Trabajo Fin de Máster
Máster Universitario en Ingeniería de Telecomunicación

Prototipo de emisor-receptor para equipo de radio definida por software

Autor: Pedro Gutiérrez Lora
Tutor: Rubén Martín Clemente

Dpto. Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2019

Trabajo Fin de Máster
Máster Universitario en Ingeniería de Telecomunicación

Prototipo de emisor-receptor para equipo de radio definida por software

Autor:
Pedro Gutiérrez Lora

Tutor:
Rubén Martín Clemente
Profesor titular

Dpto. de Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2019

Trabajo Fin de Máster: Prototipo de emisor-receptor para equipo de radio definida por software

Autor: Pedro Gutiérrez Lora

Tutor: Rubén Martín Clemente

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Sevilla, 2019

A mi familia

A mis maestros

Resumen

Con la intención de acabar con las ataduras que imponían los componentes *hardware* a los sistemas de radio, surgió la radio definida por *software*, donde todo el procesado de la señal se realiza digitalmente.

El objetivo de este trabajo pasa por estudiar los bloques básicos que componen a un dispositivo transmisor y receptor en un sistema de comunicación, proponiendo a su vez posibles implementaciones que sentarían las bases de un sistema de radio definida por *software*. Estas implementaciones se realizarán empleando el entorno de simulación MATLAB e irán acompañadas de ejemplos que muestren su funcionamiento.

Abstract

Hardware components imposed so many restrictions to radio systems that software-defined radio was born to overcome such difficulties, allowing to implement new telecommunication standards in a more flexible way.

The purpose of this project is to study the basic components of a transmitter and receiver device in a communication system and propose several ways to implement them. These implementations will be coded using MATLAB accompanied by some practical examples that will show how they work.

Índice

Resumen **ix**

Abstract **xi**

Índice **xiii**

Índice de Tablas **xv**

Índice de Figuras **xvii**

Índice de Códigos **xix**

Acrónimos **xxi**

1 Introducción **1**

2 Radio definida por software **3**

2.1. Definición y estructura básica *3*

2.2. Evolución a lo largo de la historia *4*

2.3. Dispositivos SDR comerciales *5*

2.1.1 RTL-SDR 6

2.1.2 HackRF One 7

2.1.3 USRP 7

3 Diseño del transmisor **9**

3.1. Mapper: conversor de bits a símbolos *9*

3.1.1 Phase-shift keying (PSK) 9

3.2. Conformador de pulsos *12*

3.2.1 Coseno alzado 14

3.3. Interpolación en múltiples etapas (opcional) *16*

3.3.1 Filtros cascaded integrator-comb (CIC) 17

3.4. Subida a frecuencia intermedia *19*

4 Implementación del transmisor **21**

4.1. Implementación del mapper *21*

4.1.1 Phase-shift keying (PSK) 23

4.2. Conformador de pulsos *25*

4.1.2 Coseno alzado 27

4.3. Interpolación en múltiples etapas *30*

4.1.3 Filtros cascaded integrator-comb (CIC) 31

4.4. Subida en frecuencia *32*

5 Consideraciones sobre el canal **35**

5.1. Ruido *35*

5.2. Propagación multitrayecto *36*

6 Diseño del receptor **41**

6.1. Sincronización *41*

6.1.1 Lazo de seguimiento de fase (PLL) 41

6.1.2 Sincronización de portadora 45

6.1.3 Sincronización de símbolo o tiempo 48

6.2. Diezmado en múltiples etapas *50*
6.3. Filtro adaptado *51*
6.4. Ecualización *52*
6.2.1 Ecualizador lineal *Zero-forcing* (ZF) 52
6.2.2 Ecualizador LMS adaptativo 54
6.5. Decisión *55*

7 Implementación del receptor **57**
7.1. PLL *57*
7.2. Sincronización de portadora *63*
7.3. Diezmado en múltiples etapas *67*
7.4. Diagrama de ojos *68*
7.5. Ecualización *72*
7.1.1 Zero-forcing 72
7.1.2 LMS adaptativo 74
7.6. Decisión *75*

8 Conclusiones y propuestas de mejora **77**

Anexo: Código de las implementaciones **79**

Referencias **91**

Índice de Tablas

Tabla 2.1 Comparativa entre diversos SDR comerciales. 6

Índice de Figuras

Figura 2.1 Estructura general de una radio definida por software. 3
Figura 2.2 Ejemplo de gráfico de flujo del software GNU Radio [5]. 5
Figura 2.3 Estructura del RTL-SDR Blog R820T2 v3. 6
Figura 2.4 Periférico HackRF One. 7
Figura 2.5 USRP Bus Series, modelo B200 mini. 7
Figura 3.1 Constelación BPSK con energía por bit unidad. 10
Figura 3.2 Frontera de decisión en BPSK para un canal AWGN. 11
Figura 3.3 Representación gráfica de la BER para BPSK, QPSK, 8-PSK, 16-PSK y 32-PSK. 12
Figura 3.4 Estructura de un filtro FIR. 13
Figura 3.5 Esquema de filtro polifase para interpolación. 14
Figura 3.6 (*a*) Pulso ideal de Nyquist en el tiempo. (*b*) Espectro del pulso de Nyquist. 15
Figura 3.7 Espectro del coseno alzado según diferentes factores de *roll-off*. 16
Figura 3.8 Parámetros típicos de diseño de filtros FIR paso bajo. 17
Figura 3.9 (*a*) Filtro CIC interpolador de N etapas. (*b*) Filtro CIC diezmador de N etapas. 18
Figura 3.10 Respuesta en frecuencia de un filtro CIC con $R = 6$, $M = 1$ y N variable. 19
Figura 4.1 Reestructurado de los datos para una modulación 8-PSK. 21
Figura 4.2 Ejemplo gráfico del funcionamiento del *mapper* para una 8-PSK. 22
Figura 4.3 BER teórica para una 8-PSK empleando el **Código 4.6**. 24
Figura 4.4 (*a*) Secuencia a interpolar. (*b*) Filtro de media móvil. (*c*) Resultado de la interpolación. 26
Figura 4.5 (*a*) Respuesta al impulso del filtro RRC con $\beta = 0.60$. (*b*) Respuesta en frecuencia. 29
Figura 4.6 Resultado de la conformación de pulsos usando un filtro RRC. 30
Figura 4.7 (*a*) Primera etapa de interpolación $L1 = 4$. (*b*) Segunda etapa de interpolación $L2 = 6$. 31
Figura 4.8 Portadora a frecuencia $fc = 8\ MHz$ muestreada a $fs = 96\ MHz$. 33
Figura 4.9 (*a*) Transformada de Fourier en banda base. (*b*) Datos en paso banda con $fc = 8\ MHz$. 33
Figura 5.1 Respuesta al impulso del canal multitrayecto de ejemplo. 39
Figura 6.1 (*a*) Diagrama de un PLL analógico. (*b*) Diagrama de un PLL digital. 41
Figura 6.2 Filtro activo con amplificador. 43
Figura 6.3 Estructura de un PLL digital de primer orden. 44
Figura 6.4 Estructura de un PLL digital de segundo orden. 45
Figura 6.5 Diagrama de bloques para la recuperación de portadora mediante la técnica *squaring loop*. 46
Figura 6.6 Diagrama de bloques para la recuperación de portadora mediante el lazo de Costas. 47

Figura 6.7 (*a*) Pulso rectangular. (*b*) Salida del filtro adaptado. 49
Figura 6.8 Diagrama de bloques de la sincronización Early-Late Gate. 50
Figura 6.9 Esquema de filtro polifase para diezmado. 51
Figura 6.10 Esquema de comunicaciones con ecualizador. 53
Figura 6.11 Estructura típica de un ecualizador adaptativo. 54
Figura 6.12 Punto de la constelación más cercano al símbolo recibido. 55
Figura 7.1 Estructura del *loop fitler* de un PLL de segundo orden. 58
Figura 7.2 Error en PLL de primer orden para misma frecuencia y fase que la referencia. 59
Figura 7.3 Error en PLL de primer orden con filtro paso bajo intermedio. 61
Figura 7.4 Representación del seno de la fase de salida del VCO. 62
Figura 7.5 Error en PLL de primer orden para una frecuencia $100\ kHz$ superior a la referencia. 62
Figura 7.6 Error en PLL de segundo orden. 63
Figura 7.7 Respuesta en frecuencia del filtro paso banda empelado en el *squaring loop*. 64
Figura 7.8 Respuesta en frecuencia del filtro paso banda empelado en el *squaring loop*. 65
Figura 7.9 Comparativa entre el mensaje original y el demodulado tras sincronización de portadora. 66
Figura 7.10 Diagrama de ojo y sus principales medidas. 68
Figura 7.11 Diagrama de ojos en el transmisor. 70
Figura 7.12 Diagrama de ojos en el receptor para $Eb/N0 = 0\ dB, 7\ dB, 14\ dB$ y $21\ dB$. 71
Figura 7.13 Diagrama de ojos en el receptor para $Eb/N0 = 14\ dB$ y diferentes canales multitrayecto. 72
Figura 7.14 (*a*) Respuesta al impulso del canal. (*b*) Respuesta al impulso total. 74

Índice de Códigos

Código 4.1 Convertir bits a símbolos. 21
Código 4.2 Superclase `constelacion`. 22
Código 4.3 Definición y propiedades de la clase `constel_psk`. 23
Código 4.4 Creación de la constelación M-PSK. 23
Código 4.5 Función `numAbin`**.** 24
Código 4.6 Obtención de la BER teórica para M-PSK. 24
Código 4.7 Funcionamiento básisco de `conformador_pulso.m`. 25
Código 4.8 Interpolación con filtrado polifase. 25
Código 4.9 Ejemplo de interpolación usando filtrado polifase. 26
Código 4.10 Definición y propiedades de la clase `coseno_alzado`. 27
Código 4.11 Ejemplo de uso del conformador de pulsos. 28
Código 4.12 Ejemplo de interpolación en múltiples etapas. 31
Código 4.13 Código de la función `gen_coef_cic.m`. 32
Código 4.14 Generación de la portadora a frecuencia fc. 32
Código 5.1 Generación de ruido AWGN. 36
Código 5.2 Generación de canal multitrayecto. 38
Código 5.3 Ejemplo de generación de canal multitrayecto. 38
Código 7.1 PLL configurable con ganancia proporcional e integral. 57
Código 7.2 Ejemplo PLL de primer orden. 58
Código 7.3 Mejora del PLL configurable con ganancia proporcional e integral. 60
Código 7.4 Ejemplo de PLL de primer orden con filtro paso bajo intermedio. 60
Código 7.5 Generación de la señal de refencia para sincronización por *squaring loop*. 64
Código 7.6 Generación del filtro paso banda en el *squaring loop*. 64
Código 7.7 Inicialización del PLL en el *squaring loop*. 65
Código 7.8 Reducción de la fase de salida a la mitad y compensación del desfase. 65
Código 7.9 Demodulación tras la sincronización de portadora. 66
Código 7.10 Diezmado con filtrado polifase. 67
Código 7.11 Ejemplo de diezmado en múltiples etapas. 68
Código 7.12 Diagrama de ojos en el transmisor. 69
Código 7.13 Generación de ruido y adición a los datos. 70
Código 7.14 Generación de coeficientes del filtro ecualizador ZF. 72

Código 7.15 Obtención de filtro ZF para un canal dado. 73
Código 7.16 Algoritmo LMS. 75
Código 7.17 Método para obtener la distancia a todos los puntos de la constelación. 75
Código 7.18 Método para la toma de decisión del símbolo recibido. 75

Acrónimos

FFT	*Fast Fourier transform* (transformada rápida de Fourier)
GGP	*General purpose processor* (procesador de propósito general)
RF	Radiofrecuencia
IF	*Intermediate frequency* (frecuencia intermedia)
HW	*Hardware*
CAD	Convertidor analógico-digital
CDA	Convertidor digital-analógico
BER	*Bit error rate* (tasa de error de bit)
AWGN	*Additive White Gaussian Noise* (ruido gausiano blanco aditivo)
FIR	*Finite impulse response* (respuesta al impulso finita)
ISI	*Intersymbol interference* (interferencia intersímbolo)
SNR	*Signal-to-noise ratio* (relación señal a ruido)
PLL	*Phase-locked loop* (lazo de seguimiento de fase)
DSB/SC	*Double-sideband/suppressed-carrier* (doble banda lateral/portadora suprimida)
RRC	*Root-raised cosine filter* (filtro raíz de coseno alzado)
DSP	*Digital signal processor* (procesador digital de señal)
FPGA	*Field-programmable gate array*

1 INTRODUCCIÓN

Las personas necesitan comunicarse más que nunca. El auge de las telecomunicaciones ha permitido que sus usuarios se comuniquen de diferentes formas y usando distintos medios: compartir un fichero por correo electrónico, llamadas de voz, videollamadas… lo cual consiguen haciendo uso de un ordenador, un teléfono, un *smart watch* que recibe la información de un dispositivo móvil, y un largo etcétera.

Sin embargo, en la variedad radica el problema. Y es que esta diversidad en cuanto a sistemas de comunicación trae consigo numerosos estándares y protocolos de radio que están basados en *hardware* específico, lo que los convierte en dispositivos poco flexibles a cambios. Por esta razón resulta imposible modificar un dispositivo que funciona en base a un estándar concreto (ya sea para añadir soporte a nuevos protocolos o cambiarlo por otro), pues está limitado por los componentes *hardware* que lo conforman.

La **radio definida por software** busca resolver este problema encargándose de realizar todo el procesado digital de señal mediante *software*, en lugar de emplear para ello circuitos integrados dedicados en *hardware*. Esta flexibilidad permitirá que, usando el mismo *hardware*, se puedan crear distintos tipos de sistemas de radio que funcionen con estándares de comunicación diferentes.

El objetivo de este proyecto es estudiar la base teórica de un sistema de comunicación (transmisor y receptor), incluyendo también el modelado del canal de comunicación por el que se transmite la señal. También se llevará a cabo su creación desde cero usando MATLAB, de manera que puede servir como modelo de alto nivel en futuros trabajos donde se desarrolle el sistema de radio definida por *software* en dispositivos reales.

La forma en que se ha organizado el trabajo es:

- Repaso de la historia de la radio definida por *software*. Cómo surge y qué dispositivos existen en el mercado son las principales preguntas que se pretende responder en este capítulo.
- Estudio teórico y matemático de los principales bloques básicos de un transmisor en un sistema de radio definida por *software*: *mapper*, modulaciones digitales, conformación de pulsos, interpolación en varias etapas (usando filtro polifase) y subida en frecuencia.
- Propuesta de implementación de cada uno de los bloques del transmisor en MATLAB.
- Investigación de los principales fenómenos que afectan a la señal al transmitirse por el canal: ruido y propagación multitrayecto. Se proponen formas de simular el ruido y de modelar el canal.
- Al igual que con el transmisor, estudio de los bloques básicos del receptor, incluyendo la sincronización y ecualización que no están presentes en el transmisor.
- Implementación de los bloques del receptor, además de introducir una herramienta de gran utilidad en el análisis de enlaces de comunicación, el diagrama de ojos.

2 RADIO DEFINIDA POR SOFTWARE

2.1. Definición y estructura básica

Pero ¿qué es la radio definida por *software* y cómo surge? Para responder a la primera pregunta, se unieron el ***Software Defined Radio Forum*** (renombrado desde 2009 como el *Wireless Innovation Forum*) junto con el grupo ***IEEE P1900.1*** con el fin de desarrollar un documento [1], que fue aprobado en noviembre del 2007, en el que se recogían las principales definiciones relacionadas con la radio definida por software (más conocida en la literatura como *software-defined radio* o SDR) y la radio cognitiva.

Atendiendo a la definición que hacen sobre el tema de este trabajo: "*Radio* in which some or all of the *physical layer* functions are *Software Defined*".

Es decir, la radio definida por *software* es aquella en la que algunas o todas las funciones de la capa física se encuentran definidas por *software*. Aquí aparecen tres conceptos clave: "radio", "capa física" y "definida por *software*". Estos términos también se encuentran definidos por el SDR Forum en el mismo documento y su comprensión ayudará a tener una mejor idea sobre qué es la SDR.

De forma resumida, se considera "radio" a aquella tecnología usada para transmitir o recibir información inalámbricamente mediante radiación electromagnética. Para llevar a cabo esa transmisión o recepción, se hace uso de protocolos que pueden organizarse siguiendo un modelo de 7 capas propuesto por la OSI, siendo la capa inferior la "capa física", donde tiene lugar todo el procesado de señales RF, IF o banda base. Estas funciones de la capa física serán implementadas mediante procesado *software* en una SDR.

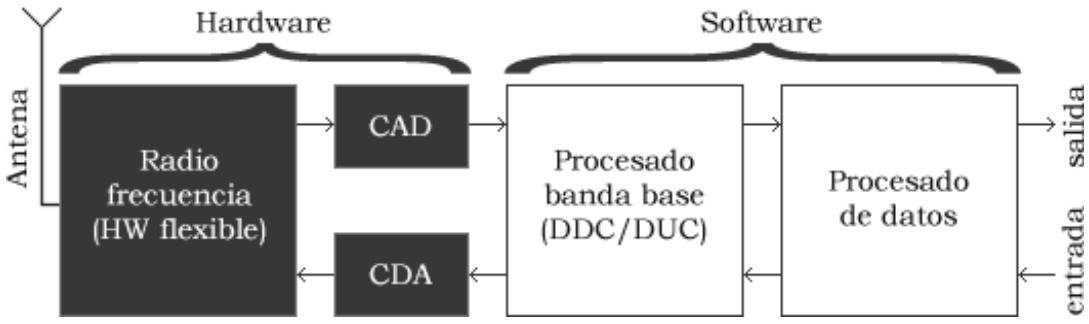


Figura 2.1 Estructura general de una radio definida por *software*.

Queda, por tanto, bien delimitado qué es y qué funciones cumple una SDR. En el esquema de la **Figura 2.1** es posible apreciar cómo sería la estructura básica de un dispositivo con soporte para radio definida por *software*. Las partes que lo componen son:

- **Antena**: transmite al medio o recibe del medio la señal RF.
- ***Hardware* de radiofrecuencia flexible**: en el caso de actuar como receptor, convertirá la señal RF recibida a una frecuencia intermedia; si actúa como transmisor, entonces subirá la señal en frecuencia y pasará por un amplificador de potencia. Es un elemento analógico y debe ser flexible para poder alojar cualquier tipo de procesado por *software*.

- **Convertidor analógico-digital/digital-analógico**: funcionando como receptor, se llevará a cabo la conversión digital-analógico; para el transmisor se convertirán los datos de analógico a digital. Aquí se podría decir que termina la parte típicamente de *hardware* y comienza todo el procesado *software* con el siguiente bloque.
- **Procesado banda base** (*Digital up/down converter*): en el receptor se produce un proceso de *digital down conversion* (DDC), en el que la señal de entrada (a frecuencia intermedia) se baja en frecuencia (por lo general, a banda base) a una tasa de muestreo menor; el transmisor realiza el proceso contrario, denominado *digital up conversion* (DUC), donde la señal en banda base se sube en frecuencia hasta frecuencia intermedia.
- **Procesado**: aquí ocurre el resto de procesado de la señal, siguiendo una serie de algoritmos que son ejecutados por dispositivos tales como FGPA, DSP o GPP (un ordenador, por ejemplo). Acciones típicas llevadas a cabo por el receptor serían la demodulación, decodificación y decisión; en el caso del transmisor haría las veces de modulador y codificador.

2.2. Evolución a lo largo de la historia

Y aunque el concepto de SDR se definió formalmente a comienzos del siglo XXI, lo cierto es que su concepción y desarrollo comenzaron en una época anterior. Ya en 1970, como fruto de los esfuerzos combinados de grupos de investigación privados y de organizaciones gubernamentales de los Estados Unidos, aparecían los primeros intentos de herramientas de radio cuyas operaciones estaban definidas por *software*.

No fue hasta 1991 cuando apareció **SpeakEASY Phase-1** [2], el primer programa militar que solicitaba la implementación de los componentes de la capa física mediante *software*. Estuvo desarrollado por la Agencia de Proyectos de Investigación Avanzados de Defensa (DARPA) y perseguía unificar en un único dispositivo radio la coexistencia de diez protocolos militares de radio diferentes, operando en cualquier frecuencia entre los 2 MHz y 2 GHz. Pero la realidad es que no lograron cumplir su objetivo [3]. Un estudio llevado a cabo determinó que no era posible implementar dicho rango de operación en un solo canal RF de banda ancha, así que se tuvo que dividir en tres secciones: una banda baja entre 2 y 30 MHz, una banda media entre 30 y 400 MHz, y una banda alta entre 400 MHz y 2 GHz. Solo se consiguió desarrollar el sistema para la banda media. Por otro lado, el dispositivo resultó no contar con la suficiente capacidad para procesado FFT. Además, la interfaz de uso, así como el *software* del módem y entorno de desarrollo eran complicados de manejar.

Posteriormente, Joseph Mitola III [4] publica un artículo en la Telesystems Conference en 1992 en el que concibe una **radio *software* ideal** que es capaz de operar con cualquier servicio de comunicaciones, ejecutando para ello diferentes algoritmos que permiten una reconfiguración al formato de la señal que emplee dicho servicio. Mitola terminaría convirtiéndose en el "padre de la radio por *software*" y en uno de los fundadores del SDR Forum.

Con el fin de mejorar las carencias del proyecto fallido de DARPA, se inicia en 1995 el **SpeakEASY Phase-2**. Esta vez su alcance es algo más amplio, buscando desarrollar una arquitectura abierta, modular y reprogramable para un sistema de radio completo, desde las entradas y salidas del usuario hasta RF.

Más adelante, el Departamento de Defensa de los Estados Unidos funda Joint Tactical Radio System (1997) con el fin de incrementar la interoperabilidad entre los diferentes estándares y protocolos de radio usados por las fuerzas militares (algo que ya se pretendía con el proyecto SpeakEASY, pero en este caso a mayor escala). Esto propulsó el avance en el desarrollo de la radio definida por *software*.

Mientras que las investigaciones académicas continuaban profundizando en la radio definida por *software* y su desarrollo se limitaba principalmente al ámbito militar, quedaba muy alejado de un posible uso comercial y/o personal. Pero en 2001 se publica uno de los proyectos *open source* más exitosos en lo que a SDR se refiere: **GNU Radio**. Este no es un dispositivo en sí, sino un *framework* de uso libre que permite desarrollar aplicaciones SDR desde un ordenador y desplegarlas en hardware RF de bajo coste, o bien realizar simulaciones en el entorno del PC. Se encuentra en constante crecimiento y recibe actualizaciones de forma frecuente (varias al año).

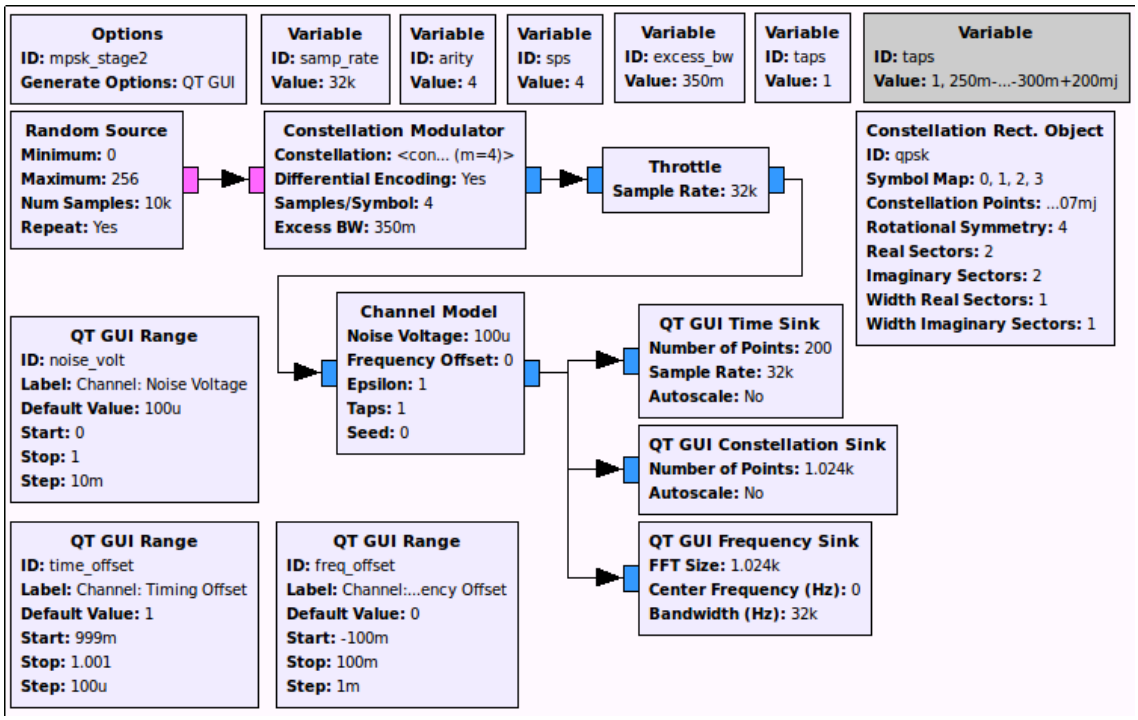


Figura 2.2 Ejemplo de gráfico de flujo del *software* GNU Radio [5].

Cuenta con una herramienta gráfica, **GNU Radio Companion** (GRC), que permite desarrollar las aplicaciones y llevar a cabo simulaciones directamente sin necesidad de tener que conocer ningún lenguaje de programación. Aunque si se quiere extender las funcionalidades (que suele ser lo habitual en aplicaciones avanzadas), será necesario emplear C++.

Por último, hay que mencionar que, aunque GNU Radio sea el *framework* mayormente usado en conjunto con dispositivos comerciales SDR, no es el único. Existen otras alternativas también gratuitas, de las cuales se listan algunas a continuación:

- **SDR#** (SDRSharp): proporciona opciones interesantes en cuanto al procesado digital de señal. Permite seleccionar el modo de demodulación (NFM, AM, DSB…), aplicación de filtros, procesado de audio, visualización de FFT y soporte para añadir *plugins*. Es compatible con gran cantidad de dispositivos: AIRSpy, FUNcube dongle, HackRF One o RTL-SD, entre otros.
- **HDSDR**: Las prestaciones son muy similares a las de SDR#, con algunas características adicionales (opciones de frecuencia ExtIO para adaptadores IF, *upconverters*, *downconverters*, *undersampling* y calibración) y sin posibilidad de añadir *plugins*. Sus aplicaciones típicas serían radioastronomía, radioaficionados o análisis del espectro de la señal. Algunos de los dispositivos con los que es compatible: AIRSpy, RTL-SD, FDM-S1, RSP2 SDRPlay.

2.3. Dispositivos SDR comerciales

Con el tiempo aparecen los primeros dispositivos comerciales que soportan SDR y se abre un abanico muy amplio en cuanto a las opciones disponibles en el mercado. Se analizarán algunos de los más populares, con el fin de conocer sus prestaciones.

Tabla 2.1 Comparativa entre diversos SDR comerciales.

Dispositivo SDR	Rango de frecuencias (MHz)	Resolución del CAD (bits)	¿Puede funcionar como transmisor?	Coste (€)
AIRSpy Mini	24-1700	12	No	138
SDRPlay RSP2	0.001-2000	12-14	No	185
RTL-SDR (R820T2)	24-1766	8	No	Hasta 22
HackRF One	1-6000	8	Sí	266
USRP B200 mini	70-6000	12	Sí	767

2.1.1 RTL-SDR

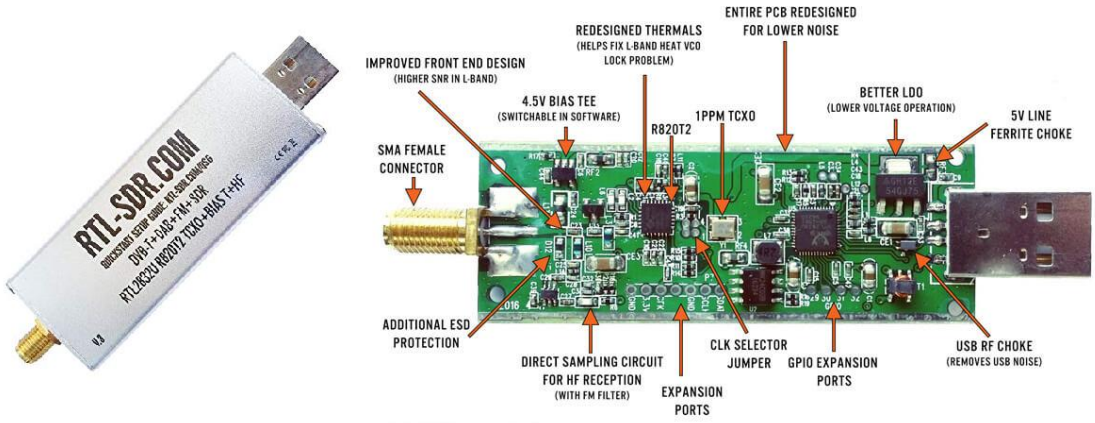


Figura 2.3 Estructura del RTL-SDR Blog R820T2 v3.

Este receptor SDR es el más barato del mercado. Inicialmente estaba pensado para funcionar como sintonizador de televisión DVB-T, pero tras algunas pruebas, se comprobó que era posible utilizarlo para la radio definida por *software*. Se puede emplear como un escáner radio para recibir señales en el área en que se encuentre. Cuenta con varios modelos, y estos se diferencian principalmente en el sintonizador que incluyen, afectando directamente al rango de frecuencias que puede recibir. A mayor rango de frecuencia, mayor será el precio del dispositivo. En concreto, el que se muestra en la **Figura 2.3** se corresponde con el sintonizador Rafael Micro R820T2, el cual posee un rango de frecuencias entre 24 y 1766 MHz. Es, junto con el sintonizador Elonics E4000, la mejor opción dentro de los diferentes modelos. Algunos de los programas con los que es compatible: GNU Radio, SDR# y HDSDR.

Como ocurre con la mayoría de los dispositivos SDR de gama baja, no es posible utilizarlo como transmisor. Dentro de la categoría de gama baja entrarían otros dispositivos como AIRSpy o SDRPlay.

2.1.2 HackRF One

Figura 2.4 Periférico HackRF One.

Desarrollado por Great Scott Gadgets, es un dispositivo SDR *open source* de gama media capaz de funcionar como receptor y transmisor. Este periférico puede trabajar con señales en el rango de 1 MHz a 6 GHz (tanto en recepción como en transmisión), lo que lo hace ideal para desarrollar y probar tecnologías radio actuales así como de las próximas generaciones. Algunos de los programas con los que es compatible: GNU Radio, SDR# y HDSDR.

2.1.3 USRP

Figura 2.5 USRP Bus Series, modelo B200 mini.

Universal Software Radio Peripheral (USRP) comprende a un conjunto de dispositivos de alta gama desarrollados por Ettus Research. Suelen funcionar conectados a un ordenador mediante alguna conexión de alta velocidad, donde se ejecuta algún *software* que controla al USRP. Otra posibilidad sería la de adquirir las versiones que integran el "ordenador" en un procesador embebido que permite que el USRP funcione por sí solo sin depender de un host externo. Dependiendo de las necesidades, resultará más interesante escoger un modelo dentro de una de las cuatro series en las que se presenta: USRP X Series (pensado para diseñar y desplegar sistemas de comunicaciones inalámbricas de la próxima generación), USRP Networked Series (útiles para aplicaciones que requieren de unas prestaciones exigentes), USRP Embedded Series y USRP Bus Series. Algunos de los programas con los que es compatible: GNU Radio, LabVIEW y MATLAB/Simulink.

3 DISEÑO DEL TRANSMISOR

Un sistema de comunicación se inicia con un dispositivo transmisor que envía información. El mensaje a enviar proviene de una fuente de datos y está codificado en bits que necesitan ser procesados y tratados previamente a su transmisión por el canal. Este es el papel del transmisor.

Se estudiarán los principales bloques que conforman al transmisor con un enfoque teórico, desde que se reciben los bits (de la fuente de información) hasta que se tiene la portadora con los datos modulados listos para atravesar el canal, pasando por la conversión de bits a símbolos, la conformación de pulsos y la interpolación de los datos.

3.1. *Mapper*: conversor de bits a símbolos

En general se distinguen dos grandes tipos de modulaciones digitales básicas: las de banda base y las de paso banda. Las primeras suelen preferirse para transmisiones de corta distancia (por cable) y también se conocen con el nombre de *line coding* o codificación de línea; mientras que las segundas son idóneas (aunque no exclusivas) para transmisiones inalámbricas y de larga distancia [6].

La función de este bloque consiste en traducir los bits entrantes en símbolos, los cuales se usarán posteriormente para formar el tren de pulsos que modularán a la señal portadora. En las modulaciones digitales, la portadora solo puede tomar una serie de estados finitos, los cuales están determinados por una amplitud, frecuencia y fase específica. Son estos estados los que se conocen como símbolos.

Cada símbolo es codificado mediante un conjunto de k bits, siendo $M = 2^k$ el número de total de combinaciones y, por tanto, símbolos posibles. Así es posible definir la relación entre la tasa de símbolos R_s (en baudios) y la tasa de bits R_b (en bits por segundo o bps) como:

$$R_s = \frac{R_b}{k} \tag{1}$$

Algunas de las modulaciones digitales más comunes son **ASK** (*Amplitude-shift keying*), **PSK** (*Phase-shift keying*) y **QAM** (*Quadrature amplitude modulation*). Según el número de símbolos empleados para codificar los bits, se suele utilizar la terminación M-ASK, M-PSK o M-QAM.

Así se deduce de la ecuación (1) que, para una misma tasa de símbolos (o ancho de banda), es posible transmitir una mayor cantidad de bits cuanto mayor es el número de símbolos. Por ejemplo, para una tasa $R_s = 200\ baudios$, en el caso de una modulación BPSK (o 2-PSK) se tendría una tasa de bits de $R_b = 200\ bps$, mientras que en una QPSK (o 4-PSK), la tasa de bits sería $R_b = 2 \cdot 200 = 400\ bps$. Aunque escoger una modulación con un alto M no son todo ventajas, pues la probabilidad de error de bit se ve afectada.

3.1.1 Phase-shift keying (PSK)

Esta técnica de modulación lineal se basa en la alteración de la fase de la portadora. Todos los posibles símbolos de la constelación tienen el mismo módulo, variando únicamente su fase. Su detección es coherente, en el sentido de que se necesita conocer la información acerca de la fase de la portadora para poder saber cuáles fueron los símbolos transmitidos, y por tanto requiere de alguna técnica de sincronización de portadora en recepción. Cuenta con una variante conocida como *Differential phase-shift keying* (DPSK), la cual emplea

el símbolo anterior como referencia para demodular el actual, por lo que la información depende de la diferencia de fase. En este caso, la detección es no-coherente y no requiere de sincronización de portadora.

Los símbolos M-PSK están representados en el espacio de señal o constelación mediante el siguiente vector señal con una energía por bit E_b:

$$\mathbf{s}_i = \begin{bmatrix} \sqrt{\log_2 M\, E_b}\cos\left(\frac{2\pi(i-1)}{M}+\varphi\right) \\ \sqrt{\log_2 M\, E_b}\sin\left(\frac{2\pi(i-1)}{M}+\varphi\right) \end{bmatrix}, \qquad i = 1,\dots,M \tag{2}$$

Por ejemplo, para BPSK se obtendrían el siguiente par de vectores:

$$\mathbf{s}_1 = \begin{bmatrix} \sqrt{E_b}\cos(\varphi) \\ \sqrt{E_b}\,\mathrm{sen}(\varphi) \end{bmatrix}, \mathbf{s}_2 = \begin{bmatrix} \sqrt{E_b}\cos(\pi+\varphi) \\ \sqrt{E_b}\,\mathrm{sen}(\pi+\varphi) \end{bmatrix} \tag{3}$$

Opcionalmente, se deja el parámetro φ de manera que se pueda ajustar la fase de la constelación, es decir, existe la posibilidad de rotarla. En la **Figura 3.1** aparece representada para $E_b = 1$ y $\varphi = 0\ rad$.

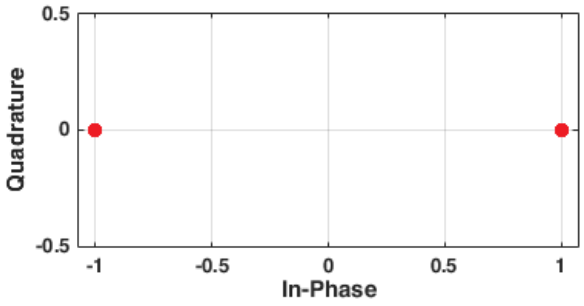


Figura 3.1 Constelación BPSK con energía por bit unidad.

Uno de los factores que permite determinar lo bueno es el sistema de comunicación diseñado es la probabilidad de error de bit o *BER*. Lo habitual es calcular dicha probabilidad según los conceptos teóricos por los que se rija la modulación empleada y, posteriormente, compararlo con los resultados reales de la implementación del sistema [7].

Si se quisiera obtener la probabilidad de error de bit en un canal AWGN para el caso de una modulación BPSK con $M = 2$ y con una energía por bit E_b y fase de rotación de la constelación $\varphi = 0\ rad$, se tendrían como vectores señal $\mathbf{s}_1 = \begin{bmatrix} \sqrt{E_b} \\ 0 \end{bmatrix}$ y $\mathbf{s}_2 = \begin{bmatrix} -\sqrt{E_b} \\ 0 \end{bmatrix}$ cuando se transmite un 0 y un 1 respectivamente.

Considerando que ambos símbolos son equiprobables y teniendo en cuenta que en un canal AWGN el ruido blanco se suma a la señal transmitida, se modela como una variable aleatoria $\mathcal{N}\sim(\mu_N = 0, \sigma_N^2 = \frac{N_0}{2})$ y tiene una función de densidad de probabilidad:

$$f_N(n) = \frac{1}{\sigma_N\sqrt{2\pi}}\, e^{-\frac{n^2}{2\sigma_N^2}} \tag{4}$$

Y en el receptor, las señales recibidas tendrían las siguientes funciones de distribución:

$$f_{Y|\boldsymbol{s}_1}(y|\boldsymbol{s}_1) = \frac{1}{\sigma_N\sqrt{2\pi}}\, e^{-\frac{(n-\sqrt{E_b})^2}{2\sigma_N^2}}, \qquad f_{Y|\boldsymbol{s}_2}(y|\boldsymbol{s}_2) = \frac{1}{\sigma_N\sqrt{2\pi}}\, e^{-\frac{(n+\sqrt{E_b})^2}{2\sigma_N^2}} \tag{5}$$

Así, la probabilidad de error de bit vendrá determinada por:

$$BER = P(\boldsymbol{s}_1)P(E/\boldsymbol{s}_1) + P(\boldsymbol{s}_2)P(E/\boldsymbol{s}_2) \tag{6}$$

Al ser símbolos equiprobables, $P(\boldsymbol{s}_1) = P(\boldsymbol{s}_2) = 0.5$. La región de decisión por tanto se sitúa en el punto medio entre ambos símbolos, o lo que es lo mismo, a la mitad de la distancia mínima entre estos. En la **Figura 3.2** se puede observar gráficamente todo lo expresado hasta ahora.

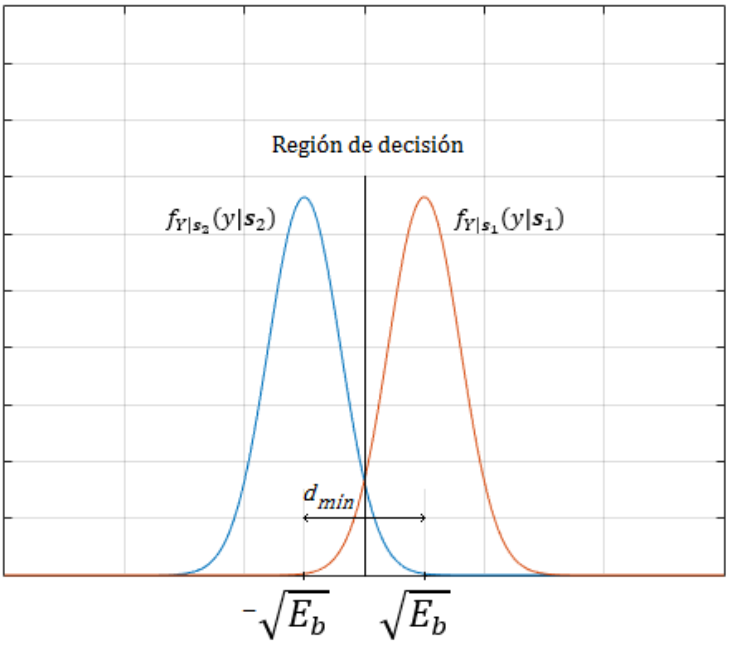


Figura 3.2 Frontera de decisión en BPSK para un canal AWGN.

La expresión de la ecuación (6) se puede reducir, por tanto, a $BER = P(E/\boldsymbol{s}_1) = P(E/\boldsymbol{s}_2)$. Así, bastará con calcular una de las dos:

$$BER = P(E/\boldsymbol{s}_2) = \frac{1}{\sigma_N\sqrt{2\pi}}\int_0^{\infty} e^{-\frac{(n+\sqrt{E_b})^2}{2\sigma_N^2}}\,dn = Q\left(\sqrt{\frac{2E_b}{N_0}}\right) \tag{7}$$

La probabilidad de error de bit se puede calcular de forma exacta para BPSK y QPSK (y, de hecho, son la misma), mientras que para el resto de M-PSK se puede obtener de forma aproximada:

$$\begin{aligned} BER_{BPSK} &= Q\left(\sqrt{\frac{2E_b}{N_0}}\right) \\ BER_{QPSK} &= Q\left(\sqrt{\frac{2E_b}{N_0}}\right) \\ BER_{M-PSK} &\approx \frac{2}{k}Q\left[\sqrt{\frac{2kE_b}{N_0}}\,\text{sen}\left(\frac{\pi}{M}\right)\right], \qquad M \geq 8 \end{aligned} \tag{8}$$

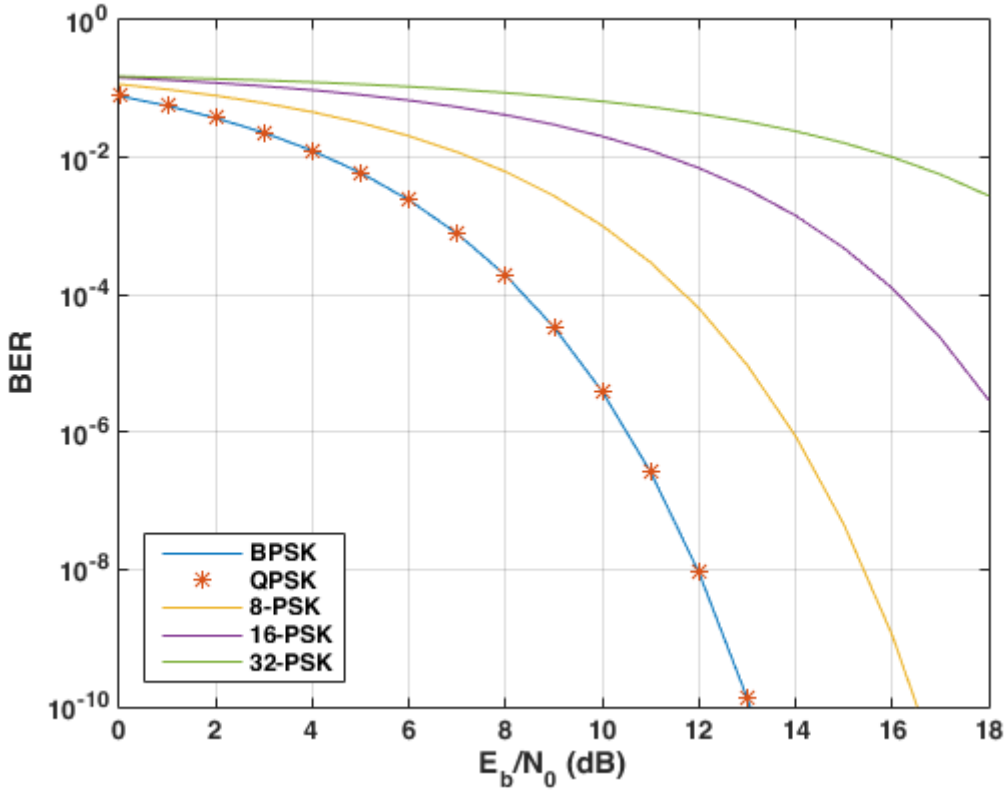


Figura 3.3 Representación gráfica de la BER para BPSK, QPSK, 8-PSK, 16-PSK y 32-PSK.

La probabilidad de error de bit de una BPSK y una QPSK son la misma (no así su probabilidad de error de símbolo que, para un E_b/N_0 dado, resulta ser mayor en una QPSK debido a que un símbolo codifica dos bits frente a BPSK que codifica un único bit). Por otro lado, no se suelen emplear modulaciones superiores a 8-PSK porque se necesita un alto E_b/N_0 para conseguir que la probabilidad de error de bit sea baja. Si se desea obtener una mayor de tasa de bits R_b, es recomendable acudir a otra modulación (aunque ello implique aumentar la complejidad del sistema).

3.2. Conformador de pulsos

Los símbolos obtenidos en el bloque anterior no se pueden transmitir directamente, es necesario adaptarlos para que su transmisión sea posible. Esta es la función del conformador de pulsos: los símbolos a la entrada se convertirán en formas de onda aptas para ser transmitidas por el canal.

La conformación de pulsos se usa principalmente con el fin de hacer un uso eficiente del ancho de banda. Las razones por las que se querría usar eficientemente el ancho de banda son diversas: coste sobre el porcentaje utilizado (a mayor ancho de banda, mayor coste), problemas de compatibilidad electromagnética, limitaciones gubernamentales…

Al usar el ancho de banda de forma eficiente, se incrementa la tasa de datos efectivas para un canal dado; aunque, por otro lado, se asume el riesgo de aumentar los errores en transmisión.

¿En qué consiste esta técnica? Desde el punto de vista teórico, si s_k $(k = 0, \dots, N)$ son cada uno de los símbolos a la salida del *mapper*, entonces la expresión matemática que rige la conformación de pulso es:

$$s(t) = \sum_{k=0}^{N} s_k p(t - kT) \tag{9}$$

donde $p(t)$ se corresponde con la forma de onda del pulso deseado y $T = \log_2 M\, T_b$ es el tiempo que dura un símbolo. De manera que $s(t)$ sería la señal resultante en banda base.

También se suele conocer como "filtro conformador de pulso". La razón es que, si se analiza la ecuación anterior, resulta obvio que los símbolos están siendo filtrados por el pulso $p(t)$. Reescribiendo (9) y haciendo uso de la función delta de Dirac $\delta(t)$, se tiene que:

$$s(t) = \sum_{k=0}^{N} p(t) * s_k \delta(t - kT) \tag{10}$$

Es decir, la señal $s(t)$ es el resultado del sumatorio del pulso conformador convolucionado con los N símbolos espaciados T segundos.

En cuanto a la elección de $p(t)$, se suele optar por el uso de un filtro de **respuesta al impulso finita** (FIR). Estos filtros, como su nombre indica, se caracterizan por tener una respuesta al impulso finita, es decir, la salida alcanza el valor cero en un tiempo finito. Su ecuación en diferencias es la siguiente [8]:

$$y[n] = \sum_{k=0}^{N-1} b_k x[n-k] \tag{11}$$

donde b_k son los coeficientes del pulso conformador y $x(n)$ son los símbolos que se quieren transmitir.

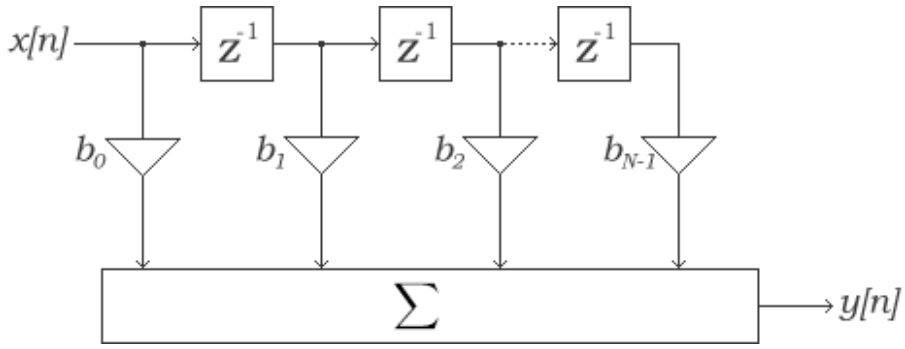


Figura 3.4 Estructura de un filtro FIR.

En digital, es posible implementar la conformación de pulso mediante la interpolación de los datos de entrada. La técnica de interpolación consta de dos pasos:

1. ***Upsample* de la señal de entrada**. Se insertan $L-1$ ceros entre cada muestra, de manera que aumenta la frecuencia de muestreo en un factor L (que debe ser un número entero) pero genera copias no deseadas del espectro original.
2. **Filtrado paso bajo**. Las copias introducidas en el proceso anterior no son de interés y conviene eliminarlas usando para ello un filtro paso bajo. Esto hará que se conserve el espectro original a la vez que se habrá conseguido aumentar la tasa de muestreo en un factor L. Desde el punto de vista del dominio del tiempo, los ceros introducidos son interpolados por el filtro.

Haciendo que el filtro paso bajo utilizado en la interpolación sea $p(t)$, es posible obtener el tren de pulsos conformados en banda base que se entregará al siguiente bloque.

Una forma eficiente de interpolar es haciendo uso de la técnica de **filtrado polifase** [9] (también se puede usar en el proceso contrario, el diezmado), la cual resulta mucho menos costosa computacionalmente frente al método explicado con anterioridad (donde se insertan $L-1$ ceros entre cada muestra y luego se filtra) cuando el factor de interpolación es elevado.

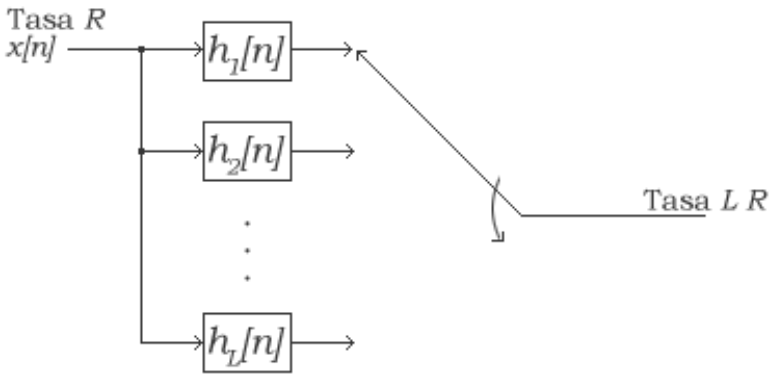


Figura 3.5 Esquema de filtro polifase para interpolación.

En la interpolación por filtrado polifase, el filtro paso bajo original con K coeficientes se divide en un total de L subfiltros, cada uno con $P = \lceil K/L \rceil$ coeficientes y se aplican de forma paralela a los datos de entrada. La razón de que esto sea posible es que al insertar $L - 1$ ceros entre cada muestra, como máximo, existirán P productos distintos de cero (es decir, el producto de los coeficientes del filtro por los datos de entrada).

Cuando entra el primer dato $x[0]$ a la cadena de filtrado, éste es multiplicado por $h[0]$. A continuación, $h[0]$ multiplicará un cero de relleno, y no se encontrará con el siguiente dato hasta que $x[0]$ haya llegado a $h[L]$. Continuando con esta lógica, no verá el siguiente hasta que el primer dato llegue a $h[2L] \dots h[K - L]$. Por tanto, estos constituirían el primer subfiltro. Partiendo ahora desde $h[1]$, y siguiendo los mismos pasos, se tendría que el segundo subfiltro lo formarían $h[1], h[L + 1], h[2L + 1] \dots h[K - L + 1]$. Y así hasta el subfiltro L-*ésimo* formado por $h(L - 1), h(2L - 1), h(3L - 1) \dots h(K - 1)$.

Concretamente, el número de multiplicaciones a realizar crece rápidamente con el factor de interpolación, mientras que en el filtrado polifase lo hace lentamente. Si se tiene un filtro con un total de K coeficientes y N_x datos de entrada siendo $N_x \geq K$, el número de multiplicaciones que hay que llevar a cabo para un factor de interpolación L mediante la inserción de $L - 1$ ceros es:

$$\sum_{i=1}^{K} i + (N_x \cdot L - K) \cdot K \tag{12}$$

Mientras que en el filtrado polifase, con N_{sf} subfiltros de P coeficientes cada uno, el número de multiplicaciones es:

$$N_{sf} \cdot \left[\sum_{i=1}^{P} i + (N_x - P) \cdot P \right] \tag{13}$$

Por ejemplo, en un sistema de comunicación dado, los datos de entrada tienen un tamaño $N_x = 300$ y se desea realizar un aumento de la frecuencia de muestreo en un factor $L = 8$, aplicando para ello un filtro paso bajo con un total de $K = 32\ taps$. Mediante inserción de ceros se tendrían que realizar 76304 multiplicaciones, y en el filtrado polifase tan solo 9552.

3.2.1 Coseno alzado

La interferencia intersímbolo (ISI) es un fenómeno indeseado en el que un símbolo dado corrompe otros símbolos transmitidos a través del canal. Este puede ocurrir principalmente por dos razones: canales limitados en banda y/o propagación multitrayecto.

No todos los pulsos son válidos para llevar a cabo la generación de la señal en banda base. Es importante que el pulso escogido introduzca la menor ISI posible, según el criterio de Nyquist.

En la teoría, existe lo que se conoce como el pulso ideal de Nyquist, el cual evita totalmente la interferencia intersímbolo en un canal paso de baja. Pero tiene un inconveniente: el espectro presenta unas transiciones muy abruptas, lo cual da lugar a un pulso infinito en el tiempo que decae lentamente [10]. Suponiendo que $p(t)$ esté limitado en banda entre las frecuencias $-B_0 \leq f \leq B_0$ y que esta se relaciona con la duración de un símbolo T, de manera que $T = \frac{1}{2B_0}$, entonces:

$$p(t) = 2B_0\mathrm{Sa}(2\pi B_0 t), \qquad P(f) = \mathrm{rect}\left(\frac{f}{2B_0}\right) \tag{14}$$

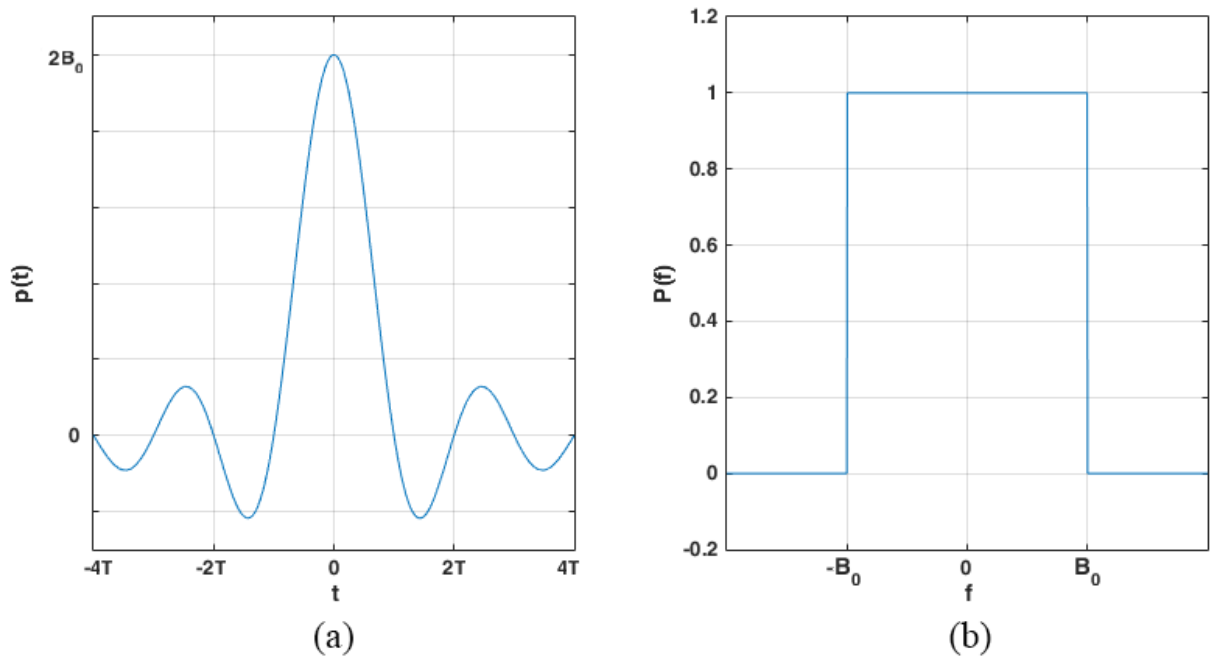


Figura 3.6 (*a*) Pulso ideal de Nyquist en el tiempo. (*b*) Espectro del pulso de Nyquist.

Con el fin de obtener una opción realizable, surge el **coseno alzado** que realiza un "suavizado" de los bordes del espectro rectangular para conseguir que la señal en el tiempo decaiga mucho más rápido, pero que aún mantenga las propiedades que lo hacen interesante de cara a la reducción de la ISI. Cuán suavizado estarán estos bordes vendrá determinado por el coeficiente β, conocido como factor de *roll-off*. Dicho coeficiente está comprendido entre $0 \leq \beta \leq 1$ e indica cuánto sobrepasa el ancho de banda del pulso de Nyquist, lo que significa que un factor $\beta = 0$ equivale a la situación ideal.

$$H(f) = \begin{cases} 1, & |f| \leq B_0(1-\beta) \\ \frac{1}{2}\left\{1 + \cos\left[\frac{\pi T}{\beta}\left(|f| - \frac{1-\beta}{2T}\right)\right]\right\}, & B_0(1-\beta) \leq |f| \leq B_0(1+\beta) \\ 0, & |f| \geq B_0(1+\beta) \end{cases} \tag{15}$$

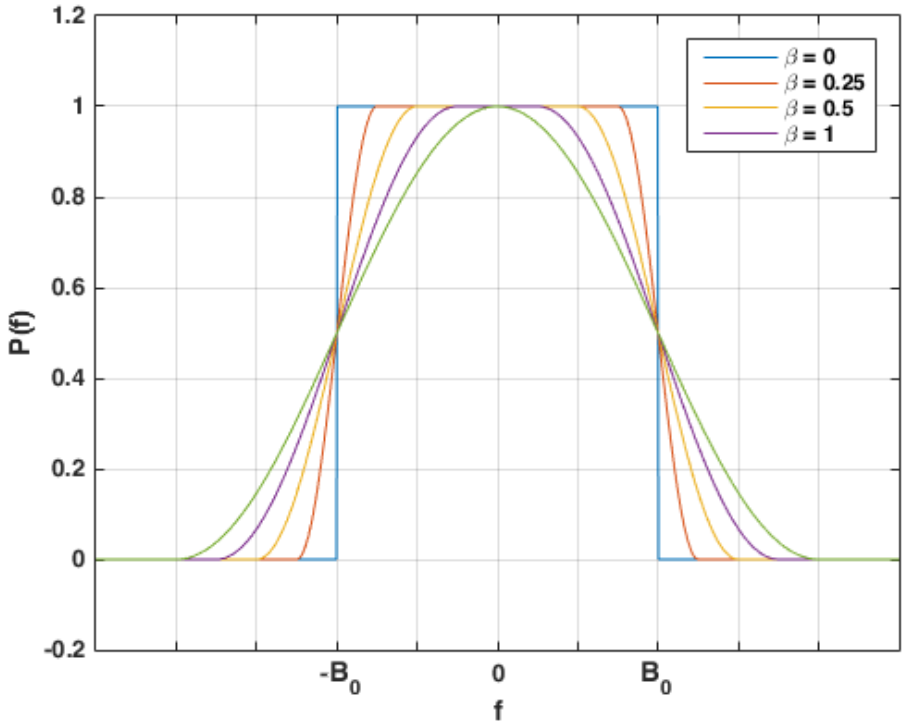


Figura 3.7 Espectro del coseno alzado según diferentes factores de *roll-off*.

La respuesta al impulso correspondiente tendrá duración infinita, por lo que es necesario realizar un truncado simétrico de la misma para obtener los coeficientes (*taps*) del filtro. Para realizar el truncado se suele aplicar algún método de enventanado, siendo el más simple el rectangular; el uso de ventanas introduce cambios en el espectro original de la señal (quedando este multiplicado por la respuesta en frecuencia de la ventana en cuestión).

Por otra parte, para tener ISI mínima ocurre que no solo se debe cumplir el criterio de Nyquist en transmisión, sino que debe ser durante todo el enlace de comunicación, lo cual incluye el canal y la recepción. Para que esto sea así, se suele emplear la raíz del coseno alzado, de manera que se hace un primer filtrado en el transmisor $H_{rca,tx}(f)$ y, posteriormente en el receptor, se aplica de nuevo como filtro adaptado $H_{rca,rx}(f)$, teniendo como respuesta final el coseno alzado $H_{ca}(f)$. Suponiendo un canal ideal $H_c(f) = 1$, entonces:

$$H_{rca,tx}(f) \cdot H_c(f) \cdot H_{rca,rx}(f) = \sqrt{H_{ca}(f)} \cdot \sqrt{H_{ca}(f)} = H_{ca}(f) \tag{16}$$

3.3. Interpolación en múltiples etapas (opcional)

Realizar una interpolación por un factor L demasiado alto resulta en un aumento del coste computacional para llevar a cabo dicha operación. En estos casos, se suele realizar una interpolación en varias etapas; resultando de especial interés cuando la implementación se hace en *hardware* (FPGAs, DSP) y no a través de una CPU. La única condición para que esto sea posible es que el factor final deseado L no sea un número primo, de manera que pueda descomponerse en un producto de factores enteros:

$$L = L_1 L_2 \dots L_N, \qquad L_1 \leq L_2 \leq \cdots \leq L_N, \tag{17}$$

Para ilustrar la mejora que supone el uso de múltiples etapas, se propone el siguiente ejemplo: se desea interpolar una señal con frecuencia de muestreo $F_0 = 400\ Hz$ para conseguir una nueva frecuencia de muestreo de $F_1 = 40\ kHz$. Para ello, se usará un filtro paso bajo diseñado con el método de rizado constante (*equiripple*), de manera que sea posible conocer el número aproximado de coeficientes que harán falta para conformar el filtro. Se quiere que el filtro tenga un rizado en la banda de paso de $\delta_P = 0.01$ y un rizado en la

banda de rechazo de $\delta_S = 0.001$ $(-60\ dB)$. Al insertar $L-1$ ceros (para interpolar), como ya se mencionó, el espectro original se copia L veces, por ello es necesario realizar el filtro paso bajo con una frecuencia de corte de la banda de paso en $F_P = 195\ Hz$ y frecuencia de corte en la banda de rechazo $F_S = 200\ Hz$ (es decir, alrededor de $F_0/2$). Esto significa que la banda de transición tendrá un ancho de $\Delta F = F_S - F_P = 5\ Hz$. Con estos datos y mediante la siguiente ecuación conocida como el método Harris [11], es posible obtener de forma aproximada el número de coeficientes necesarios para conformar el filtro:

$$N \approx \left| \frac{\delta_S\ (dB)}{22 \cdot\ \Delta F/F} \right| \tag{18}$$

Así se tiene que se necesitarían alrededor de $N \approx 21818$ coeficientes para una interpolación de $L = 100$.

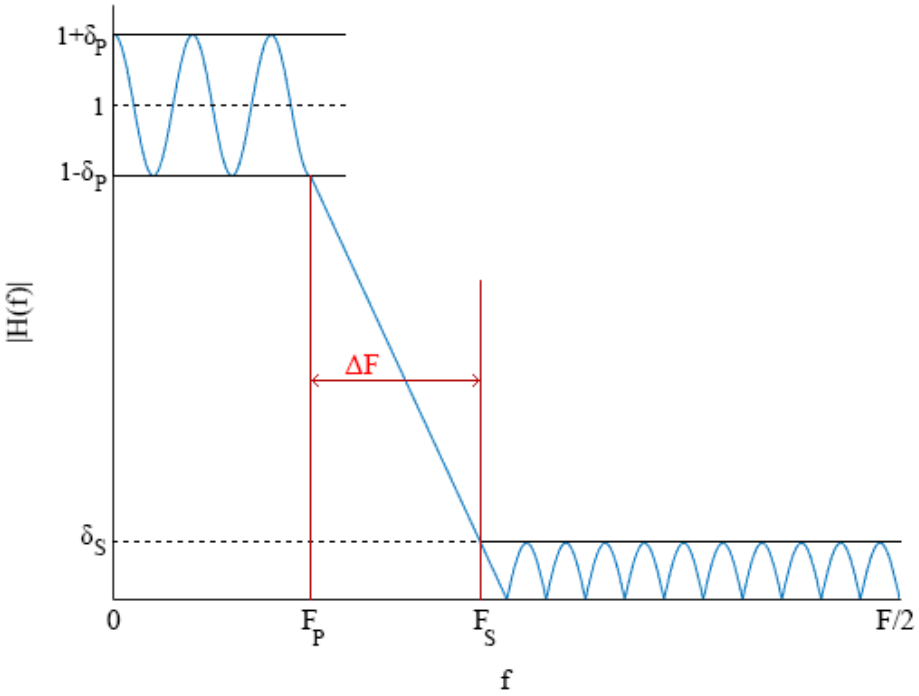


Figura 3.8 Parámetros típicos de diseño de filtros FIR paso bajo.

Continuando con el ejemplo, en lugar de hacerlo en una única etapa, se interpolará en dos etapas con $L_1 = 2$ y $L_2 = 50$. De esta manera, primero se pasará de una frecuencia de muestreo $F_0 = 400\ Hz$ a $F_1 = 800\ Hz$, y en la segunda etapa a $F_2 = 40\ kHz$. En primer lugar, se insertan $L_1 - 1$ ceros y se aplica un filtro paso bajo con N_1 coeficientes, para posteriormente insertar $L_2 - 1$ ceros y usar un filtro paso bajo de N_2 coeficientes. El primer filtro tendrá una frecuencia de muestreo $F_1 = 800\ Hz$, y una banda de transición $\Delta F_1 = 200 - 195 = 5\ Hz$, con un rizado en la banda de paso de $\delta_P = 0.005$ y un rizado en la banda de rechazo de $\delta_S = 0.001$ $(-60\ dB)$. Esto implica que aproximadamente se tendrían $N_1 \approx 436$ coeficientes. En el segundo filtro, la frecuencia de muestreo cambia a $F_2 = 40\ kHz$, y en este caso se admite que la banda de transición sea más ancha debido a que el filtrado anterior eliminó las copias pertinentes, lo que relajará en gran medida el número de coeficientes resultantes; esto es, ahora $\Delta F_2 = 600 - 195 = 405\ Hz$ y los rizados se mantienen por igual. Entonces se tendrán $N_2 \approx 269$ coeficientes. El total de coeficientes necesarios sería $N = N_1 + N_2 = 705$, frente a los 21818 del caso anterior interpolando en una sola etapa.

3.3.1 Filtros cascaded integrator-comb (CIC)

Este tipo de filtros son ampliamente utilizados cuando se necesita cambiar la frecuencia de muestreo en un **factor grande**. Son sencillos en lo que a complejidad de cómputo se refiere ya que no emplea ningún multiplicador (solo sumadores), lo que los hace realmente interesantes en implementación *hardware* donde los

recursos para el procesado digital a veces escasean. Son, además, filtros de fase lineal.

$$H(z) = H_I^N(z)H_C^N(z) = \left(\frac{1}{1-z^{-1}}\right)^N \cdot (1-z^{-RM})^N = \left(\frac{1-z^{-RM}}{1-z^{-1}}\right)^N = \left(\sum_{k=0}^{RM-1} z^{-k}\right)^N \quad (19)$$

Los filtros CIC están conformados por dos bloques básicos que son: un integrador y un filtro *comb* (de peine). Cada uno de estos bloques se repite N veces y, según se organicen, harán las veces de interpolador o de diezmador (ver **Figura 3.9**).

Tal y como puede observarse en la igualdad final de la ecuación (19), dicha expresión es equivalente a la implementación de N filtros FIR, por lo que la generación de los coeficientes del filtro es una tarea sencilla.

Los parámetros (enteros positivos) que conforman el filtro son:

- N: número de etapas integradoras y *comb*.
- R: factor de interpolación/diezmado.
- M: retraso diferencial. Suele fijarse a 1.

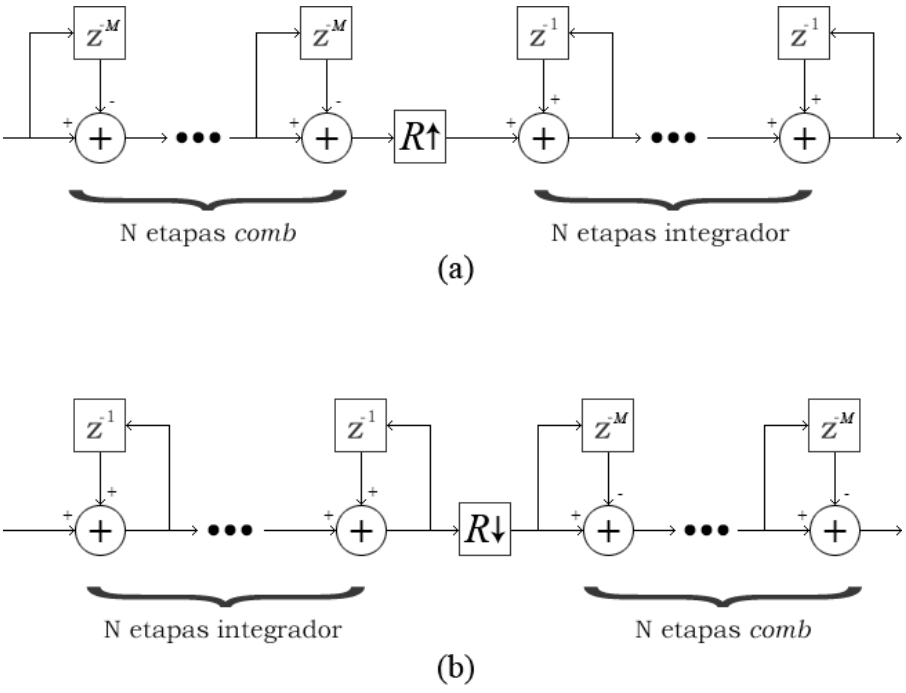


Figura 3.9 (*a*) Filtro CIC interpolador de N etapas. (*b*) Filtro CIC diezmador de N etapas.

A pesar de las virtudes de este filtro, también tiene un gran inconveniente: su respuesta en frecuencia no es plana en la banda de paso (ver **Figura 3.10**), lo que implica que la banda de interés se ve atenuada. Para ello suele ser necesario la acción de un filtro FIR que compense el efecto del CIC. Este filtro compensador se situaría antes de la interpolación CIC o después del diezmado CIC (siempre en el lado que funciona a menor frecuencia de muestreo) [12].

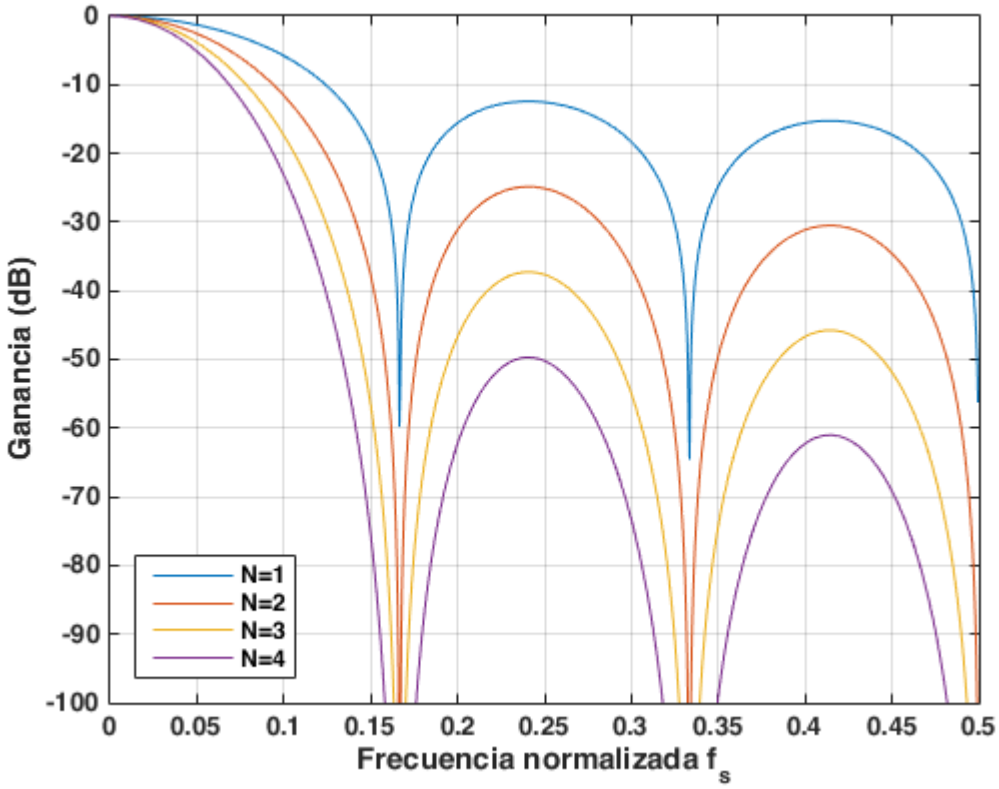


Figura 3.10 Respuesta en frecuencia de un filtro CIC con $R = 6$, $M = 1$ y N variable.

3.4. Subida a frecuencia intermedia

Hasta el momento, todo el procesado de señal ha tenido lugar en banda base (es decir, donde la frecuencia central es 0 Hz), sin embargo, para llevar a cabo una transmisión radio es necesario que la señal en banda base se module a una banda de frecuencias mucho más alta, la cual se suele denominar con el nombre de radiofrecuencia (RF). Sin embargo, y de acuerdo con lo estudiado en la **sección 2.1**, la conversión a RF es una tarea de la que se encargará la parte *hardware* (después de pasar por el CDA). En *software* se realiza una conversión a frecuencia intermedia (IF) menor que la mitad de la frecuencia de muestreo final.

$$f_s = f_{s,fuente\ datos} \prod_{k=1}^{N} L_i \tag{20}$$

donde L_i se corresponde con cada uno de los factores de interpolación de las diferentes etapas según se mencionó en la sección anterior.

Por lo que, para asegurar el cumplimiento del teorema del muestreo de Nyquist, la frecuencia de la portadora debe ser:

$$f_c = f_{IF} < \frac{f_s}{2} \tag{21}$$

Si la señal en banda base es $s_b(t) = s_{b,I}(t) + j s_{b,Q}(t)$, entonces $s(t)$ será la señal con frecuencia central f_c:

$$s(t) = Re\{s_b(t) e^{j2\pi f_c t}\} = s_{b,I}(t)\cos(2\pi f_c t) - s_{b,Q}(t)\,\mathrm{sen}(2\pi f_c t) \tag{22}$$

De esta forma, la señal estaría lista para su transmisión por el canal.

4 IMPLEMENTACIÓN DEL TRANSMISOR

Tras el estudio teórico de los principales bloques del transmisor del capítulo anterior, se propondrá un código para la implementación en MATLAB de cada uno de ellos, así como algunos resultados de la ejecución y simulación de los bloques.

Se aprovechará la posibilidad que ofrece MATLAB de emplear clases y objetos, de forma que todo quede mucho más compacto y permita la adición de nuevos componentes en el futuro (por ejemplo, una nueva modulación en el mapper, un tipo de pulso diferente a emplear durante la conformación, etc.).

4.1. Implementación del *mapper*

Esta función tiene dos parámetros principales, los datos serie en binario (es decir, una fila de $N\ bits$) y el tipo de modulación digital que se desea emplear. Además de un tercer parámetro opcional de depuración y testeo que, en caso de estar activado, realizará la representación de la constelación transmitida y de la BER teórica para la modulación escogida.

Código 4.1 Convertir bits a símbolos.

```
datos = reshape(datos', modulacion.k, [])';
[~,index] = ismember(datos, modulacion.bits, 'rows');
simbolos = modulacion.alfabeto(index);
```

El *mapper* supone que los datos llegan debidamente tratados para poder realizar el `reshape` sin complicaciones. De manera que reorganiza los bits, que inicialmente están en serie, en filas de k columnas y compara cada una con cada estado o punto de la constelación de la modulación elegida (`ismember`), asignando el símbolo correspondiente a dicho conjunto de bits.

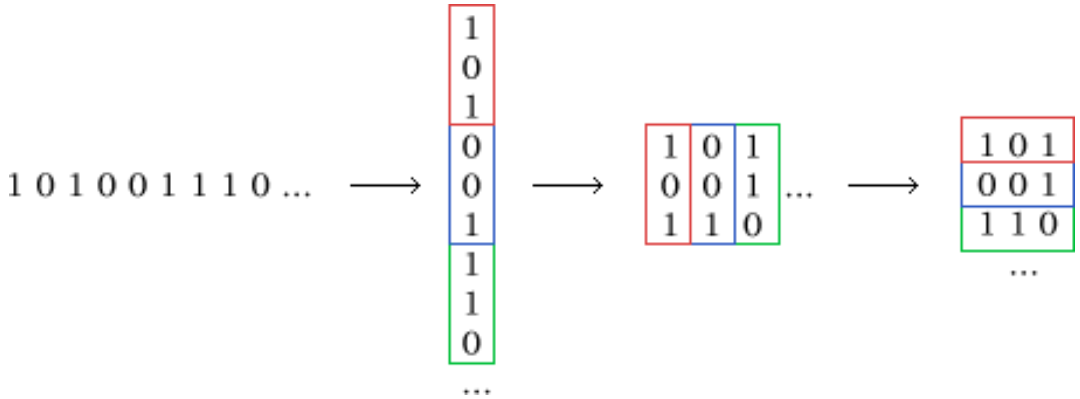


Figura 4.1 Reestructurado de los datos para una modulación 8-PSK.

El parámetro referente a la modulación tiene una característica especial, y es que debe ser un objeto perteneciente a la superclase `constelacion`. Esta superclase hará de plantilla para dar soporte a todas las

modulaciones que se quieran integrar en el sistema, y por tanto deberán heredar de la misma sus propiedades y el método constructor. La razón por la que se ha optado por usando clases es porque admite un gran modularidad que, a fin de cuentas, es lo que se busca en una radio definida por *software*. De esta manera, se podrían añadir modulaciones adicionales sin demasiada complicación y sin tener que realizar grandes modificaciones al código base.

Código 4.2 Superclase `constelacion`.

```
classdef (Abstract) constelacion
    properties
        M        % constelacion M-aria
        k        % numero de bits de cada simbolo
        bits     % bits de la constelacion
        alfabeto % simbolos de la constelacion
    end

    methods
        function obj = constelacion(M)
            obj.k = ceil(log2(M));
            obj.M = 2^obj.k;
        end
    end
end
```

Lo primero a destacar sobre esta clase es el término `(Abstract)` que aparece justo antes del nombre. Esto indica que se usará para describir una serie de funcionalidades que serán comunes a un grupo de clases pero que deben ser implementadas de manera específica en cada una de ellas, por tanto, `constelacion` no puede ser instanciada directamente.

Cuenta con las siguientes propiedades que serán comunes a todas las subclases:

- `M`: número de símbolos de la constelación.
- `k`: número de bits que codifica cada símbolo. Se calcula a partir de `M`, de manera que se escoge un `k` para poder tener un número entero de bits.
- `bits`: matriz $M \times k$ que almacena el conjunto de bits correspondientes a cada símbolo.
- `alfabeto`: equivalente en símbolo de la propiedad anterior.

Cada vez que se desee crear una clase que herede de `constelacion`, se usará el símbolo < detrás del nombre e indicando a continuación el nombre de la superclase.

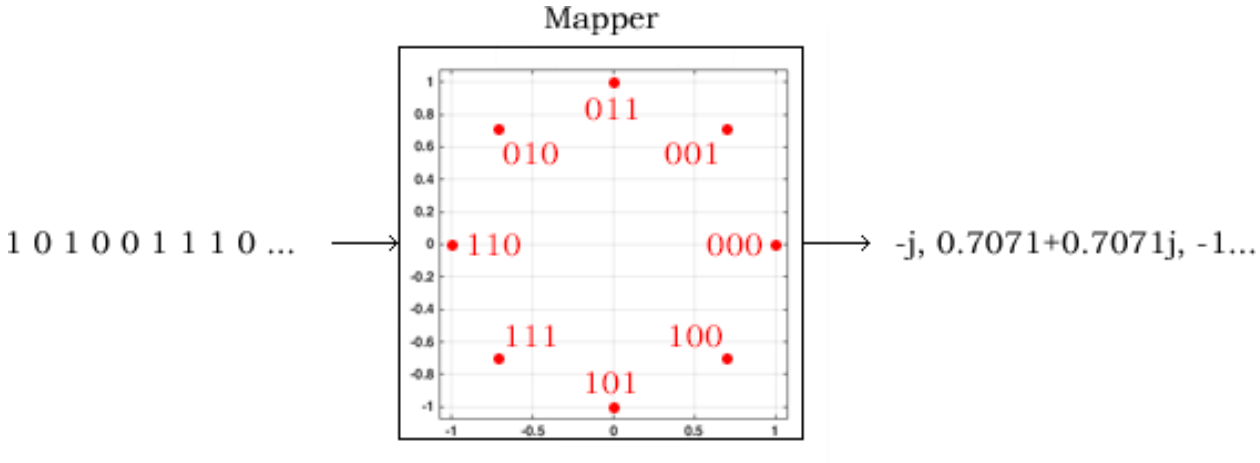


Figura 4.2 Ejemplo gráfico del funcionamiento del *mapper* para una 8-PSK.

4.1.1 Phase-shift keying (PSK)

Se creará una nueva clase `constel_psk` que permitirá generar una constelación M-PSK a partir de los argumentos de entrada. Tal y como se comentaba, esta clase heredará las propiedades y métodos de la superclase `constelacion`. Además, se le añadirá una nueva propiedad, `fase`, que permitirá ajustar la fase de rotación de la constelación (por defecto será cero):

Código 4.3 Definición y propiedades de la clase `constel_psk`.

```
classdef constel_psk < constelacion
    properties
        fase     % fase de la constelacion
    end

    methods
        function obj = constel_psk(M, fase)
            if (nargin < 2)
                fase = 0;
            end

            obj = obj@constelacion(M);
            obj.fase = fase;
            [obj.bits, obj.alfabeto] = obj.crear_constelacion;
        end

    %[. . .]

    end
end
```

El constructor de `constel_psk` llama al constructor de la superclase y, a continuación, establece la fase y genera la constelación haciendo uso del método `crear_constelacion`.

Código 4.4 Creación de la constelación M-PSK.

```
function [bits,alfabeto] = crear_constelacion(obj)
    alfabeto = zeros(1, obj.M);
    bits = zeros(obj.M, obj.k);
    for s=1:obj.M
        bits(s,:) = numAbin(s-1, obj.k);
        alfabeto(s) = exp(1i*(2*pi*(s-1)/obj.M + obj.fase));
    end

    % Convertimos a gray
    bits = xor(bits,[zeros(obj.M,1), bits(:,1:end-1)]);
end
```

Por lo que bastará con las propiedades `M` y `k` para generar el alfabeto correspondiente y sus bits asociados. Para obtener el alfabeto se usa la expresión (2) y los bits son convertidos a codificación Gray (esto se hace para evitar la existencia de señales ilegales en las transiciones de un símbolo a otro).

Código 4.5 Función `numAbin`.

```
function bin = numAbin( num , bits )
    if (nargin < 2)
        bits = 0;
    end

    bin = mod(floor(num.*2.^-(floor(log2(num)):-1:0)),2);
    tam = length(bin);

    if (bits > tam)
        bin = [zeros(1,bits-tam), bin];
    end
end
```

A partir de las ecuaciones estudiadas en el capítulo anterior (8), es posible añadir un nuevo método a la constelación PSK que permita obtener la probabilidad de error de bit para cualquier valor de `M`:

Código 4.6 Obtención de la BER teórica para M-PSK.

```
function ber_teorica = calcular_ber_teorica(obj, EbNo_dB)
    EbNo = 10.^(EbNo_dB/10);

    switch (obj.M)
        case {2, 4}
            ber_teorica = qfunc(sqrt(2.*EbNo));
        otherwise
            ber_teorica =(2/obj.k)*qfunc(sqrt(2.*obj.k.*EbNo).* ...
            sin(pi/obj.M));
    end
end
```

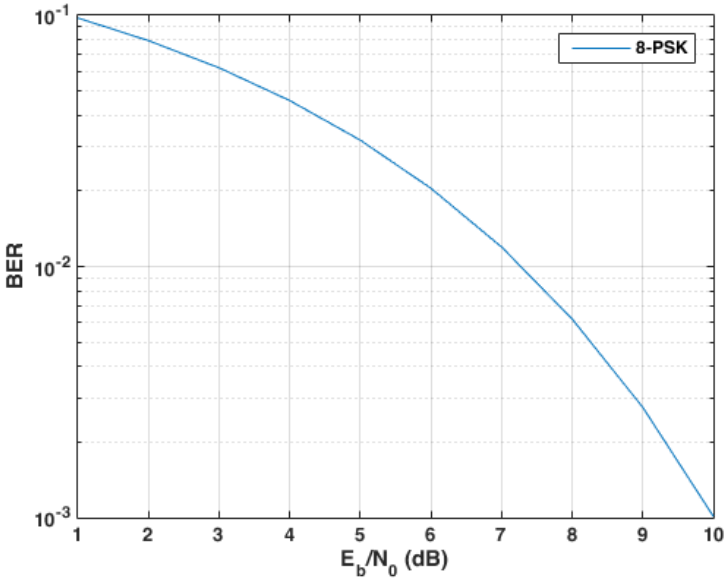


Figura 4.3 BER teórica para una 8-PSK empleando el **Código 4.6**.

4.2. Conformador de pulsos

La función `conformador_pulso` cuenta con tres parámetros obligatorios y dos opcionales. Los datos de entrada (que serán los símbolos de la salida del *mapper*), un objeto de la superclase `pulso_conf` y el factor de interpolación son requeridos mientras que, opcionalmente, se podrá facilitar directamente los coeficientes del filtro conformador (en lugar del objeto) y el parámetro que permite activar la depuración del código.

Código 4.7 Funcionamiento básisco de `conformador_pulso.m`.

```
if (coef == 0)
    coef = pulso.generar_coeficientes;
end

datos_pulsoconf = filtrado_polifase(datos_mod, coef, ...
                                    factor_interpolacion);
```

En primer lugar, se comprueba si han pasado los coeficientes directamente; en caso contrario, se llamará al método `generar_coeficientes` que de forma obligatoria deben implementar todos los objetos que hereden de la superclase `pulso_conf`.

Según se observa en el **Código 4.7**, la función `filtrado_polifase` admite como entradas los datos a filtrar, los coeficientes del filtro, y el factor de interpolación/diezmado. Opcionalmente es posible pasar un cuarto parámetro que indique si se está interpolando o diezmando (en caso de no incluirlo se hará por defecto la interpolación).

Código 4.8 Interpolación con filtrado polifase.

```
function datos_filtrados = filtrado_polifase(datos, coef,
factor_intdec, interp, retraso )
    tam_coef = length(coef);
    rellenar_ceros=factor_intdec*ceil(tam_coef/factor_intdec)-tam_coef;
    b = [coef zeros(1,rellenar_ceros)];
    b_p = reshape(b, factor_intdec, []);

    if (interp == 1)
        % Interpolado
        tam_datos_filtrados = length(datos)*factor_intdec;
        datos_filtrados = zeros(1, tam_datos_filtrados);

        for i=1:factor_intdec
            datos_filtrados(i:factor_intdec:end)= filter(b_p(i,:),1,...
                                                  datos);
        end
    else
        % [ . . . ] esta parte se discute en el diezmado
    end
end
```

Lo primero que se hace es comprobar si es necesario rellenar con ceros el filtro. Esto es porque al crear los subfiltros, todos deben tener el mismo número de coeficientes por lo que, en caso de que `tam_coef/factor_intdec` no sea un número entero, será necesario rellenar con ceros hasta conseguir que lo sea.

Mediante un `reshape` se reorganizan los coeficientes en una matriz LxM, donde cada fila representa un subfiltro.

El último paso sería filtrar los datos de entrada (sin la inserción de $L-1$ ceros, es decir, intactos) con cada uno de los subfiltros y reorganizar el resultado de cada uno de ellos para formar la salida interpolada. Denominando como $y_m(n), m = 0 \dots M-1$ a la salida de los subfiltros, y sabiendo que el número de datos a la salida es el mismo que el de entrada, entonces estos se reorganizaran de la manera siguiente:

$$y(n) = [y_0(0), y_1(0), \dots, y_{M-1}(0), y_0(1), y_1(1) \dots, y_{M-1}(1), y_0(N-1), y_1(N-1) \dots, y_{M-1}(N-1)] \tag{23}$$

Para ejemplificar cómo funciona el código anterior, se muestra a continuación una secuencia numérica que se desea interpolar por un factor de 5. Como filtro paso bajo se usará uno de fácil implementación, el filtro de media móvil. El número de puntos de este filtro dependerá del factor de interpolación.

Código 4.9 Ejemplo de interpolación usando filtrado polifase.

```
X          = [1 2 3 4 5 1 0 1 2 3 1 1 2 3 3];
factor_int= 5;
b          = ones(1, factor_int)./factor_int;

resultado = filtrado_polifase(x, b, factor_int, 1, 0);
```

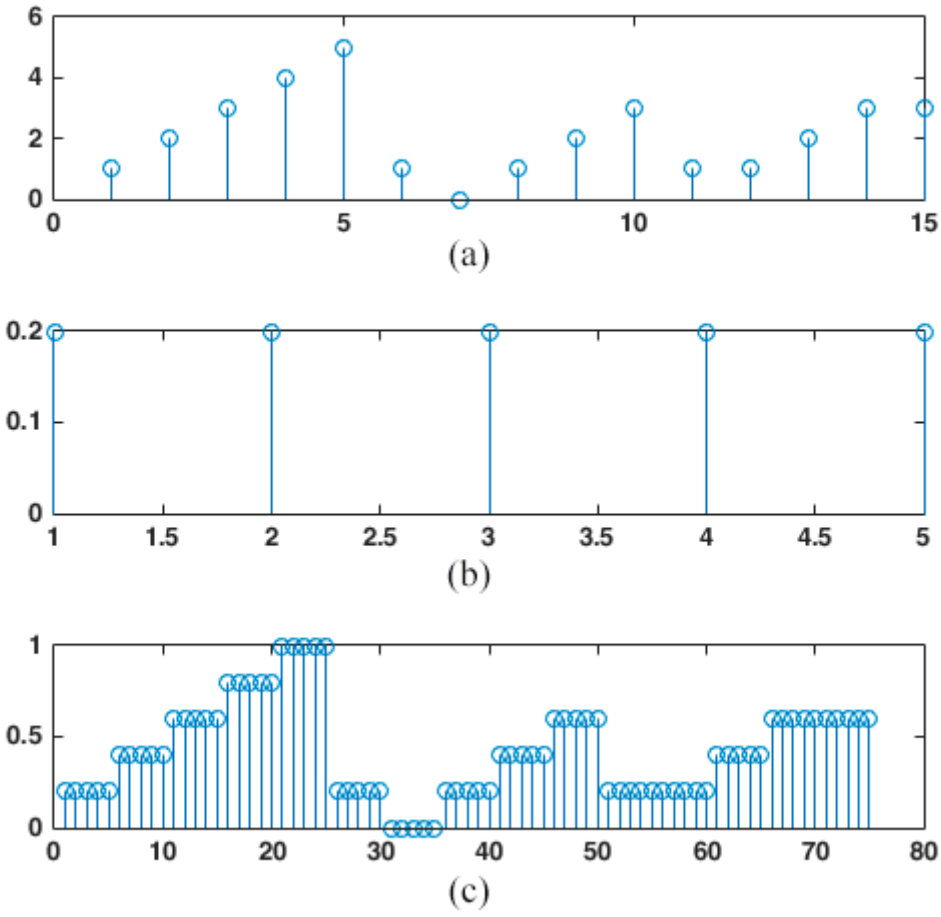


Figura 4.4 (*a*) Secuencia a interpolar. (*b*) Filtro de media móvil. (*c*) Resultado de la interpolación.

4.1.2 Coseno alzado

Para desarrollar la clase `coseno_alzado` (que hereda de `pulso_conf`) se ha optado por el uso de la función `rcosdesign` implementada en MATLAB [13]. Esta función admite cuatro parámetros:

- `beta`: es el factor de *roll-off*, valor comprendido entre 0 y 1.
- `span`: indica en cuántos símbolos se truncará la respuesta al impulso.
- `sps`: este parámetro se usa para establecer cuántas muestras habrá por símbolo.
- `shape`: admite dos valores, `normal` o `sqrt`. En el primer caso genera coeficientes correspondientes al coseno alzado, mientras que en el segundo lo hace para la raíz del coseno alzado.

El orden del filtro vendrá determinado por $span * sps$. Debido a que el truncado es simétrico, el producto anterior debe ser par, así se obtendrá siempre un número impar de coeficientes. A mayor número de coeficientes, mayor coste computacional (una de las razones por la que resulta útil implementar el filtrado polifase ya comentado).

Código 4.10 Definición y propiedades de la clase `coseno_alzado`.

```
properties
    tipo            = 'sqrt'  % normal ('normal') o raiz ('sqrt')
    rolloff         = 0.5     % factor de rolloff
    truncSimbolos   = 4       % numero de simbolos a los que se trunca
    muestrasPorSimb = 4       % numero de muestras por simbolo
end

methods
    function obj = coseno_alzado(rolloff, span, sps, tipo)
        if (mod(span*sps,2) ~= 0)
            error('truncSimbolos por muestrasPorSimb debe ser par.');
        else
            if (nargin < 4 && nargin > 0)
                obj.tipo = 'normal';
            else
                obj.tipo = tipo;
            end
            obj.rolloff = rolloff;
            obj.truncSimbolos = span;
            obj.muestrasPorSimb = sps;
        end
    end

    function [coef,retraso] = generar_coeficientes(obj)
        coef = rcosdesign(obj.rolloff, obj.truncSimbolos, ...
                          obj.muestrasPorSimb, obj.tipo);
        retraso = mean(grpdelay(coef));
    end
end
```

Las propiedades de la clase son equivalentes a los parámetros necesarios para usar la función `rcosdesign` de MATLAB. El método `generar_coeficientes`, además de los *taps* del filtro, también devuelve el retardo de grupo medio introducido por este.

El valor de la propiedad `muestrasPorSimb` deberá coincidir con el factor de interpolación de la función principal `conformador_pulso`.

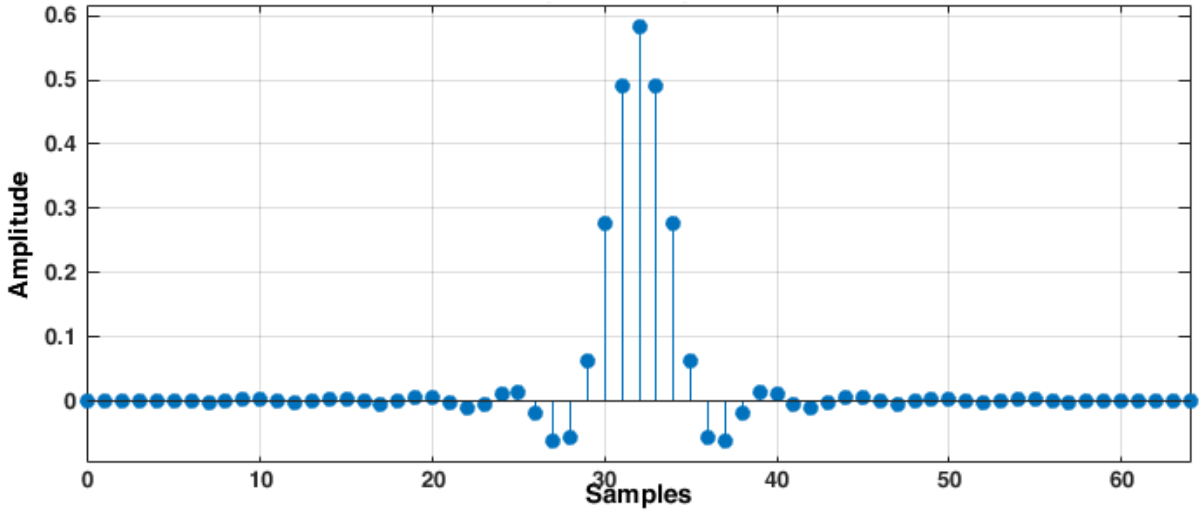
Suponiendo que se tiene una secuencia de datos que ha sido modulada usando BPSK y se quiere usar el conformador de pulsos de manera que el pulso sea la raíz del coseno alzado con un factor de roll-off $\beta = 0.60$, un total de 16 símbolos y 4 muestras por símbolo, entonces el código a emplear sería:

Código 4.11 Ejemplo de uso del conformador de pulsos.

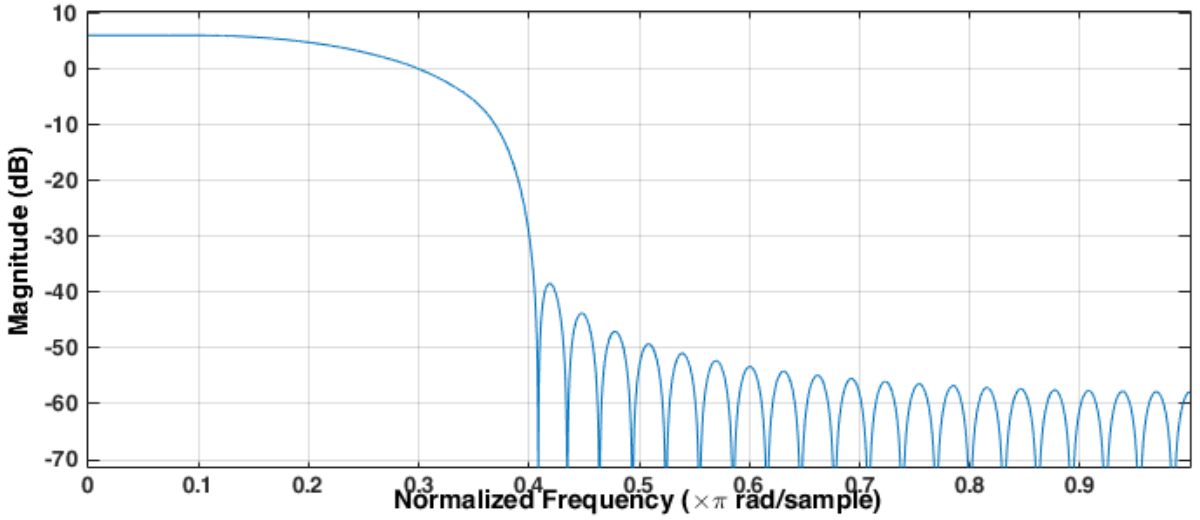
```
datos           = randi([0 1], 1, 200).*2-1;

% Definición de las propiedades del objeto 'coseno_alzado'
tipo            = 'sqrt';  % normal ('normal') o raiz ('sqrt')
rolloff         = 0.60  ;  % factor de rolloff
truncSimbolos   = 16    ;  % numero de simbolos a los que se trunca
muestrasPorSimb = 4     ;  % numero de muestras por simbolo

pulso = coseno_alzado(rolloff, truncSimbolos, muestrasPorSimb, tipo);

[coef, retraso] = pulso.generar_coeficientes();

resultado = filtrado_polifase(datos, coef, muestrasPorSimb, 1, 0);
```

La respuesta al impulso y en frecuencia del filtro generado se puede observar en la **Figura 4.5**, y la forma de onda resultante del conformador de pulsos en la **Figura 4.6**.

Para poder realizar la representación de la forma de onda de manera que coincida con los datos superpuestos, es necesario tener en cuenta el retraso que introduce el filtro (en el código es la variable de salida `retraso` del método empleado para la generación de coeficientes) e introducir `muestrasPorSimb-1` ceros entre cada una de las muestras de los datos de entrada.



(a)



(b)

Figura 4.5 (*a*) Respuesta al impulso del filtro RRC con $\beta = 0.60$. (*b*) Respuesta en frecuencia.

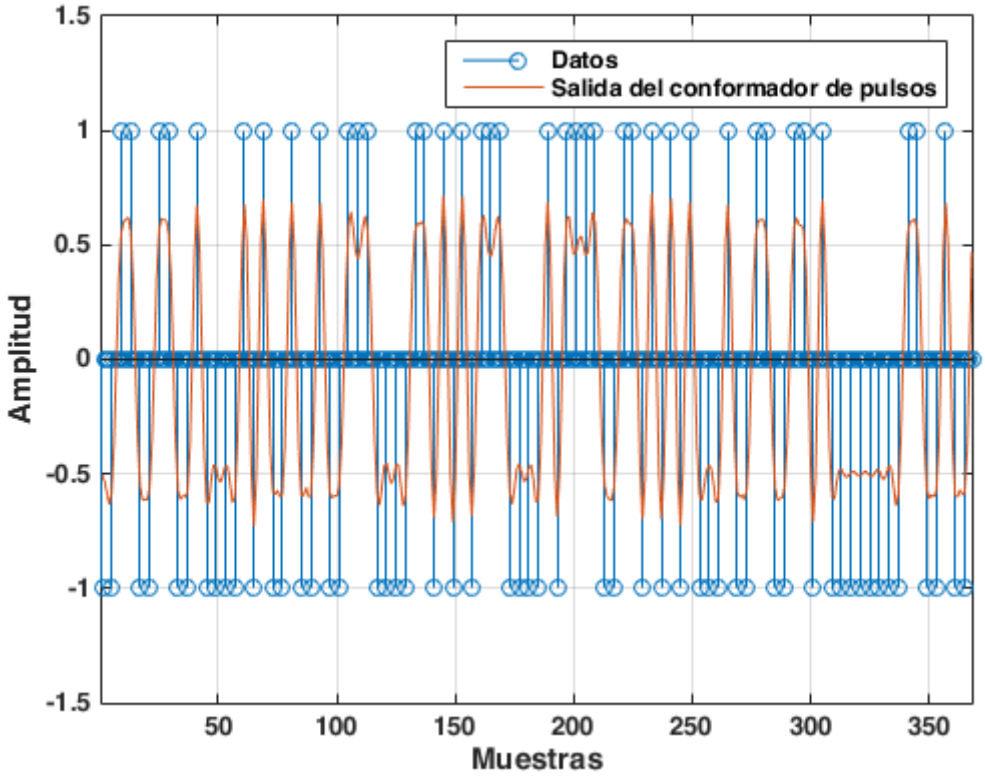


Figura 4.6 Resultado de la conformación de pulsos usando un filtro RRC.

4.3. Interpolación en múltiples etapas

La función `transmisor` acepta `factor_int` como uno de sus parámetros de entrada. Este indica el factor de interpolación que, en caso de ser un número, implementará en una única etapa y usará como filtro paso bajo el pulso conformador. Sin embargo, es posible pasar ese parámetro como un vector $1xN$, lo que supondrá que se haga una interpolación en varias etapas, siendo la primera de ellas la de la conformación del pulso. El filtrado del resto de etapas dependerá de otro parámetro pasado al transmisor llamado `coef_multi_etapa`, el cual deberá ser de tipo `cell` (celda, estructura predefinida por MATLAB) de tamaño $1x(N-1)$. Cada una de las columnas de la celda será, a su vez, un vector $1xK_i$ con los coeficientes de cada uno de los filtros. La razón por la que se usan celdas es que estas permiten contener vectores de diferentes anchuras, mientras que en las matrices todas las filas deben tener el mismo número de columnas.

Es necesario recalcar que las $N-1$ interpolaciones realizadas después de la conformación de pulso deberán usar filtros con ganancia L_i pues, de lo contrario, la amplitud a la salida de estos se vería escalada por un factor $1/L_i$ [14].

Para ilustrar la interpolación en múltiples etapas, se presenta un pequeño código necesario para interpolar una secuencia de datos por un factor $L = L_1 L_2 = 4 \cdot 6 = 24$. Se llevará a cabo en dos etapas, siendo la primera usada para la conformación de pulsos (usando la raíz del coseno alzado).

Código 4.12 Ejemplo de interpolación en múltiples etapas.

```
datos = randi([0 1], 1, 200).*2-1;

L1=4
% generacion del pulso coseno alzado como en la seccion anterior
% [ . . . ]
b_1 = pulso.generar_coeficientes();

L2 = 6;
b_2 = L2.*[0.0008 0.0031 0.0077 0.0154 0.027 0.0432 0.0617 0.0802
0.0965 0.108 0.1127 0.108 0.0965 0.0802 0.0617 0.0432 0.027 0.0154
0.0077 0.0031 0.0008];

interp1 = filtrado_polifase(datos, b_1, L1, 1, 0);
interp2 = filtrado_polifase(interp1, b_2, L2, 1, 0);
```

Observando la siguiente figura, es posible notar como, para una misma cantidad de muestras representadas, el espacio entre dichas muestras es mayor en *(b)* que en *(a)*.

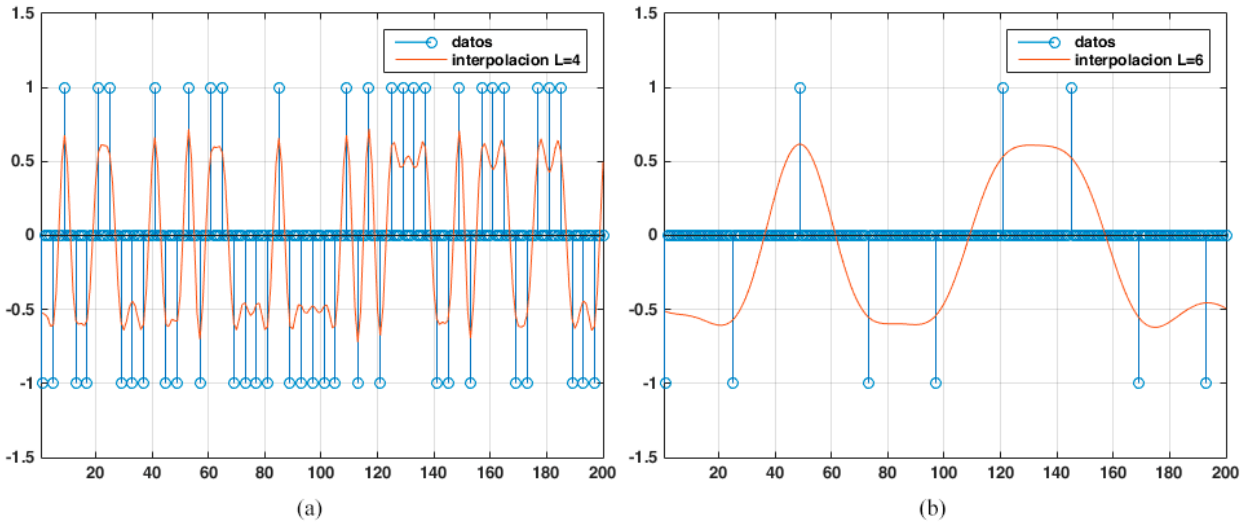


Figura 4.7 (*a*) Primera etapa de interpolación $L_1 = 4$. (*b*) Segunda etapa de interpolación $L_2 = 6$.

4.1.3 Filtros cascaded integrator-comb (CIC)

La forma de obtener los coeficientes del filtro CIC es muy sencilla. Se comienza generando un total RM coeficientes todos con valor 1. Esta secuencia inicial se convolucionará tantas veces como indique el parámetro N, de manera que el filtro resultante tendrá un número final de coeficientes que se puede obtener como:

$$NRM - (N - 1) \tag{24}$$

Código 4.13 Código de la función `gen_coef_cic.m`.

```
function [coef_normalizados,retardo] = gen_coef_cic(N, R, M)
    if (nargin < 3)
        M=1;
    end

    RM=R*M;

    hcic_orden1 = ones(1,RM);
    coeficientes = hcic_orden1;

    if (N >= 1)
        for ordenN=2:N
            coeficientes = conv(coeficientes,hcic_orden1);
        end
    end

    % Ganancia del filtro
    g=sum(coeficientes);
    coef_normalizados=coeficientes./g;

    retardo = mean(grpdelay(coef_normalizados));
end
```

4.4. Subida en frecuencia

De acuerdo con la ecuación (22), es necesario generar la exponencial compleja que funcionará como portadora de los datos que se quieren transmitir. Será necesario definir una frecuencia de portadora f_c que cumpla con la ecuación (21). Puesto que la portadora se está generando en un entorno discreto, esta estará debidamente muestreada a la frecuencia de muestreo f_s obtenida tras la interpolación en múltiples etapas.

Siguiendo con el ejemplo anterior (esto es, empleando el mismo factor de interpolación y datos interpolados), el siguiente código muestra cómo generar la portadora.

Código 4.14 Generación de la portadora a frecuencia f_c.

```
fs        =                    4e6*L1*L2; % 4MHz * 4 * 6 = 96 MHz
fc        =                          8e6;
n         =    0:(length(datos_interp)-1);
portadora =   exp(1i.*(2.*pi.*fc./fs.*n));
datos_tx  = real(datos_interp.*portadora);
```

Con una frecuencia de muestreo $f_s = 96\,MHz$ y una frecuencia de portadora $f_c = 8\,MHz$, se tendrán un total de $\frac{f_s}{f_c} = 12\,muestras$ por período de la portadora.

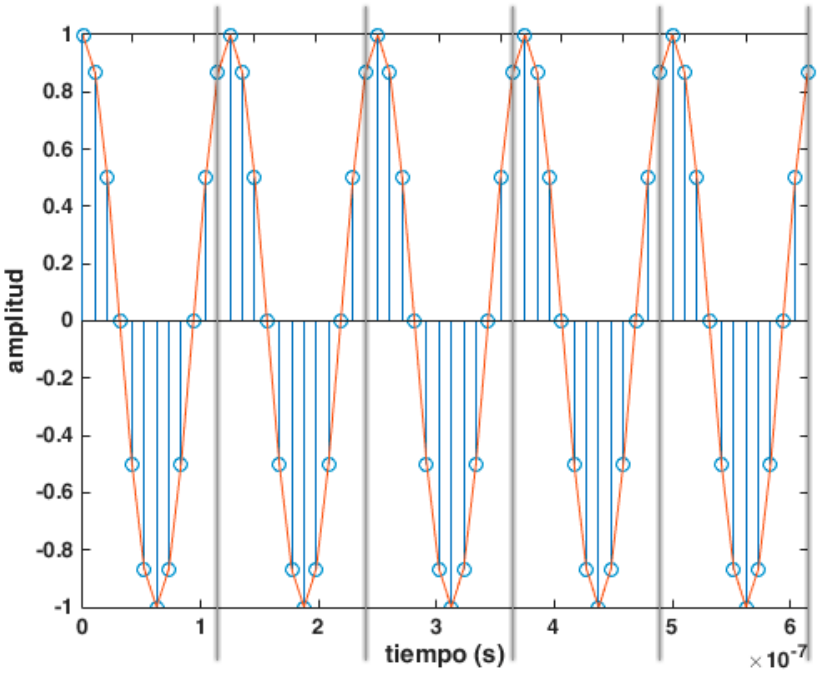


Figura 4.8 Portadora a frecuencia $f_c = 8\ MHz$ muestreada a $f_s = 96\ MHz$.

Desde el punto de vista del dominio frecuencial, inicialmente el ancho de banda que ocupan los datos en banda base (tras la conformación de pulso de coseno alzado y el interpolado en múltiples etapas) se puede calcular como:

$$B = \frac{R_s}{2}(1+\beta) = \frac{R_b}{2k}(1+\beta) \tag{25}$$

Para una modulación BPSK ($k = 1$), un $R_b = 4 \cdot 10^6\ bps$, y un factor de *roll-off* $\beta = 0.6$, realizando las sustituciones oportunas en la ecuación anterior, se tiene que $B = 3.2\ MHz$.

Tras realizar la subida a paso banda donde la frecuencia central es $f_c = 8\ MHz$, se tiene que el ancho de banda ocupado corresponde con $2B = 6.4\ MHz$.

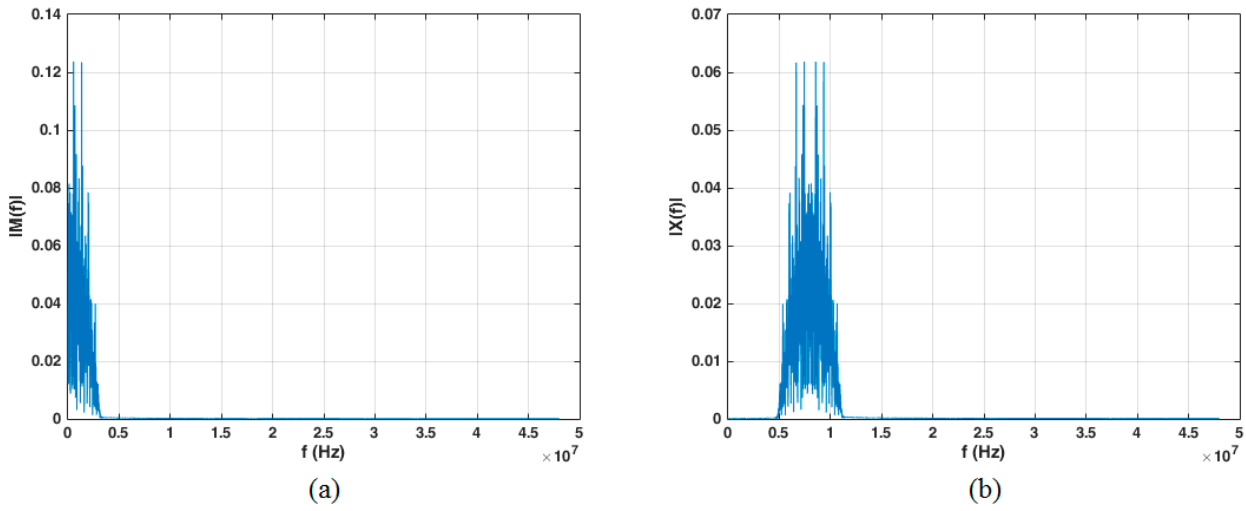


Figura 4.9 (*a*) Transformada de Fourier en banda base. (*b*) Datos en paso banda con $f_c = 8\ MHz$.

5 CONSIDERACIONES SOBRE EL CANAL

A la hora de estudiar las prestaciones de un sistema transmisor-receptor, es necesario someterlo a diferentes condiciones tratando de imitar la situación real de funcionamiento. Por ello resulta indispensable modelar el canal por el que se transmitirá la información hasta el receptor.

En este capítulo se considerarán dos fenómenos de gran importancia: el ruido y la propagación multitrayecto. Ambos fenómenos afectan negativamente al comportamiento del sistema y siempre estarán presentes (en menor o mayor medida) en una aplicación real. Por lo que, además de presentar las bases teóricas de ambos, se propondrá una forma de implementación en MATLAB.

5.1. Ruido

Generalmente el canal que atraviese la señal de interés será un canal ruidoso. Mediante su modelado en el entorno de simulación será posible averiguar cómo afectará al sistema. Para ello se considerará que el ruido que afecta a la señal transmitida es un ruido AWGN:

- Se suma a la señal.
- Afecta por igual a todas las frecuencias.
- Se puede modelar como una variable aleatoria $\sim\mathcal{N}(\mu_N = 0, \sigma_N^2 = \frac{N_0}{2})$.

Dependiendo de la potencia del ruido respecto a la de la señal de interés, este afectará en mayor o menor medida al resultado final. Para estudiar su efecto, se suele representar la BER en función de la relación E_b/N_0 (relación energía de bit a densidad espectral de ruido). Para simular un valor concreto de E_b/N_0, es necesario generar el ruido de la forma correcta; esto es, a partir de la potencia de la señal P_S y de un valor E_b/N_0 dado, se obtendrá la potencia de ruido con la que se generarán un conjunto de números aleatorios bajo la distribución $\sim \mathcal{N}(\mu_N = 0, \sigma_N^2)$.

Para ello, es necesario acudir a otra relación muy utilizada en este ámbito que es la SNR. Esta se define como el cociente entre la potencia de señal y la de ruido (a menos que se indique entre paréntesis lo contrario, las unidades empleadas son las naturales) que, para señales de media cero, equivalen a la varianza de estas:

$$SNR = \frac{P_S}{P_N} \tag{26}$$

La relación que existe entre SNR y E_b/N_0, para una señal $s(t)$ real, es [15]:

$$\frac{E_b}{N_0} = \frac{SNR}{k}\frac{T_{simbolo}}{2T_{muestreo}} \tag{27}$$

donde $k = \log_2 M$ es el número de bits en un símbolo, $T_{simbolo}$ es el tiempo de símbolo y $T_{muestreo}$ es el periodo de muestreo. Sustituyendo en la ecuación el valor de SNR y teniendo en cuenta que al ser señales de media cero $P_S = \sigma_S^2$ y $P_N = \sigma_N^2$, entonces:

$$\sigma_N^2 = \frac{\sigma_S^2}{k\frac{E_b}{N_0}} \frac{T_{simbolo}}{2T_{muestreo}} \tag{28}$$

El cociente $T_{simbolo}/T_{muestreo}$ indica el número de muestras por símbolo (esto se hace para tener en cuenta el factor de interpolación, si lo hubiere. Por lo que, dicho cociente será igual al factor de interpolación final L).

Una vez conocida la forma de obtener σ_N^2 a partir de E_b/N_0, es necesario generar el ruido de forma apropiada en MATLAB. Para ello es posible usar la función `randn`, la cual genera números aleatorios a partir de una distribución normal de media cero y varianza unidad. Por ello habrá que adaptar el resultado a la varianza σ_N^2 deseada. Si la variable `varNd` contiene el valor deseado A, entonces solo habrá que hacer `sqrt(varNd).*randn(…)`. Esto se puede demostrar de forma que, si la varianza de $\mathcal{N}$ se define como:

$$\sigma_N^2 = \frac{1}{N}\sum_{i=1}^{N} n_i^2 \tag{29}$$

siendo n_i cada uno de los valores que conforman el vector ruido (por defecto, $\sigma_N^2 = 1$). Si, a continuación, se define un nuevo vector ruido siendo $n_{deseado,i} = \sqrt{A} \cdot n_i$ cada uno de los elementos de dicho vector, entonces, al calcular la varianza:

$$\frac{1}{N}\sum_{i=1}^{N} n_{deseado,i}^2 = \frac{1}{N}\sum_{i=1}^{N} A \cdot n_i^2 = A\sigma_N^2 = A = \boldsymbol{\sigma_{Nd}^2} \tag{30}$$

La función `gen_ruido` devolverá un vector del mismo tamaño que la señal, con el ruido correspondiente a un nivel E_b/N_0 (dB) dado como parámetro. Este vector se deberá sumar a la señal generada por el transmisor.

Código 5.1 Generación de ruido AWGN.

```
function [n, varN] = gen_ruido(datos, EbN0_dB, k, factor_inttotal)

    EbN0 = 10^(EbN0_dB/10);
    varS = var(datos);

    varN = varS*factor_inttotal / (2*EbN0*k);

    n = randn(size(datos)) * sqrt(varN);
end
```

5.2. Propagación multitrayecto

Otro de los problemas que afectan a la señal de interés al transmitirse por el canal es la propagación multitrayecto [16]. Este efecto ocurre cuando la señal que emite el transmisor no sigue (únicamente) una ruta directa para alcanzar el receptor, sino que además llegan a éste múltiples copias de la señal atenuadas y retrasadas en el tiempo, teniendo como consecuencia indeseada la aparición de interferencia intersímbolo. La propagación multitrayecto ocurre tanto en canales inalámbricos como cableados.

Una forma de modelar este fenómeno de forma aproximada es mediante la construcción de un filtro FIR. En el caso más simple, suponiendo un canal lineal e invariante en el tiempo (LTI) con propagación multitrayecto, se puede describir como:

$$h_c(t) = \sum_{k=0}^{N-1} \alpha_k \delta(t - \tau_k) \leftrightarrow H_c(\omega) = \sum_{k=0}^{N-1} \alpha_k e^{-j\omega\tau_k} \tag{31}$$

Si $s(t)$ es la señal que se obtiene a la salida del transmisor, entonces $r(t)$ será la recibida por el receptor tras atravesar el canal, suponiendo que no hay ruido:

$$r(t) = s(t) * h_c(t) = \sum_{k=0}^{N-1} \alpha_k s(t - \tau_k) \tag{32}$$

donde $\tau_0, \tau_1 \ldots \tau_{N-1}$ representan los N retrasos y $\alpha_0, \alpha_1 \ldots \alpha_{N-1}$ las atenuaciones de la señal.

La diferencia $D = \tau_{N-1} - \tau_0$ se conoce con el nombre de ***delay spread*** (extensión del retraso) que representa la diferencia entre el tiempo de llegada de la componente multitrayecto más tardía y la más temprana. Su inverso, es decir, $B = \frac{1}{D} = \frac{1}{\tau_{N-1} - \tau_0}$, se denomina ***coherence bandwidth*** (ancho de banda de coherencia).

Se puede asumir que, si el ancho de banda de la señal transmitida es inferior al ancho de banda de coherencia, el comportamiento en frecuencia del canal será aproximadamente plano; mientras que, si ocurre lo contrario, el ancho de banda de la señal es mayor que el de coherencia, la señal sufrirá desvanecimiento selectivo en frecuencia.

Puesto que lo interesante sería poder modelar un canal multitrayecto en un entorno de simulación [17], será necesario, mediante aproximación, convertir el sistema de la ecuación anterior a tiempo discreto. Así, para un periodo de muestreo T_s y evaluando la respuesta en frecuencia del canal de la ecuación (31), es posible obtener la transformada Z teniendo en cuenta que $z = e^{j\omega T_s}$:

$$H_c(z) = \sum_{k=0}^{N-1} \alpha_k z^{-\frac{\tau_k}{T_s}} \tag{33}$$

Se va a considerar, además, que existe un camino principal que no atenúa la señal y que llega sin retraso al receptor, esto es, $\alpha_0 = 1$ y $\tau_0 = 0$. Entonces:

$$H_c(z) = 1 + \alpha_1 z^{-\frac{\tau_1}{T_s}} + \alpha_2 z^{-\frac{\tau_2}{T_s}} + \cdots + \alpha_{N-1} z^{-\frac{\tau_{N-1}}{T_s}} \tag{34}$$

Suponiendo que los retrasos están ordenados de menor a mayor, es decir, $\tau_1 < \tau_2 < \cdots < \tau_{N-1}$ y que T_s es mucho menor que el menor de los retrasos τ_1, es posible aproximar las fracciones τ_1/T_s, τ_2/T_s, …, τ_{N-1}/T_s por números enteros $m_1, m_2, \ldots, m_{N-1}$:

$$H_c(z) = 1 + \alpha_1 z^{-m_1} + \alpha_2 z^{-m_2} + \cdots + \alpha_{N-1} z^{-m_{N-1}} \tag{35}$$

Así resultaría posible implementar el canal como un filtro FIR en tiempo discreto siendo

$$N_c = \frac{\tau_{N-1}}{T_s} + 1 \tag{36}$$

el número total de coeficientes de ese filtro. No todos los N_c coeficientes serán distintos de cero, solo aquellos que coincidan con algunos de los retrasos del sistema real en la ecuación (31).

Código 5.2 Generación de canal multitrayecto.

```
if (sum(Ts > retrasos) > 0)
    error('Los retrasos deben ser (mucho) mayores que Ts');
end

%se ordenan los retrasos de menor a mayor
[ret, idx] = sort(retrasos);
g = g(idx);

% numero de coeficientes
Nc = round(ret(end)/Ts)+1;
coef = zeros(1,Nc);
coef(1) = 1;

% posiciones en el filtro
enteros = round(ret/Ts)+1;
coef(enteros) = g;

datos_multi = filter(coef, 1, datos);
```

Para generar el filtro FIR que modela al canal multitrayecto se necesitarán tres entradas: un vector `retrasos` que contengan los diferentes retrasos, un vector `g` que indica las ganancias asociadas a los retrasos y el tiempo de muestreo `Ts`.

Inicialmente los retrasos se reordenan de menor a mayor (esto incluye también a las ganancias de manera que su posición o índice del vector coincida con los retrasos) y se calcula el número de *taps* que tendrá el canal según lo visto en la ecuación (36).

En la generación del canal se supone que el primer *tap* o coeficiente tiene una ganancia de 1, esto es, la señal llega al receptor sin sufrir atenuación ni retraso (además del resto de copias de esta). Por lo que, a la hora de formar los vectores `retrasos` y `g`, no es necesario tener en cuenta este caso.

Como ejemplo, si se quisiera generar un canal multitrayecto donde las copias retrasadas llegarían $500\ ns$ y $800\ ns$ después de la principal y con unas ganancias de 0.7 y 0.1 respectivamente, teniendo una frecuencia de muestreo de $10\ MHz$:

Código 5.3 Ejemplo de generación de canal multitrayecto.

```
Ts = 100e-9;
retrasos = [500e-9 800e-9];
g = [0.7 0.1];

[~,coef]=gen_multitrayecto(1,g,retrasos,Ts)
```

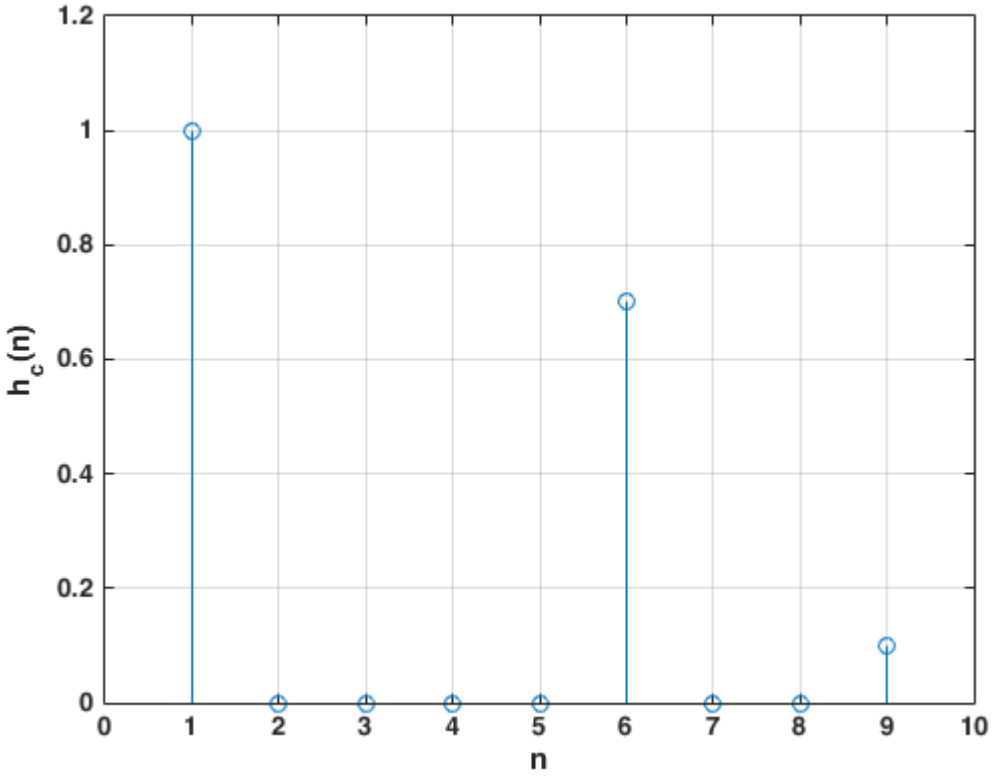


Figura 5.1 Respuesta al impulso del canal multitrayecto de ejemplo.

6 DISEÑO DEL RECEPTOR

El destino de la información transmitida es el receptor. Este no solo se encarga de "deshacer" los pasos que ha dado el transmisor para procesar la información y hacerla apta para su transmisión, sino que además deberá hacer frente a los efectos introducidos por atravesar el canal de comunicación.

Es por esta razón por la que el diseño del receptor resulta de mayor complejidad que el del transmisor, introduciendo nuevos bloques como los de sincronización o los de ecualización. También se tratará desde el punto de vista teórico el resto de bloques del receptor, como la aplicación del filtro adaptado o la toma decisión sobre el símbolo transmitido.

6.1. Sincronización

6.1.1 Lazo de seguimiento de fase (PLL)

Conviene introducir de forma previa el concepto del ***phase-locked loop*** o lazo de seguimiento de fase (PLL) pues resultará un componente esencial a la hora de realizar la sincronización de portadora y de tiempo.

Un PLL es un sistema de control realimentado (de lazo cerrado) que lleva a cabo el seguimiento y estimación de la fase de referencia de una señal de entrada, generando para ello una señal que guarda cierta relación con la primera. Su cometido es "engancharse" a la entrada de manera que la diferencia entre la fase de referencia (que es desconocida) y la generada por el PLL sea cero.

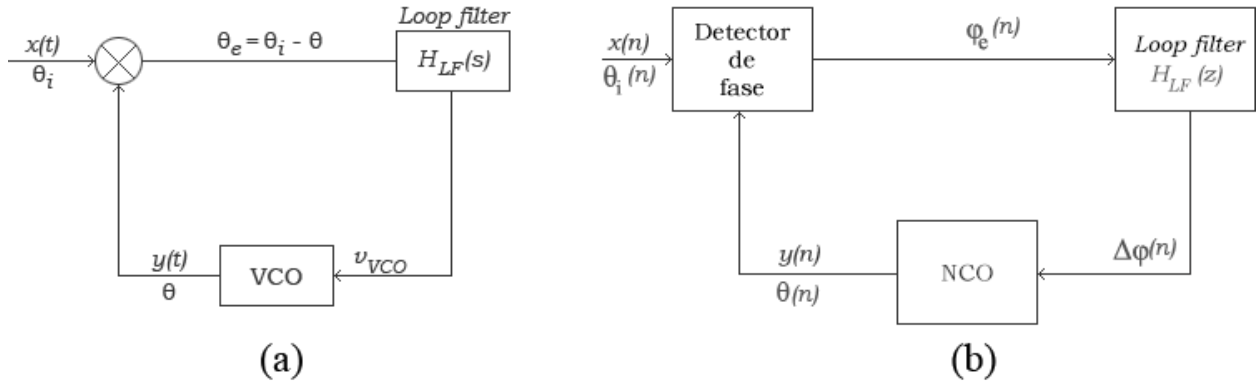


Figura 6.1 (*a*) Diagrama de un PLL analógico. (*b*) Diagrama de un PLL digital.

La estructura de un PLL se resume en tres bloques básicos:

- **Detector de fase**: recibe la señal de entrada $x[n]$ con una fase de referencia θ_i y la señal generada por el controlador de fase $y[n]$ con fase θ. Su función es generar la diferencia de fase entre θ_i y θ. La forma de hacerlo depende de la aplicación y la tecnología que se use en la implementación.
- ***Loop filter* (filtro de lazo)**: por lo general suele ser un filtro paso bajo y contiene integradores perfectos (con un polo a frecuencia cero, además de otros para mejorar la estabilidad). Puesto que

las señales de entrada habitualmente estarán afectadas por ruido y/o interferencias, con este filtro se intenta reducir el efecto de estos sobre la señal de interés.

- **Controlador de fase**: genera la salida del PLL y ajusta la fase. En analógico sería un VCO (*voltage-controlled oscillator*) el encargado de esta función, mientras que en digital se suele conocer con el nombre de NCO (*numerically controlled oscillator*).

Una característica importante de un PLL es el orden, el cual queda determinado por el *loop filter*. En general, en un PLL debería haber al menos un integrador y, debido a que el controlador de fase contiene uno (haciendo las veces de acumulador de fase) ocurre que el *loop filter* es simplemente una constante en el caso del PLL de orden uno (el denominador de la función de transferencia resultante del PLL es un polinomio de primer orden).

Se revisarán las ecuaciones relativas a los PLL de primer y segundo orden, tanto en su variante analógica como digital

6.1.1.1 PLL analógico

La entrada al PLL [18] es una señal sinusoidal $x(t) = \text{sen}\,(2\pi ft + \theta_i)$ y la salida es $y(t) = \cos\,(2\pi ft + \theta)$. El detector de fase más simple que se puede usar es un multiplicador, de manera que, al multiplicar la señal de entrada con la generada por el PLL, se obtiene:

$$z(t) = x(t)y(t) = \frac{1}{2}[\text{sen}(4\pi ft + \theta_i + \theta) + \text{sen}(\theta_i - \theta)] \tag{37}$$

El error de fase está en el segundo término de la expresión anterior $\theta_e = \theta_i - \theta$. El primer término, sin embargo, que es un seno a frecuencia $2f\ Hz$, se eliminará al pasar por el *loop filter* (con efecto paso bajo) con función de transferencia $H_{LF}(s)$.

Si el error de fase está cerca de cero, la salida del *loop filter* será un valor proporcional a dicho error, pues $\text{sen}\,(a) \approx a$ si $0 < a \ll 1$. Dicha salida se envía al VCO, que actúa de controlador de fase en el caso de un PLL analógico. Un VCO espera a la entrada una tensión de control v_{VCO}, la cual emplea para cambiar la frecuencia de la señal de salida de manera proporcional (de manera que se comporta como un integrador perfecto).

Tradicionalmente, en los sistemas de control analógicos se ha empleado la transformada de Laplace para describir las funciones de transferencias en lugar de la de Fourier, por lo que, tras atravesar el *loop filter* se tiene la tensión de control:

$$v_{VCO}(t) = h_{LF}(t) * \theta_e(t) \leftrightarrow V_{VCO}(s) = H_{LF}(s)\Theta_e(s) = H_{LF}(s)[\Theta_i(s) - \Theta(s)] \tag{38}$$

Para poder obtener la función de transferencia del sistema completo $H_{PLL}(s) = \frac{\Theta(s)}{\Theta_i(s)}$ es necesario establecer una relación más entre $v_{VCO}(t)$ y $\theta(t)$. Si se usa la tensión de control para cambiar la frecuencia de salida (teniendo en cuenta la ganancia k_{VCO} introducida por el controlador), eso significa que el cambio en la fase de salida es igual al cambio en frecuencia integrado en el tiempo:

$$\frac{d\theta(t)}{dt} = f_o(t) = k_{VCO}v_{VCO}(t) \tag{39}$$

Así, obteniendo la transformada de Laplace de la expresión anterior:

$$s\Theta(s) = k_{VCO}V_{VCO}(s) = k_{VCO}H_{LF}(s)[\Theta_i(s) - \Theta(s)] \tag{40}$$

Ahora ya es posible obtener la función de transferencia del PLL al completo:

$$H_{PLL}(s) = \frac{\Theta(s)}{\Theta_i(s)} = \frac{k_{VCO}\text{H}_{\text{LF}}(s)}{s + k_{VCO}\text{H}_{\text{LF}}(s)} \tag{41}$$

El orden del PLL dependerá del número de polos de la función de transferencia del *loop filter* $H_{LF}(s)$.

Un PLL de primer orden resulta útil cuando se tiene un error de fase inicial constante o un cambio en escalón de la fase de entrada. En ese caso, $H_{LF}(s) = K_p$, el filtro no es más que una constante, quedando la función de transferencia final como:

$$H_{PLL}(s) = \frac{\Theta(s)}{\Theta_i(s)} = \frac{k_{VCO}K_p}{s + k_{VCO}\mathrm{K_p}} \tag{42}$$

Si la fase de entrada cambiase linealmente con el tiempo (por ejemplo, una función rampa), se produciría un error de fase imposible de corregir en régimen permanente, por lo que en esos casos hay que recurrir a un PLL de segundo orden.

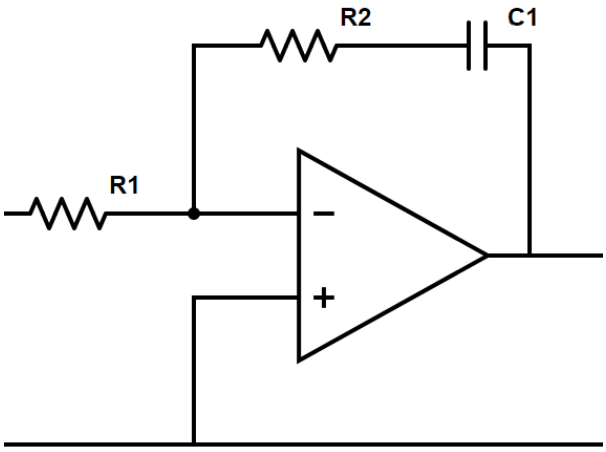


Figura 6.2 Filtro activo con amplificador.

En esta situación, el *loop filter* suele ser un filtro paso bajo RC pasivo o bien empleando componentes activos (usando un amplificador de ganancia A) como el de la **Figura 6.2** con función de transferencia $H_{LF}(s) = \frac{1+s\tau_2}{1/A+s\tau_1}$, donde τ_1 y τ_2 son las constantes de tiempo del circuito. Esta expresión se puede aproximar por $H_{LF}(s) \approx \frac{1+s\tau_2}{s\tau_1}$ cuando $A \gg 1$.

Sustituyendo en la expresión (41) la función de transferencia del *loop filter* para el caso activo:

$$H_{PLL}(s) = \frac{\Theta(s)}{\Theta_i(s)} = \frac{s\tau_2 k_{VCO} + k_{VCO}}{s^2\tau_1 + s\tau_2 k_{VCO} + k_{VCO}} \tag{43}$$

En la literatura esta ecuación se suele expresar en función de la frecuencia natural ω_n y el factor de *damping* ξ, por lo que dividiendo el numerador y denominador por τ_1:

$$H_{PLL}(s) = \frac{\Theta(s)}{\Theta_i(s)} = \frac{(s\tau_2 k_{VCO} + k_{VCO})/\tau_1}{s^2 + s\tau_2 k_{VCO}/\tau_1 + k_{VCO}/\tau_1} = \frac{2\xi\omega_n s + \omega_n^2}{s^2 + 2\xi\omega_n s + \omega_n^2} \tag{44}$$

donde $\omega_n = \sqrt{k_{VCO}/\tau_1}$ y $\xi = \frac{\tau_2}{2}\sqrt{\frac{k_{VCO}}{\tau_1}}$.

Además de ser un PLL de segundo orden, también es un PLL tipo 2 (porque tiene un total de dos integradores perfectos).

6.1.1.2 PLL digital

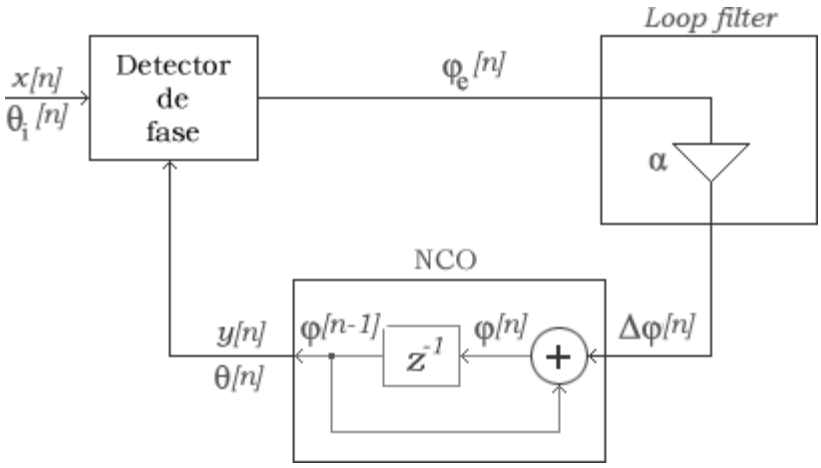


Figura 6.3 Estructura de un PLL digital de primer orden.

Por su parte, un PLL digital [19] tendrá como señal de entrada $x[n]$ cuya fase $\theta_i[n]$ es desconocida. El detector de fase se encarga de calcular la diferencia entre la fase de referencia $\theta_i[n]$ y la generada por el NCO, $\theta[n]$, obteniendo así el error de fase $\varphi_e[n]$. Posteriormente es escalado por la constante α y se denota como el incremento de fase $\Delta\varphi[n] = \alpha\varphi_e[n]$ que se añadirá en el acumulador de fase (integrador del NCO):

$$\varphi[n] = \varphi[n-1] + \Delta\varphi[n] = \varphi[n-1] + \alpha\varphi_e[n] \tag{45}$$

Previo a la puesta en marcha del sistema, el PLL debe ser inicializado con un valor de fase $\varphi(-1)$ (generalmente, cero) que se usará para obtener $\varphi[0]$. La fase de salida del NCO será $\theta[n] = \varphi[n-1]$.

Para comprender cómo opera el PLL digital de primer grado, se propone la siguiente situación: imaginando que el detector de fase se implemente de la forma más simple (e ideal) donde $\varphi_e[n] = \theta_i[n] - \theta[n]$, se tiene que la fase de referencia al inicio es $\theta_i[0] > 0$ y el PLL se inicializa con $\varphi[-1] = 0$. Al procesar el error, se tiene que $\varphi_e[0] > 0$ por lo que, al actualizar el NCO según la ecuación (45), se obtendrá $\varphi[0] > \varphi[-1]$. Suponiendo que $\theta_i[n]$ es constante, se habrá conseguido disminuir la diferencia entre la fase de referencia y la generada por el controlador de fase, de manera que para la siguiente muestra $\varphi_e[1] < \varphi_e[0]$. Cada paso hará que el error de fase disminuya con respecto al anterior, tendiendo a cero. Una vez que el error es nulo, el acumulador de fase deja de actuar y el PLL permanece enganchado.

A partir del esquema es posible obtener la función de transferencia linealizada desde que el error de fase entra al *loop filter* hasta que sale del integrador en el NCO:

$$\Phi(z) = \frac{\alpha}{1 - z^{-1}}\Phi_e(z) \tag{46}$$

Para obtener la función de transferencia global del sistema es necesario definir la manera en que se realiza la detección de fase (que, tal y como se ha comentado, es dependiente de la aplicación). Siguiendo con el ejemplo donde $\varphi_e[n] = \theta_i[n] - \theta[n]$ y teniendo en cuenta que $\theta[n] = \varphi[n-1] \leftrightarrow \Theta(z) = z^{-1}\Phi(z)$:

$$H_{PLL}(z) = \frac{\Theta(z)}{\Theta_i(z)} = \frac{\alpha z^{-1}}{1 - (1-\alpha)z^{-1}} \tag{47}$$

Analizando la expresión anterior se puede deducir que existe un polo real en $z = 1 - \alpha$ y que α controla la posición de dicho polo y, por tanto, el ancho de banda del filtro.

Igual que ocurría con el PLL analógico de primer grado, el digital tiene el mismo inconveniente, y es que es incapaz de corregir un error de fase que incrementa linealmente (una rampa) provocado por un *offset* de frecuencia constante. Para solucionar este problema, se propone el uso de un PLL de segundo grado.

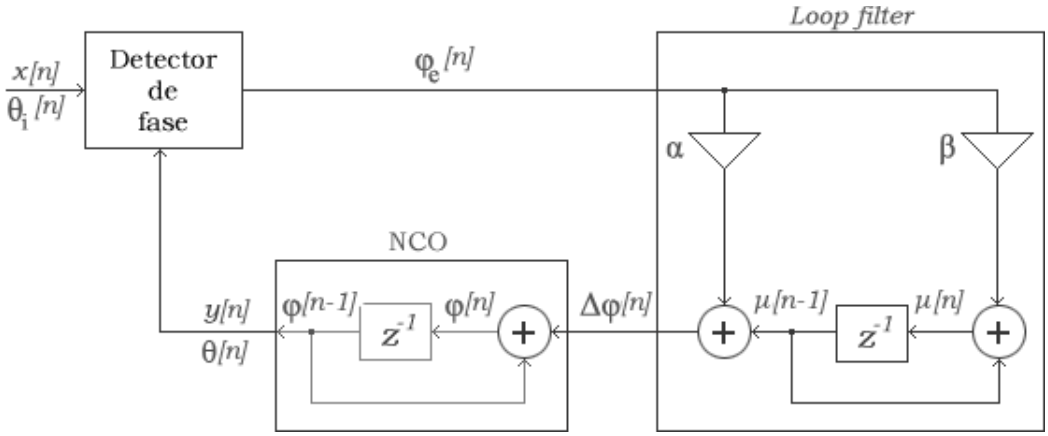


Figura 6.4 Estructura de un PLL digital de segundo orden.

La diferencia con respecto al de primer orden radica únicamente en la adición de una rama adicional en el *loop filter*, la cual incluye un integrador. En este caso serán dos las ecuaciones de actualización del NCO, una para la fase y otra para la frecuencia de salida:

$$\begin{aligned} \mu[n] &= \mu[n-1] + \beta\varphi_e[n] \\ \varphi[n] &= \varphi[n-1] + \Delta\varphi[n] = \varphi[n-1] + \alpha\varphi_e[n] + \mu[n-1] \end{aligned} \tag{48}$$

Al igual que en el caso anterior, se requieren de unos valores de inicialización tanto para la fase como para la frecuencia, $\varphi[-1]$ y $\mu[-1]$. Por lo general ocurre que $\beta \ll \alpha$ (cuando $\beta = 0$, el PLL de segundo orden se comporta como uno de primer orden).

La función de transferencia de la nueva sección añadida al PLL es:

$$\mathrm{M}(z) = \frac{\beta}{1 - z^{-1}}\Phi_e(z) \tag{49}$$

Teniendo en cuenta que $\Theta(z) = z^{-1}\Phi(z)$ y que $\Phi(z) = z^{-1}\Phi(z) + \alpha\Phi_e(z) + z^{-1}\mathrm{M}(z)$, entonces la función de transferencia final quedaría como:

$$H_{PLL}(z) = \frac{\Theta(z)}{\Theta_i(z)} = \frac{\alpha z + (\beta - \alpha)}{z^2 + (\alpha - 2)z + (1 - \alpha + \beta)} \tag{50}$$

6.1.2 Sincronización de portadora

Tras atravesar el canal, la señal llega al receptor. En un primer lugar, la parte *hardware* se encarga de convertir de RF a frecuencia intermedia la señal recibida. Partiendo de lo expuesto en la sección 3.4, para obtener la señal en banda base $s_b(t)$ será necesario emplear un oscilador local en el receptor que generará las señales adecuadas a la frecuencia de portadora exacta con la que lo hizo el transmisor. Estas señales generadas se mezclarán con la recibida, de forma que:

$$\begin{aligned} s_1(t) &= 2\,s(t)\cos(2\pi f_c t) = s_{b,I}(t)[1 + \cos(4\pi f_c t)] - s_{b,Q}(t)\,\mathrm{sen}(4\pi f_c t) \\ s_2(t) &= -2\,s(t)\,\mathrm{sen}(2\pi f_c t) = -s_{b,I}(t)\,\mathrm{sen}(4\pi f_c t) + s_{b,Q}(t)[1 - \cos(4\pi f_c t)] \end{aligned} \tag{51}$$

Tras pasar $s_1(t)$ y $s_2(t)$ por un filtro paso bajo que elimine las componentes generadas al doble de la frecuencia de la portadora, se obtendrían las señales $s_{b,I}(t)$ y $s_{b,Q}(t)$.

Esto funcionaría a la perfección si la frecuencia (y fase) de la portadora generada por el oscilador local del transmisor coincidiera exactamente con la generada por el oscilador del receptor. Sin embargo, esto no ocurre en la realidad. Además, el hecho de atravesar el canal de transmisión supondrá que la frecuencia con la que inicialmente se transmitió difiera de la que recibirá el receptor. En modulaciones coherentes como la M-PSK es necesario la existencia de un sistema capaz de detectar las diferencias entre la señal recibida y la generada por el receptor y actuar en consecuencia para hacer que el error sea lo más pequeño posible (idealmente, cero). Esto es lo que se conoce como recuperación o sincronización de portadora.

En la mayoría de los sistemas, la información de fase y frecuencia de la portadora no se puede extraer de la señal recibida mediante operaciones lineales.

Si a la entrada del receptor se tiene $y(t) = Re[x(t)e^{j(2\pi f_c t+\theta)}]$ y se realiza la media de la misma $E[y(t)] = Re\{E[x(t)e^{j(2\pi f_c t+\theta)}]\} = Re\{\bar{x}e^{j(2\pi f_c t+\theta)}\} = \bar{x}\cos(2\pi f_c t + \theta)$, sería relativamente fácil extraer información sobre la portadora si $\bar{x} \neq 0$. Sin embargo, transmitir con $\bar{x} \neq 0$ no resulta eficiente porque parte de la potencia de transmisión se dedica a la portadora (se suele usar supresión de portadora de manera que no se dedique potencia a la transmisión de esta). Así, no es posible obtener la información deseada promediando $y(t)$, pero aprovechando que $y(t)$ es cicloestacionaria [20], dicha información se puede extraer mediante operaciones no lineales.

6.1.2.1 Squaring loop

Esta técnica puede ser empleada en modulaciones DSB/SC (como BPSK y PAM) con $x(t) \in \mathbb{R}$. La información de la fase se puede extraer elevando al cuadrado la señal de entrada $y(t)$ para generar un segundo armónico a la frecuencia $2f_c$.

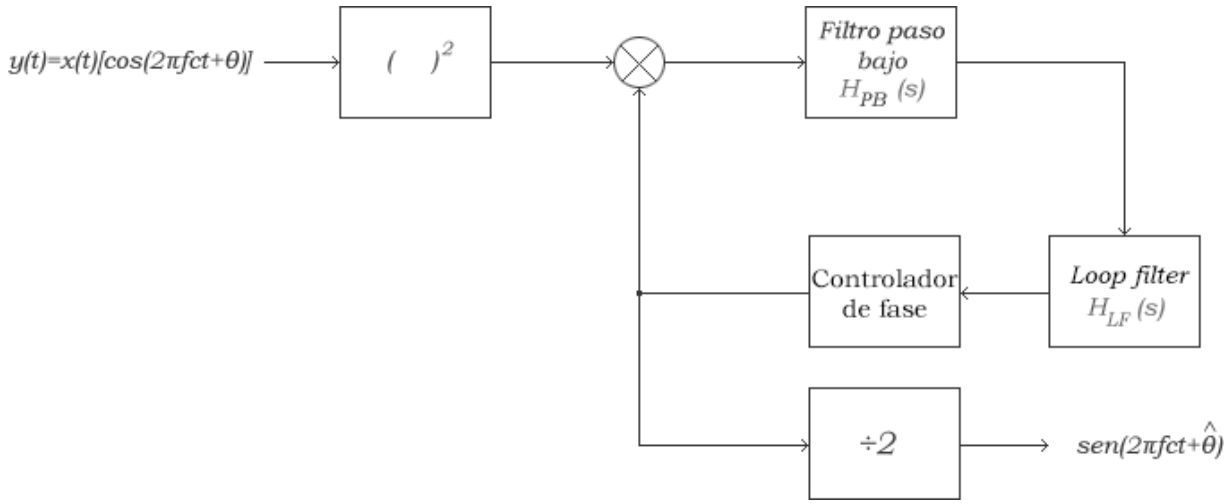


Figura 6.5 Diagrama de bloques para la recuperación de portadora mediante la técnica *squaring loop*.

De esta forma, aunque $E[x(t)] = \bar{x} = 0$, se tendrá que:

$$E[y^2(t)] = E\{[x(t)\cos(2\pi f_c t + \theta)]^2\} = \frac{1}{2}E[x^2(t)][1 + \cos(4\pi f_c t + 2\theta)] \tag{52}$$

Como $E[x^2(t)] > 0$, será posible obtener la frecuencia y fase de la portadora haciendo uso de un PLL cuya salida estará al doble de la frecuencia de la portadora.

Siguiendo el diagrama de la **Figura 6.5**, lo primero es elevar la señal de entrada al cuadrado, de manera que el resultado es:

$$y^2(t) = \frac{1}{2}x^2(t)[1 + \cos(4\pi f_c t + 2\theta)] \tag{53}$$

El PLL genera una señal sinusoidal al doble de la frecuencia (y fase) de la portadora $y_{PLL}(t) = \text{sen}\,(4\pi f_c t + 2\hat{\theta})$ que, tras mezclarlo con la señal de entrada al cuadrado queda como:

$$y^2(t)y_{PLL}(t) = \frac{x^2(t)}{2}\left[\text{sen}\left(4\pi f_c t + 2\hat{\theta}\right) + \frac{1}{2}\text{sen}\,(8\pi f_c t + 2\theta + 2\hat{\theta})\right] + \frac{x^2(t)}{4}\text{sen}\,[2(\theta - \hat{\theta})] \tag{54}$$

Posteriormente pasa por un filtro paso bajo que elimina las componentes en $4f_c$ y $8f_c$, resultando en:

$$\frac{x^2(t)}{4}\text{sen}\,[2(\theta - \hat{\theta})] \tag{55}$$

El *loop filter* se encargará de eliminar las altas frecuencias, entregando al controlador de fase una expresión proporcional al error. Si $2(\theta - \hat{\theta}) \ll 1$, entonces $\text{sen}\,[2(\theta - \hat{\theta})] \approx 2(\theta - \hat{\theta})$.

6.1.2.2 Lazo de costas

Una alternativa a la técnica anterior es el lazo de Costas, desarrollado por John Costas [21]. A diferencia del *squaring loop*, la salida del NCO va a la misma frecuencia que la de entrada (y no al doble), esto es, $y_{PLL}(t) = \cos\,(2\pi f_c t + \hat{\theta})$.

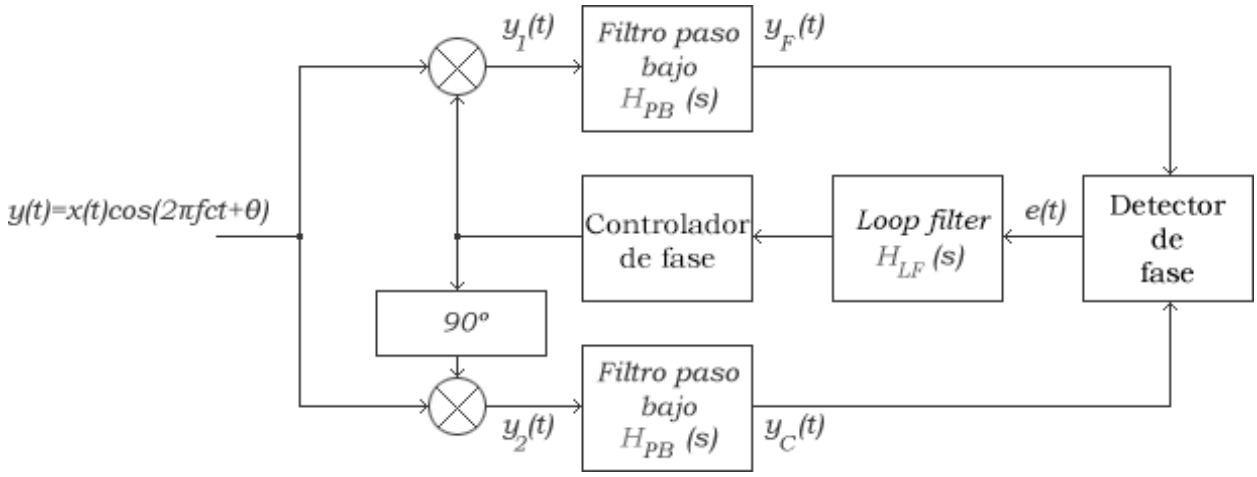


Figura 6.6 Diagrama de bloques para la recuperación de portadora mediante el lazo de Costas.

Se distinguen dos ramas que son idénticas, siendo la única diferencia la señal con la que se multiplica la entrada. De manera que:

$$\begin{aligned} y_1(t) &= x(t)\cos\,(2\pi f_c t + \theta)\cos\,(2\pi f_c t + \hat{\theta}) \\ y_2(t) &= x(t)\cos(2\pi f_c t + \theta)\cos\left(2\pi f_c t + \hat{\theta} + \frac{\pi}{2}\right) \\ &= -x(t)\cos(2\pi f_c t + \theta)\,\text{sen}\left(2\pi f_c t + \hat{\theta}\right) \end{aligned} \tag{56}$$

Desarrollando las identidades trigonométricas de las expresiones anteriores:

$$\begin{aligned} y_1(t) &= \frac{x(t)}{2}[\cos\left(4\pi f_c t + \theta + \hat{\theta}\right) + \cos\,(\theta - \hat{\theta})] \\ y_2(t) &= \frac{x(t)}{2}[-\text{sen}\left(4\pi f_c t + \theta + \hat{\theta}\right) + \text{sen}\,(\theta - \hat{\theta})] \end{aligned} \tag{57}$$

Con el fin de eliminar las componentes generadas al doble de la frecuencia de la portadora, se hace pasar las señales $y_1(t)$ y $y_2(t)$ a través de un filtro paso bajo (idénticos) que elimine las altas frecuencias, de forma que:

$$y_F(t) = \frac{x(t)}{2}\cos(\theta - \hat{\theta})$$
$$y_C(t) = \frac{x(t)}{2}\text{sen}(\theta - \hat{\theta}) \quad (58)$$

Posteriormente, estas señales entrarían al detector de fase. La implementación de este suele variar, siendo la más típica:

$$e(t) = y_F(t)y_C(t) = \frac{x(t)}{2}\cos(\theta - \hat{\theta})\frac{x(t)}{2}\text{sen}(\theta - \hat{\theta}) = \frac{x^2(t)}{8}\text{sen}(2\theta - 2\hat{\theta}) \quad (59)$$

Finalmente, $e(t)$ atravesaría el *loop filter* (que, como se mencionó con anterioridad, actúa de filtro paso bajo) y quedaría $K\text{sen}(2\theta - 2\hat{\theta})$, siendo K un valor de continua. Si ocurre que $2\theta - 2\hat{\theta}$ tiende a 0, entonces $\text{sen}(2\theta - 2\hat{\theta}) \approx 2\theta - 2\hat{\theta}$, por lo que la entrada del NCO sería aproximadamente $2K(\theta - \hat{\theta})$, es decir un valor proporcional a la diferencia de fase.

Estando el PLL enganchado a la señal de entrada, es posible proceder tal y como se indica al comienzo de la sección 6.1.2 para realizar la conversión a banda base y poder continuar con el procesado de la misma.

6.1.3 Sincronización de símbolo o tiempo

¿Cuál es el mejor instante para muestrear la señal recibida? La sincronización de símbolo o de tiempo (o *timing recovery*) es otro de los grandes problemas que requieren de solución en los receptores de los sistemas de comunicación. El receptor, además de conocer a qué frecuencia se muestrea la salida del filtro adaptado, debe saber también dónde tomar la muestra dentro de cada periodo de símbolo.

La elección de ese instante de muestreo dentro del intervalo de símbolo de duración $T = 1/R_s$ es lo que se conoce como fase de tiempo o *timing phase*. La mejor fase de tiempo se correspondería con el instante de tiempo dentro del periodo de símbolo donde la salida del filtro receptor es máxima.

En general se utilizan tres esquemas básicos para la sincronización de tiempo:

- Mediante procesado analógico se decide cuándo tomar las muestras.
- Mediante procesado digital se decide cuándo tomar las muestras.
- Las muestras se toman con un reloj independiente y en un procesado digital posterior se escogen las muestras oportunas.

La tendencia hacia lo digital hace que normalmente se consideren los dos últimos para implementar en sistemas de comunicación.

Los algoritmos existentes para sincronización de tiempo son numerosos, aunque pueden clasificarse según dos grandes categorías:

- ***Feedforward*** (bucle abierto): estos están enfocados en una sincronización por bloques, operando en muestras de un cierto número de símbolos en un momento dado (normalmente con símbolos conocidos). Solo emplea el estado actual del sistema, pudiendo resultar su implementación computacionalmente costosa. Suele emplearse en comunicaciones que usen un modo ráfaga o como inicialización de la sincronización.
- ***Feedback*** (bucle cerrado): se centran en la sincronización por flujo, operando de forma inmediata en la muestra o símbolo entrante. Se utiliza en situaciones donde existe un flujo continuo de símbolos, o después de la inicialización de la sincronización.

Es común en estos algoritmos implementar bloques de remuestreo (*resampling*) y estimación de error de tiempo.

6.1.3.1 Sincronización Early-Late Gate

Uno de los métodos más empleados en sincronización de tiempo es el Early-Late Gate (E-L Gate), el cual explota las propiedades de simetría de la señal a la salida del filtro adaptado.

Por simplicidad, se considera un pulso rectangular $p(t)$ como el de la **Figura 6.7** (*a*), con una duración de T segundos. La salida del filtro adaptado $y(t)$ tendrá su máximo en $t = T$, siendo este el tiempo ideal para realizar el muestreo (en el pico de la función de autocorrelación del pulso $p(t)$).

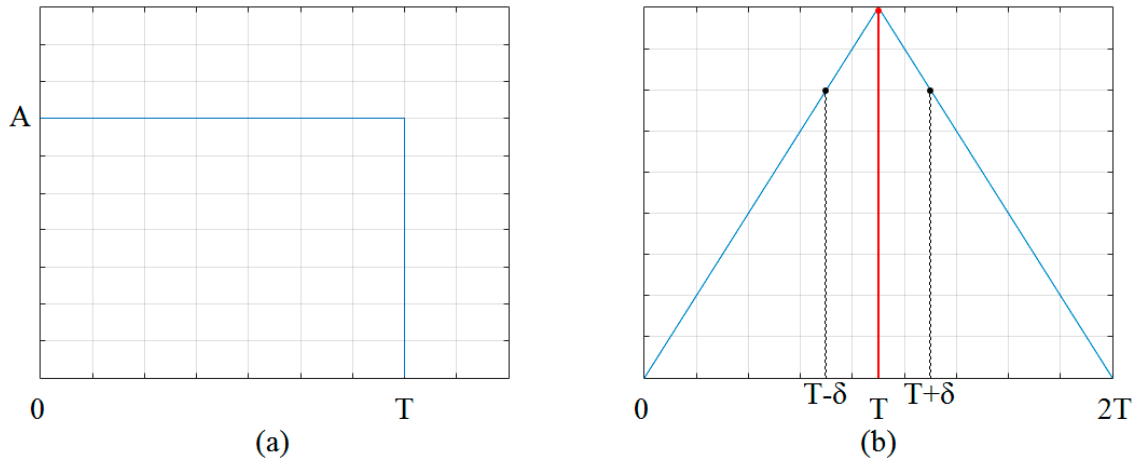


Figura 6.7 (*a*) Pulso rectangular. (*b*) Salida del filtro adaptado.

Sin embargo, cuando hay ruido no resulta tan sencillo identificar el valor máximo, así que la sincronización E-L Gate propone tomar una muestra antes y después de T, es decir, en $T - \delta$ (designada como *early sample*) y $T + \delta$ (llamada *late sample*), siendo $0 \leq \delta \leq T$. A partir de estas muestras se genera el error de tiempo comparando las amplitudes a la salida del filtro y se actuará en consecuencia aumentando o disminuyendo el siguiente instante de muestreo:

- Si $|y(T - \delta)| > |y(T + \delta)|$, el muestreo se ha realizado demasiado tarde, por lo que el próximo muestreo se adelantará una fracción de δ.
- Si $|y(T - \delta)| = |y(T + \delta)|$, no es necesario ajustar el próximo instante de muestreo.
- Si $|y(T - \delta)| < |y(T + \delta)|$, el muestreo se ha realizado demasiado pronto, por lo que el próximo muestreo se atrasará una fracción de δ.

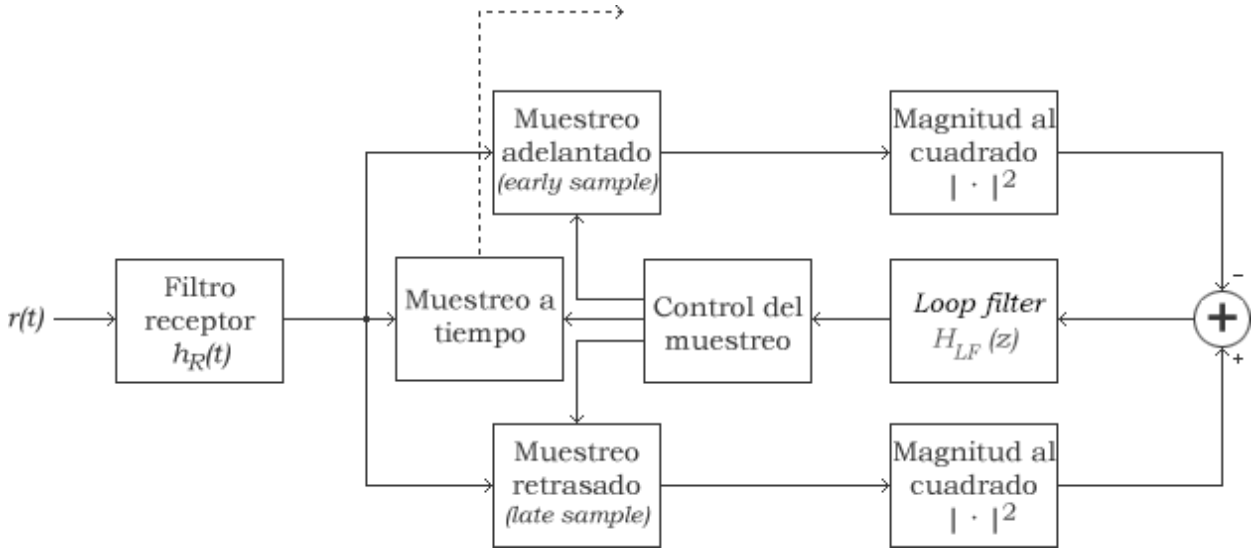


Figura 6.8 Diagrama de bloques de la sincronización Early-Late Gate.

La estructura de un sincronizador E-L Gate es un sistema de control en bucle cerrado con un PLL como el que se puede ver en la figura anterior, pudiendo ser implementados tanto en analógico como en digital.

Un algoritmo muy similar al E-L Gate que ofrece un mejor rendimiento en situaciones de alto SNR es el propuesto por Gardner [22], que funciona principalmente con las modulaciones BPSK y QPSK.

6.2. Diezmado en múltiples etapas

La técnica de diezmado es similar a la de interpolación, pero en sentido inverso. Si se quiere diezmar una señal de entrada por un factor M, entonces habría que realizar:

1. **Filtrado paso bajo.** Puesto que lo que se pretende es reducir la frecuencia de muestreo, el primer paso consistiría en aplicar un filtro *anti-aliasing* que asegure que se cumplirá el teorema del muestreo de Nyquist para la nueva frecuencia de muestreo, eliminando las altas frecuencias (concretamente, aquellas superiores a $\frac{f_s^{nueva}}{2}$).

2. ***Downsample* de la señal de entrada**. Comenzando desde la primera, se conservará cada M-ésima muestra, descartando así las $M-1$ muestras intermedias, de manera que disminuye la frecuencia de muestreo en un factor M (que debe ser un número entero).

   Cuando se eliminan las $M-1$ muestras entre cada muestra de interés, el espectro original que iba de $-\frac{f_s}{2}$ a $\frac{f_s}{2}$, ahora se ve reducido al rango comprendido entre $-\frac{f_s}{2M}$ y $\frac{f_s}{2M}$. Es aquí cuando se produciría el efecto indeseado de *aliasing* si en el paso anterior no se hubieran eliminado las frecuencias superiores a $\frac{f_s}{2M}$.

Al igual que se explicó en la implementación del transmisor para la interpolación, es posible aplicar un filtrado polifase para hacer el diezmado una tarea menos costas desde el punto de vista computacional cuando el factor M es elevado.

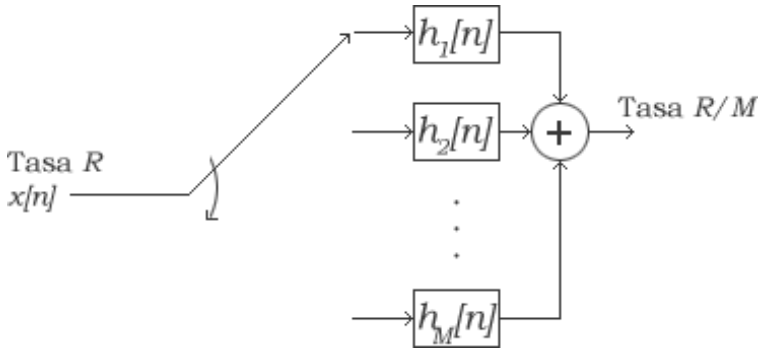


Figura 6.9 Esquema de filtro polifase para diezmado.

En el diezmado por filtrado polifase, el filtro paso bajo original con K coeficientes se divide en un total de M subfiltros, cada uno con $S = \lceil K/M \rceil$ coeficientes. A diferencia de la interpolación (donde por cada subfiltro pasaban la totalidad de los datos), en el diezmado, diferentes subconjuntos de los datos atravesarán los M subfiltros.

Puesto que se desea volver a la frecuencia de muestreo original de la fuente de datos, será necesario "deshacer" los cambios en frecuencia que se introdujeron en el transmisor por interpolación en múltiples etapas. Para ello, se aplicará el diezmado en múltiples etapas, y el factor M deberá ser igual al factor L escogido en la interpolación, con la siguiente variación:

$$M = M_1 M_2 \dots M_N = L = L_1 L_2 \dots L_N, \qquad M_N = L_1 \leq M_{N-1} = L_2 \leq \cdots \leq M_1 = L_N, \tag{60}$$

Esto es así con el objetivo de tener que manejar una menor cantidad de datos, resultando en un procesado eficiente de estos. Es decir, si primero se diezmara por el factor menor M_N, la cantidad de muestras resultantes sería superior a la que se tendría al empezar por el factor M_1, lo que desde un punto de vista funcional no supondría ningún problema, pero sí afectaría la eficiencia computacional que se viene persiguiendo a lo largo de este trabajo.

6.3. Filtro adaptado

Tal y como se ha mencionado en capítulos anteriores, el canal se simulará bajo la suposición de que el ruido que afecte a la señal transmitida sea AWGN. En estas condiciones, existe un esquema de detección óptimo (en el sentido de máxima verosimilitud) para modulaciones lineales conocido como filtro adaptado [23]. Este filtro actuará maximizando la SNR a la salida del mismo en los instantes de muestreo de la señal.

Suponiendo que a la entrada de un hipotético receptor se tiene:

$$y(t) = s(t) + n(t) \tag{61}$$

siendo $s(t)$ la señal en banda base y $n(t)$ el ruido que se suma a la señal de interés al atravesar el canal.

Si se busca un filtro $h_R(t)$ tal que maximice la potencia de $s(t)$ en comparación con $n(t)$ en un instante τ, sería posible demostrar que, haciendo uso de la desigualdad de Cauchy-Schwarz, la expresión de $h_R(t)$ en frecuencia sería de la forma [24]:

$$H_R(f) = kS^*(f)e^{-j2\pi f\tau} \tag{62}$$

Aplicando la transformada de Fourier inversa y las propiedades de simetría y desplazamiento en el tiempo:

$$\mathcal{F}^{-1}\{kS^*(f)e^{-j2\pi f\tau}\} = ks^*(\tau - t) \tag{63}$$

Entonces, un filtro adaptado a $s(t)$ es aquel con respuesta al impulso tal que:

$$h_R(t) = ks^*(\tau - t) \tag{64}$$

De la ecuación (9) en la sección 3.2 se observa que $s(t)$ es el tren de pulsos conformado a partir de $p(t)$. Siendo $p(t)$ una señal real, entonces:

$$h_R(t) = kp(\tau - t) \tag{65}$$

Si el pulso conformador $p(t)$ tiene simetría par, entonces $h_R(t)$ no es más que dicho pulso con un cierto retraso.

Es por esta razón por lo que, tal y como se mencionaba en el capítulo sobre la implementación del transmisor, se emplea la raíz del coseno alzado como pulso conformador (en lugar del coseno alzado como tal), pues el filtro adaptado a este es él mismo, resultando en una respuesta global (a lo largo de todo el canal de transmisión) de un coseno alzado, el cual cumple con el criterio de Nyquist para tener ISI mínima.

6.4. Ecualización

Con el fin de hacer frente a la ISI introducida al atravesar un canal donde existe propagación multitrayecto, la señal recibida debe ser debidamente ecualizada para reducir (o eliminar por completo) la interferencia.

El tipo de ecualizador empleado dependerá en gran medida de la aplicación, de las prestaciones del sistema, etc. En general, existen varias formas de clasificar un ecualizador [25] siendo una de ellas:

- **Ecualizadores lineales (EL)**: este tipo de ecualizadores son sencillos de implementar y su salida no es más que una versión escalada y retrasada de la entrada (de ahí que sean lineales). No funcionan bien cuando la ISI es severa. Ejemplos de EL: *zero-forcing* (ZF), mínimo error cuadrático medio (MMSE).
- **Ecualizadores no lineales (ENL)**: de una complejidad mayor que los anteriores, suelen añadir una estructura de realimentación, de manera que la salida ya no es lineal. Son más efectivos contra casos de ISI severa. Ejemplos de ENL: estimación de secuencia de máxima verosimilitud (MLSE), ecualización de decisión realimentada (DFE).

También es posible distinguir entre:

- **Ecualizadores ciegos (EC)**: trabajan directamente sobre la señal recibida sin necesidad de una secuencia de entrenamiento previa. Ejemplos de EC: *least mean squares* (LMS), *constant-modulus algorithm* (CMA).
- **Ecualizadores adaptativos (EA)**: a partir de una secuencia de entrenamiento, el ecualizador estima los parámetros del canal y estos se van actualizando conforme el canal va cambiando. Ejemplos de EA: *least mean squares* (LMS), *stochastic gradient descent* (SGD), algoritmo Kalman.

Para exponer el funcionamiento de algunas de estas técnicas, se explicarán con más detalle los ecualizadores *zero-forcing* y LMS adaptativo.

6.2.1 Ecualizador lineal *Zero-forcing* (ZF)

Tal y como su nombre indica, el ecualizador ZF [26] funciona forzando a cero todas las contribuciones a la respuesta al impulso total del transmisor, canal, filtro adaptado y ecualizador en los instantes nT para $n \neq 0$, donde T es el tiempo de muestreo.

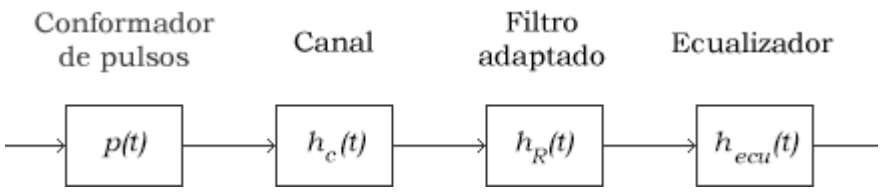


Figura 6.10 Esquema de comunicaciones con ecualizador.

Suponiendo un canal no ruidoso, la respuesta al impulso total de la cadena de comunicación es:

$$h_o(t) = p(t) * h_c(t) * h_R(t) * h_{ecu}(t) \quad \leftrightarrow \quad H_o(f) = P(f)H_c(f)H_R(f)H_{ecu}(f) \tag{66}$$

De acuerdo con lo mencionado anteriormente, debe ocurrir que:

$$h_o(nT) = \begin{cases} 1, & n = 0 \\ 0, & n \neq 0 \end{cases} \tag{67}$$

Idealmente, se conseguiría ISI nula con un filtro inverso (en frecuencia) del tipo:

$$H_{ecu}(f) = \frac{1}{P(f)H_c(f)H_R(f)} \tag{68}$$

Puesto que no siempre será posible encontrar un sistema inverso y estable (causal), la solución de ZF trata de aproximar $H_{ecu}(f)$ por un filtro FIR, logrando de este modo mitigar la ISI en una cantidad determinada de puntos. El número total de coeficientes del filtro será $N_{ecu} = 2N + 1$, siendo N un parámetro dependiente de la aplicación.

$$h_{ecu}(t) = \sum_{n=-N}^{N} \beta_n \delta(t - nT) \leftrightarrow H_{ecu}(f) = \sum_{n=-N}^{N} \beta_n e^{-j2\pi f nT} \tag{69}$$

Si se agrupan el conformador de pulsos, canal y filtro adaptado en $h(t) = p(t) * h_c(t) * h_R(t)$, entonces la respuesta total muestreada cada T segundos, sería:

$$h_o(kT) = \sum_{n=-N}^{N} \beta_n h_c[(k - n)T] \tag{70}$$

La ecuación anterior se puede desarrollar en formato matricial, de forma que (se usará la notación $\underline{v}$ para vectores y $\underline{\underline{M}}$ para matrices):

$$\underline{h_o} = \underline{\underline{H}}\ \underline{h_{ecu}} \rightarrow \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} h(0) & h(-T) & \cdots & h(-2NT) \\ h(T) & h(0) & \cdots & h((-2N+1)T) \\ \vdots & \vdots & \ddots & \vdots \\ h(2NT) & h((2N-1)T) & \cdots & h(0) \end{bmatrix} \begin{bmatrix} \beta_{-N} \\ \beta_{-N+1} \\ \vdots \\ \beta_0 \\ \vdots \\ \beta_{N-1} \\ \beta_N \end{bmatrix} \tag{71}$$

Es interesante notar que la matriz de convolución de canal $\underline{\underline{H}}$ es una matriz de Toeplitz, lo que facilitará su generación cuando se implemente el ecualizador en el siguiente capítulo.

Así, para obtener los coeficientes del filtro ecualizador habría que resolver la siguiente ecuación:

$$\underline{h_{ecu}} = \underline{\underline{H}}^{-1}\ \underline{h_o} \tag{72}$$

Puesto que es necesario conocer $\underline{\underline{H}}$, sería necesario la aplicación de técnicas de estimación de canal que permitieran obtener una buena aproximación del mismo.

Esta técnica de ecualizado presenta el inconveniente de no ser efectiva cuando existen nulos en frecuencia (o se tienen niveles muy bajos), ya que para compensarlo hace que la ganancia crezca de forma desmedida.

¿Y en qué afecta ese aumento de la ganancia del filtro ante nulos en frecuencia? Los cálculos anteriores se han realizado suponiendo que el canal no es ruidoso. La realidad es que el canal lo será. Para simplificar la situación, suponiendo que no exista conformación de pulso ni filtro adaptado, tan solo el canal, la señal que recibe el receptor es:

$$r(t) = s(t) * h_c(t) + n(t) \tag{73}$$

Si se tiene un filtro $h_{ecu}(t)$ que cumple con la teoría expuesta anteriormente, entonces tras atravesar el ecualizador se tiene:

$$y(t) = s(t) * h_c(t) * h_{ecu}(t) + n(t) * h_{ecu}(t) = s(t) + n(t) * h_{ecu}(t) \tag{74}$$

Aplicando la transformada de Fourier:

$$Y(f) = S(f) + N(f)H_{ecu}(f) \tag{75}$$

De esta forma queda patente que si $|H_{ecu}(f)|$ es grande, el ruido quedaría amplificado y degradaría enormemente la relación SNR.

Es por ello por lo que en la práctica el ZF no es muy útil, aunque su funcionamiento es sencillo y sirve como primera toma de contacto con los ecualizadores.

6.2.2 Ecualizador LMS adaptativo

Lo normal en muchas situaciones reales es emplear algún tipo de ecualización adaptativa [27], esto es, un filtro que va actualizando sus coeficientes con el objetivo de minimizar una cierta función de coste o error que, por lo general, suele ser la diferencia entre una señal de referencia (secuencia de entrenamiento) y la salida del ecualizador.

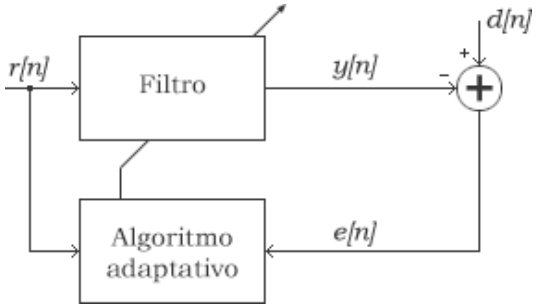


Figura 6.11 Estructura típica de un ecualizador adaptativo.

El filtro adaptativo que se busca es FIR y, al igual que en la anterior sección, tiene la siguiente estructura:

$$h_{ecu}(t) = \sum_k \beta_k \delta(t - kT) \tag{76}$$

La señal de error que se desea minimizar se puede definir como:

$$e[n] = d[n] - y[n] = d[n] - \sum_m \beta_m r[n - m] \tag{77}$$

donde $d[n]$ es la secuencia de entrenamiento o señal de referencia, y la salida del filtro ecualizador es $y[n]$.

Para obtener los coeficientes β_k del filtro se empleará la técnica de ***steepest descent*** o *gradient descent*. Es un algoritmo iterativo que busca mínimo local más cercano de una función a partir de su gradiente. La función de actualización de los coeficientes es de la forma:

$$\beta_k[n+1] = \beta_k[n] + \frac{\mu}{2}\left[-\frac{\partial C(\beta_k)}{\partial \beta_k}\right]_{\beta_k=\beta_k[n]} \tag{78}$$

siendo μ un parámetro ajustable que controla la velocidad de convergencia (*step size*) y $C(\beta_k)$ es la función de coste que se quiere minimizar.

Puesto que se trata de un ecualizador LMS (valor mínimo esperado del cuadrado de la señal de error), la función de coste es:

$$C = E\{|e[n]|^2\} \tag{79}$$

Después de resolver el gradiente en la ecuación de actualización, esta resulta de la forma:

$$\beta_k[n+1] = \beta_k[n] + \mu e[n] r[n-k] \tag{80}$$

Así, a partir de la secuencia de entrenamiento que le llega al receptor $r[n]$, se calculan los coeficientes del filtro adaptativo β_k y se obtiene la salida de dicho filtro $y[n]$. Será conveniente ajustar el parámetro μ dependiendo del sistema a controlar, y fijar un valor inicial para los coeficientes del filtro $\beta_k[0]$.

Como conclusión sobre este ecualizador, mencionar su facilidad de implementación, así como el buen funcionamiento que ofrece frente al ruido (a diferencia del *Zero-forcing*). Como inconveniente, están los problemas de convergencia según se escoja el parámetro μ, y además la necesidad de usar una secuencia de entrenamiento (aumento del ancho de banda) para lograr el correcto funcionamiento del algoritmo.

6.5. Decisión

El paso final consiste en la conversión de símbolos en bits. Para ello, se procederá a tomar la decisión de acuerdo con la mínima distancia de la muestra recibida a los diferentes puntos de la constelación. En caso de que el símbolo obtenido se encuentre a la misma distancia de dos o más puntos, se decidirá aleatoriamente entre cuales quiera de esos puntos.

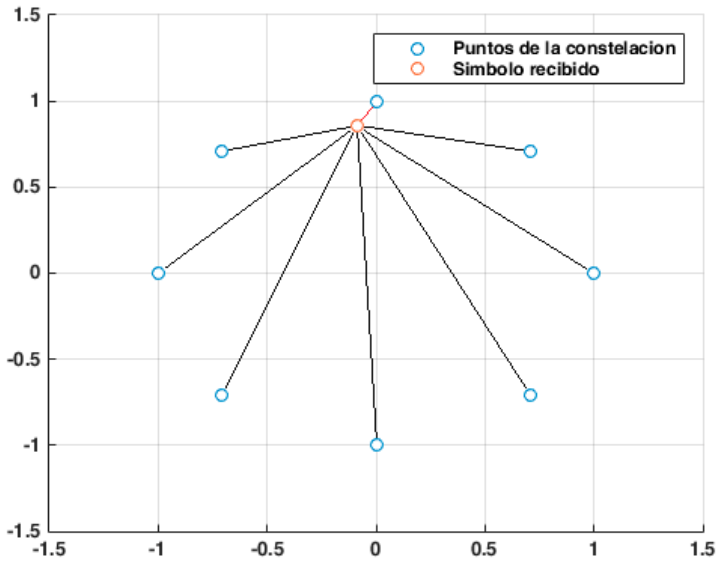


Figura 6.12 Punto de la constelación más cercano al símbolo recibido.

7 IMPLEMENTACIÓN DEL RECEPTOR

De nuevo, y tras lo visto en el capítulo sobre diseño del receptor, se realizarán propuestas de implementación de los diferentes bloques acompañados de demostraciones, ejemplos y gráficas que ilustren el código desarrollado.

También se aprovecha para introducir el uso del diagrama de ojos, una herramienta muy empelada para evaluar las prestaciones del enlace de comunicación.

7.1. PLL

Partiendo de lo explicado sobre PLLs analógico en la sección 6.1.1.1 se desarrollará el código correspondiente que permitirá simular un PLL de primer y segundo orden.

Un PLL de primer orden puede ser visto como un controlador proporcional con una función de transferencia para el *loop filter* $H_{LF}(s) = K_P$ siendo K_P la ganancia proporcional del controlador. Por su parte, un PLL de segundo orden se comporta como un controlador proporcional e integral (PI) con una función de transferencia $H_{LF}(s) = K_P + K_I/s$ (sustituyendo $\tau_1 = 1/K_I$ y $\tau_2 = K_P/K_I$ en las ecuaciones referentes al PLL de segundo orden visto en el capítulo anterior se llegaría a la expresión del *loop filter* propuesta), siendo K_I la ganancia integral del controlador.

La función `pllorden12` contará con las siguientes entradas:

- `ref`: señal de referencia con la que se pretende que el PLL se sincronice.
- `fpll`: frecuencia con la que se inicializará el PLL.
- `fasepll`: fase con la que se inicializará el PLL.
- `Kvco`: ganancia del VCO (se corresponde con la constante k_{VCO} vista en el capítulo anterior).
- `KI`: ganancia integral del controlador (en el caso del PLL de primer orden, esta debe ir a cero).
- `KP`: ganancia proporcional del controlador.
- `fs`: frecuencia de muestreo de la señal de referencia.

Código 7.1 PLL configurable con ganancia proporcional e integral.

```
% Senal de entrada  ------
n_muestras = length(ref);

% Inicializacion PLL -----
VCO=zeros(1,n_muestras);
incr_fase=zeros(1,n_muestras);
fase=zeros(1,n_muestras);

detector_fase = zeros(1,n_muestras);

Int_error=zeros(1,n_muestras);
```

```
interror = 0;
PI_error = zeros(1,n_muestras);

for n=1:n_muestras
    Int_error(n) = interror;
    incr_fase(n) = fasepll;
    fase(n) = 2*pi*fpll/fs*n+incr_fase(n);
    VCO(n) = cos(fase(n));

    detector_fase(n)=ref(n)*VCO(n);

    interror=Int_error(n)+KI*detector_fase(n);
    PI_error(n)=KP*detector_fase(n)+interror;

    fasepll=incr_fase(n)+2*pi*PI_error(n)*Kvco;

end
```

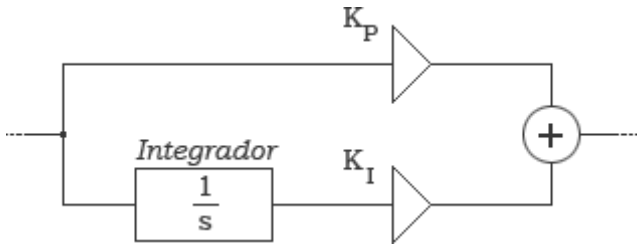


Figura 7.1 Estructura del *loop fitler* de un PLL de segundo orden.

La **Figura 7.1** ejemplifica las operaciones que se llevan a cabo en las variables `interror` y `PI_error` dentro del bucle `for`.

Para probar el código, se generará una señal seno de referencia de 1 MHz muestreada a $f_s = 70\ MHz$. En un primer intento, se configura con `KP=1` y `KI=0`, de manera que actúa como PLL de primer orden. Se configura para que arranque con la misma frecuencia y fase que la referencia (por lo que se podría esperar que el PLL se enganchara sin problemas):

Código 7.2 Ejemplo PLL de primer orden.

```
fs = 70e6;

% generacion de la entrada
f=1e6;
fase_ref=0;
n_muestras=2000;
t = 0:1/fs:n_muestras/fs-1/fs;

ref=sin(2*pi*f*t+fase_ref);

% PLL
Kvco = 0.01;
KP=1;
KI=0;
fpll=f+0e6;
```

```
fasepll=fase_ref;

% VCO: salida del VCO
% df: error de fase (después del detector de fase)
% fase: fase de salida en cada muestra
[VCO,df,fase] = pllorden12(ref,fpll,fasepll,Kvco,KI,KP,fs);
```

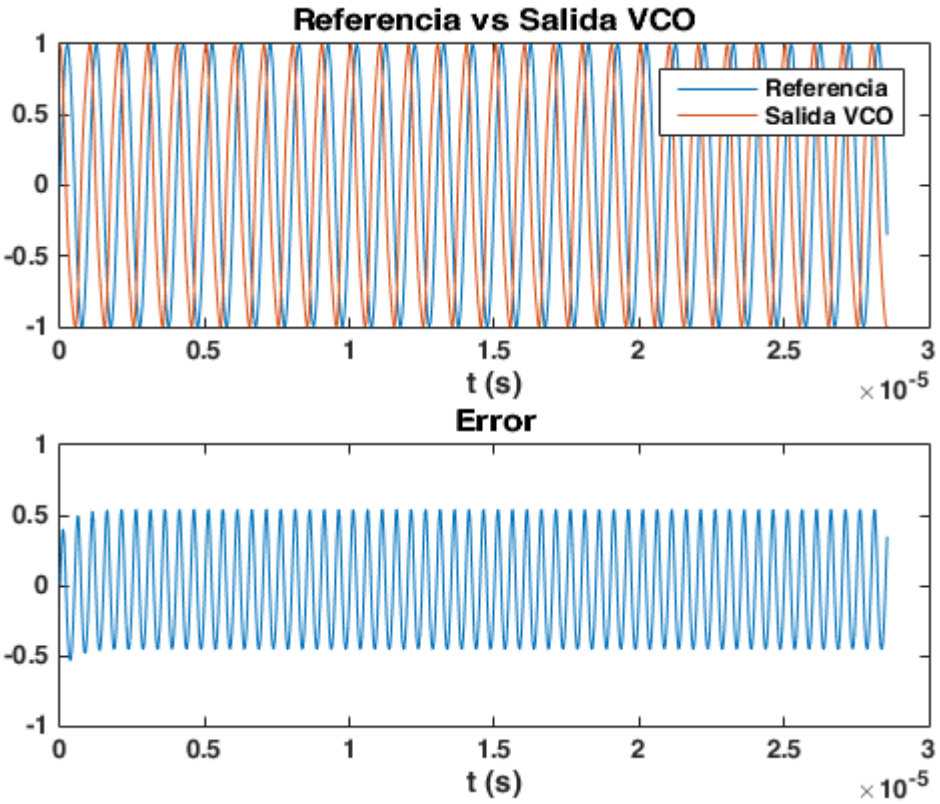


Figura 7.2 Error en PLL de primer orden para misma frecuencia y fase que la referencia.

Tal y como se puede observar en la **Figura 7.2**, lo más notable es que la señal de error tiene una forma sinusoidal. Esto es debido a que al realizar la detección de fase se lleva a cabo una multiplicación entre la señal de referencia y la generada por el VCO, resultando en una componente en DC además de otra al doble de frecuencia de la de la señal de referencia (como se explicó en la sección de PLL analógico del capítulo anterior). Puesto que el *loop filter* no ha sido capaz de filtrar dicha componente indeseada, se modificará el código propuesto para añadir un filtro paso bajo previo al *loop filter*. Así, se añaden estos nuevos parámetros a la función `pllorden12`:

- `orden_pb`: orden del filtro paso bajo a generar.
- `fcorte_pb`: frecuencia de corte del filtro paso bajo (que debe ser menor que el doble de la frecuencia de la señal de referencia).

Los coeficientes de este filtro se generarán empleando la función interna de MATLAB `fir1`.

Código 7.3 Mejora del PLL configurable con ganancia proporcional e integral.

```
% Senal de entrada  ------
n_muestras = length(ref);

% Filtro Paso bajo PLL ---
b = fir1(orden_pb,fcorte_pb/(fs/2));

% Inicializacion PLL -----
VCO=zeros(1,n_muestras);
incr_fase=zeros(1,n_muestras);
fase=zeros(1,n_muestras);

error = zeros(1,n_muestras);
detector_fase = zeros(1,n_muestras);

Int_error=zeros(1,n_muestras);
interror = 0;
PI_error = zeros(1,n_muestras);

for n=1:n_muestras
    Int_error(n) = interror;
    incr_fase(n) = fasepll;
    fase(n) = 2*pi*fpll/fs*n+incr_fase(n);
    VCO(n) = cos(fase(n));

    detector_fase(n)=ref(n)*VCO(n);

    filtradoPB = filter(b,1,detector_fase(1:n));
    error(n) = filtradoPB(end);

    interror=Int_error(n)+KI*error(n);
    PI_error(n)=KP*error(n)+interror;

    fasepll=incr_fase(n)+2*pi*PI_error(n)*Kvco;

end
```

Modificando ahora el **Código 7.2** para añadir estos nuevos parámetros:

Código 7.4 Ejemplo de PLL de primer orden con filtro paso bajo intermedio.

```
fs = 70e6;

% generacion de la entrada
f=1e6;
fase_ref=0;
n_muestras=2000;
t = 0:1/fs:n_muestras/fs-1/fs;

ref=sin(2*pi*f*t+fase_ref);

% PLL
Kvco = 0.01;
KP=1;
```

```
KI=0;
fpll=f+0e6;
fasepll=fase_ref;
fcorte_pb=100e3;
orden_pb=100;

% VCO: salida del VCO
% df: error de fase (después del detector de fase)
% fase: fase de salida en cada muestra
[VCO,df,fase] = pllorden12(ref,fpll,fasepll,Kvco,KI,...
                           KP,fs,orden_pb,fcorte_pb);
```

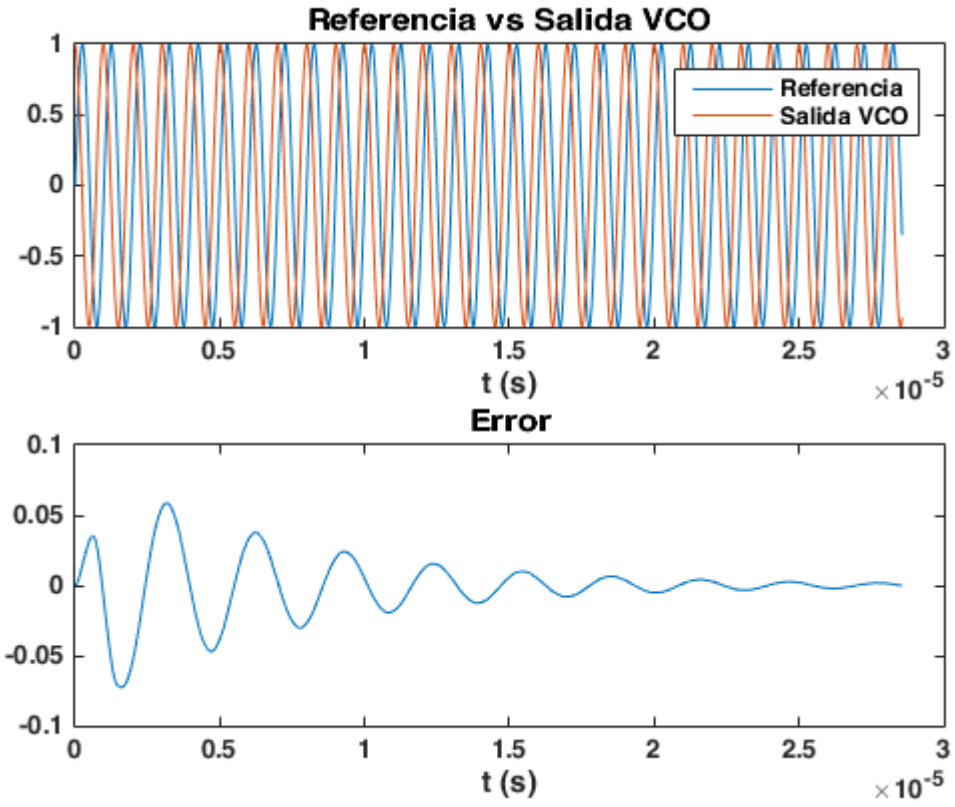


Figura 7.3 Error en PLL de primer orden con filtro paso bajo intermedio.

Observando la figura anterior, el resultado mejora hasta el punto de tener un error máximo de 0.05.

Sin embargo, a pesar de que del error se hace cero, la salida del VCO no parece estar en fase con la referencia. De la forma que se explicó en la sección 6.1.1.1, el VCO genera un coseno mientras que la entrada es un seno, por lo que hay una discrepancia entre estos de $\pi/2$. Para solucionarlo, la tercera salida de la función `pllorden12` es `fase`, la cual contiene únicamente la información de la fase para cada muestra (a diferencia de la señal `VCO` que devuelve el coseno), por lo que se puede usar para representar el `sin(fase)` como se aprecia en la **Figura 7.4**.

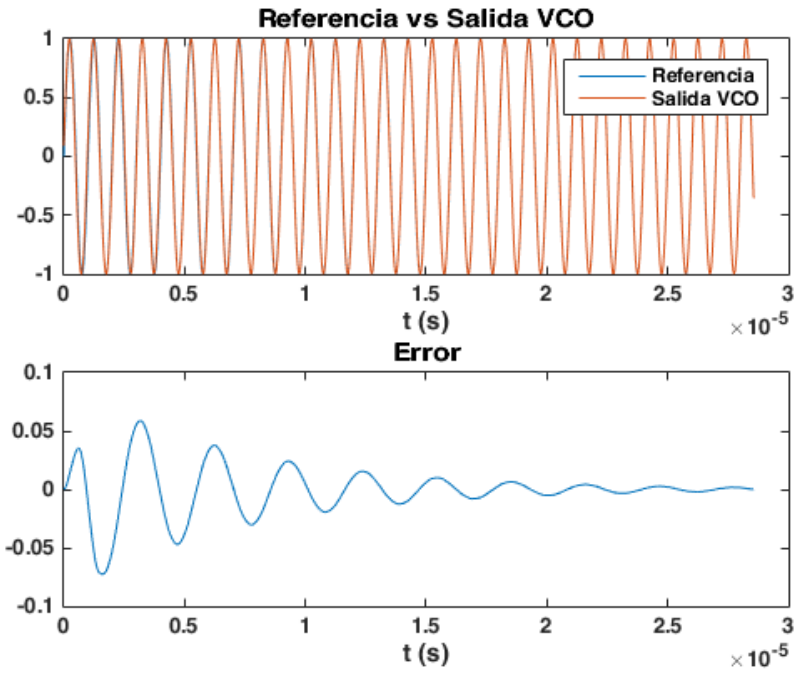


Figura 7.4 Representación del seno de la fase de salida del VCO.

A partir de aquí todas las pruebas se harán con el filtro paso bajo activado y se representará la señal de referencia frente a la salida del VCO empleando `sin(fase)`.

Si ahora se prueba a inicializar el PLL con una frecuencia ligeramente superior a la de referencia (concretamente con 100 kHz más), se puede apreciar el defecto del PLL de primer grado (consigue engancharse, pero teniendo un error de fase constante):

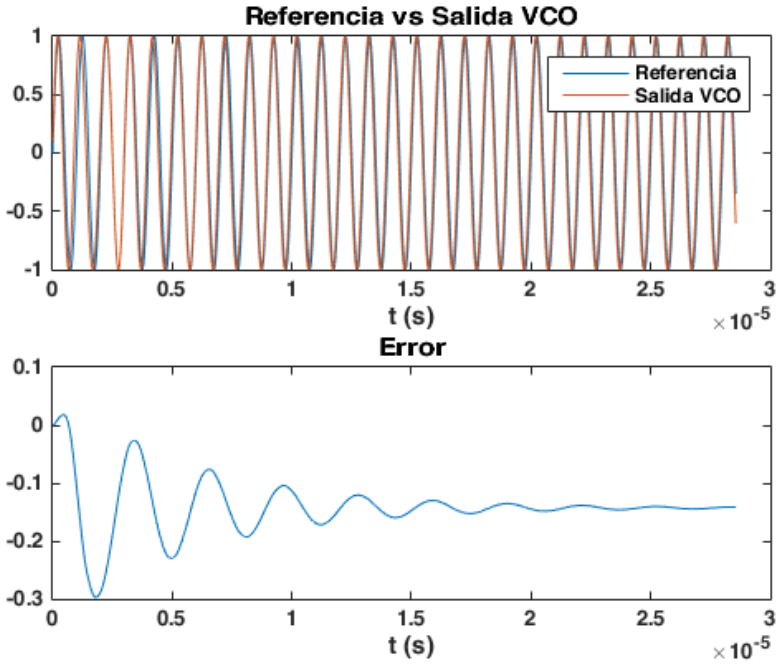


Figura 7.5 Error en PLL de primer orden para una frecuencia 100 kHz superior a la referencia.

Para solucionar esto, hay que acudir al PLL de segundo orden, que conseguirá engancharse siempre y cuando la diferencia de frecuencias sea pequeña. Esto se consigue dando un valor a la ganancia integral de la función. Así que, tomando `KP=0.4467` y `KI=0.0018`, ahora se corrige el error de fase aun siendo la frecuencia de inicialización del PLL 100 kHz mayor que la de la señal de referencia:

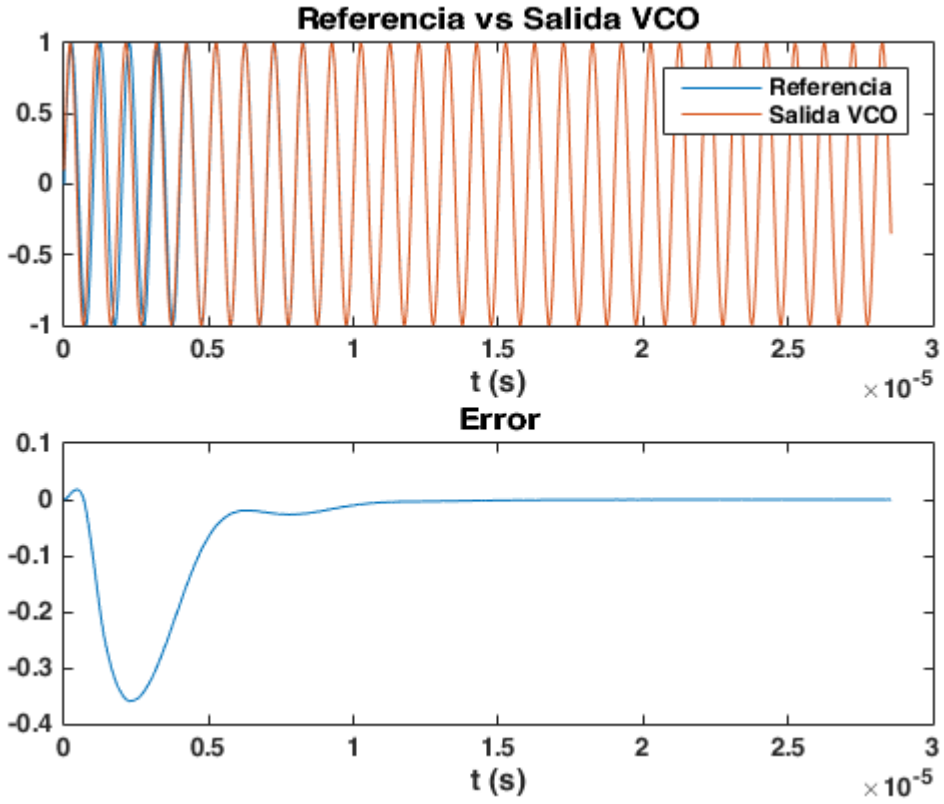


Figura 7.6 Error en PLL de segundo orden.

7.2. Sincronización de portadora

Comprender el funcionamiento del PLL es importante debido a que la gran mayoría de técnicas de sincronización de portadora (y de símbolo) hacen uso de este. Es por ello por lo que para la implementación del método *squaring loop* [28] se hará uso de la implementación propuesta en la sección anterior.

A la entrada del bloque sincronizador de portadora se tendrá la señal:

$$y(t) = Am(t)\text{sen}\,(2\pi f_c t + \theta_i) \tag{81}$$

donde $m(t)$ es el mensaje, f_c es la frecuencia de la portadora y θ_i es la fase de referencia. El mensaje será otra señal sinusoidal a una frecuencia mucho menor f_m.

Código 7.5 Generación de la señal de refencia para sincronización por *squaring loop*.

```
fs = 10e6;

% generacion de la entrada

fc = 0.501e6;
fm = 0.07e6;
n_muestras = 1500;
t = 0:1/fs:n_muestras/fs-1/fs;

fase_ref=0.35;

c=sin(2*pi*fc*t+fase_ref); %portadora
m=0.8*sin(2*pi*fm*t); %mensaje
y=c.*m; %mensaje modulado
```

El esquema que se va a implementar es el que aparece en la **Figura 6.5**, con la única diferencia que se introducirá un filtro paso banda entre el bloque que eleva al cuadrado la señal de entrada y el detector de fase, centrado en la frecuencia $2f_c$, con el fin de eliminar la componente indeseada.

Código 7.6 Generación del filtro paso banda en el *squaring loop*.

```
% Filtro Paso banda -------
orden_pasobanda = 400;
fbaja_pasobanda = 0.99e6;
falta_pasobanda = 1.01e6;
desfase_FPBanda = 320*2*pi/360;
bpb = fir1(orden_pasobanda, ...
          [fbaja_pasobanda/(fs/2) falta_pasobanda/(fs/2)]);
```

Hay que tener en cuenta que el filtro paso banda introduce un desfase en la señal, por lo que debe ser considerado más adelante. La forma de obtener este desfase es representando la respuesta en frecuencia del filtro (usando, por ejemplo, `freqz(bpb)`) y probando valores cercanos a la fase correspondiente a la frecuencia central del filtro.

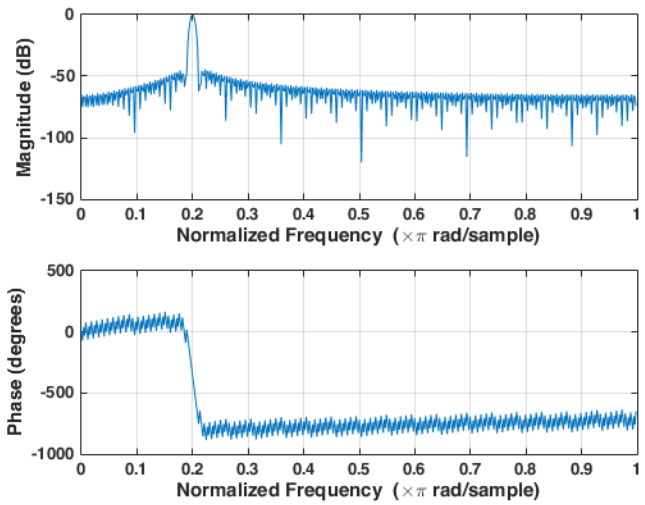


Figura 7.7 Respuesta en frecuencia del filtro paso banda empelado en el *squaring loop*.

Ya que la señal de referencia al cuadrado y filtrada por el paso banda será la que entre al PLL, este se debe inicializar con una frecuencia cercana al doble de la portadora.:

Código 7.7 Inicialización del PLL en el *squaring loop*.

```
% PLL
Kvco = 0.08;
KP=0.4467; %0.4467
KI=0.0178; %0.0018
fpll=1e6;
fasepll=fase_ref;
fcorte_pb=0.2e6;
orden_pb=20;
bypass_pb=0;

% senal al cuadrado
ref_cuadrado = y.^2;

% filtro paso banda para quedarnos con la componente 2fc
salida_FPBanda = filter(bpb,1,ref_cuadrado);

% PLL
[VCO,df,fase] = pllorden12(salida_FPBanda,fpll,fasepll,...
                           Kvco,KI,KP,fs,bypass_pb,orden_pb,fcorte_pb);
```

Para terminar, será necesario reducir la frecuencia/fase resultante a la mitad. Además, es en este punto donde hay que restar el desfase introducido por el filtro paso banda:

Código 7.8 Reducción de la fase de salida a la mitad y compensación del desfase.

```
VCO_mitad_fase = sin(fase./2 - desfase_FPBanda);
```

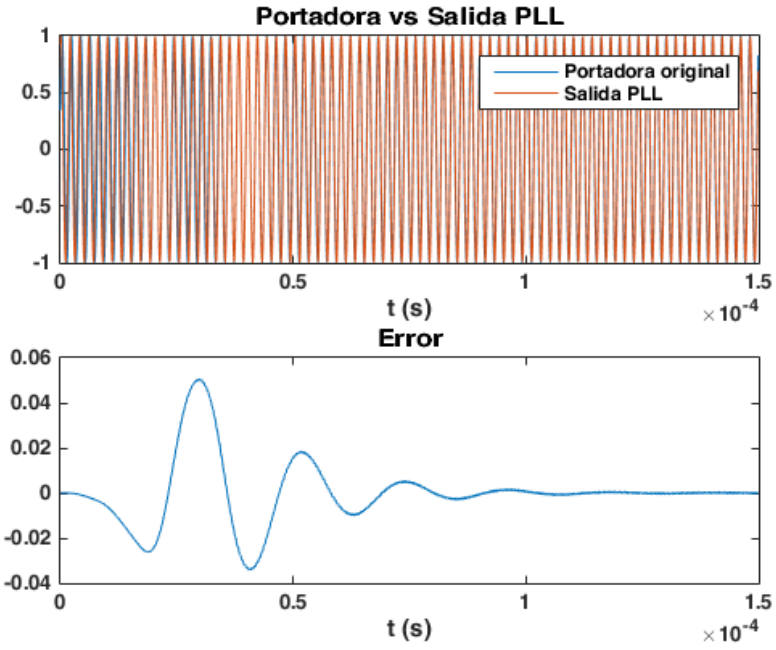


Figura 7.8 Respuesta en frecuencia del filtro paso banda empelado en el *squaring loop*.

Se alcanza un error cercano a cero transcurridos unos $100\ \mu s$ y la salida del PLL queda sincronizada de forma casi perfecta a la portadora (debido al rizado que tiene la señal de error, existe un pequeño desfase constante entre la portadora y la salida del PLL).

Finalmente, se aprovecha para demodular el mensaje de la portadora utilizando la información proporcionada por el PLL. Si $y_{PLL}(t) = \text{sen}\ (2\pi f_c t + \theta)$ es la señal de salida del proceso de sincronización, entonces:

$$x(t) = y(t)y_{PLL}(t) = Am(t)\,\text{sen}(2\pi f_c t + \theta_i)\,\text{sen}(2\pi f_c t + \theta) = \frac{A}{2}m(t)[\cos(\theta_i - \theta) - \cos\,(4\pi f_c t + \theta_i + \theta)] \tag{82}$$

Al pasar por un filtro paso bajo, se eliminará la componente en frecuencia $2f_c$, de manera que solo quedará $x_{PB}(t) = \frac{A}{2}m(t)\cos(\theta_i - \theta)$. Debido a la acción del PLL, el error $\theta_i - \theta \approx 0$, por lo que el resultado debería ser $x_{PB}(t) \approx \frac{A}{2}m(t)$. Para compensar la reducción a la mitad de la amplitud, se puede multiplicar por 2, quedando $Am(t)$:

Código 7.9 Demodulación tras la sincronización de portadora.

```
% Filtro paso bajo para demodular ----
orden_pb_demod=50;
fcorte_pb_demod=0.4e6;
bdemod = fir1(orden_pb_demod,fcorte_pb_demod/(fs/2));

demod=y.*VCO_mitad_fase;
demod_filtrado=2.*filter(bdemod,1,demod);
```

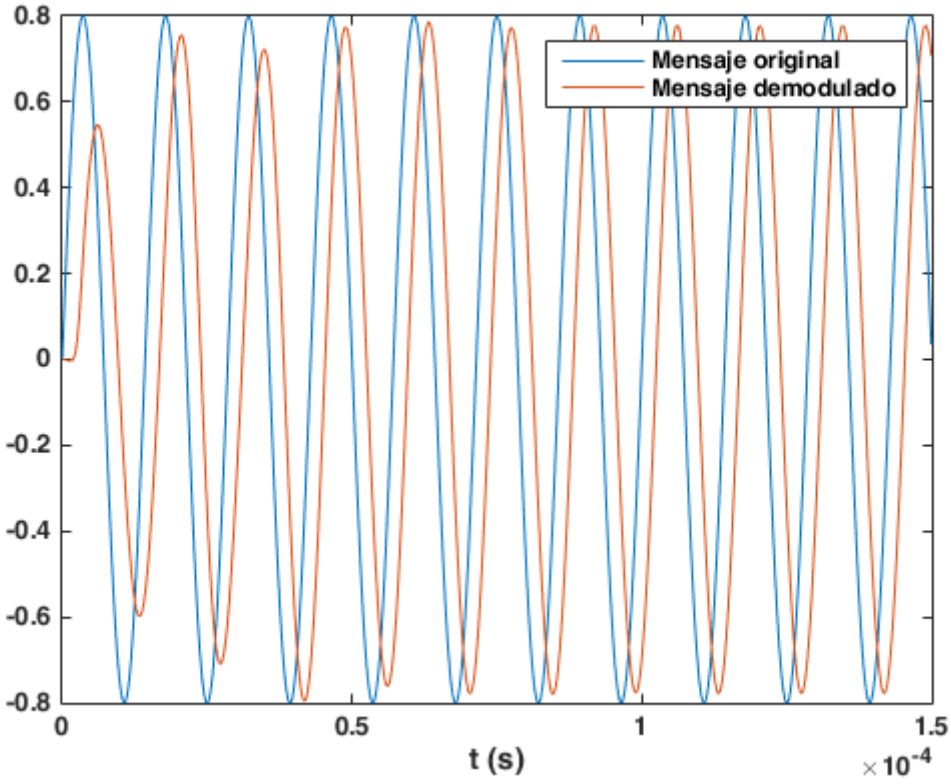


Figura 7.9 Comparativa entre el mensaje original y el demodulado tras sincronización de portadora.

7.3. Diezmado en múltiples etapas

Al igual que ocurría en el transmisor, la función `receptor` recibe como parámetro de entrada (entre otros) un vector `factor_dm`, de tamaño $1xN$ que indica los diferentes factores a tener en cuenta para realizar el diezmado en múltiples etapas (en caso de ser un vector $1x1$ solo se aplicaría el filtro adaptado). El orden en que se realiza el diezmado es inverso al de interpolación. Otro parámetro de tipo `cell` será el que se use para especificar los coeficientes de los diferentes filtros usados en las etapas del diezmado (sin incluir la última etapa obligatoria de filtro adaptado donde se deshace la conformación de pulso) de tamaño $1x(N-1)$. Esta celda `coef_multi_etapa` contendrá a su vez vectores de distintos tamaños correspondientes a los coeficientes de los filtros.

En el diezmado será necesario también eliminar el retraso generado al aplicar los filtros. No solo hay que eliminar el introducido en el diezmado, también el producido durante la interpolación. Puesto que son los mismos filtros (en interpolación y diezmado), el retraso a eliminar será dos veces el introducido por el filtro en diezmado.

Código 7.10 Diezmado con filtrado polifase.

```
function datos_filtrados = filtrado_polifase(datos, coef,
factor_intdec, interp, retraso )
    tam_coef = length(coef);
    rellenar_ceros=factor_intdec*ceil(tam_coef/factor_intdec)-tam_coef;
    b = [coef zeros(1,rellenar_ceros)];
    b_p = reshape(b, factor_intdec, []);

    if (interp == 1)
        % [ . . . ] ya desarrolado en la conformacion de pulsos
    else
        % Diezmado
        datos_ajustados = [zeros(1,factor_intdec-1) datos];
        tam_datos_final = length(datos_ajustados);

        datos_ajustados(1:mod(retraso,factor_intdec))=[];
        tam_datos_filtrados_ajustado=floor(length(datos_ajustados) ...
                                           /factor_intdec);

        datos_ajustados=datos_ajustados(...
                        1:tam_datos_filtrados_ajustado*factor_intdec);

        datos_filtrados = zeros(1, tam_datos_filtrados_ajustado);

        for i=1:factor_intdec
            i_inverso = factor_intdec - (i-1);
            datos_diezm=filter(b_p(i,:),1,...
                        datos_ajustados(i_inverso:factor_intdec:end));
            datos_filtrados = datos_filtrados + datos_diezm;
        end

        tam_datos_final = floor((tam_datos_final-retraso)...
                                /factor_intdec);
        datos_filtrados = datos_filtrados(length(datos_filtrados)-...
                                          tam_datos_final+1:end);

    end
end
```

Continuando con el ejemplo expuesto en el **Código 4.12**, se realizará el proceso de diezmado empleando para ello los factores $M = M_1 M_2 = 6 \cdot 4 = 24$.

Código 7.11 Ejemplo de diezmado en múltiples etapas.

```
% continuando desde el Codigo 4.11 usando la variable interp2
% [ . . . ]
M1 = 6;
[b_1, retraso1] = [coef_normalizados,retardo] = gen_coef_cic(4, M1, 1);

M2 = 4;
% generacion del pulso coseno alzado como en secciones anteriores
% [ . . . ]
[b_2, retraso2] = pulso.generar_coeficientes();

diezm1 = filtrado_polifase(interp2, b_1, M1, 0, 2*retraso1);
diezm2 = filtrado_polifase(diezm1, b_2, M2, 0, 2*retraso2);
```

7.4. Diagrama de ojos

Es necesario mencionar en este punto una herramienta de gran utilidad para analizar un enlace de comunicación, conocida como **diagrama de ojos**. Aunque es más interesante representar dicho diagrama en el receptor, se realizará la prueba también en el transmisor.

El diagrama de ojos se forma dividendo la señal en varias partes de igual longitud, y superponiendo todas sobre el mismo eje de tiempo. De esta manera, la figura que se forma tiene un cierto parecido al ojo humano.

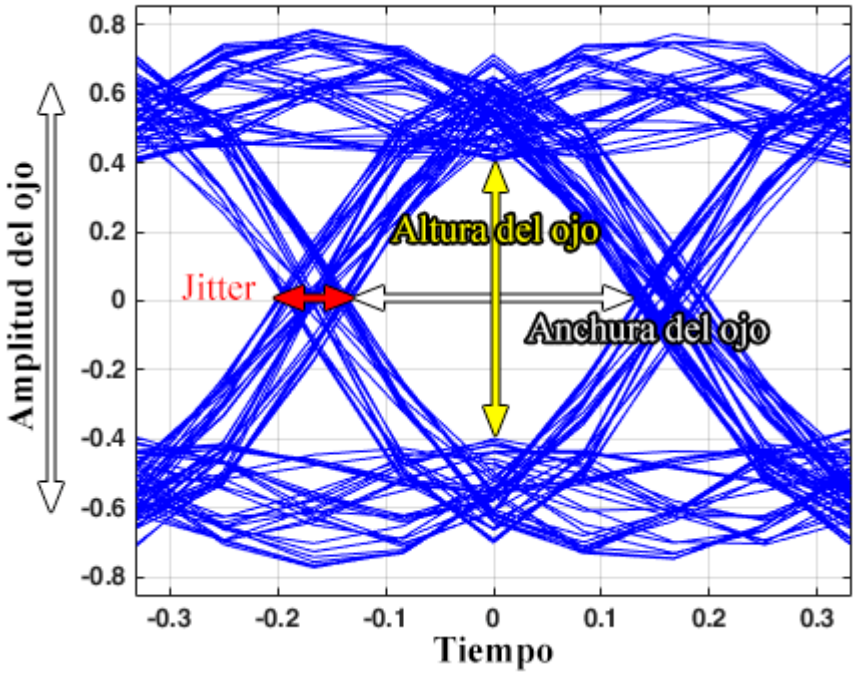


Figura 7.10 Diagrama de ojo y sus principales medidas.

Sobre este diagrama se pueden realizar diversas medidas que sirven para caracterizar la calidad de la señal. Algunas de estas medidas son:

- ***Jitter***: variaciones en los cruces por cero. Cuando se transmite un bit, las transiciones de un estado al otro se producen en un momento dado. Idealmente, el envío repetido del mismo bit significaría que estas transiciones ocurren siempre en los mismos instantes de tiempo. Pero debido a diferentes causas (reflexiones, interferencia intersímbolo, ruido térmico electrónico, etc.), un mismo bit no siempre realizará la transición al siguiente estado en el mismo tiempo, produciendo así las variaciones en los cruces por cero o *jitter*.
- **Amplitud del ojo**: diferencia entre el nivel lógico '1' y el '0'.
- **Altura del ojo**: medida vertical de la apertura del ojo. Idealmente, sería igual a la amplitud, pero la presencia de ruido provoca que el ojo se cierre. Esta característica se encuentra estrechamente ligada con la relación señal a ruido (SNR).
- **Anchura del ojo**: medida horizontal de la apertura del ojo. Idealmente, se mediría en los puntos de cruce por cero, sin embargo, la existencia de *jitter* hace que esto no sea así.

Como ejemplo, se propone un sencillo caso donde el transmisor, empleando una modulación BPSK, realiza la conformación de pulsos y, antes de transmitir los datos empleando una portadora, se hace un estudio del diagrama de ojos. Posteriormente, los datos atraviesan un canal ruido y una vez en el receptor, tras demodular la portadora y justo antes de aplicar el filtro adaptado, se volverá a estudiar el diagrama de ojos para ver en qué medida ha afectado el ruido a la señal.

Para representar el diagrama de ojos se hará uso de la función de MATLAB `eyediagram`.

Código 7.12 Diagrama de ojos en el transmisor.

```
datos           = randi([0 1], 1, 200).*2-1;

% Definición de las propiedades del objeto 'coseno_alzado'
tipo            = 'sqrt';  % normal ('normal') o raiz ('sqrt')
rolloff         = 0.60  ;  % factor de rolloff
truncSimbolos   = 16    ;  % numero de simbolos a los que se trunca
muestrasPorSimb = 4     ;  % numero de muestras por simbolo

pulso = coseno_alzado(rolloff, truncSimbolos, muestrasPorSimb, tipo);

[coef, retraso] = pulso.generar_coeficientes();

resultado = filtrado_polifase(datos, coef, muestrasPorSimb, 1, 0);

% se elimina el retraso
resultado = resultado(retraso+1:end);
eyediagram(resultado, muestrasPorSimb);
```

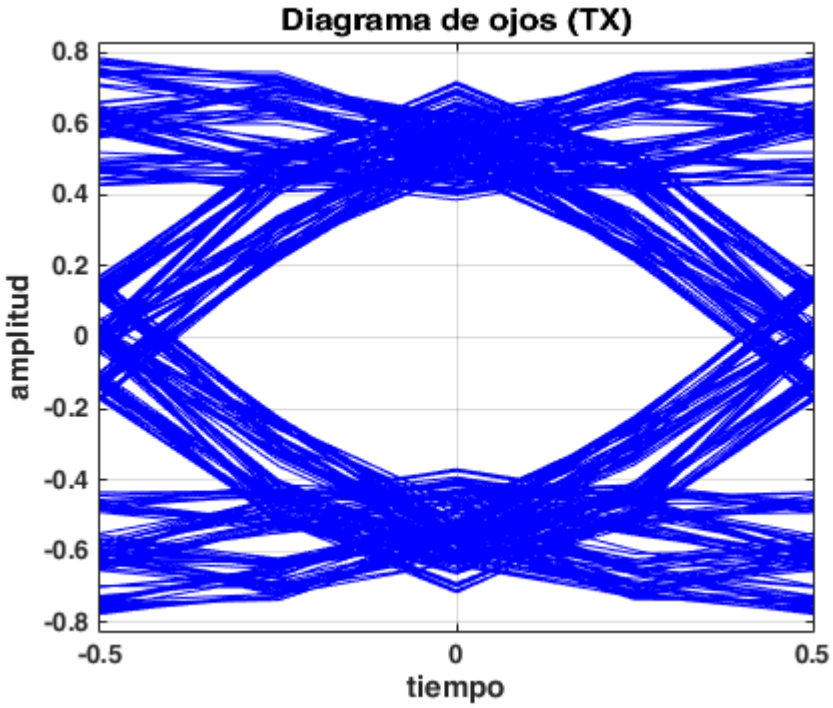


Figura 7.11 Diagrama de ojos en el transmisor.

Después de realizar el resto de interpolaciones y la subida en frecuencia por medio de una portadora, los datos atraviesan un canal ruidoso:

Código 7.13 Generación de ruido y adición a los datos.

```
EbN0_dB = 0;
n = gen_ruido(datos_tx,EbN0_dB,cpsk.k,factor_interpolacion_final);

datos_canal = datos_tx + n;
```

Estos datos llegan al receptor que, tras todo el procesado necesario, llegará al filtro adaptado. Justo antes de aplicarlo, se representa el diagrama de ojo para ver cómo ha afectado el ruido a la señal. Esto puede observarse en la **Figura 7.12**.

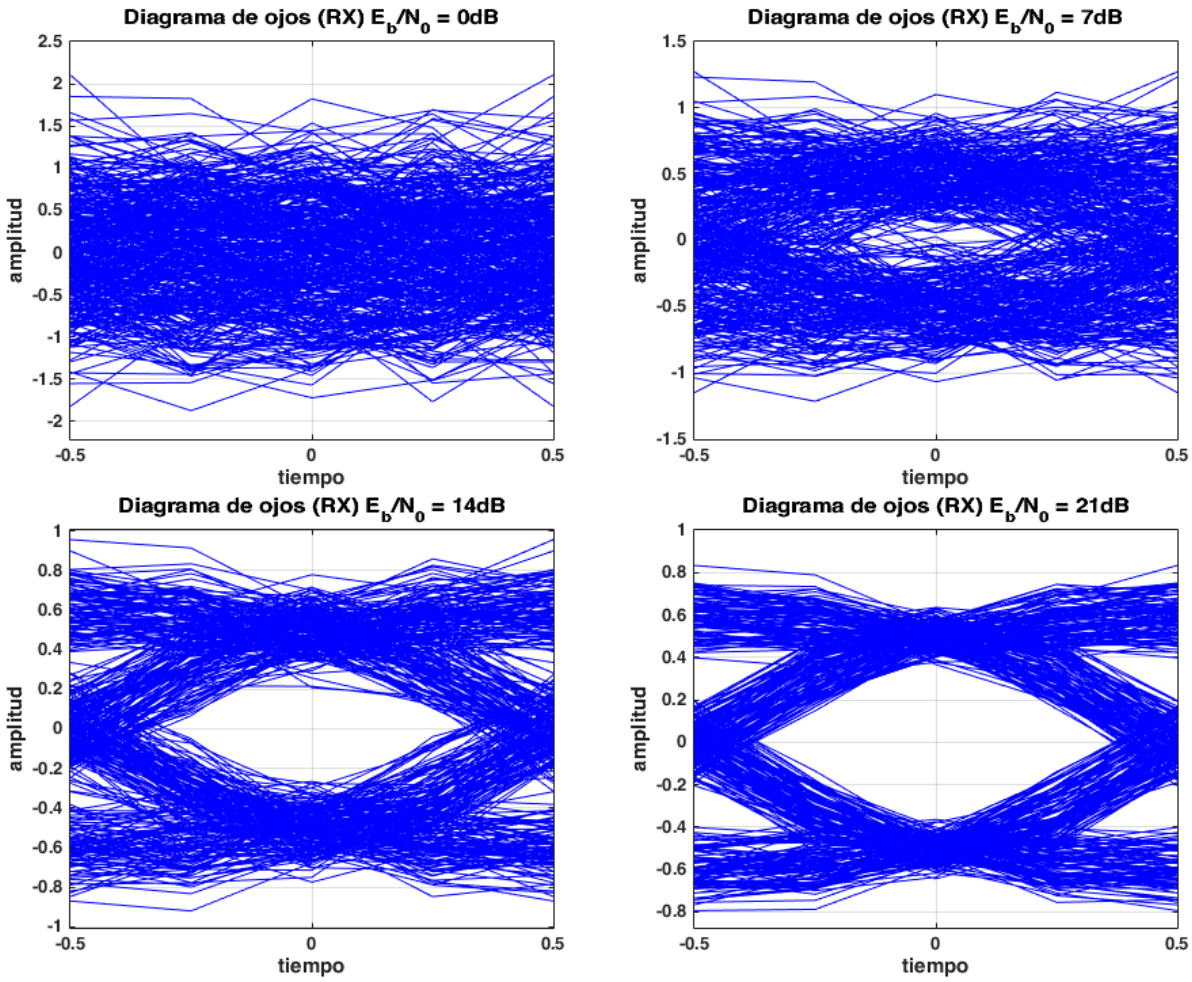


Figura 7.12 Diagrama de ojos en el receptor para $E_b/N_0 = 0\ dB, 7\ dB, 14\ dB$ y $21\ dB$.

También es posible observar el efecto de atravesar un canal multitrayecto. Para ello, se probará un canal donde, además de la principal que llega sin retraso y con ganancia unidad, existirán dos copias retrasadas $250\ ns$ y $500\ ns$, con una frecuencia de muestreo de $128\ MHz$. Las ganancias de las copias irán variando para ver el efecto en el diagrama de ojos, tal y como se observa en la **Figura 7.13**.

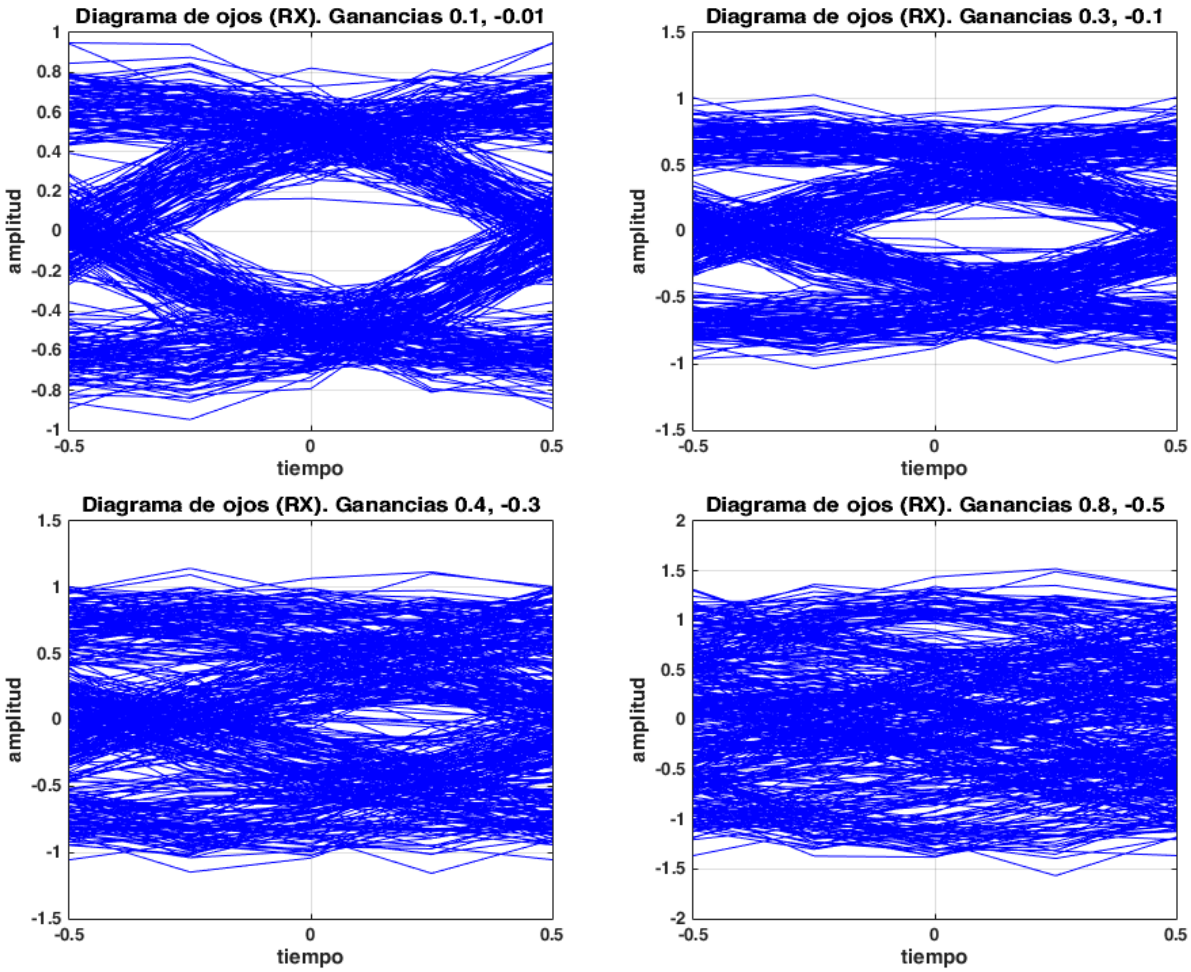


Figura 7.13 Diagrama de ojos en el receptor para $E_b/N_0 = 14\ dB$ y diferentes canales multitrayecto.

Tras el estudio del patrón de ojos, se procede a la aplicación del filtro adaptado, que no difiere del diezmado, teniendo en cuenta que el filtro empleado debe cumplir con la ecuación (65).

7.5. Ecualización

7.1.1 Zero-forcing

La implementación del ZF se hará en base a lo visto en la sección 6.2.1, ecuación (72). Se asume conocido el canal (típicamente se obtendría mediante técnicas de estimación de canal).

Código 7.14 Generación de coeficientes del filtro ecualizador ZF.

```
function [ecu,H] = zeroforcing( canal, longitud_ecu )
    if (mod(length(canal),2) == 0)
        error('El numero de taps del canal debe ser impar');
    end

    tap_cero = (length(canal)+1)/2;

    colidx = tap_cero:tap_cero+2*longitud_ecu;
    colend = min(tap_cero+2*longitud_ecu, length(canal));
    colcanal = canal(tap_cero:colend);
```

```
    filidx = tap_cero:-1:tap_cero-2*longitud_ecu;
    filend = max(1, tap_cero-2*longitud_ecu);
    filcanal = canal(tap_cero:-1:filend);

    pcol = zeros(1,length(colidx)); % primera columna
    pcol(1:length(colcanal)) = colcanal;
    pfil = zeros(1,length(filidx)); % primera fila
    pfil(1:length(filcanal)) = filcanal;

    delta = zeros(2*longitud_ecu+1,1);

    H = toeplitz(pcol, pfil);
    delta(longitud_ecu+1) = 1;

    ecu = H\delta;

end
```

Se considerará el coeficiente central del canal como $h(0)$, por lo que se calcula su posición en `tap_cero`.

Para construir la matriz de Toeplitz se necesita definir la primera columna y fila que formará parte de la matriz:

$$\begin{aligned} pcol &= [h(tap\_cero), h(tap\_cero+1), \dots, h(tap\_cero+2N) \\ pfila &= [h(tap\_cero), h(tap\_cero-1), \dots, h(tap_{cero}-2N)] \end{aligned} \tag{83}$$

El número de coeficientes del filtro será $2N+1$, siendo N un parámetro que se escoge según convenga. En la función anterior se corresponde con el parámetro de entrada `longitud_ecu`. Puesto que N puede ser cualquier valor, si se supera el tamaño de la variable de entrada `canal` al intentar construir la matriz de Topelitz, las posiciones restantes se rellenarán con ceros.

A continuación, se expone un código de ejemplo para un canal dado:

Código 7.15 Obtención de filtro ZF para un canal dado.

```
c = [0.07 -0.3 0.4 1.25 0.3 -0.3 0.1];
longitud = 3;

h_ecu = zeroforcing(c,longitud);
```

En la siguiente figura se puede observar la respuesta al impulso del canal, y la total. En esta última, es posible comprobar como el punto central tiene una amplitud de 1 además de N puntos a su derecha y N a su izquierda (en total $2N$ puntos) están forzados a cero.

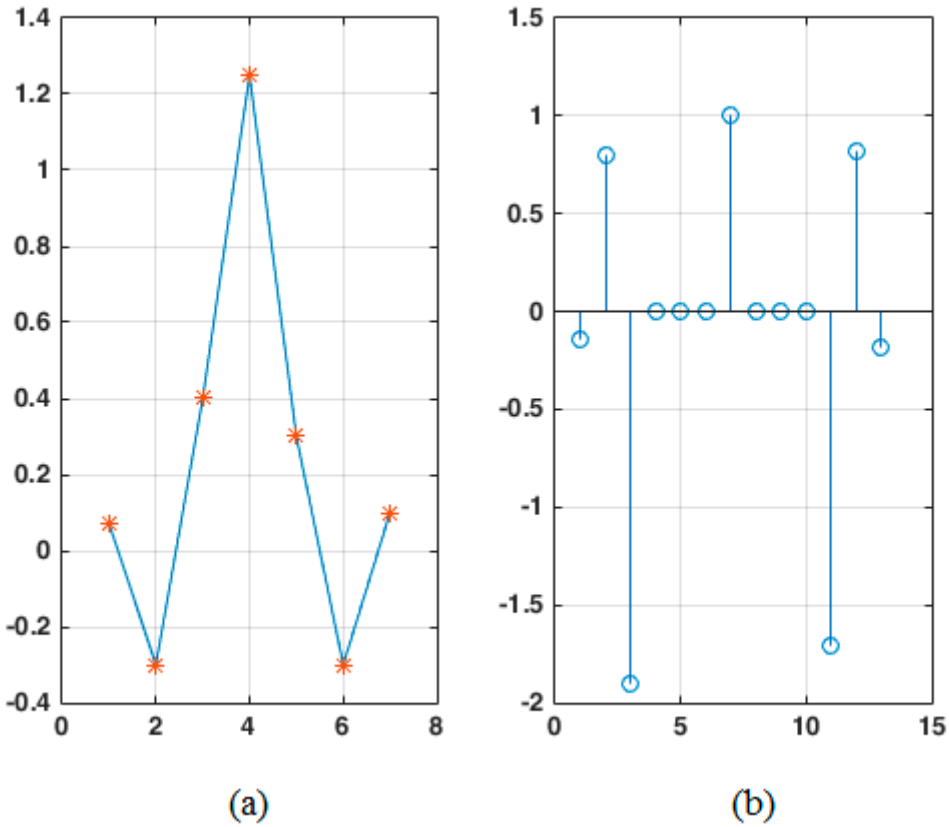


Figura 7.14 (*a*) Respuesta al impulso del canal. (*b*) Respuesta al impulso total.

7.1.2 LMS adaptativo

El objetivo principal es adaptar la ecuación (80) para convertirla en una función que ejecute el algoritmo LMS de forma iterativa. En concreto, esta función tendrá cuatro parámetros de entrada:

- `mu`: la constante de adaptación o *step size*.
- `K`: el número de coeficientes del filtro adaptativo resultante.
- `r`: la señal recibida (de entrada).
- `d`: la señal de referencia o deseada.

La diferencia entre la señal `r` y la señal `d`, es que la primera ha atravesado el canal ruidoso y multitrayecto, mientras que `d` es la señal tal cual generada por el transmisor.

Por otro parte, la función cuenta con tres salidas:

- `y`: salida del filtro adaptativo.
- `b`: coeficientes del filtro adaptativo.
- `e`: error, tal y como se definió en la ecuación (77), en cada una de las iteraciones del algoritmo.

Las señales de entrada y la de referencia deben ser del mismo tamaño, N, que será el número de veces que se ejecutará el algoritmo.

Código 7.16 Algoritmo LMS.

```
function [y,b,e]=lms1(mu,K,r,d)
    b=zeros(1,K);
    N=length(r);
    y=zeros(1,N);
    e=zeros(1,N);

    %LMS
    for n=K:N
        rvec=r(n:-1:n-K+1).';
        y(n)=b*rvec;
        e(n)=d(n)-y(n);
        b=b+mu*e(n)*rvec';
    end
end
```

Los coeficientes del filtro se van actualizando para intentar conseguir que $y[n] \approx d[n]$.

Si se aplica la media al vector de salida `e`, se obtendrán valores muy cercanos a cero. El algoritmo LMS converge para todas las señales, pero es una convergencia en la media, es decir, puede ocurrir que el filtro se acerque al valor óptimo, pero se quede "rebotando" alrededor del mismo. No se garantiza que se alcance en el valor óptimo.

La mayor restricción al respecto es la constante de adaptación μ. Es importante que sea lo suficientemente grande para asegurar la convergencia, pero lo suficientemente pequeña para que no se aleje del valor óptimo.

7.6. Decisión

La decisión sobre el símbolo transmitido se toma en base a la distancia mínima a cada uno de los puntos de la constelación. Es por ello por lo que en la clase `constelacion_psk` se añaden dos nuevos métodos.

El primero de ellos calcula la distancia de una muestra a todos los puntos de la constelación, esto es, devuelve un vector de tamaño $1xM$, siendo M el número de símbolos de la constelación.

Código 7.17 Método para obtener la distancia a todos los puntos de la constelación.

```
function dist = obtener_distancia(obj, muestra)
    dist = sqrt(real(muestra - obj.alfabeto).^2 + ...
                imag(muestra - obj.alfabeto).^2);
end
```

El segundo método hace uso del primero y busca entre todas las distancias obtenidas, la menor. Esto le permite identificar a qué símbolo corresponde la muestra que se pasa como parámetro. En caso de que la distancia mínima no sea única, se elegirá aleatoriamente entre los puntos de la constelación para los que se obtuvo el mínimo valor.

Código 7.18 Método para la toma de decisión del símbolo recibido.

```
function bits = tomar_decision(obj, muestra)
    if (obj.k == 1)
        bits = sign(muestra)<0;
    else
        dist = obj.obtener_distancia(muestra);
```

```
        minval = min(dist);
        idxs = find(dist == minval);
        idx = idxs(randi(length(idxs)));
        bits = obj.bits(idx,:);
    end
end
```

8 CONCLUSIONES Y PROPUESTAS DE MEJORA

Con la realización de este trabajo se pretende, por un lado, ahondar en los fundamentos de comunicación básico sobre los que se basan transmisor y receptor para poder desarrollar una radio definida por *software*, y por otro lado facilitar posibles formas de implementación que ayuden a su desarrollo en un dispositivo real.

El repaso a lo largo de la historia de la SDR deja de manifiesto que, a pesar de los numerosos avances, aún queda por hacer. Los dispositivos comerciales presentan dos limitaciones principales: o bien solo funcionan como receptor y su rango de frecuencias no es muy amplio, o resultan bastantes caros a cambio de una doble funcionalidad transmisor-receptor.

Relativo al estudio del transmisor, es de suma importancia decidir la modulación que se empleará en un posible dispositivo final dependiendo de la aplicación y la situación (por ejemplo, podría ser que se prefiera usar una BPSK por su robustez, aunque implique una menor tasa de transmisión porque solo se va a usar en tareas de control), así como elegir un pulso que introduzca la mínima ISI posible, como es el caso del coseno alzado. A su vez se ha introducido la técnica de la interpolación mediante filtrado polifase que permite aumentar la frecuencia de muestreo, una técnica muy eficiente y fácil de implementar en dispositivos finales reales.

Un aspecto fundamental a tener en cuenta a la hora de realizar simulaciones para hacer que todo sea lo más parecido al entorno real es el canal. El canal que atraviesa la información es un canal ruidoso que dificulta la detección por parte del receptor y además puede provocar que la señal de interés se refleje en diversos obstáculos y terminen llegando copias retrasadas en el tiempo, aumentando la interferencia intersímbolo.

El receptor plantea el mayor reto de todo el sistema, debido a sus bloques dedicados a la sincronización. Por un lado, hay que diseñar un sistema que permita engancharse a los cambios en la frecuencia de la portadora, y para esto se han estudiado algunos métodos básicos en la literatura como el *squaring loop* y el lazo de Costas. Por otro, es necesario averiguar el momento de muestreo idóneo para tomar la decisión correcta, de lo que se encarga el bloque de sincronización de tiempo, con métodos como el Early-Late Gate. Ambos dependen en gran medida de un PLL, sistema de control necesario para hacer un seguimiento de la fase de una señal. Los PLL de primer orden no resultan del todo efectivos para este cometido, siendo el de segundo orden el más indicado.

En el receptor también se encuentra otro bloque esencial, el ecualizador, que tratará de deshacer los cambios del canal. Existen numerosas técnicas para hacer frente a este problema, desde filtros adaptativos que se van actualizando conforme reciben la información, a filtros ciegos que no necesitan de ninguna señal de entrenamiento para funcionar.

Finalmente, a lo largo de los ejemplos de implementación del receptor se ha introducido el diagrama de ojos que permite estudiar de una manera más visual cómo afecta el ruido, el *jitter* o la interferencia intersímbolo a la señal recibida.

Como propuestas de mejora o continuación de este proyecto se propone:

- Añadir técnicas de detección y corrección de errores para poder asegurar la integridad de los datos. Una de las técnicas de corrección más populares es el *forward error correction* (FEC), que se basa en añadir información redundante al mensaje (codificación de fuente) de manera que permita una recuperación fiable en el receptor si se ha producido un número de errores.
- En cierto modo, gran parte de este proyecto se ha realizado pensando en una posible implementación en un sistema real como podría ser una FPGA o un DSP, de forma que en muchas

implementaciones se busca la simplicidad o la eficiencia computacional. Teniendo las bases de proyecto, sería posible usarlo como herramienta de comprobación a alto nivel y comprobar cómo de parecido (o diferente) resulta con los datos obtenidos en la realidad.

ANEXO: CÓDIGO DE LAS IMPLEMENTACIONES

constelacion.m

```
classdef (Abstract) constelacion
    properties
        M        % constelacion M-aria
        k        % numero de bits de cada simbolo
        bits     % bits de la constelacion
        alfabeto % simbolos de la constelacion
    end

    methods
        function obj = constelacion(M)
            obj.k = ceil(log2(M));
            obj.M = 2^obj.k;
        end
    end
end
```

constel_psk.m

```
classdef constel_psk < constelacion
    properties
        fase     % fase de la constelacion
    end

    methods
        function obj = constel_psk(M, fase)
            if (nargin < 2)
                fase = 0;
            end

            obj = obj@constelacion(M);
            obj.fase = fase;
            [obj.bits, obj.alfabeto] = obj.crear_constelacion;
        end

        function [bits,alfabeto] = crear_constelacion(obj)
            alfabeto = zeros(1, obj.M);
            bits = zeros(obj.M, obj.k);
            for s=1:obj.M
                bits(s,:) = numAbin(s-1, obj.k);
                alfabeto(s) = exp(1i*(2*pi*(s-1)/obj.M + obj.fase));
            end
```

```
            % Convertimos a gray
            bits = xor(bits,[zeros(obj.M,1), bits(:,1:end-1)]);
        end

        function dist = obtener_distancia(obj, muestra)
            dist = sqrt(real(muestra - obj.alfabeto).^2 + ...
                        imag(muestra - obj.alfabeto).^2);
        end

        function bits = tomar_decision(obj, muestra)
            if (obj.k == 1)
                bits = sign(muestra)<0;
            else
                dist = obj.obtener_distancia(muestra);
                minval = min(dist);
                idxs = find(dist == minval);
                idx = idxs(randi(length(idxs)));
                bits = obj.bits(idx,:);
            end
        end

        function ber_teorica = calcular_ber_teorica(obj, EbNo_dB)
            EbNo = 10.^(EbNo_dB/10);

            switch (obj.M)
                case {2, 4}
                    ber_teorica = qfunc(sqrt(2.*EbNo));
                otherwise
                    ber_teorica = (2/obj.k)*...
                                  qfunc(sqrt(2.*obj.k.*EbNo).*...
                                  sin(pi/obj.M));
            end

        end

        function representar_ber(~, EbNo_dB, …
                                 ber_teorica, estilo_representacion)
            if nargin < 4
               estilo_representacion = '-';
            end
            semilogy(EbNo_dB, ber_teorica, estilo_representacion);
            grid on;
            xlabel('E_b/N_0 (dB)');
            ylabel('BER');
        end
    end

end
```

pulso_conf.m

```
classdef (Abstract) pulso_conf
    methods (Abstract)
        generar_coeficientes(obj)
```

```
    end
end
```

coseno_alzado.m

```
classdef coseno_alzado < pulso_conf
    properties
        tipo           = 'sqrt'% normal ('normal') o raiz ('sqrt')
        rolloff        = 0.5   % factor de rolloff
        truncSimbolos  = 4     % numero de simbolos a los que se trunca
        muestrasPorSimb= 4     % numero de muestras por simbolo
    end

    methods
        function obj = coseno_alzado(rolloff, span, sps, tipo)
            if (mod(span*sps,2) ~= 0)
                error('El producto de truncSimbolos por muestrasPorSimb
debe ser par.');
            else
                if (nargin < 4 && nargin > 0)
                    obj.tipo = 'normal';
                else
                    obj.tipo = tipo;
                end
                obj.rolloff = rolloff;
                obj.truncSimbolos = span;
                obj.muestrasPorSimb = sps;
            end
        end

        function [coef,retardo] = generar_coeficientes(obj)
            coef = rcosdesign(obj.rolloff, obj.truncSimbolos, ...
                              obj.muestrasPorSimb, obj.tipo);
            retardo = mean(grpdelay(coef));
        end
    end

end
```

conformador_pulso.m

```
function  datos_pulsoconf  =  conformador_pulso(  datos_mod,  pulso,
factor_interpolacion, coef, debug )
    % Si se proporcionan los coeficientes se procede
    % a realizar el filtrado e interp autimaticamente
    if (nargin < 4)
        coef = 0;
    end

    % debug es opcional, en caso de
    % no usarla tomara el valor 0
    if (nargin < 5)
        debug = 0;
    end
```

```
    if (coef == 0)
        coef = pulso.generar_coeficientes;
    end

    datos_pulsoconf      =      filtrado_polifase(datos_mod,      coef,
factor_interpolacion);
end
```

filtrado_polifase.m

```
function    datos_filtrados    =    filtrado_polifase(    datos,    coef,
factor_intdec, interp, retardo )
    if (nargin <4)
        interp=1;
    end

    if (nargin <5)
       retardo=0;
    else
        if (retardo > length(datos))
            error('El  retardo  no  puede  ser  mayor  que  el  numero  de
datos.');
        end
    end

    tam_coef = length(coef);
    rellenar_ceros     =     factor_intdec*ceil(tam_coef/factor_intdec)-
tam_coef;
    b = [coef zeros(1,rellenar_ceros)];
    b_p = reshape(b, factor_intdec, []);

    if (interp == 1)
        % Interpolado
        tam_datos_filtrados = length(datos)*factor_intdec;
        datos_filtrados = zeros(1, tam_datos_filtrados);

        for i=1:factor_intdec
            datos_filtrados(i:factor_intdec:end)  =  filter(b_p(i,:),  1,
datos);
        end
    else
        % Diezmado
        datos_ajustados = [zeros(1,factor_intdec-1) datos];
        tam_datos_final = length(datos_ajustados);

        datos_ajustados(1:mod(retardo,factor_intdec))=[];
        tam_datos_filtrados_ajustado                                    =
floor(length(datos_ajustados)/factor_intdec);

        datos_ajustados                                                 =
datos_ajustados(1:tam_datos_filtrados_ajustado*factor_intdec);

        datos_filtrados = zeros(1, tam_datos_filtrados_ajustado);

        for i=1:factor_intdec
```

```
            i_inverso = factor_intdec - (i-1);
            datos_diezm          =          filter(b_p(i,:),          1,
datos_ajustados(i_inverso:factor_intdec:end));
            datos_filtrados = datos_filtrados + datos_diezm;
        end

        tam_datos_final = floor(...
                          (tam_datos_final-retardo)/factor_intdec);
        datos_filtrados = datos_filtrados(...
                          length(datos_filtrados)-...
                          tam_datos_final+1:end);
    end
end
```

gen_ruido.m

```
function [n, varN] = gen_ruido(datos, EbN0_dB, k, factor_inttotal,
bypass)
% GENERADOR_RUIDO
    if (nargin < 5)
        bypass = 0;
    end

    if (bypass == 0)
        EbN0 = 10^(EbN0_dB/10);
        varS = var(datos);

        varN = varS*factor_inttotal / (2*EbN0*k);

        n = randn(size(datos)) * sqrt(varN);
    else
        n = 0;
        varN = 0;
    end
end
```

mapper.m

```
function simbolos = mapper(datos, modulacion, debug)
    % Esta variable es opcional, en caso de
    % no usarla tomara el valor 0
    if (nargin < 3)
        debug = 0;
    end

    % Se supone que los datos vienen debidamente tratados para ser
    % procesado por el modulador digital, toda la insercion de ceros
    % iria antes.
    s = superclasses(modulacion);
    s = char(s);

    switch s
        case 'constelacion'
            datos = reshape(datos', modulacion.k, [])';
            [~,index] = ismember(datos, modulacion.bits, 'rows');
```

```
            simbolos = modulacion.alfabeto(index);
        otherwise
            simbolos = 0;
    end

    if (debug == 1)
        % representamos la constelacion transmitida
        scatterplot(unique(simbolos));
        title('Constelacion transmitida');
        % representamos la BER teorica
        EbNo_dB = 1:10;
        ber_teorica = modulacion.calcular_ber_teorica(EbNo_dB);
        figure
        modulacion.representar_ber(EbNo_dB, ber_teorica);
        legend([num2str(modulacion.M),'-PSK']);
    end
end
```

demapper.m

```
function bits = demapper(datos, modulacion, debug)
    if (nargin < 4)
        debug = 0;
    end

    % Se supone que los datos vienen debidamente tratados para ser
    % procesado por el modulador digital, toda la insercion de ceros
    % iria antes.
    s = superclasses(modulacion);
    s = char(s);

    switch s
        case 'constelacion'
            tam_datos = length(datos);
            bits = zeros(1,tam_datos*modulacion.k);

            if (modulacion.k > 1)
                for i=1:tam_datos
                    bits(1+(i-1)*modulacion.k:modulacion.k*i)=...
                        modulacion.tomar_decision(datos(i));
                end
            else
                bits = modulacion.tomar_decision(datos);
            end
        otherwise
            bits = 0;
    end
end
```

gen_mulitrayecto.m

```
function [datos_multi, coef] = gen_multitrayecto(datos, at, retrasos,
Ts, bypass)
% GEN_MULTITRYAYECTO
    if (nargin < 5)
```

```
        bypass = 0;
    end

    if (bypass == 0)
        if (sum(Ts > retrasos) > 0)
            error('Los retrasos deben ser (mucho) mayores que Ts');
        end

        %se ordenan los retrasos de menor a mayor
        [ret, idx] = sort(retrasos);
        at = at(idx);

        % numero de coeficientes
        Nc = round(ret(end)/Ts)+1;
        coef = zeros(1,Nc);
        coef(1) = 1;

        % posiciones en el filtro
        enteros = round(ret/Ts)+1;
        coef(enteros) = at;

        datos_multi = filter(coef, 1, datos);
    else
        coef = 1;
        datos_multi = datos;
    end
end
```

gen_coef_cic.m

```
function [coef_normalizados,retardo] = gen_coef_cic(N, R, M)
% GENERADOR_FILTRO_CIC
    if (nargin < 3)
        M=1;
    end

    RM=R*M;

    hcic_orden1 = ones(1,RM);
    coeficientes = hcic_orden1;

    if (N >= 1)
        for ordenN=2:N
            coeficientes = conv(coeficientes,hcic_orden1);
        end
    end

    % Ganancia del filtro
    g=sum(coeficientes);
    coef_normalizados=coeficientes./g;

    retardo = mean(grpdelay(coef_normalizados));
end
```

transmisor.m

```
function [datos_tx, portadora, datos_mod] = transmisor(datos,
modulacion, fc, fs, ...
                              tipo_pulso, factor_int, debug, ...
                              coef_multi_etapa, filtro_comp)
    % Comprobaciones iniciales
    if (nargin < 7)
        debug = 0;
    end

    if (nargin < 8)
        coef_multi_etapa = {};
    end

    if (nargin < 9)
        filtro_comp = 0;
    end

    if (~iscell(coef_multi_etapa))
       error('Los coeficientes multi etapa deben ser una celda 1xN');
    end
    %-------------------------

    % Modulacion digital de los datos
    datos_mod = mapper(datos, modulacion, debug);

    % Conformacion de pulsos
    datos_interp_etapa1 = conformador_pulso(datos_mod, tipo_pulso,
factor_int(1));
    datos_interp = datos_interp_etapa1;

    if (size(factor_int,2) > 1)
        % si tiene mas de 1 columna, interp en varias etapas
        for i=2:size(factor_int,2)
            coef_multi = coef_multi_etapa{i-1}.*factor_int(i);
            datos_interp_etapa2 = filtrado_polifase(datos_interp, ...
                                                    coef_multi, ...
                                                    factor_int(i));
            datos_interp = datos_interp_etapa2;
        end
    end

    % Filtro compensador (si se uso CIC)
    if (size(filtro_comp,2) > 1)
        datos_interp = filter(filtro_comp,1,datos_interp);
    end

    % Subida en frecuencia
    n = 0:(length(datos_interp)-1);
    portadora = exp(1i.*(2.*pi.*fc./fs.*n+0));
    datos_tx = real(datos_interp.*portadora);
end
```

receptor.m

```
function [datos_rx, ecu_taps] = receptor(datos, modulacion, fc, fs, ...
                                tipo_pulso, factor_dm, portadora, ...
                                debug, coef_multi_etapa, bypass_equ, ...
                                canal)
    n = 0:(length(datos)-1);

    % Comprobaciones iniciales
    if (nargin < 7)
        portadora = exp(-1i.*2.*pi.*fc./fs.*n);
    end

    if (nargin < 8)
        debug = 0;
    end

    if (nargin < 9)
        coef_multi_etapa = {};
    end

    if (nargin < 10)
        bypass_equ = 1;
    end

    if (nargin < 11)
        canal = 0;
    end

    if (~iscell(coef_multi_etapa))
       error('Los coeficientes multi etapa deben ser una celda 1xN');
    end
    %------------------------

    % Conversion a banda base
    datos_bb = 2.*datos.*portadora;

    % Diezmado (deshacemos interpolacion en orden inverso)
    datos_diezm = datos_bb;

    if (size(factor_dm,2) > 1)
        % si tiene mas de 1 columna, interp en varias etapas
        for i=1:size(factor_dm,2)-1
            coef = coef_multi_etapa{i};
            retardo = mean(grpdelay(coef));
            datos_diezm = filtrado_polifase(datos_diezm, coef, ...
                                            factor_dm(i),             0,
2*retardo);
            %datos_diezm = filter(coef,1,datos_diezm);
            %datos_diezm                                               =
downsample(datos_diezm(retardo+1:end),factor_dm(i));
        end
    end

    % Filtro adaptado
    [cp,rp] = tipo_pulso.generar_coeficientes();
    datos_diezm = filtrado_polifase(datos_diezm, cp, ...
```

```
                                        factor_dm(end), 0, 2*rp);
    %datos_diezm = filter(cp,1,datos_diezm);
    %datos_diezm = downsample(datos_diezm(2*rp+1:end),factor_dm(end));

    % Ecualizacion
    datos_decision = datos_diezm;
    if (bypass_equ == 0)
        ecu_taps = zeroforcing(canal, (length(canal)-1)/2);
        ret_ecu = abs(round(mean(grpdelay(ecu_taps))));
        datos_decision = filter(ecu_taps, 1, datos_decision);
        %datos_decision = datos_decision(ret_ecu+1:end);
    else
        ecu_taps = 1;
    end

    datos_rx = demapper(datos_decision,modulacion,debug);
end
```

numAbin.m

```
function bin = numAbin( num , bits )
    if (nargin < 2)
        bits = 0;
    end

    bin = mod(floor(num.*2.^-(floor(log2(num)):-1:0)),2);
    tam = length(bin);

    if (bits > tam)
        bin = [zeros(1,bits-tam), bin];
    end
end
```

hacerFFT.m

```
function [ f,P1 ] = hacerFFT( x, Fs )
    n=length(x);

    Y=fft(x);
    P2=abs(Y/n);
    P1 = P2(1:n/2+1);
    P1(2:end-1) = 2*P1(2:end-1);
    f = Fs*(0:(n/2))/n;

    if nargout < 1
       plot(f,P1);
       f='Hecho';
    end
end
```

sistema_completo.m

```
close all;
```

```
% Modulacion BPSK
M_simbolos = 2;
fase_modulacion = 0;
cpsk = constel_psk(M_simbolos, fase_modulacion);

% Fuente de datos
RS = 4e6; % tasa de simbolos
TS = 1/RS; % periodo de simbolo

Init_bits = repmat(reshape(cpsk.bits,1,[]),1,50*log2(M_simbolos));
rng('default')
N_bits = 400*log2(M_simbolos); % de esta manera los bits se pueden
reshapear
datos = [ randi([0 1], 1, N_bits)];

% Primera etapa: Pulso conformador: SRRC
tipo            = 'sqrt' ;  % normal ('normal') o raiz ('sqrt')
rolloff         = 0.35   ;  % factor de rolloff
truncSimbolos   = 16     ;  % numero de simbolos a los que se trunca
muestrasPorSimb = 4      ;  % numero de muestras por simbolo

tipo_pulso = coseno_alzado(rolloff, truncSimbolos, muestrasPorSimb,
tipo);

% Segunda etapa: Filtro CIC
N = 4;
R = 8;
coef_multi_etapa = {gen_coef_cic(N,R)};

factor_int = [muestrasPorSimb R];

% Frecuencia muestreo final
fs = prod(factor_int)*RS;

% Frecuencia portadora
fc = 8e6;

debug = 0;

% TRANSMISOR ---------------------------------
[datos_tx,portadora,datosPSK] = transmisor(datos, cpsk, fc, fs, ...
                     tipo_pulso,factor_int, debug, ...
                     coef_multi_etapa);

% CANAL --------------------------------------
bypass_multitrayecto = 1;
bypass_ecualizador   = 1;
bypass_ruido         = 0;

% Multicamino
retrasos = [250e-9 500e-9];
atenuaciones = [0.8 -0.5];

[datos_multi, cmulti] = gen_multitrayecto(datos_tx, atenuaciones,...
                       retrasos, 1/fs, bypass_multitrayecto);

% Ruido
```

```
EbN0_dB=14;
[n,varN]=gen_ruido(datos_multi,EbN0_dB,cpsk.k,prod(factor_int),
bypass_ruido);

datos_canal = datos_multi + n;

% RECEPTOR ----------------------------------
factor_dm = [R muestrasPorSimb];

[datos_rx, cmulti2] = receptor(datos_canal, cpsk, fc, fs, ...
                      tipo_pulso,factor_dm, conj(portadora), ...
                      debug, coef_multi_etapa, bypass_ecualizador, ...
                      cmulti);

n_bits_rx = length(datos_rx);

% Calculamos error de bit
error_absoluto = datos_rx - datos(1:n_bits_rx);
n_bits_erroneos = sum(error_absoluto ~= 0);

ber = n_bits_erroneos/n_bits_rx

if (debug == 1)
    figure
    stem(datos_rx(1:100));
    hold on;
    stem(datos(1:100));
    hold off;
    grid
    legend('Datos RX','Datos TX');

    %Diagrama de ojos
    %eyediagram(datos_tx(1:1000),prod(factor_int));
end
```

Referencias

[1] SDR Forum, «SDRF Cognitive Radio Definitions,» 8 noviembre 2007. [En línea]. Available: http://www.sdrforum.org/pages/documentLibrary/documents/SDRF-06-R-0011-V1_0_0.pdf. [Último acceso: 20 junio 2019].

[2] R. J. Lackey y D. W. Upmal, «Speakeasy: the military software radio,» *IEEE Communications Magazine,* vol. 33, nº 5, pp. 55-61, 1995.

[3] P. G. Cook y W. Bonser, «Architectural overview of the SPEAKeasy system,» *IEEE Journal on Selected Areas in Communications,* vol. 17, nº 4, pp. 650-661, 1999.

[4] J. Mitola, «Software radios-survey, critical evaluation and future directions,» de *NTC-92: National Telesystems Conference*, Washington, 1992.

[5] GNU Radio, «GNU Radio - Main Page,» [En línea]. Available: https://wiki.gnuradio.org/index.php/Main_Page. [Último acceso: 4 julio 2019].

[6] F. Xiong, Digital Modulation Techniques, Artech House, 2006.

[7] A. Goldsmith, «Performance of Digital Modulation over Wireless Channels,» de *Wireless Communications*, 2004, pp. 172-175.

[8] T. B. Welch, C. H. G. Wright y M. G. Morrow, «FIR Digital Filters,» de *Real-Time Digital Signal Processing from MATLAB to C with the TMS320C6x DSPs*, CRC Press, 2012, pp. 31-52.

[9] J. G. Proakis y D. G. Manolakis, «Polyphase filter structures,» de *Digital Signal Processing*, Prentice-Hall, 1996, pp. 794-800.

[10] S. Haykin y M. Moher, «The Nyquist channel,» de *Introduction to Analog and Digital Communications*, Wiley, 2007, pp. 235-237.

[11] F. J. Harris, «Interpolating with Low Pass Half-band Filters,» de *Multirate Signal Processing for Communication Systems*, Prentice-Hall, 2004, p. 216.

[12] Altera Corporation, «Understanding CIC Compensation Filters (AN-455-1.0),» abril 2007. [En línea]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/an/an455.pdf. [Último acceso: 8 agosto 2019].

[13] The MathWorks, Inc., «rcosdesign,» [En línea]. Available: https://es.mathworks.com/help/signal/ref/rcosdesign.html. [Último acceso: 1 agosto 2019].

[14] A. V. Oppenheim y R. W. Schafer, «Increasing the Sampling Rate by an Integer Factor,» de *Discrete-Time Signal Processing*, Prentice-Hall, pp. 172-173.

[15] W. Damm, «Signal-to-Noise, Carrier-to-Noise, EbNo Webinar,» 2 noviembre 2010. [En línea]. Available: https://www.noisecom.com/news-home/general/signal-to-noise-webinar. [Último acceso: 3 septiembre 2019].

[16] C. R. Johnson, W. A. Sethares y A. G. Klein, «13.1 Multipath Interference,» de *Software Receiver Design*, Cambridge University Press, 2011, pp. 272-273.

[17] N. Robertson, «Multipath Channel Model Using DSP,» 14 febrero 2011. [En línea]. Available: https://dspguru.com/dsp/tutorials/multipath-channel-model-using-dsp/. [Último acceso: 23 octubre 2019].

[18] F. Gardner, «Transfer Functions of Analog PLLs,» de *Phaselock Techniques*, Wiley, 2005, pp. 6-28.

[19] F. Ling, «Digital PLLs,» de *Synchronization in Digital Communication Systems*, Cambridge University Press, 2017, pp. 147-166.

[20] L. Franks, «Carrier and Bit Synchronization in Data Communication - A Tutorial Review,» *IEEE Transactions on Communications,* vol. 28, nº 8, pp. 1107-1121, 1980.

[21] J. P. Costas, «Synchronous communications,» *Proceedings of the IEEE,* vol. 90, nº 8, pp. 1461-1466, 2002.

[22] F. Gardner, «A BPSK/QPSK Timing-Error Detector for Sampled Receivers,» *IEEE Transactions on Communications,* vol. 34, nº 5, pp. 423-429, 1986.

[23] G. Turin, «An introduction to matched filters,» *IRE Transactions on Information Theory,* vol. 6, nº 3, pp. 311-329, 1960.

[24] C. R. Johnson y W. A. Sethares, «Matched Filtering,» de *Telecommunication Breakdown*, Prentice Hall, 2003, pp. 237-239.

[25] L. Hanzo, C. H. Wong y M. S. Yee, «Introduction to Equalizers,» de *Adaptive Wireless Transceivers: Turbo-Coded, Turbo-Equalized and Space-Time Coded TDMA, CDMA and OFDM Systems*, Wiley, 2002, pp. 21-23.

[26] L. Hanzo, C. H. Wong y M. S. Yee, «2.3.1 Zero Forcing Equalizer,» de *Adaptive Wireless Transceivers: Turbo-Coded, Turbo-Equalized and Space-Time Coded TDMA, CDMA and OFDM Systems*, Wiley, 2002, pp. 32-34.

[27] A. D. Poularikas, «8.2 The LMS Algorithm,» de *Adaptive Filtering: Fundamentals of Least Mean Squares with MATLAB*, CRC Press, 2015, pp. 203-204.

[28] A. Scher, «AM-DSB-CS coherent demodulators with carrier recovery,» octubre 2015. [En línea]. Available: http://aaronscher.com/wireless_system_simulations/DSB_CS_1.html. [Último acceso: 13 noviembre 2019].