcosflt.m'* es usada tanto en el Bloque Transmisor como en el Bloque Receptor, para implementar precisamente los filtros transmisor y receptor, dicho problema de compilación nos afectaba en ambos extremos del canal.

Empezamos a analizar el código de dicha función, y las funciones que son llamadas durante el código, y llegamos a la conclusión de que la función *'eval'* sólo intervenía en el tratamiento de los parámetros de entrada y salida de las funciones donde estaban incluidas, para simular lo que en lenguaje orientado a objetos se denomina sobrecarga de funciones, esto es, que dependiendo del número o tipo de argumentos de

entrada, la función se comportara de una manera u otra. Por ejemplo, si introducimos el parámetro *'sqrt'*, indicamos la opción de pulso raíz de coseno alzado, y si no introducimos dicha opción, el programa generará un pulso de coseno alzado.

Así, la solución que adoptamos fue crear, a partir de todas estas funciones, una función denominada *'z_filtro'*, que únicamente calcula un filtro raíz de coseno alzado a partir de un filtro FIR y, mediante la función *'filter'*, genera la convolución con la señal de entrada. Esta función pierde en versatilidad, puesto que no podemos usarla para construir, por ejemplo, un pulso mediante un filtro IIR. Sin embargo dicha versatilidad no la necesitamos, ya que siempre ejecutaremos los mismos parámetros de entrada a la hora de construir el filtro raíz de coseno alzado.

De esta manera, ***el código de Matlab mostrado en el capítulo III.3 queda modificado***, de tal manera que la función 'rcosflt' utilizada tanto en la función 'z_oscillator' del Bloque Transmisor, como en la función 'z_bloq_dem' del Bloque Receptor, la hemos sustituido por la función ***'z_filtro'***. Es decir:

- tram_env_fas=z_filtro(v_c_f.',N,0,alfa); (rama de fase. Transmisor)
- tram_env_quad=z_filtro(v_c_q.',N,0,alfa); (rama de cuadrat. Transmisor)
- trama_env_fase= z_filtro(tr_fase,N,1,alfa); (rama de fase. Receptor)
- trama_env_quad= z_filtro(tr_quad,N,1,alfa); (rama de cuadrat. Receptor)

Los parámetro de salida son los mismos, pero los parámetros de entrada varían entre la funciones '*z_filtro*' y *'rcosflt'*. Dichos argumentos de entrada serán:

- La señal vectorial de entrada, que por ejemplo será *v_c_f* para la rama de fase del Bloque Transmisor.
- *N*: el número de muestras por símbolo.
- El tercer parámetro indicará en qué bloque estamos. Así controlamos el sobremuestreo, en lugar de emplear *'Fs'*, como vimos en *'rcosflt'*:
  - 0: Estamos en el Bloque Transmisor. Existirá sobremuestreo.
  - 1: Estamos en el Bloque Receptor. No existirá sobremuestreo.
- *alfa*: será el valor del factor de Roll-off.
- El *retraso* por defecto seguirá siendo de 3 intervalos de símbolos por cada cola, al igual que en *'rcosflt'*.

Con este código, además de evitar usar la función *'eval'*, que no tolera el compilador de Matlab, también simplificamos el código, acelerando la ejecución tanto del código interpretado de Matlab, como del código compilado en C, puesto que eliminamos líneas de código y reducimos el número de llamadas a funciones.

El código de nuestra función z_filtro es el siguiente:

- **z_filtro:**

```
function y = z_filtro(x,N,bloque,alfa)

    % z_filtro: variante de rcosflt, que unicamente realiza la convolucion de x con el pulso raiz...
    % ...de coseno alzado.
    % alfa: factor de roll-off.Rango entre 0 y 1. Si excede, lo saturamos a esos limites.
    % y=muestras a la salida del filtro transmisor.
```

```
% bloque=0...estamos en transmisor...existe sobre muestreo.
% N muestras de salida por simbolo entrante.
% retraso: retraso de grupo del filtro=3 muestras de entradas (3·T sg).(por defecto).
% T=tiempo de simbolo.
% exisitiran 2 colas de 3 simbolos=3·N muestras en cada extremo.

 retraso = 3;

%factor de Roll-off
if alfa < 0
   alfa=0;
elseif alfa > 1
   alfa=1;
end



%procesamiento de la señal de entrada
len_x=length(x);


%Construccion del filtro
%Construccion del numerador

time_T = [0:1/N :retraso]; %time_T: numero de simbolos. T: tiempo de simbolos
%time_T_r = alfa * time_T;

% raiz de coseno alzado
num=z_generar_filtro(N*2*retraso,1/2,alfa,N,retraso*N)*sqrt(N);
% fin de construccion del numerador

%denominador
den = 1;


if bloque~=0          %receptor: bloque~=0 no existe sobremuestreo
   %xx: señal de entrada modificada
   xx = zeros(len_x + (retraso*2*N),1);
   xx(1:len_x) = x(1:len_x);
else                  %transmisor: bloque==0 existe sobremuestreo
    % xx: señal de entrada modificada
    xx = zeros((len_x + retraso*2)*N,1);
    for i = 1 : len_x
       xx((i - 1) * N + 1) = x(i);
    end
end

% Señal de salida
   y= filter(num, den, xx);
```

Podemos observar que esta función realiza una única llamada, concretamente a la función *'z_generar_filtro'*, para construir el filtro raíz de coseno alzado, y que también ha sido construida tomando como base las funciones llamadas dentro de la función *'rcosflt'*, eliminando la parafernalia referente al tratamiento de los parámetros de entrada.

- **z_generar_filtro:**

```
function b=z_generar_filtro(grado_num,fo,alfa,fs,delay)
%B=FIRRCOS(grado_num,fo,alfa,fs,,DELAY)
%funcion que implementa la construccion del filtro raiz de coseno alzado.

%grado_num: grado del numerador.
%   DELAY must be an integer in the range [0, grado_num+1].
% alfa: rollof
%fs:frecuencia de muestreo(para D=1hz., fs=N hz.)

L = grado_num+1; % Length of window

%fo:frec. de corte (D/2 en nuestro sistema...en nuestro programa D=1 hz.).

% alfa is now always a rolloff factor - DF has been converted
if alfa == 0,
  alfa = realmin;
end

%n = -delay/fs : 1/fs : (L-delay-1)/fs;
n = ((0:L-1)-delay) ./ fs;


% square root raised cosine design

  ind1 = find(n == 0);
  if ~isempty(ind1),
    b(ind1) = - sqrt(2.*fo) ./ (pi.*fs) .* (pi.*(alfa-1) - 4.*alfa );
  end

  ind2 = find(abs(abs(8.*alfa.*fo.*n) - 1.0) < sqrt(eps));
  if ~isempty(ind2),
    b(ind2) = sqrt(2.*fo) ./ (2.*pi.*fs) ...
      * (   pi.*(alfa+1)  .* sin(pi.*(alfa+1)./(4.*alfa)) ...
          - 4.*alfa     .* sin(pi.*(alfa-1)./(4.*alfa)) ...
        + pi.*(alfa-1)  .* cos(pi.*(alfa-1)./(4.*alfa)) ...
       );
  end

  ind = 1:length(n);
  ind([ind1 ind2]) = [];
  nind = n(ind);

  b(ind) = -4.*alfa./fs .* ( cos((1+alfa).*2.*pi.*fo.*nind) + ...
                   sin((1-alfa).*2.*pi.*fo.*nind) ./ (8.*alfa.*fo.*nind) ) ...
         ./ (pi .* sqrt(1./(2.*fo)) .* ((8.*alfa.*fo.*nind).^2 - 1));

  b = sqrt(2.*fo) .* b;
```

Wavplay

El error más importante fue el que generó la función 'wavplay', en cuanto que supuso mayor dificultad averiguar el origen de dicho error y solucionarlo.

Dicha función es la encargada de transmitir las muestras a través de la tarjeta de sonido, y producía un ***warning en tiempo de compilación***, que nos advertía que nos iba a generar un ***error en tiempo de ejecución***, y efectivamente así se producía.

El ***warning*** decía lo siguiente:

"Warning: File: F:\matlabR12\toolbox\matlab\audio\private\playsnd.m Line: 15 Column: 18 The second output argument from the "unix" function is only available in MEX mode. A run-time error will occur if this code is executed in stand-alone mode".

Se ve claramente que la función que generaba dicho warning era ***'playsnd.m'***, y también se ve que avisaba de un error en tiempo de compilación.

Cuando intentábamos ejecutar el archivo 'z_main.exe', se producía el siguiente ***mensaje de error***, al tiempo de transmitir las muestras, esto es, al ejecutarse *'wavplay'*:

"Audio capabilities are not available on this machine".

Se empezó una línea de investigación en torno a la función *'wavplay'*, y comprobamos que efectivamente esta función llamaba a la función *'playsnd'*.

Intentamos usar otras funciones en lugar de *'wavplay'*, como *'sound'*, pero comprobamos que todas acababan haciendo una llamada a *'playsnd'*.

Finalmente nos dimos cuenta de que *'playsnd.m'* estaba en el directorio:

- *F:\matlabR12\toolbox\matlab\audio\private*

y que en dicho directorio existían tres funciones de nombre *'playsnd'* y extensión distinta:

- 'playsnd.m': archivo propio de Matlab.
- 'playsnd.c': archivo que contiene un código fuente en lenguaje C.
- 'playsnd.dll': archivo externo de Matlab pero ejecutable dicho entorno.

Analizamos el código de *'playsnd.m'* (que era el único accesible, en cuanto que está en código de texto ASCII), y comprobamos que era una función propia para sistema Unix, y que al detectar un sistema distinto a Unix, proporcionaba exactamente un mensaje idéntico al que nos salía en tiempo de ejecución.

Entonces empezamos a investigar en la ayuda del compilador y llegamos a varias conclusiones:

- Si en un mismo directorio existen dos archivos con el mismo nombre y distintas extensiones, 'archivo.m' y 'archivo.dll', el archivo externo de Matlab se ejecuta primero.
  Por esta razón, si ejecutamos en Matlab el archivo *'z_main.m'*, dicha función homónima llamará *a 'wavplay'*, y ésta a su vez llamará a *'playsnd.dll'*, con lo que no se produce ningún mensaje de error, ya que este archivo sí es compatible con el sistema Windows.
- Sin embargo, a la hora de compilar, en la situación anterior, el *'archivo.m'* es compilado primero.
  Entonces, al ejecutarse *'z_main.exe'*, se producirá el error en tiempo de ejecución que se mostraba en nuestro experimento, pues previamente se ha compilado la función *'playsnd.m'*, y es la que aparece en *'z_main.exe'*.
- Además, con la función *'wavrecord'*, que realiza el proceso contrario a *'wavplay'*, no ocurría ningún problema, ya que esta función llamaba a la función *'recsnd.dll'* y no existía ninguna función *'recsnd.m'* en el mismo directorio. Esta última observación daba fuerza a nuestra teoría.

A raíz de estas reflexiones, decidimos ***renombrar temporalmente*** el fichero ***'playsnd.m'***, y lo nombramos de forma diferente, concretamente 'playsnd2.m'. Reiniciamos el ordenador y procedimos a compilar. La compilación se realizó sin ningún problema.

Ya teníamos el fichero ejecutable *'z_main'* en el Bloque Transmisor (que era el único que estaba generando problemas a causa de dicha función *'playsnd.m'*, pues el Bloque Receptor funcionaba perfectamente), y lo habíamos obtenido sin ningún problema en la compilación. Pero a la hora de realizar pruebas, comprobamos con la ayuda de Cool Edit como Bloque Receptor de las muestras, que se transmitía una muestra, pero luego se cortaba la transmisión.

En un principio se creyó que era a causa de transmitir de forma asíncrona, mediante el fichero 'playsnd.m' (introducir parámetro de entrada *'async'* en *'wavplay'*), Decidimos transmitir de forma síncrona, usando en su lugar el fichero 'playsndb.m' (omitir el parámetro 'async'), pero el corte seguía produciéndose.

Aquí mostramos un ejemplo de lo que ocurría, a partir de una grabación realizada con el editor de sonido 'Cool Edit':

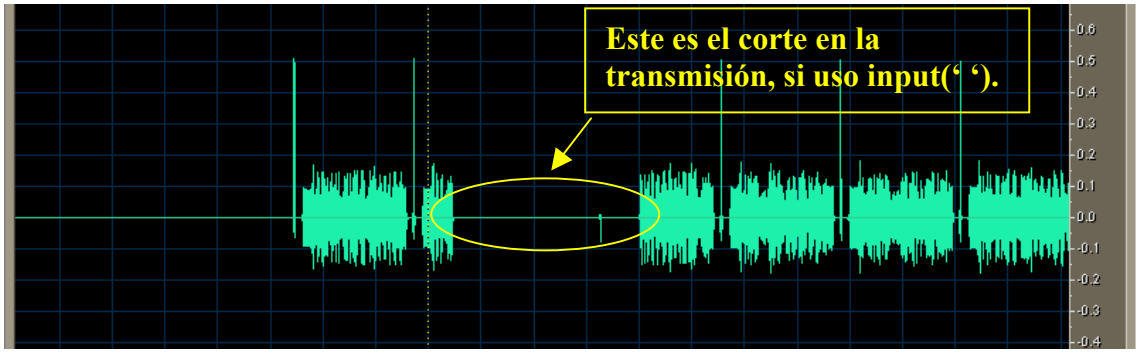


Fig. 112.- Colisión en bus de datos

Haciendo pruebas, descubrimos que la verdadera razón de este corte momentáneo en la transmisión era el uso de la función input(' ') para parar el programa, independientemente de usar transmisión síncrona o asíncrona. La verdadera causa apunta a una colisión en el bus de Datos, durante el tiempo en que se intenta almacenar el valor [] en la variable 'ans'. Si intentáramos usar la función *'getch()'* de lenguaje C, en lugar de *'input'*, probablemente el efecto sería el mismo.

Así la solución que le hemos dado tiene que ver con el hecho de que el corte se produce siempre entre 0.03 sg y 0.05 sg del comienzo de la transmisión. Así, podemos insertar unos 0.07 sg. de silencio, para que dicho corte afecte solo al silencio.

Por último, tan sólo comentar que si usamos la transmisión síncrona, y compilamos sin camuflar el archivo *'playsnd.m'*, si bien no desaparece el *'warning'*, evitamos el error en tiempo de ejecución, ya que transmitimos de manera síncrona y, si analizamos el código del archivo *'wavplay.m'*, comprobaremos que en lugar de llamar a la función *'playsnd'* (que se usa en transmisión asíncrona), se realiza una llamada a la función *'playsndb'*, que está contenida en un archivo homónimo *'playsndb.dll'* situado en el mismo directorio que *'playsnd.dll'* y *'playsnd.m'*.

Clc

La función *'clc'* borra la pantalla. Al traducirla el compilador no genera error, pero lo cierto es que no la implementa en el sentido de que no se borra la pantalla de la consola de MS-DOS, donde ejecutamos *'z_main.exe'*. Esto es un mal menor, y simplemente realizamos un *'cls'* antes de ejecutar dicho archivo.

Acentos

Otro contratiempo es que no traduce bien los caracteres acentuados con sus respectivas tildes en las cadenas de caracteres. La solución adoptada es no usar tildes, para evitar la presencia de caracteres extraños, que da un matiz antiestético a nuestro entorno gráfico en MS-DOS.

Tampoco representa bien la letra *'Ñ'* , en la palabra Diseño, por lo que se ha sustituido por la letra *'N'*, que queda más estético que el carácter extraño, y no deja de ser un problema sin ninguna importancia, pues la legibilidad es perfecta.

Con esto terminamos esta última fase práctica del proyecto. Pasaremos a exponer en el siguiente capítulo una especie de manual, para que se pueda ejecutar nuestra aplicación de una manera eficiente.

Nota: No se ha creído necesario exponer las líneas en código generadas en C por el compilador Compiler 2.1, ya que ocupan una gran extensión, y la traducción es muy tediosa y poco intuitiva en el sentido de que se van enumerando las variables mediante un sufijo numérico de forma creciente. Además la explicación de dichas líneas de código no aportarían nada nuevo, pues realizan las mismas operaciones que las líneas de código de Matlab ya depuradas durante el capítulo III.3.

IV Entornos de funcionamiento

En este capítulo vamos a explicar el ***modo de funcionamiento*** de nuestra aplicación, en los dos entornos en los que ha sido desarrollado, esto es, en ***Matlab*** y en la consola ***MS-DOS de Windows***. Mostraremos algunas gráficas para que se tenga una idea del ***entorno gráfico*** en el que nos vamos a mover durante la aplicación.

IV.1.- ENTORNO DE MATLAB

TRANSMISOR

Para llevar a cabo la ejecución del proyecto en ***Matlab*** es necesario abrir dicha aplicación y situarse en el directorio en el cual se ubiquen los '*archivos.m*' que implementen el ***Bloque Transmisor***. A continuación debemos teclear ***'z_main'*** (sin comillas), para llamar a la función principal y arrancar la aplicación transmisora.

Seguidamente, el programa nos preguntará si queremos introducir los valores de los ***parámetros de diseño*** de nuestro sistema, y que ya conocemos sobradamente, o bien preferimos los que están por defecto incluidos en el código de la función *'z_main'*. En caso de que queramos introducir los datos, éstos aparecerán en el siguiente orden:

- Nombre del Fichero a transmitir (si no se encontrase generaría un error).
- Número de símbolos de la constelación.
- Número de muestras por símbolo.
- Número de bytes del campo DATOS por cada trama.
- Frecuencia de muestreo.
- Número de bits de resolución.
- Factor de Roll-off.

Tras introducir estos datos, las tramas se irán procesando y, a medida que esto ocurre, irá apareciendo en pantalla el número de trama que acaba de ser procesada. Posteriormente aparece un mensaje pidiéndonos que pulsemos *"enter"* para transmitir el fichero. Finalmente aparece un mensaje que indica que el fichero ha sido transmitido.

A modo de ***ejemplo*** ilustrativo, mostraremos un gráfico del entorno de *Matlab* de nuestro proyecto durante la transmisión de un fichero de texto denominado *'prueba.txt'*.

Los parámetros usados, como puede verse en la imagen, son los siguientes:

- Nombre del fichero: 'prueba.txt'.
- M = 256 símbolos.
- N = 10 muestras por símbolo.
- BDT = 70 bytes de datos por trama.
- Fs = 44100 hz.
- NB = 16 bits por muestra.
- alfa = 0.5.

Para mostrar el procesamiento en transmisión hemos tenido que crear una 'megapantalla' a partir de 3 impresiones gráficas, con la ayuda de las herramientas de Word.

La pantalla que resulta es la siguiente:

Fig. 113.- Entorno Matlab. Transmisor

RECEPTOR

Al igual que en el Transmisor, para ejecutar el ***Bloque Receptor*** del proyecto, deberemos abrir otra sesión de *Matlab* y situarnos en el directorio que contenga la implementación de dicho bloque mediante nuestros *'archivos.m'* de Matlab. A continuación hay que volver a teclear ***'z_main'*** para llamar al programa principal del Bloque Receptor, y ejecutar la cadena receptora. Acto seguido, el programa nos volverá a preguntar si queremos introducir o no los datos, o bien dejar los que están contenidos por defecto en la función *'z_main'*. Hay que volver a insistir en que, excepto el nombre de fichero destino, el resto de ***parámetros de diseño*** deben ser ***iguales a*** los del ***Bloque Transmisor***, para una correcta recepción de los datos. Éstos, en caso de que se decida introducirlos manualmente, aparecerán en el siguiente orden:

- Nombre del Fichero destino (si no existiera se crearía).
- Número de símbolos de la constelación.
- Número de muestras por símbolo.
- Número de bytes del campo DATOS por cada trama.
- Frecuencia de muestreo.
- Número de bits de resolución.
- Número de bytes totales del fichero origen.
- Factor de Roll-off.

A continuación, el programa nos pedirá que pulsemos la tecla *"enter"* para empezar a grabar. Ya sabemos que grabaremos 3 segundos más de lo necesario, esto es, $44100 \cdot 3 = 132300$ muestras más de lo necesario (para una Fs = 441000 hz.), y tener así el tiempo sufiente para poder pulsar y transmitir los datos en el Bloque Transmisor.

Luego se nos vuelve a pedir que pulsemos la tecla "enter" para procesar los datos registrados, tras lo cual empiezan a enumerarse, al igual que en la pantalla del Bloque Transmisor, las tramas que van siendo procesadas. Por último, muestra un mensaje para informar de la creación del nuevo fichero, y muestra los resultados, es decir, el número de tramas procesadas, y el número de tramas erróneas.

Para ilustrar todo lo comentado en este bloque mediante un ***ejemplo***, volveremos a usar el fichero *'prueba.txt'* que transmitimos anteriormente en el Bloque Transmisor. Es importante destacar, como ya lo hicimos durante el capítulo III.2, que es indiferente el tipo de fichero que quiera transmitirse, pero ya que expusimos el ejemplo de un fichero de texto en transmisión, vamos a mostrar la parte de recepción de dicho fichero, utilizando, obviamente, los mismos parámetros, es decir:

- Nombre del fichero destino: 'pruebarx.txt'.
- M = 256 símbolos.
- N = 10 muestras por símbolo.
- BDT = 70 bytes de datos por trama.
- Fs = 44100 hz.
- NB = 16 bits por muestra.
- Tamaño del fichero origen: 1000 bytes.
- alfa = 0.5.

La pantalla que resulta es la siguiente:

.....

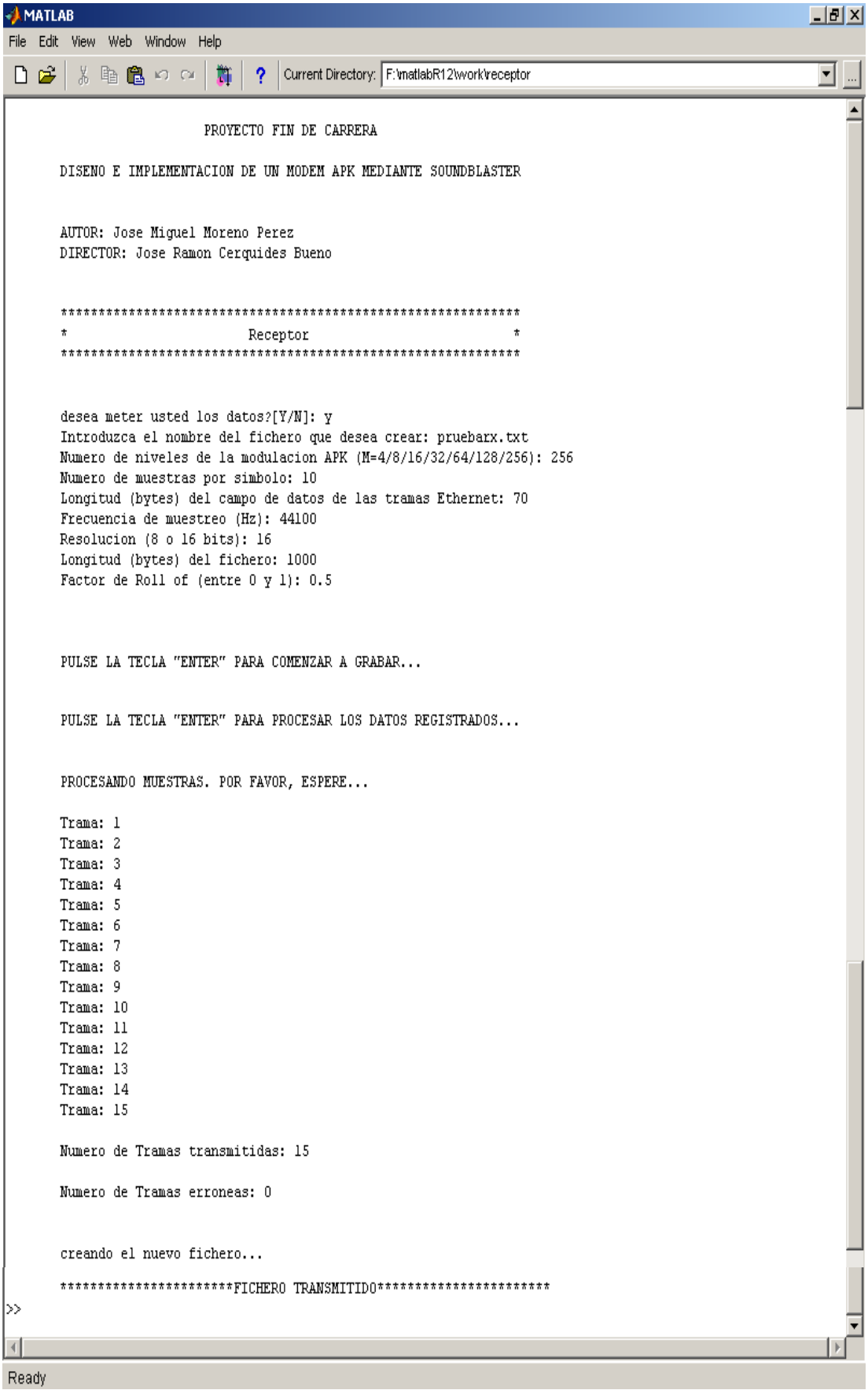


Fig. 114.-Entorno Matlab. Receptor

IV.2.- ENTORNO DE MS-DOS

TRANSMISOR

Para ejecutar nuestra aplicación en la ***consola de MS-DOS de Windows***, para el ***Bloque Transmisor***, tendremos que ubicarnos en dicha consola, dentro del directorio donde se encuentre el fichero ejecutable generado por el compilador, y escribir su nombre, es decir, ***'z_main.exe'***, y el Transmisor comenzará a ejecutarse siguiendo los pasos idénticos a su ejecución en Matlab.

Realmente ***no es necesario abrir la consola de MS-DOS***, sino que directamente hacemos ***doble clic*** sobre el icono del fichero *'z_main.exe'*, y la consola se abrirá automáticamente para que el fichero pueda ser ejecutado.

Para mostrar un ***ejemplo*** ilustrativo del entorno de la *consola de MS-DOS*, vamos a transmitir un fichero gráfico escogido al azar, y que se denomina *'Pica.gif'*, cuya imagen es la siguiente:

Fig.115.- Pica.gif

Vamos a tratar de transmitir la imagen de este lindo ratoncito a través de la tarjeta de sonido. El resultado es el siguiente (los parámetros usados se reflejan en la gráfica):

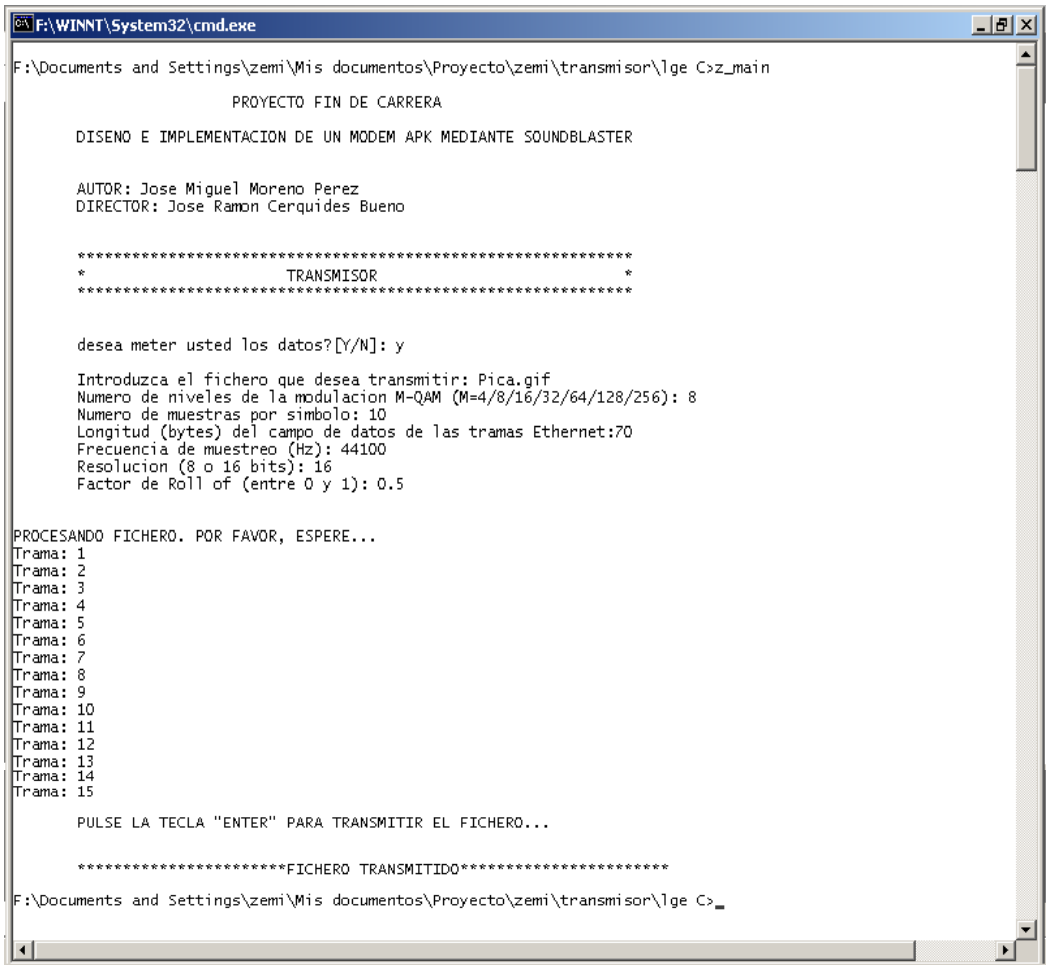


Fig. 116.-Entorno MS-Dos. Transmisor

RECEPTOR

Para llevar a cabo la ejecución en el ***Bloque Receptor***, volvemos a repetir los mismos pasos que en el Bloque Transmisor, de manera que, o bien hacemos ***doble clic*** en el fichero ***'z_main.exe'*** correspondiente al archivo ejecutable del bloque receptor, o bien ***abrimos la consola de MS-DOS***, nos situamos en el directorio de dicho fichero, y tecleamos su nombre.

El resultado será, en ambos casos, la captura y el procesamiento de las muestras. Lógicamente, los pasos que se siguen en este extremo de la comunicación son los mismos que se dieron durante la ejecución del Bloque Receptor en el entorno *Matlab*.

Como ***ejemplo*** gráfico, vamos a recibir el archivo gráfico que transmitimos en el Bloque Transmisor, y vamos a comprobar si efectivamente "el ratón llega sano y salvo a recepción":

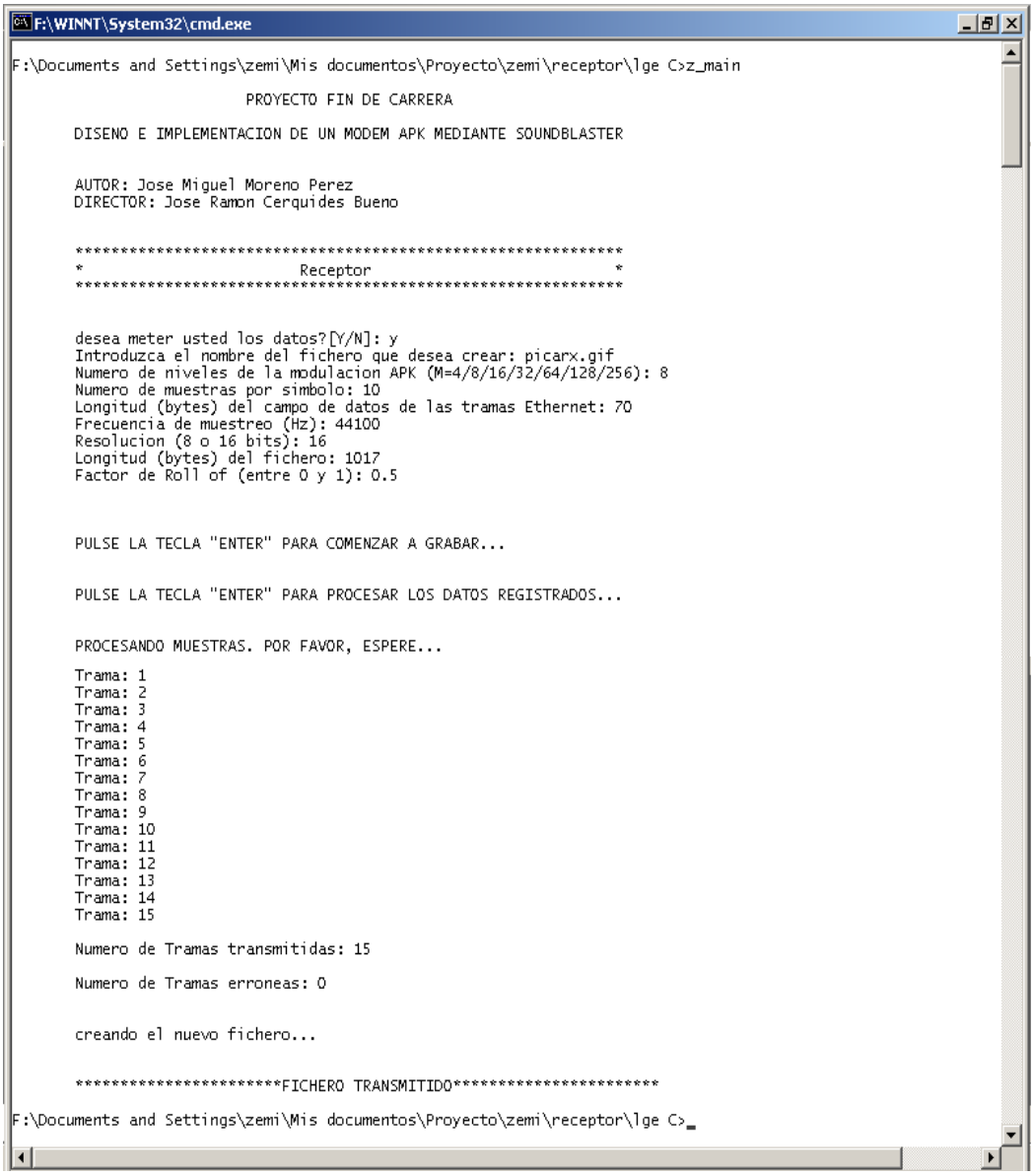


Fig. 117.-Entorno MS-Dos. Receptor

A juzgar por los resultados (ninguna trama errónea), parece ser que el fichero *'Pica.gif'* es idéntico al fichero generado en destino, *'picarx.gif'*. De todos modos, vamos a mostrar un dibujo de dicho fichero *'picarx.gif'*, para verificar los óptimos resultados:

Fig.118.- picarx.gif

Nota:Volvemos a insistir que si queremos transportar el archivo 'z_main.exe', además de dicho fichero hacen falta los ficheros playsnd.dll, y rcsnd.dll (encargados de la transmisión y recepción de las muestras, y que situaremos en el mismo directorio de su ejecutable correspondiente como ya explicamos), y las librerías dinámicas ('run-time libraries'), que se encuentran, como buenamente sabemos, en el archivo instalador de dichas librerías denominado mglinstaller.exe.

V Conclusiones finales y propuestas de mejora

Concluidas las fases de experimentación y elaboración de la memoria, pasemos a exponer las ***conclusiones*** más importantes de este proyecto a modo de resumen final.

Para terminar, escribiremos un apartado donde se propondrán una serie de ***líneas futuras de trabajo***, así como algunas posibles ***propuestas de mejora***.

V.1.- CONCLUSIONES FINALES

Recordemos que nuestro objetivo es crear un ***modulador/demodulador M-APK*** para transmitir datos. La ***fuente*** de estos datos son ***ficheros*** presentes en el disco duro de un PC, o bien en alguna unidad extraíble como pudieran ser un diskette de $3_{1/2}$, ZIPs o CDROM.

Deberemos transmitirlos hacia otro PC cercano (o bien hacia el mismo PC, si la tarjeta es full duplex), teniendo como único medio de transmisión un ***cable de audio*** (mono o estéreo), terminado en dos conectores macho. Estos conectores se insertarán en las clavijas LINE-IN (PC Receptor) y LINE-OUT (PC Transmisor) de las tarjetas de sonido.

Para diseñar nuestro mómem M-APK, necesitamos obtener nuestras variables de diseño del sistema, para ello empezamos haciendo un ***estudio del canal***. Dicho estudio del canal nos proporciona una información muy valiosa, a saber:

- ***Característica BP*** (paso de banda) del canal.
- ***Ancho de banda*** del canal comprendido entre 20 hz. y ***20 khz***. Como consecuencia inmediata, situamos nuestra frecuencia portadora en 11025 hz.
- El canal introduce una ***distorsión en amplitud***, por lo que habrá que ecualizar dicha respuesta impulsiva.
- El análisis del ruido nos indica que el nivel de éste no es importante como para considerarlo a la hora de diseñar el ecualizador de canal. Así, descarto usar filtros Wiener y realizamos un filtro igualador de cero forzado (***Zero-Forcing Equalizer***).
- Existen ***problemas de sincronismo***, derivados del desajuste de frecuencias entre los relojes Transmisor y Receptor. Este es el problema más importante y el que hay que evitar o paliar a toda costa.

Ya en la fase de Diseño de nuestro sistema, y tras una breve introducción a la modulación digital, y a pesar de tener elegida de antemano la técnica de modulación a emplear, o sea, la Modulación Digital M-QAM, se creyó conveniente establecer una ***justificación comparativa*** de nuestra técnica con otros tipos de técnicas de modulación alternativas perfectamente eficaces. Todas estas comparaciones se discutieron desde varios puntos de vista:

- La ***probabilidad de error de símbolo***, P_e.
- El ***ancho de banda*** de la señal.
- ***Energía media*** y ***energía pico*** del sistema.
- ***Tasa de bits*** del sistema, R_b.
- ***Nº de puntos del mensaje*** de la constelación, M.

Pero antes se realizó una ***parametrización*** del sistema, para conocer la relaciones existentes entre las distintas variables de diseño, y empezamos a ***dimensionar*** un poco el sistema, con valores típicos (N=10, D=44100 hz.), para hacernos una idea del rango numérico en el que se moverían cada una de estas variables. De aquí sacamos en claro algunos conceptos:

- El ***ancho de banda*** de nuestra señal depende de nuestra tasa de símbolos, D.
- ***Para aumentar la velocidad de transmisión***, tenemos varias opciones:
  - ***Aumentamos M***, de manera que la tasa de bits, Rb aumenta sin aumentar D. Pero esto aumentar la probabilidad de error del sistema, P_e. ***Para mantener P_e constante***, podemos:
    - ***Aumentar la energía de cada símbolo***, pero nuestra potencia está limitada al rango de nuestro canal y tenemos que evitar dicha saturación.
    - Aumentar M usando ***otra técnica de modulación*** que mantenga los símbolos repartidos en la constelación.
  - ***Aumentar*** la ***frecuencia de muestreo***, ***Fs***, teniendo en cuenta que D también aumenta y nuestro canal tiene una frecuencia de corte considerablemente baja.
- Establecimos un rango dinámico ***el número de muestras por símbolo***, ***N***. Al final obtuvimos un valor práctico para el caso más crítico (factor de Roll-off unidad y M = 256 símbolos) de N=9 muestras por símbolo.
- Además hallamos los ***extremos teóricos de Fs***, para valores típicos de N:
  - ***N=10 → $Fs|_{min}$ = 27563 hz.***
  - ***N=10 → $Fs|_{max}$ = 69750 hz***.
- Decidimos 7 posibles valores de ***M: 4,8,16,32,64,128 y 256 símbolos***.
- Por último, para dichos valores típicos , N=10 y Fs=44100 hz, establecimos el ***rango de velocidades de nuestro sistema***, en función del rango de M:
  - M = 4→ Rb = 8820 bits/sg.
  - M = 256→ Rb= 35280 bits/sg.

Como desenlace de las comparaciones entre la técnica de modulación M-APK, llegamos a las siguientes conclusiones:

- La técnica de modulación ***M-QAM*** es la que puede soportar, en las mismas condiciones de contorno, un ***mayor número*** de símbolos en el sistema, ***M***, concretamente M = 256, debido entre otras cosas a que su constelación tiene dos grados de libertad (amplitud y fase), y permite repartir mejor los puntos de mensaje en el plano.
- Esto sitúa a nuestra modulación como la ***más versátil*** en cuanto a número de ***constelaciones*** diferentes ,tanto por la distribución de sus símbolos como por su número de constelaciones M-arias viables.

- También, y junto con la técnica M-PSK, es la que tiene ***mejor eficiencia espectral***, ***ρ***.
- Por el contrario, al estar contenida la información tanto en la fase como en la amplitud de la portadora, también es la ***más vulnerable*** a las ***no linealidades del canal***, por lo que el Ecualizador va a ser pieza clave en nuestro sistema, para eliminar esas no linealidades.

Tras dicha serie de comparaciones, dividimos nuestro sistema en dos bloques, ***Bloque Transmisor*** y ***Bloque Receptor***, mostramos un dibujo genérico de cada uno de ellos, e hicimos una ruta muy detallada de cada tramo perteneciente en cada bloque, planteando los problemas y soluciones que les fuimos dando, incluyendo además dibujos aclarativos y aplicaciones prácticas cada vez que teníamos oportunidad.

Las observaciones más importantes de esta excursión descriptiva por nuestro sistema iban a ser las siguientes:

- Respecto a las ***constelaciones***:
  - Para asegurar una ***constelación variable***, tendremos ***7 tablas*** de constelación distintas, una para cada valor de M.
  - La tasa binaria más alta se dará para M = 256 símbolos. Sin embargo será el sistema con más tendencia al error en cuanto a la detección de los símbolos. Sin embargo, para M = 4 símbolos ocurre todo lo contrario.
  - Las constelaciones con un número de ***M = 4, 16, 64 ó 256*** símbolos serán de ***forma cuadrada***.
  - Las constelaciones con un número de símbolos ***M = 32 ó 128*** serán constelaciones ***"cuasi-cuadradas"***.
  - La ***energía pico***, o energía del símbolo más alejado del sistema es igual a la ***unidad***, debido al rango digital de la tarjeta de sonido, ubicado en el intervalo [-1,1].
  - El código para codificar los puntos de mensaje de la constelación es el ***código Gray***, para disminuir el número bits erróneos en un símbolo, ante un fallo en la detección de dicho símbolo.
  - Usamos tablas para aumentar la configurabilidad de las ***constelaciones*** y poder ***dotarla de otra fisonomía***. Esto lo conseguimos modificando las coordenadas de los símbolos en dichas las tablas de mapeo.
  - Además, podríamos ***cambiar*** fácilmente el ***tamaño*** de las ***constelaciones***, y la distancia, bien cambiando el valor de la mínima ***distancia intersimbólica***, L, o bien escalando su valor por una constante.

- En relación al ***Inicio y Fin de Trama***:
  - Para devolver el rango dinámico de las muestras recibidas, al rango original donde se movían en transmisión, [-1, 1], nos inventamos unos ***símbolos ficticios*** de Inicio de trama, ***de valor 0.4***, que nos servirán para realizar el ***reajuste de niveles***.

- Además los primeros 4 símbolos del Inicio de Trama, y los 4 símbolos de Fin de trama absorberán los ***efectos del transitorio***, debido al retraso de grupo que introduce el Filtro Transmisor.

- En relación a los ***Filtros Transmisor y Receptor***:

  - Para evitar la ***Interferencia intersimbólica o ISI***, hacemos uso de la familia de ***pulsos raíz de coseno alzado***.
  - El Filtro Transmisor ejerce la labor de ***pulso conformador***, y el Filtro Receptor hace a su vez de ***filtro matcheado***, y de ***filtro Paso de Baja*** para recuperar la envolvente de la señal una vez demodulada.
  - Ambos filtros son pulsos raíz de coseno alzado ,y en cascada (siempre suponiendo un canal ideal en conjunto con el ecualizador), forman un ***pulso en coseno alzado***, que cumple las condiciones de Nyquist (en tiempo y en frecuencia) para evitar la ISI.
  - El factor de ***Roll-off*** será ***modificable***, siendo manipulable de este modo la anchura de nuestra señal.
  - Ambos filtros, al convolucionar con la señal de entrada, generan unas ***colas*** denominadas ***retraso de grupo***, y que hay que tener en cuenta en nuestro diseño. Dicho retraso de grupo será de tres tiempos de símbolos, esto es, de 3·N muestras.

- En lo referente a la ***señal Piloto***:

  - Tiene dos misiones igualmente importantes:

    - Establecer el ***sincronismo de trama***.
    - Actualizar la respuesta impulsiva del canal antes de cada trama, para conseguir así un ***ecualizador dinámico***.

  - Intentamos transmitir las tramas de una en una y nos resultó imposible debido a que no podíamos controlar los ***tiempos de CPU***.
  - Así, con el ecualizador dinámico simulamos que transmitimos las tramas de una en una, por un canal distinto cada vez, y ecualizamos cada una de ellas de manera óptima.

- Respecto los ***Convertidores Analógico-Digital y Digital-Analógico***:

  - Teniendo en cuenta el rango dinámico de Fs para valores típicos de N, y que las frecuencias de muestreo estándares para los dispositivos de audio para PC son ***8000, 11025, 22050 y 44100 hz.***, llegamos a la conclusión de que ***Fs = 44100 hz.*** es la ***máxima*** y la ***única frecuencia viable*** en nuestro sistema de transmisión, ya que inferiormente está acotada por el ***efecto "aliasing"***.
  - Nuestro proyecto utilizará siempre un número de bits de resolución ***NB = 16***, ya que en el formato PCM que usaremos para transmitir, NB dependerá del tipo de datos que se transmitan. De todos modos dejamos explicado cómo transmitir a NB=8 muestras/símbolo, esto es,

haciendo uso del comando 'uint', para cambiar de formato 'double' a 'uint' los datos a transmitir.
- El ***retraso del ecualizador*** habrá que tenerlo en cuenta en recepción.

- Respecto a la ***modulación y demodulación***, decir simplemente que ambas han de hacerse, como es lógico a la misma frecuencia de portadora, que será fija e igual a ***11025 hz.***, por el hecho de usar un submúltiplo de la frecuencia de muestreo, Fs, que será de 44100 hz.

- En relación al ***Ecualizador***:

  - Ante la relación señal a ruido (***SNR***) tan ***elevada*** que presenta el sistema, ***descartamos*** el uso de ***filtros Wiener***.
  - Al tratarse de un ***sistema LTI*** (aunque la falta de sincronismo lo camuflara como sistema LTV), ***rechazamos*** la necesidad de usar ***filtros adaptativos***.
  - Usaremos un Igualador de cero forzado (***zero-forcing equalizer***), que en conjunto con el canal se comportarán como un sistema ideal. El igualador básicamente desplaza las muestras a su posición original, para paliar los efectos del desplazamiento de las muestras a causa de la falta de sincronismo.
  - Construiremos ***Ecualizadores óptimos*** para cada trama, gracias a la señal piloto que precede a cada una de ellas.

- El ***Detector*** que usaremos será el de ***Máxima Verosimilitud***.

- El ***Destino Digital*** será un ***fichero*** que crearemos y guardaremos en el Disco duro o en un diskette. Dicho fichero, caso de no producirse errores, ha de ser idéntico al fichero procedente de la Fuente Digital.

- Respecto a las ***anomalías del sistema***, enumeraremos las más frecuentes e influyentes:

  - La ***distorsión en amplitud*** del canal: la solucionamos con la ayuda del Ecualizador.
  - La ***interferencia intersimbólica*** (***ISI***): Para remediarla hacemos uso de los filtros raíz de coseno alzado principalmente. El ecualizador también ayuda.
  - La ***ausencia de total sincronismo***: Es el problema más grave.

    - Se solucionó ***acortando la longitud de las tramas***, para conseguir que el error de desplazamiento no superara el ***1%.***
    - De este modo solo podemos mandar ***882 muestras*** totales por trama.
    - Así, para un caso crítico de probabilidad de error (M=256) y un valor típico de N=10 muestras / símbolo, establecemos el valor del número de bytes de datos por cada trama, ***BDT***, que será de ***60 a 70 bytes*** aproximadamente.

- En relación a la ***estructuración en tramas*** de nuestra información:
  - Ésta surge como consecuencia de dos razones fundamentales:
    - ***Facilitar la computación*** de la CPU.
    - Para ***controlar los trozos de información*** en que se divide el fichero origen, como solución a la ausencia de sincronismo.
  - El formato de trama en el que nos inspiramos es el denominado ***Ethernet***, es decir, el ***IEEE 802.3***, en el que realizamos los siguientes cambios:
    - Eliminamos los campos Inicio, Preámbulo, Relleno, Dirección Origen y Dirección Destino, porque no los necesitábamos.
    - El CRC-32 lo cambiamos por el ***CRC-16*** por los siguientes motivos:
      - CRC-16 es ya bastante un código bastante potente.
      - El número de bytes del campo Datos es muy bajo (60 a 70 bytes < 1500 bytes permitidos en campo DATOS).
      - El CRC-16 introduce menor número de cálculos.
- Así, cada una de las ***tramas*** adoptará el siguiente ***aspecto***:

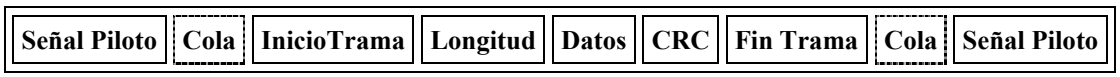


Fig. 119.- Campos de la trama

Respecto a la ***compilación***, usamos una herramienta de Matlab, denominada ***Compiler 2.1***, que nos ahorrará toda la labor de programación en C y de estudio de las APIs de Windows. Lo más importante a destacar en esta fase son los siguientes puntos:

- Para invocar al compilador utilizamos el ***comando mcc***.
- Para crear los ejecutables 'C stand alone' para los Bloques Transmisor y Receptor, empleamos la ***macro –m***, que equivale a todas las opciones necesarias para el comando mcc.
- Para la ***compilación*** era necesario:
  - El entorno de Matlab.
  - Compiladores de Matlab y de C.
  - La librería MATLAB C/C++ MATH library.
- Por otro lado, para la ***portabilidad del código***, fue necesario:
  - Los ficheros *'z_main.exe'* de ambos bloque Transmisor y Receptor.
  - Los archivos *'playsnd.dll'* y *'rcsnd.dll'* para la transmisión y recepción de las muestras a través de la tarjeta de sonido.
  - Todas las librerías dinámicas MATLAB Math run-time, contenidas en el archivo '*mglinstaller.exe'*.

- Los ***problemas*** que tuvimos en la ***compilación*** se debieron principalmente a las ***limitaciones del compilador***, que no traducía correctamente algunas funciones, o bien no las admitía:

  - Para evitar usar la función 'eval.m', cambiamos la función 'z_rcosflt.m' de Matlab por nuestra función 'z_filtro.m'.
  - Sustituimos la función 'pause.m' por la función 'input.m'.
  - Tuvimos que transmitir silencio antes de la señal para evitar colisiones en el bus de datos.
  - Otros contratiempos sin importancia fueron que no se recogía la función 'clc.m' de Matlab, ni las tildes en las cadenas de caracteres.

En lo referente al capítulo de los ***entornos de funcionamiento*** es interesante comentar dos cosas únicamente:

- Reiterar que se necesitan ***dos sesiones*** para ejecutar nuestro proyecto, una para el ***Bloque transmisor***, y otra para el ***bloque Receptor***.
- El entorno usado en Windows es la ***consola de MS-DOS***.

Tras la observación empírica del proyecto culminado en distintos PC's, llegamos a la conclusión de que es necesario disponer de ***PC's relativamente potentes***, para que pueda soportar de una manera eficiente todo el procesamiento de la señal en ambos Bloques Transmisor y Receptor. Esto es así porque ***nuestro módem*** es eminentemente un ***producto Software***, y no disponemos de un dispositivo Hardware que nos facilite la alguna tarea. Por este motivo, hay que tener paciencia en PC's con micros bastante lentos, o poca memoria RAM, donde los Bloques de cálculo del CRC-16, detección de trama errónea y extracción del igualador óptimo se hacen notar bastante, porque ralentizan mucho la velocidad de proceso de cada trama.Esto supone un gran contraste frente a las ***tarjetas de Ethernet***, por ejemplo, donde todos estos bloques se implementan ***vía Hardware***, de la misma forma que en un módem para Internet.

Se trata de un ***proyecto*** que, tanto por sus conocidas ***limitaciones de ancho de banda***, como por tratarse de un ***sistema plesiócrono***, está enfocado a ***ficheros de poca extensión*** (sin contar además con las ***limitaciones en velocidad de procesamiento*** a las que aludimos en el párrafo anterior). Ya hemos apuntado que la máxima velocidad de transmisión la podemos alcanzas sin errores, para N=10 muestras y un número de símbolos M=256, es de Rb= 35280 bits/sg., ya que para N=9 aparecen algunas tramas erróneas esporádicamente.

V.2.- LÍNEAS DE MEJORA

La ***propuesta de mejora más importante*** que planteamos en esta sección está relacionada con la ***rapidez del sistema***. Se trata de aprovechar de transmitir las tramas una a una. Con ello solapamos los tiempos de procesamiento de trama en transmisión y en recepción, incrementando notablemente la velocidad de ejecución de la transmisión, aspecto bastante importante en un sistema cuyo escaso ancho de banda apenas permite una tasa de unos cuantos Kbaudios.

Como ya comentamos durante la fase de Diseño, ***intentamos sincronizar ambos Bloques Transmisor y Receptor***. Pero fue un ***rotundo fracaso***, por varias razones:

- Ambos bloques estaban en ***sesiones de Matlab diferentes***, y teníamos que jugar con los tiempos de CPU de sistemas para su sincronización, utilizando para ello funciones de Matlab que controlaran dichos tiempos.
- Además, el procesamiento en la cadena receptora era mucho más lento que en la transmisora, puesto que teníamos que incluir segundos de silencio en cada trama recibida para evitar la pérdida de muestras, y teníamos que controlar bien los tiempos de transmisión / recepción, para que no se perdieran muestras.
- Aún así, no contábamos con la ***labor de multiprocesamiento de la CPU***, que no atendía instantáneamente nuestras peticiones de transmisión y recepción de las tramas, por lo que algunas se perdían.
- Así, para evitar pérdidas de muestras y transitorios indeseados, los ***tiempos de espera*** entre los procesamientos y las transmisiones se nos antojaban ***bastante grandes***, lo cual provocó que esta técnica de transmisión en tiempo real fuera mucho más lenta que la técnica que finalmente usamos.

De todos modos dejamos esta opción en el aire, por si alguien se anima a ***programar los drivers de sonido en tiempo real***, esto es, a permitir el acceso a los distintos buffers de la tarjeta de sonido receptora de modo que no hubiera que esperar a que terminara la transmisión del fichero para comenzar a procesarlo en recepción.

Otra forma de mejorar el proyecto para que éste gane en rapidez es implementar un ***sistema de detección y corrección de errores***. De esta forma, podríamos aumentar un poco más la longitud de las tramas, aun a riesgo de cometer algún error en alguna que otra trama, ya que ésta sería corregida en recepción, gracias a dicho campo.

Una tercera propuesta podría ir dirigida a ***mejorar la interfaz gráfica*** de la aplicación, esto es, aprovechar las herramientas de Matlab de interactuación con el programa ***Visual Studio C++ 6.0***, para poder dotar a nuestra aplicación, por ejemplo, de:

- Ventanas de diálogo: para los Bloques Transmisor y Receptor.
- Etiquetas de texto: para enumerar las distintas opciones de diseño.
- Cuadros de texto: para escribir el nombre de los ficheros origen y destino.
- Grupos de texto: para agrupar las distintas opciones en un menú vistoso.
- Botones de opción: para mostrar los distintos grupos de opciones para cada variable de diseño.
- Barra de progreso: para mostrar el porcentaje de tramas procesadas.

Aunque no sea una propuesta de mejora en el estricto sentido de la ingeniería, sin embargo supondría una interfaz más agradable para el usuario.

Por último nos gustaría indicar una vía de ***ampliación de nuestro proyecto***, integrando nuestra aplicación dentro de un ***sistema de funcionamiento en red***. Es decir, varios PC's interconectados entre sí, en cualquier topología de red (en estrella, en anillo, interconexión total...), ***unidos vía cable de audio***, y usando nuestro propio proyecto como base de la transmisión de los datos entre PC's. La ampliación de nuestro proyecto de comunicación punto a punto hacia una red de interconexión vía audio se reflejaría, entre otros aspectos, en los siguientes puntos:

- Tendríamos que implementar algún ***protocolo de red*** (bien orientado a conexión o bien servicio de Datagrama) para ,por ejemplo, evitar colisiones, encaminar las tramas, o para adquirir el testigo (caso de que se usara ese sistema), etc...
- Además, también sería necesario identificar cada uno de los PC's de alguna manera.
- Nos resultaría vital la recuperación de los ***campos Dirección de Origen*** y ***Dirección de Destino*** de la Trama Ethernet, desechados en un principio por su inutilidad, y que en este nuevo proyecto tendrían un papel decisivo para el enrutamiento de los datos. Incluso podríamos codificar estos campos para una transmisión en multidifusión.
- Podríamos además emplear alguna ***codificación de fuente***, para compactar y encriptar los datos, lo cual serviría como ejercicio práctico muy instructivos para afianzar conocimientos teóricos.

Aquí concluye la exposición del proyecto. A quien pudiera interesarle continuar con lo desarrollado hasta aquí y quisiera consultar algún tipo de dudas, simplemente comentar que estaría encantado en colaborar en ese nuevo proyecto.

José Miguel Moreno Pérez
Dirección de correo electrónico: jmmoreno@linuxmail.org

BIBLIOGRAFÍA

[†1] Haykin, Simon
Digital communications
New York [etc.] : John Wiley and Sons, 1988

[†2] Haykin, Simon
Señales y sistemas
México : Limusa Wiley, 2001

[†3] Autor Lang, Serge
Cálculo
Argentina [etc.] Addison-Wesley Iberoamericana, 1990

[†4] Proakis, John G.
Digital communications
Boston [etc.] : McGraw-Hill, 2001

[†5] Juan Francisco Izquierdo León
Tx de datos mediante sound Blaster
Universidad Sevilla (Proyecto Fin Carrera)

[†6] The MathWorks
Manual de Ayuda del Entorno Matlab

[†7] www.conozcasuhardware.com
La Tarjeta de sonido SoundBlaster
Página de internet y similares

[†8] us.creative.com
Tx de datos mediante sound Blaster
Página oficial de Creative Labs

[†9] Sanjurjo Navarro, Rafael
Teoría de circuitos eléctricos
Madrid [etc.] : McGraw-Hill, 1997

[†10] García Teodoro, Pedro
Transmisión de datos y redes de computadores
Madrid : Pearson Educación, 2003