

**ANÁLISIS DEL DESEMPEÑO DE LA MODULACIÓN QPSK EN EL
USRP COMO HERRAMIENTA DIDÁCTICA EN LA ENSEÑANZA
DE TELECOMUNICACIONES**

Por

Jesús Espinoza Hernández

Tesis sometida en cumplimiento parcial de los requerimientos para el grado de

MAESTRÍA EN CIENCIAS

en

INGENIERÍA EN ELECTRÓNICA

UNIVERSIDAD AUTÓNOMA DE BAJA CALIFORNIA
FACULTAD DE CIENCIAS QUÍMICAS E INGENIERÍA

Agosto, 2010

ANÁLISIS DEL DESEMPEÑO DE LA MODULACIÓN QPSK EN EL USRP COMO HERRAMIENTA DIDÁCTICA EN LA ENSEÑANZA DE TELECOMUNICACIONES

Autor: Jesús Espinoza Hernández
Tutor: M.C. Susana Burnes Rudecino

RESUMEN

Este proyecto de tesis analiza el desempeño de la plataforma USRP (*Universal Software Radio Peripheral*) como herramienta para el estudio de las modulaciones digitales y el concepto de radio reconfigurable por software en un ambiente académico. La caracterización del sistema se realizó utilizando el software *GNURadio*. Se implementó un módem de una sola vía utilizando la técnica de modulación QPSK (modulación por corrimiento de fase en cuadratura) para transmitir datos generados por medio de un archivo y se evaluó su desempeño observando gráficamente su espectro y la constelación de fases así como también su densidad espectral de potencia. La transferencia del archivo se realizó a diferentes tasas de bits para determinar la máxima velocidad de transferencia confiable entre la PC y el USRP y analizar su tasa de error de bit utilizando un cable coaxial como medio de transferencia.

Palabras Clave: Modulador, Demodulador, Radio reconfigurable por software, USRP, QPSK

ANALYSIS OF THE PERFORMANCE OF THE USRP PLATFORM USING THE QPSK MODULATION SCHEME AS A TOOL FOR TEACHING TELECOMMUNICATIONS

Author: Jesús Espinoza Hernández
Advisor: M.C. Susana Burnes Rudecino

ABSTRACT

This thesis project analyses the performance of the USRP (Universal Software Radio Peripheral) as a tool for the study of digital modulation techniques and the software defined radio concept. The characterization of the system was done using the open source GNURadio software. A one-way modem was implemented using QPSK modulation (Quadrature Phase Shift Keying) to transmit generated data from a file and its performance was evaluated by analyzing its spectrum and signal constellation as well as its power spectral distribution. The file transmission was done at different bitrates to determine the maximum reliable transfer rate between the host PC and the USRP as well as analyzing its bit error rate over a wired coaxial medium.

Keywords: Modulator, Demodulator, Software defined radio, USRP, QPSK

	<u>Índice general</u>
	<u>página</u>
RESUMEN EN ESPAÑOL	II
ABSTRACT ENGLISH	III
Índice de figuras	VI
Índice de tablas	IX
Índice de acrónimos	X
1. Introducción	1
1.1. Antecedentes	1
1.2. Planteamiento del problema	2
1.3. Justificación	3
1.4. Importancia del estudio	3
1.5. Objetivos	4
1.6. Metodología	4
2. Fundamentos teóricos	5
2.1. Concepto de señal	5
2.2. Sistema de comunicaciones digitales	7
2.3. Concepto de modulador	9
2.4. Envolvente compleja	10
2.5. El ruido como factor de degradación de la señal transmitida	10
2.6. Concepto de demodulador	13
2.7. Modulación QPSK	15
2.8. Tasa de error de bits QPSK	19
2.9. Herramientas de medición de desempeño	19
2.9.1. Diagrama de ojo	19
2.9.2. Espectrograma	20
2.10. Sincronización	22
2.11. Concepto de radio reconfigurable por software	26
3. Estudio de la plataforma USRP y GNURadio	30
3.1. Estructura del sistema USRP	30
3.2. <i>GNURadio</i>	38
3.3. Características del lenguaje <i>Python</i>	38
3.4. Programación en <i>GNURadio</i> con <i>Python</i>	39

3.5.	Descripción del hardware y software	43
3.5.1.	Hardware utilizado	43
3.5.2.	Código fuente de GNURadio	44
3.6.	Descripción del flujo del software	47
3.6.1.	Estructura del transmisor	47
3.6.2.	Estructura del modulador DQPSK	52
3.6.3.	Modulador DQPSK con filtros polifásicos	58
3.6.4.	Estructura del receptor	61
3.6.5.	Estructura del demodulador DQPSK	62
3.6.6.	Demodulador DQPSK con filtros polifásicos	64
4.	Resultados	67
4.1.	Material utilizado	68
4.2.	Parámetros utilizados	69
4.3.	Tasa de error de bits	71
4.4.	Espectrograma de la transmisión	85
4.5.	Diagrama de ojo	87
4.6.	Constelación observada	92
4.7.	Transmisión de un archivo	100
5.	Conclusiones	104
5.1.	Trabajo futuro	105
	APÉNDICES	106
A.	Instalación del ambiente de desarrollo	107
B.	GNURadio Companion	116
	Referencias	129

<u>Figura</u>	Índice de figuras	<u>página</u>
2–1. Ejemplo de una señal continua y una señal discreta	6	
2–2. Diagrama a bloques de un sistema de comunicación digital con transmisión de datos pasa banda	8	
2–3. Función de densidad de probabilidad de la distribución Gaussiana (Sklar, 2001)	12	
2–4. Diagrama simple de un demodulador digital.	13	
2–5. Probabilidades condicionales de las funciones $p(z s_1)$ y $p(z s_2)$	15	
2–6. Diagrama de constelación para la modulación BPSK	17	
2–7. Diagrama de constelación de la modulación QPSK	18	
2–8. El diagrama de ojo formado al sobreponer varias secuencias de bits. .	20	
2–9. Parámetros de medición de un diagrama de ojo.	21	
2–10. Ejemplo de un espectrograma.	21	
2–11. Diagrama a bloques de un circuito cuadrado. (Pérez, 2004)	23	
2–12. Diagrama a bloques de un sincronizador de Lazo de Costas. (Pérez, 2004)	24	
2–13. Diagrama a bloques del sincronizador M&M. (Meyr, 1998)	26	
2–14. Diagrama a bloques de un sistema ideal de SDR	27	
2–15. Hardware proporcionado por Rockwell Collins para el cluster JTRS (McHale, 2004)	28	
2–16. Diagrama a bloques de un nodo del sistema Hydra (Wireless Networking Communications Group, 2007)	29	
3–1. Placa principal del dispositivo USRP.	31	
3–2. Estructura de un DDC implementado en el USRP	34	
3–3. Respuesta a la frecuencia del filtro CIC de 4 etapas y $R = 4$	36	
3–4. Respuesta a la frecuencia del filtro de media banda	37	

3–5. Ejemplo de un grafo de flujo en <i>GNURadio</i>	40
3–6. Grafo que envía dos senoidales a la tarjeta de audio de una PC.	43
3–7. Árbol de directorios de los ejemplos que ofrece <i>GNURadio</i>	45
3–8. Relación de módulos del ejemplo <i>benchmark</i> de <i>GNURadio</i>	46
3–9. Diagrama a bloques del transmisor DQPSK.	48
3–10. Estructura del paquete de datos.	51
3–11. Estructura del modulador DQPSK de <i>GNURadio</i>	53
3–12. Representación en el dominio de la frecuencia de filtros coseno elevado	56
3–13. Forma de onda en el dominio del tiempo para filtros de coseno elevado	57
3–14. Estructuras básicas de filtros multi tasa.	59
3–15. Descomposición del filtro decimador en su representación polifásica. .	60
3–16. Estructura del modulador DQPSK con filtros polifásicos	61
3–17. Diagrama a bloques del receptor DQPSK	61
3–18. Estructura del demodulador DQPSK	63
3–19. Estructura del demodulador DQPSK con filtros polifásicos.	64
4–1. Grafo generador de datos modulados para la evaluación del BER. . .	72
4–2. Grafo que simula un canal AWGN	74
4–3. Grafo que realiza la demodulación de la señal simulada	77
4–4. Respuesta del filtro pre-selector en el receptor para la medición del BER.	78
4–5. Gráfica comparativa del BER para la primera versión de los esquemas DQPSK y DBPSK.	83
4–6. Gráfica comparativa del BER para la segunda versión de los esquemas DQPSK y DBPSK.	84
4–7. Resultados teóricos de la herramienta <i>BERTool</i> de Matlab.	85
4–8. Aplicación en <i>Python</i> que muestra el espectrograma de lo que el USRP está capturando.	87
4–8. Señales QPSK con filtrado de coseno elevado a diferentes valores de α . .	90

4–9. Resultados de la transmisión con parámetros por defecto del programa a 100kbits/s	95
4–10. Constelación obtenida con parámetros óptimos de una transmisión a 100kbits/s	96
4–11. Programa <i>benchmark</i> TX y RX en modo consola.	97
4–12. Constelación observada al transmitir un archivo a 30kbit/s.	103
A–1. Opciones del instalador <i>Wubi</i>	108
A–2. Menú de arranque con la opción de <i>Ubuntu</i>	109
A–3. Escritorio <i>GNOME</i> de la distribución <i>Ubuntu</i>	109
A–4. Administrador de software Synaptic	112
B–1. Pantalla principal de <i>GNURadio Companion</i>	117
B–2. Pantalla de propiedades del bloque signal_source	118
B–3. Ejemplo de un grafo desarrollado en GRC	119
B–4. Pantalla para guardar el trabajo en GRC	121
B–5. Ejemplo de un grafo en ejecución	121
B–6. Bloque FFT en conjunto con el osciloscopio en GRC	122
B–7. Modificación de parámetros por medio de bloques variables en GRC .	124

	Índice de tablas	
<u>Tabla</u>		<u>página</u>
2–1. Ejemplos de señales	7	
3–1. Tarjetas auxiliares que soporta el dispositivo USRP.	33	
3–2. <i>Endpoints</i> implementados por el controlador FX2	34	
A–1. Dependencias requeridas para la instalación de <i>GNURadio</i>	111	

Índice de acrónimos

ADC	Convertidor Análogo a Digital.
AWGN	Ruido Blanco Gaussiano Aditivo.
BER	Tasa de error de bits (<i>Bit error rate</i>).
BPSK	Modulación por corrimiento de fase binaria.
CIC	Filtro Peine Integradores en Cascada.
CRC	Comprobación de redundancia cíclica.
DAC	Convertidor Digital a Análogo.
DDC	Convertidor Digital de Frecuencia hacia Abajo.
DSP	Procesador Digital de Señales.
DUC	Convertidor Digital de Frecuencia hacia Arriba.
FPGA	Matriz de compuertas programables (<i>Field Programable Gate Array</i>).
HB	Filtro de media banda.
IF	Frecuencia intermedia.
ISI	Interferencia entre símbolos.
NCO	Oscilador controlado numéricamente.
PAM	Modulación por amplitud de pulso.
PCI	Interconexión de Componentes Periféricos.
PCIe	Interconexión de Componentes Periféricos express.
QPSK	Modulación por Corrimiento de Fase en Cuadratura.
SDR	Radio Reconfigurable por Software.
SMA	Conector coaxial SubMiniatura versión A.
SNR	Relacion Señal a Ruido.
USB	Bus serial universal (<i>Universal serial bus</i>).
USRP	Periférico Universal de Radio Definido por Software.
VCO	Oscilador controlado por voltaje.

Capítulo 1

INTRODUCCIÓN

El proyecto que se documenta evalúa la plataforma USRP (*Universal Software Radio Peripheral*) como herramienta útil para la enseñanza de las técnicas de modulación digital. La evaluación consiste en introducir alguna secuencia de datos al sistema (aleatoria, datos extraídos de un archivo, etc.), generar la modulación y demodulación para la transmisión y observar los resultados.

Los sistemas de comunicaciones digitales implementados en su mayor parte por software se les conocen como SDR (Radio Definido por Software) ([Mitola, 1992](#)). El sistema se desarrolló dentro de la plataforma Linux, utilizando el software *GNURadio*.

1.1. Antecedentes

Gracias a los avances tecnológicos recientes, han surgido varios esfuerzos por desarrollar plataformas versátiles para la implementación de SDRs. La complejidad de los algoritmos de modulación, codificación, filtrado, etc., los hacen buenos candidatos para ser implementados en DSPs (Procesador Digital de Señales) como el procesador C6711 de Texas Instruments ([Abendroth, 2003](#)).

Existen varias soluciones comerciales que ofrecen un nivel más avanzado y completo para la investigación y desarrollo de aplicaciones de comunicaciones basadas en SDR. Pentek ofrece plataformas basadas en la interfaz PCI y PCIe entre otras ([Pentek Inc., 2009](#)). Estas plataformas utilizan FPGAs (del Inglés *Field Programmable Gate Arrays*) de Xilinx para implementar los bloques de DDC (Conversión digital hacia abajo) y DUC (Conversion Digital hacia Arriba), así como también

filtros de interpolación. La compañía ofrece drivers para Windows y Linux, ofreciendo así mucha flexibilidad de escoger un ambiente adecuado para el desarrollo de la aplicación.

El mundo de *Open Source* también ha realizado avances notorios en este ramo de investigación. El proyecto más sobresaliente es *GNURadio* ([GNURadio, 2009](#)). Este proyecto consiste en una biblioteca extensa con varias funciones matemáticas, así como también funciones para la implementación de algoritmos de procesamiento digital de señales, basada en los lenguajes Python y C++. La biblioteca motivó el desarrollo de un dispositivo de bajo costo, conocido como el USRP, con el propósito de utilizarlo para la transmisión y recepción de varios tipos de señales generadas por *GNURadio* ([LLC, 2009](#)). Este dispositivo está también basado en un FPGA y un par de ADCs (Convertidor Análogo a Digital) y DACs (Convertidor Digital a Análogo) respectivamente. La señal es enviada a la PC por medio del puerto USB (del Inglés *Universal Serial Bus*) utilizando el controlador CY7C68013 de la compañía Cypress. A pesar de ser un sistema *Open Source*, el USRP y *GNURadio* han sido utilizados en diversos campos de investigación y desarrollo como académicos, comerciales, militares y gubernamentales.

1.2. Planteamiento del problema

Los conceptos teóricos del área de comunicaciones digitales no consideran algunos elementos físicos sobre su implementación; por ejemplo, normalmente suponemos filtros ideales, los cuales no son posible implementar físicamente debido a varios factores como la cantidad limitada de ancho de banda, efectos de ruido, etc., por lo que se requiere alguna herramienta que nos muestre lo físicamente realizable, y contrastarlo contra lo puramente teórico, para así proporcionar al estudiante una experiencia de campo en el tema de las comunicaciones digitales.

1.3. Justificación

El estudio de los conceptos de comunicaciones normalmente ha sido dado de una manera teórica. En la mayoría de las instituciones se realizan simulaciones en Matlab o en algún otro ambiente de desarrollo. Estos paquetes proporcionan datos simulados del estudio en cuestión, ya sea la transmisión de alguna señal o el análisis de algún canal de transmisión. El USRP ofrece la oportunidad de llevar a cabo estos conceptos a un ambiente más práctico, es decir, los datos obtenidos del estudio son de los efectos reales de una implementación física de un sistema de comunicaciones. El ambiente de desarrollo *GNURadio* ofrece incluso un ambiente poderoso y flexible de utilizar ya que el lenguaje de programación utilizado, Python, es muy similar al estilo de programación de Matlab.

1.4. Importancia del estudio

El uso de una plataforma más amigable y más dinámica para el entendimiento de los conceptos de comunicaciones digitales es de gran importancia para el desarrollo académico del alumno que estudia dichos temas. Estos conceptos son normalmente estudiados dentro de ambientes de simulación para ver sus efectos y comprobar de forma analítica la teoría. La habilidad de poder llevar estos análisis teóricos a la práctica ofrece la oportunidad importante de poder visualizar los resultados obtenidos en la simulación en una implementación real donde se puede observar con mayor exactitud los efectos y resultados que se están estudiando. El uso del USRP permite no sólo aprender conceptos de comunicaciones, sino integrar también los conceptos de sistemas digitales y de procesamiento digital de señales, conectando con ellos áreas del conocimiento que faciliten al estudiante de ingeniería su preparación para la solución de problemas que requieren un conocimiento multidisciplinario. Con esto, la importancia de este estudio se vuelve más evidente al tener la posibilidad de usar un sistema que permita concentrar la atención en el problema de comunicación sin distraerse en problemas técnicos de implementación. Además proporciona

al estudiante el conocimiento de una plataforma flexible que le permita en el futuro investigar otros temas no incluidos en el curso tradicional de comunicaciones digitales, sin requerir de un equipo especializado.

Con esto, la importancia de este estudio se vuelve más evidente cuando el alumno que utilice esta plataforma pueda llevar sus conocimientos básicos a la práctica e implementar un sistema de comunicaciones completo de una manera rápida y sencilla.

1.5. Objetivos

Los objetivos del presente proyecto son los siguientes:

- Implementar la modulación y demodulación QPSK (Modulación por Corrimiento de Fase en Cuadratura) en el sistema USRP.
- Graficar la SNR (Relación señal a ruido) del rendimiento del sistema.
- Cálculo del BER (Tasa de error de bit).
- Cálculo y gráficas de densidad espectral de potencia.
- Diagrama de ojo y de constelación del esquema de modulación.

1.6. Metodología

La metodología a seguir es la siguiente:

- El desarrollo se llevará a cabo en el Sistema Operativo GNU/Linux utilizando el software *GNURadio*.
- Instalación del sistema operativo y el software.
- Validación de la instalación. Ejecución de programas de prueba proporcionados por *GNURadio*.
- Estudio de los bloques de procesamiento de *GNURadio*.
- Experimentos con el sistema USRP.
- Generación de gráficas para medir y visualizar los resultados de los experimentos.

Capítulo 2

FUNDAMENTOS TEÓRICOS

2.1. Concepto de señal

Una señal se puede definir como una cantidad que se puede medir a través del tiempo ([Sklar, 2001](#)). Esta cantidad puede representar diversas cosas dependiendo de la aplicación. Las señales pueden ser de dos tipos: análogas ó discretas. Las señales análogas, o también conocidas como continuas, se pueden definir como una función de valores reales o complejos, las cuales están definidas dentro de un intervalo de tiempo t . Este intervalo es normalmente infinito aunque también se puede acotar a un intervalo específico, dependiendo del instante de tiempo que se desea medir. Las señales discretas se definen como una función derivada de un conjunto de números acotados a un intervalo específico. Estos números son las muestras que se tomaron de la señal en tiempos discretos. Esta información es almacenada como dato digital para posteriormente ser analizado y / o manipulado. Un ejemplo de una señal continua y discreta se muestra en la figura [2-1](#).

Una señal puede ser determinística o aleatoria. Se le denomina determinística a una señal la cual no existe ninguna incertidumbre con respecto a su valor dentro de un lapso de tiempo. Las señales determinísticas pueden ser expresadas por ecuaciones matemáticas, tales como $x(t) = 5 \cos(10t)$. No es posible utilizar este tipo de ecuaciones para describir señales aleatorias, también conocidas como procesos aleatorios, aunque sí es posible analizarlas utilizando métodos estadísticos y de probabilidad ([Sklar, 2001](#)).



Figura 2–1: Ejemplo de una señal continua y una señal discreta

Las señales también pueden ser periódicas o no periódicas. Se dice que una señal es periódica si existe una constante $T_0 > 0$ tal que la siguiente ecuación se pueda satisfacer:

$$x(t) = x(t + T_0) \quad -\infty < t < \infty \quad (2.1)$$

donde t es el tiempo. El valor más pequeño de T_0 que pueda satisfacer la ecuación (2.1) se le conoce como el periodo de $x(t)$. Una señal a la cual no existe un valor T_0 que satisfaga la ecuación (2.1) se le denomina no periódica ([Sklar, 2001](#)).

Algunos ejemplos de varios tipos de señales se muestran en la tabla 2–1.

Variaciones en tiempo y espacio en una dimensión (1D)	<ul style="list-style-type: none"> ■ Acústicos: Señales sonoras, eco de murciélagos y delfines. ■ Ingeniería Electrónica: Señales emitidas por una antena de transmisión. ■ Química: Temperatura de una reacción química. ■ Finanzas: Historial del mercado de acciones. ■ Medicina: Señales de un electrocardiograma.
Variaciones en el espacio de dos dimensiones (2D)	<ul style="list-style-type: none"> ■ Ingeniería de Agricultura: vegetación y evaporación en un campo. ■ Geografía: temperatura de la superficie y mapas de presión. ■ Ingeniería de Tráfico: tasa de accidentes en un día de una ciudad.
Variaciones volumétricas de tres dimensiones (3D)	<ul style="list-style-type: none"> ■ Mecánica de fluidos: perfil de flujo-velocidad de un ala de un avión. ■ Biología: distribución CO₂ de un bioreactor.

Tabla 2–1: Ejemplos de señales

2.2. Sistema de comunicaciones digitales

Los componentes típicos de un sistema de comunicaciones digital se muestran en la figura 2–2.

La información que se desea enviar es convertida en dígitos binarios en la etapa de Formato de Mensaje. Estos dígitos son agrupados para formar mensajes digitales o símbolos para poder asegurar compatibilidad entre la información y el procesamiento de señales dentro de un sistema de comunicaciones digital. Hasta este punto la información permanece en forma de una cadena de bits. La modulación transforma



Figura 2–2: Diagrama a bloques de un sistema de comunicación digital con transmisión de datos pasa banda

los símbolos del mensaje en formas de onda compatibles con el canal de transmisión. El bloque modulación por pulsos es importante porque los símbolos deben ser convertidos de una representación binaria a una señal banda base. Este término se refiere a una señal cuyo espectro va desde 0 (o casi 0 o dc) hasta un valor finito, normalmente menos de unos cuantos megahertz. Para una aplicación que involucre la transmisión por RF, el siguiente bloque es requerido cuando el canal no soporta la propagación de señales en forma de pulsos. Para tales casos se requiere la presencia de señales pasa bandas. Este término se refiere a una señal banda base cuya frecuencia es traducida por una señal portadora a una frecuencia mucho mayor que su frecuencia original. La señal recibida puede ser interpretada como

$$r(t) = s_i(t) * h_c(t) + n(t) \quad i = 1, \dots, M \quad (2.2)$$

donde el operador $*$ representa una operación de convolución, $h_c(t)$ representa la respuesta al impulso del canal, $s_i(t)$ es la señal transmitida y $n(t)$ representa un proceso de ruido.

En el lado del receptor, el demodulador restablece $r(t)$ en pulsos banda base óptimamente formados en preparación para ser detectados.

En comunicaciones digitales los términos demodulación y detección se utilizan de una manera intercambiable ([Sklar, 2001](#)) aunque la demodulación hace énfasis

en la recuperación de la forma de onda y la detección se refiere al proceso de toma de decisión de símbolos.

2.3. Concepto de modulador

En el campo de las telecomunicaciones, la modulación es el proceso de variar una forma de onda periódica para enviar algún tipo de mensaje([Sklar, 2001](#)). La necesidad de modular una secuencia de datos para ser transmitida es relativa al tamaño de la antena. Es posible transmitir una señal sin antes haber sido modulada, pero el tamaño de la antena para poderla transmitir sería muy grande que no sería factible la transmisión. Por ejemplo, las señales de teléfonos celulares. El tamaño de las antenas son típicamente 1/4 de la longitud de onda ([Sklar, 2001](#)). Si consideramos una señal de 3000Hz acoplándola directamente sin haberla modulado antes, el tamaño de la antena se puede calcular utilizando la siguiente expresión:

$$\begin{aligned}\lambda &= \frac{c}{f} = \frac{3 \times 10^8 m/s}{3000 Hz} \\ \frac{\lambda}{4} &= 2.5 \times 10^4 m\end{aligned}\tag{2.3}$$

Si la señal se modula antes de enviarse con una portadora o envolvente de mayor frecuencia a la señal que se desea enviar, por ejemplo 900Mhz, la longitud de la antena será de 8cm. Por esta razón la modulación es un paso esencial en todos los sistemas que involucran transmisión de datos por radio.

Otro beneficio que ofrece la modulación es de poder colocar la frecuencia de la señal en otra donde los requerimientos de filtrado y amplificación, entre otros requerimientos de diseño, puedan ser logrados de una manera más eficiente. Este es el caso donde por ejemplo, en el receptor, la señal de RF que es recibida es convertida a una IF (frecuencia intermedia) para que el sistema pueda procesarla correctamente.

2.4. Envolvente compleja

La descripción de moduladores y demoduladores en la vida real se ha facilitado por el uso de la notación compleja. Cualquier forma de onda pasa banda $s(t)$ puede ser representada utilizando la notación compleja ([Sklar, 2001](#))

$$s(t) = \operatorname{Re}\{g(t)e^{j\theta(t)}\} \quad (2.4)$$

donde $g(t)$ se conoce como la envolvente compleja, expresada como

$$g(t) = x(t) + jy(t) = \|g(t)\|e^{j\theta(t)} = R(t)e^{j\theta(t)} \quad (2.5)$$

la magnitud de la envolvente es

$$R(t) = \|g(t)\| = \sqrt{x^2(t) + y^2(t)} \quad (2.6)$$

y su fase es

$$\theta(t) = \tan^{-1} \frac{y(t)}{x(t)} \quad (2.7)$$

Observando la ecuación (2.4) se dice que $g(t)$ es el mensaje de la información que se desea transmitir en forma compleja y $e^{j\theta t}$ es la envolvente en su forma compleja. El producto de estas dos representa la modulación y $s(t)$, la parte real del producto, es la forma de onda transmitida. Con esto podemos expresar $s(t)$ de la siguiente manera ([Sklar, 2001](#))

$$\begin{aligned} s(t) &= \operatorname{Re}\{(x(t) + jy(t))(\cos \omega_0 t + j \sin \omega_0 t)\} \\ s(t) &= x(t) \cos \omega_0 t - y(t) \sin \omega_0 t \end{aligned} \quad (2.8)$$

2.5. El ruido como factor de degradación de la señal transmitida

El proceso de demodulación consiste en recuperar la información (bits) de la señal transmitida con la menor cantidad de errores posible. Estos errores son causados por el ruido, el cual se puede definir como cualquier distorsión no deseada en la señal ([Sklar, 2001](#)). Podemos considerar dos causas principales de ruido. La primera

es debido al filtrado que se emplea en el transmisor, canal y receptor. Si la función de transferencia de estos filtros no es la ideal pueden llegar a causar un fenómeno conocido como ISI (interferencia entre símbolos). La otra causa de degradación de la señal es debido a la interferencia generada por el medio ambiente incluyendo interferencia con otras señales. Existe un fenómeno natural que no se puede eliminar ni controlar denominado *ruido térmico*. Este tipo de ruido es generado por el movimiento térmico de los electrones en cualquier medio conductor y puede ser descrito como un proceso aleatorio Gaussiano con media cero, es decir, $\mu = 0$. Un proceso Gaussiano $n(t)$ es una función aleatoria el cual su valor n en cualquier punto en un tiempo t esta caracterizado por la PDF (función de densidad de probabilidad)

$$p(n) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{1}{2}\left(\frac{n}{\sigma}\right)^2\right] \quad (2.9)$$

donde σ^2 es la varianza de la muestra n . Su representación gráfica se muestra en la figura 2–3.

Otra característica del ruido térmico es que su distribución de potencia es igual para todas las frecuencias, en otras palabras, su potencia esta uniformemente distribuida a través de todo el espectro. Esta característica le da el nombre ruido blanco.

Este ruido está presente en todos los sistemas de comunicación y recibe el nombre de AWGN, así agrupando todas las características anteriormente mencionadas (A de aditivo, W *white* o blanco y G de Gaussiano).

Si consideramos un sistema de comunicaciones binario que envia un 1 o un 0 en un intervalo de símbolo T

$$s_i(t) = \begin{cases} s_1(t) & 0 \leq t \leq T \quad \text{binario 1} \\ s_2(t) & 0 \leq t \leq T \quad \text{binario 0} \end{cases} \quad (2.10)$$

y le agregamos ruido aditivo blanco la señal recibida será

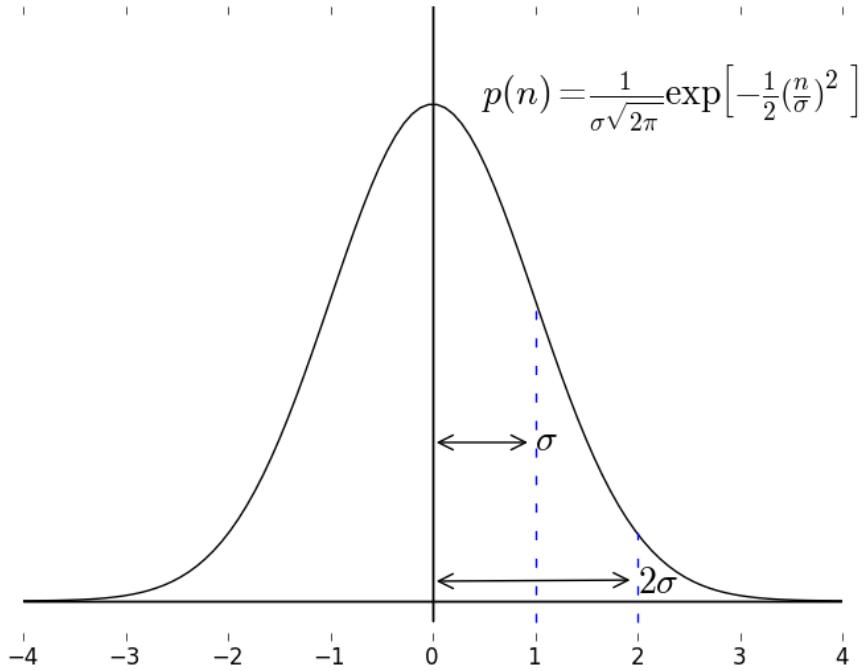


Figura 2–3: Función de densidad de probabilidad de la distribución Gaussiana ([Sklar, 2001](#))

$$r(t) = s_i(t) + n(t) \quad i = 1, 2 \quad 0 \leq t \leq T \quad (2.11)$$

donde $s_i(t)$ es el símbolo recibido y $n(t)$ es el ruido aleatorio causado por el movimiento de los electrones.

Las características del ruido AWGN son comunmente utilizadas para modelar la interferencia en el proceso de detección y tambien para el diseño de receptores, ya que este tipo de ruido siempre está presente en todos los sistemas de comunicaciones y, por lo tanto, se han desarrollado modelos de canales de transmisión que simulan el comportamiento de este tipo de ruido sobre la señal transmitida. Estos canales de transmisión se llaman canales AWGN y representan un canal donde la única causa de degradación de la señal es el ruido térmico.



Figura 2–4: Diagrama simple de un demodulador digital.

2.6. Concepto de demodulador

La demodulación es el proceso inverso de la modulación y consiste en extraer la información que se transmitió de la señal modulada. El método utilizado para llevar a cabo la recuperación de la información depende del tipo de modulación que se empleó para la transmisión. A los demoduladores también se les llama detectores. Un sistema que puede modular y demodular una señal se conoce como módem, el cual su nombre es una contracción de las palabras modulador y demodulador.

Un sistema típico para la demodulación y detección de señales se muestra en la figura 2–4.

Dentro de la etapa de demodulación, el bloque de conversión de frecuencia hacia abajo (FDC) realiza la operación de mover la frecuencia central de la señal a otra mas baja. Esta nueva frecuencia puede ser 0Hz (DC o corriente directa) o una intermedia. Este bloque normalmente se emplea para señales de tipo pasa bandas. Su empleo es totalmente opcional ya que se utiliza para acondicionar la frecuencia de la señal a una que el sistema de detección pueda utilizar adecuadamente. El siguiente bloque implementa un filtro acoplado que se utiliza para recuperar los pulsos de banda base con la mejor tasa de señal a ruido (SNR). En algunos casos se utiliza un filtro equalizador para reducir el fenómeno de ISI inducido por el canal y se introduce después del filtro acoplado. La mayoría de veces es preferible implementar un solo filtro que realice ambas operaciones para que de esa manera pueda compensar las distorsiones generadas por ambos, el transmisor y el canal.

Asumiendo que el ruido que contiene la señal es Gauseano, la salida del demodulador en la figura 2–4 entrega la siguiente prueba estadística:

$$z(T) = a_i(T) + n_0(T) \quad i = 1, 2 \quad (2.12)$$

para un sistema binario, donde $a_i(T)$ es la señal deseada, n_0 es el ruido y T es la duración de un símbolo. Como el ruido es un proceso Gauseano aleatorio, $z(T)$ es una variable Gauseana aleatoria que tendrá una media de a_1 o a_2 dependiendo si un 1 o un 0 binario fue enviado. Con esto podemos expresar las funciones de densidad de probabilidad condicionales para cada símbolo de la siguiente manera:

$$p(z|s_1) = \frac{1}{\sigma_0\sqrt{2\pi}} \exp \left[-\frac{1}{2} \left(\frac{z - a_1}{\sigma_0} \right)^2 \right] \quad (2.13)$$

y

$$p(z|s_2) = \frac{1}{\sigma_0\sqrt{2\pi}} \exp \left[-\frac{1}{2} \left(\frac{z - a_2}{\sigma_0} \right)^2 \right] \quad (2.14)$$

Una representación gráfica de estas probabilidades condicionales se muestra en la figura 2–5.

El eje x representa todo el rango de valores posibles que pueden ser recibidos y las distribuciones representan la densidad de probabilidad dado que s_1 o s_2 haya sido transmitido. Debido a que $z(T)$ es un voltaje proporcional a la energía del símbolo recibido, entre más grande sea la magnitud de $z(T)$ habrá menos errores en el proceso de decisión de símbolos. Esta decisión se realiza tomando la hipótesis resultante de la medición del umbral dada por la ecuación 2.15 (Sklar, 2001):

$$z(T) \begin{matrix} < \\ H_1 \\ > \\ H_2 \end{matrix} \gamma \quad (2.15)$$



Figura 2–5: Probabilidades condicionales de las funciones $p(z|s_1)$ y $p(z|s_2)$

donde H_1 y H_2 son dos posibles hipótesis binarias e indica que la hipótesis H_1 es la que se escoge si $z(T) > \gamma$ y H_2 se escoge si $z(T) < \gamma$. Esto es equivalente a escoger un 0 o un 1 binario dependiendo si s_1 o s_2 fue enviado.

2.7. Modulación QPSK

Este método de modulación agrupa la información en pares de bits para ser transmitidos por un canal. Cada par representa lo que se denomina un símbolo, es decir, un posible estado que puede tomar la señal modulada. QPSK es derivada de la modulación PSK que es una de las tres principales modulaciones digitales:

- ASK = *Modulación por corrimiento de amplitud*
- FSK = *Modulación por corrimiento de frecuencia*
- PSK = *Modulación por corrimiento de fase*

La modulación PSK utiliza una cantidad finita de cambios de fase derivados de una lista de patrones de bits denominada *alfabeto*. Esta lista se determina de la

siguiente manera:

$$M = 2^b \quad (2.16)$$

donde M es el total de elementos en el conjunto y b es la cantidad de bits asignados a cada elemento del conjunto. El tamaño del alfabeto lo determina la aplicación e implica consideraciones en el espacio entre símbolo en el plano complejo y la probabilidad de error de bit, y que un alfabeto muy grande aumenta la probabilidad de errores pero a su vez aumenta la eficiencia espectral de la transmisión.

La modulación PSK tiene la siguiente expresión general:

$$s_i(t) = \sqrt{\frac{2E}{T}} \cos(\omega_0 t + \phi_i(t)) \quad (2.17)$$

donde E es la energía por símbolo, T es la duración de cada símbolo, $\omega_0 t$ es la frecuencia de la portadora y el término $\phi_i(t)$ representa los cambios de fase en la señal. Estos valores son típicamente dados por la siguiente expresión:

$$\phi_i(t) = \frac{2\pi i}{M}; \quad i = 1, \dots, M \quad (2.18)$$

donde M es la cantidad de elementos del alfabeto.

La forma más simple de PSK se llama BPSK (Modulación por corrimiento de fase binaria). Este esquema utiliza un alfabeto de dos posibles valores separados por 180 grados. Estos valores se pueden observar claramente en el diagrama de constelación mostrado en la figura 2–6. Este diagrama muestra la señal en forma vectorial, proyectada en el plano complejo. La parte real se le denomina I, o componente “en fase” y la parte imaginaria se le denomina Q, o componente en “quadratura”. La señal BPSK se observa como una señal unidimensional, es decir, únicamente tiene proyección en el eje X. Cada símbolo puede modular únicamente un bit, lo cual hace esta modulación ineficiente para aplicaciones que requieran una tasa de transferencia muy alta.



Figura 2–6: Diagrama de constelación para la modulación BPSK

QPSK es otra variación de PSK que utiliza cuatro puntos en el plano complejo y por lo tanto, se considera una señal bidimensional ya que también tiene proyección en el eje Y. Como se mencionó anteriormente, este esquema agrupa los bits en pares por cada símbolo, haciendo esta modulación espectralmente más eficiente que BPSK ya que se puede transmitir más información utilizando el mismo ancho de banda. El diagrama de constelación se muestra en la figura 2–7.

La señal transmitida tiene la siguiente forma ([Sklar, 2001](#))

$$s(t) = I(t) \cos(2\pi f_0 t) + Q(t) \sin(2\pi f_0 t) \quad (2.19)$$

donde $I(t)$ y $Q(t)$ son las señales moduladas y f_0 es la frecuencia de la envolvente.

En el receptor estas dos señales son demoduladas utilizando un demodulador coherente, es decir, un demodulador que sincroniza la fase de su oscilador local con la fase del oscilador del modulador. Tal receptor multiplica la señal por separado con una señal seno y coseno para producir los estimados de $I(t)$ y $Q(t)$ respectivamente.



Figura 2–7: Diagrama de constelación de la modulación QPSK

Matemáticamente $I(t)$ puede ser demodulada multiplicando la señal transmitida con una señal coseno

$$\begin{aligned} r_i(t) &= s(t) \cos(2\pi f_0 t) \\ &= I(t) \cos(2\pi f_0 t) \cos(2\pi f_0 t) + Q(t) \sin(2\pi f_0 t) \cos(2\pi f_0 t) \end{aligned} \quad (2.20)$$

Utilizando las identidades trigonométricas

$$\begin{aligned} \cos \theta \cos \varphi &= \frac{1}{2}[\cos(\theta + \varphi) + \cos(\theta - \varphi)] \\ \sin \theta \cos \varphi &= \frac{1}{2}[\sin(\theta + \varphi) + \sin(\theta - \varphi)] \end{aligned} \quad (2.21)$$

se puede escribir

$$\begin{aligned} r_i(t) &= \frac{1}{2}I(t)[1 + \cos(4\omega_0 t)] + \frac{1}{2}Q(t)\sin(4\omega_0 t) \\ r_i(t) &= \frac{1}{2}I(t) + \frac{1}{2}[I(t)\cos(4\omega_0 t) + Q(t)\sin(4\omega_0 t)] \end{aligned} \quad (2.22)$$

Aplicando un filtro pasa bajas a $r_i(t)$ se pueden eliminar los términos de frecuencia (los que tienen el término ω_0) dejando únicamente el término $I(t)$. Similarmente se puede multiplicar $s(t)$ por una señal senoidal y luego por un filtro pasa bajas para obtener $Q(t)$.

2.8. Tasa de error de bits QPSK

El estudio de la tasa de error de bits es de gran importancia ya que nos proporciona una medida del rendimiento del sistema de comunicaciones. Esta medida se interpreta en términos de probabilidad de error de un símbolo enviado P_E dado que fue corrompido por el ruido del canal. El ruido se puede definir como una perturbación no deseable en la señal que se está transmitiendo ([Sklar, 2001](#)). Este fenómeno siempre existe en un canal de transmisión y generalmente no se puede evitar, por lo que es necesario tomarlo en cuenta durante el diseño de un sistema de comunicación.

2.9. Herramientas de medición de desempeño

Las herramientas que se mencionan en ésta sección proporcionan un panorama completo del rendimiento de un sistema de comunicaciones, ya sea de la etapa de transmisión o recepción y son ampliamente utilizadas para verificar el análisis teórico o práctico.

2.9.1. Diagrama de ojo

Una manera eficiente de visualizar la calidad de la señal en el receptor es por medio de un diagrama de ojo. Este diagrama contiene todas las posibles secuencias de bits que muestran debilidades en el diseño del sistema. Un ejemplo se muestra en la figura 2–8([Foster, 2004](#)).

La forma convencional de generar el diagrama de ojo es por medio de un osciloscopio en modo de persistencia. Cada captura es obtenida mediante un reloj de sincronización obtenido de la misma forma de onda recibida. El modo de persistencia causará que los trazos de la onda se sobrepongan una con otra hasta formar el

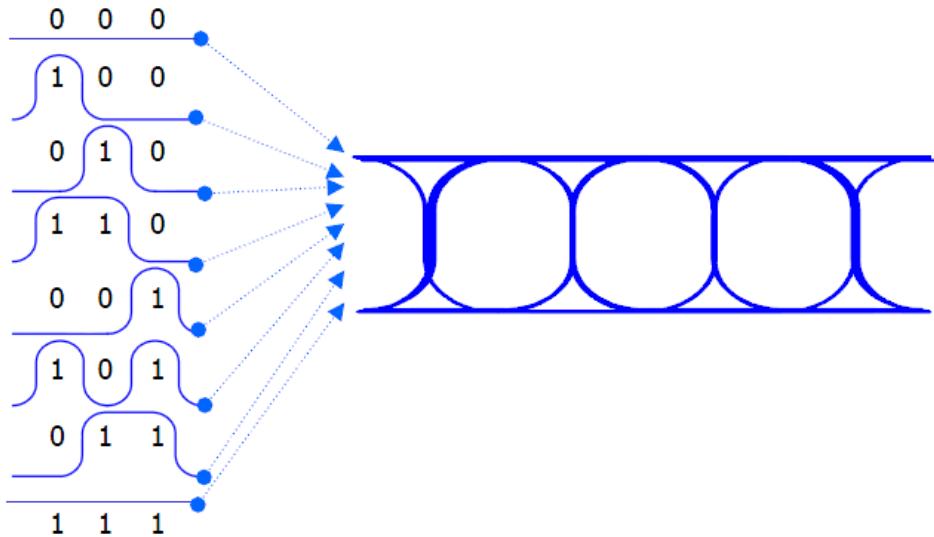


Figura 2–8: El diagrama de ojo formado al sobreponer varias secuencias de bits.

patrón de ojo sobre la pantalla. Los parámetros que se pueden medir se muestran en la figura 2–9(Breed, 2005).

La distorsión en el tiempo de muestreo es generada por el fenómeno de ISI. La diferencia en tiempo pico a pico de los cruces por cero se le conoce también como *jitter*. El momento óptimo para iniciar el muestreo de un símbolo es en la parte que está más abierta del ojo. Conforme el SNR es más bajo el ojo comenzará a cerrarse haciendo así más difícil y, por consecuencia, más erróneo el muestreo de la señal recibida.

2.9.2. Espectrograma

El espectrograma, también conocido como gráfica de cascada (*waterfall*), muestra las variaciones de la densidad espectral de la señal a través del tiempo. Utiliza tres parámetros para mostrar una imagen representando la distribución espectral: el tiempo, la frecuencia y la amplitud. El eje X representa el tiempo, el eje Y la frecuencia y una escala de colores muestra la intensidad de cada punto en la imagen. Los ejes X y Y pueden ser invertidos para visualizar la imagen de abajo hacia arriba. Un ejemplo de un espectrograma se muestra en la figura 2–10.



Figura 2–9: Parámetros de medición de un diagrama de ojo.

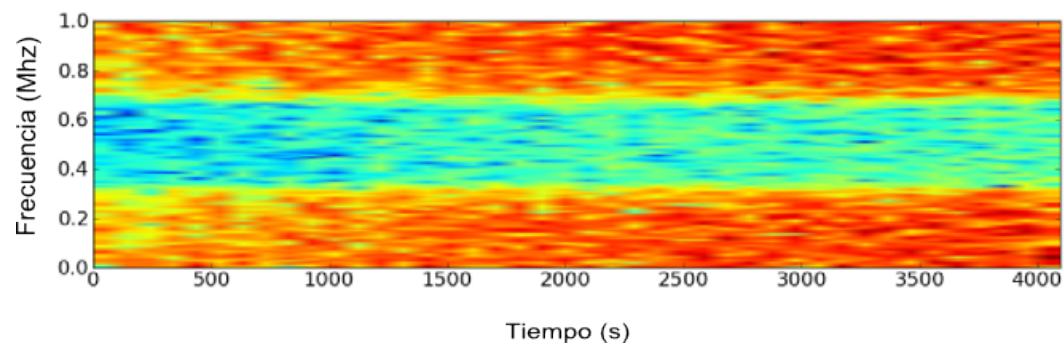


Figura 2–10: Ejemplo de un espectrograma.

2.10. Sincronización

El propósito de un receptor es tomar la señal transmitida y recuperar el mensaje que se transmitió. Para lograr esto el receptor tiene que muestrear la señal para convertirla en un formato digital. Como se observa en los parámetros del diagrama de ojo 2–9, el punto óptimo para muestrear un símbolo es en la zona más abierta que corresponde al punto medio de cada símbolo. Si el muestreo se realiza en la transición de un símbolo a otro los valores resultantes pueden llegar a ser erróneos.

Si asumimos que T es la tasa de muestreo, τ es el tiempo que tarda en llegar el primer símbolo y δ el tiempo que tarda la señal en viajar del transmisor al receptor entonces el símbolo $(k + 1)$ empieza en $\tau + kT$ y tarda $\tau + kT + \delta$ en llegar a su destino. El receptor comienza a muestrear en el momento n y a partir de ese punto muestrea cada $n + kT$. Si el mejor punto para muestrear ocurre en $\tau + kT + \delta + T/2$ entonces se puede hacer que n sea:

$$n = \tau + \delta + T/2 \quad (2.23)$$

y de esta manera se logra recuperar el mensaje transmitido sin ningún problema. Mas sin embargo, al observar bien la ecuación 2.23 se puede ver dos problemas inmediatamente: el receptor no sabe en qué tiempo se inició la transmisión y adicionalmente, no sabe cuánto tarda en llegar la señal. Por lo tanto la ecuación 2.23 contiene dos variables desconocidas. Para resolver este problema se requiere emplear métodos de sincronización que permitan extraer la frecuencia y el reloj de la señal recibida.

Existen varios métodos de sincronización en receptores digitales, tales como:

- Sincronización de fase de símbolos: Escoger el momento adecuado para tomar la muestra (Escoger T).
- Sincronización de frecuencia de símbolos: Compensar por las diferencias en frecuencia de ambos osciladores (TX y RX).

- Sincronización de fase de portadora: Alinear la fase de la portadora en el receptor con la portadora del transmisor.
- Sincronización de frecuencia de portadora: Comenzar la frecuencia de la portadora en el receptor con la del transmisor.
- Sincronización de trama: Detectar el inicio de cada mensaje.

Para poder realizar la sincronización de fase se pueden emplear circuitos basados en lazos de amarre de fase (PLL por sus siglas en Inglés *phase-locked loop*). Dos circuitos comunes son el circuito cuadrado y el lazo de costas ([Pérez, 2004](#)). El diagrama a bloques representativo de un circuito cuadrado se muestra en la figura [2–11](#).

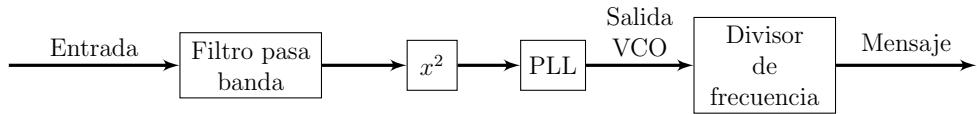


Figura 2–11: Diagrama a bloques de un circuito cuadrado. ([Pérez, 2004](#))

La señal entra al circuito donde se filtra para reducir el ancho de banda y se eleva al cuadrado para quitar la modulación y generar la segunda armónica de la frecuencia portadora, la cual es rastreada con la fase por el PLL. La frecuencia de salida del VCO (Oscilador controlado por voltaje) del PLL se divide entre dos y se utiliza como referencia de fase para los detectores de producto.

El lazo de costas utiliza dos circuitos de rastreo en paralelo, I y Q, para derivar el producto de los componentes I y Q de la señal que maneja el VCO. Una vez que la frecuencia del VCO sea igual a la frecuencia de la portadora suprimida, el producto de las señales I y Q producirá un voltaje de error proporcional a cualquier error de fase del VCO. Este voltaje controla la fase y la frecuencia del VCO. El diagrama a bloques del lazo de costas se muestra en la figura [2–12](#)



Figura 2–12: Diagrama a bloques de un sincronizador de Lazo de Costas. ([Pérez, 2004](#))

Desde el punto de vista operacional, los sincronizadores de reloj se pueden catalogar de dos formas: sincronizadores de rastreo de error (lazo cerrado) y sincronizadores de lazo abierto ([Meyr, 1998](#)). Los sincronizadores de rastreo de error siguen el principio del circuito PLL para extraer el reloj. La señal de entrada es comparada con una señal generada localmente por un VCO por un detector de error de sincronía el cual su salida produce una estimación del signo y la magnitud del error de sincronía

$$e = \varepsilon - \hat{\varepsilon} \quad (2.24)$$

donde ε es el error de la señal PAM (Modulación por amplitud de pulso) de entrada y $\hat{\varepsilon}$ es la estimación de sincronía. La salida filtrada del detector de error de sincronía ajusta la estimación $\hat{\varepsilon}$ para reducir el error e . Esta estimación es el retraso normalizado de la señal de referencia, la cual activa la etapa de muestreo.

En el sincronizador de lazo abierto, la señal PAM entra al detector de error de sincronía el cual mide el valor instantáneo de ε ([Meyr, 1998](#)). Las mediciones a la salida del detector de error pasan por un filtro de promedio para producir la estimación $\hat{\varepsilon}$ y así activar la etapa de muestreo.

Como se verá en el capítulo [3](#), *GNURadio* utiliza un sincronizador de lazo cerrado llamado *sincronizador de Mueller y Müller* (M&M por las iniciales de sus autores). Su diagrama a bloques funcional se muestra en la figura [2–13](#).

Como el sincronizador utiliza los resultados de decisión de símbolos del receptor se dice que es un sincronizador *dirigido por decisiones* ([Litwin, 2011](#)). El algoritmo requiere únicamente una muestra para representar cada símbolo y el error se calcula de la siguiente manera

$$e_n = (y_n \cdot \hat{y}_{n-1}) - (\hat{y}_n \cdot y_{n-1}) \quad (2.25)$$

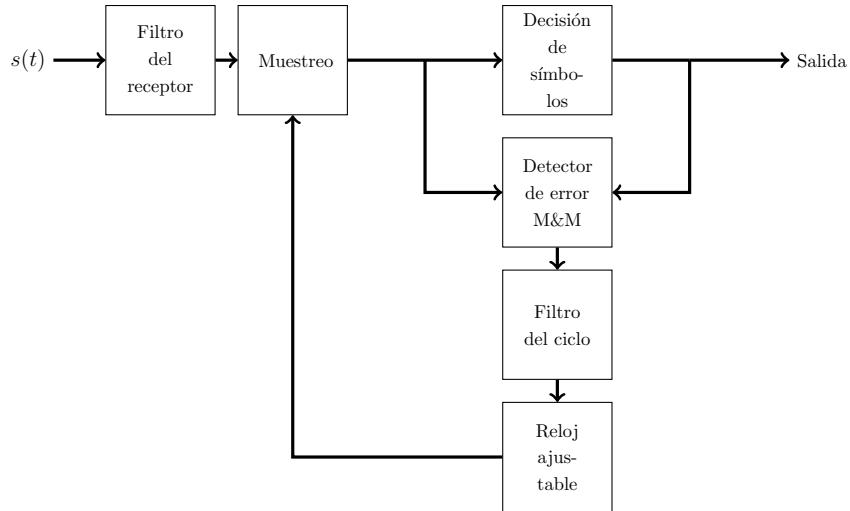


Figura 2–13: Diagrama a bloques del sincronizador M&M. ([Meyr, 1998](#))

donde y_n es la muestra del símbolo actual y y_{n-1} es la muestra del símbolo previo. El símbolo actual y previo producido por el sincronizador dirigido por decisiones esta representado por \hat{y}_n y \hat{y}_{n-1} respectivamente. El sincronizador requiere que previamente se haya realizado la recuperacion de la fase ya que el algoritmo es muy sencible a offsets en la portadora. La discusión matemática de este sincronizador esta fuera del alcance de este trabajo pero se invita al lector a consultar la bibliografía si desea aprender mas sobre la composición y funcionamiento interno de este y otros sincronizadores.

2.11. Concepto de radio reconfigurable por software

El concepto de radio definido por software fue primeramente establecido por Joseph Mitola([Mitola, 1992](#)). Comunmente se les llaman SDR por sus siglas en Inglés (*Software Defined Radio*). Este tipo de radio consiste en reemplazar los moduladores, filtros, amplificadores, entre otros componentes, de un sistema típico de RF y reemplazarlos por software. Lo óptimo es conectar la antena lo más cerca al procesador. Un sistema básico puede consistir en una PC, una tarjeta de sonido y alguna circuitería de RF para acoplar la antena. Un diagrama a bloques del sistema básico

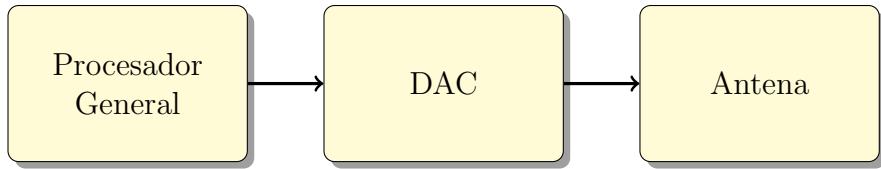


Figura 2–14: Diagrama a bloques de un sistema ideal de SDR

de un SDR se muestra en la figura 2–14. Para un sistema receptor SDR lo único que se tiene que cambiar es el bloque de DAC por un ADC. Con el apoyo del procesamiento digital de señales dentro del procesador, este tipo de radio puede trabajar con diferentes tipos de señales, codificaciones y modulaciones con tan solo cambiar el software. Algunas ventajas que estos sistemas ofrecen son: bajo costo, un factor de tamaño reducido y una mayor facilidad de mantenimiento y actualización.

Los requerimientos del procesador general para la implementación de algoritmos que llevan a cabo las tareas de modulación, codificación, entre otras, son tan grandes que hoy en día aún no hay plataformas suficientemente poderosas para implementar el SDR ideal. Sin embargo, los avances tecnológicos de hoy están permitiendo el desarrollo de nuevos procesadores con capacidades suficientemente poderosas para esta aplicación en especial. Algunos procesadores como el OMAP3 de la Texas Instruments ([Texas Instruments, 2008](#)) y los FPGAs Virtex de Xilinx ([Lyrtech, 2009](#)) ofrecen plataformas altamente flexibles y con capacidad suficiente para la implementación de algoritmos sofisticados de modulación y codificación, así como también filtrado entre otros.

Algunas de las áreas donde se aplican los conceptos de los SDRs son por ejemplo en el área militar. El sistema de comunicaciones del ejército militar de Estados Unidos está siendo remplazado por un sistema basado en SDRs con la capacidad de poder establecer un enlace sin importar el tipo de radio y con capacidad de trabajar con diversas redes de comunicación. La ventaja principal que los líderes militares ven en el uso de SDRs es la habilidad de poder configurar el sistema para utilizar diversas capas de protocolos y formatos sin actualizar el hardware. El programa JTRS (por



Figura 2–15: Hardware proporcionado por Rockwell Collins para el cluster JTRS (McHale, 2004)

sus siglas en Inglés *Joint Tactical Radio Systems*) es un SDR que permitirá a los soldados comunicarse con una amplia cantidad de redes de comunicación así como redes que utilizan arquitecturas viejas ([McHale, 2004](#)). El plan de este programa es remplazar el hardware tradicional de los radios utilizados en el ejército militar con dispositivos que puedan emular las capacidades de cualquier radio.

En la figura 2–15 se muestra el hardware proporcionado al ejército militar por el sector de comunicaciones de Rockwell Collins.

Otro ejemplo de una aplicación en el área académica es el proyecto Hydra de la Universidad de Texas en Austin ([Wireless Networking Communications Group, 2007](#)). Este proyecto consiste en un sistema de pruebas enfocado a redes que soportan múltiples saltos inalámbricos y donde la red toma ventaja de técnicas sofisticadas como OFDM (Multiplexación por División de Frecuencias Ortogonales) y MIMO (Múltiples Entradas Múltiples Salidas). El sistema se basa en el proyecto de *GNU-Radio* y su plataforma de desarrollo USRP ([LLC, 2009](#)) para la implementación de la etapa de RF y del software. La estructura de este sistema se muestra en la figura 2–16.

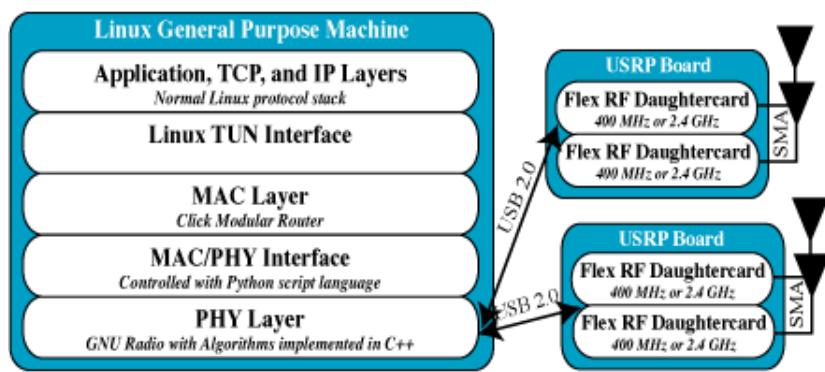


Figura 2–16: Diagrama a bloques de un nodo del sistema Hydra ([Wireless Networking Communications Group, 2007](#))

Capítulo 3

ESTUDIO DE LA PLATAFORMA USRP Y GNURADIO

En este capítulo se aplican los conceptos de la modulación QPSK para analizar el rendimiento de la plataforma USRP. El experimento consistió en elaborar un programa que realice la modulación utilizando el lenguaje *Python* y *GNURadio* bajo el ambiente Linux. Este programa interactúa con el USRP por medio de un puerto USB, intercambiando la información que se transmite y recibe. El análisis se visualiza en la PC utilizando las herramientas que *GNURadio* proporciona.

La transmisión se realizó utilizando las tarjetas LFTX y LFRX de la compañía Ettus. Estas tarjetas proporcionan acceso directo a las salidas del DAC y entradas del ADC respectivamente sin ningún tipo de amplificación y filtrado. Dependiendo de la aplicación será necesario acoplar una etapa de RF que permita acondicionar la señal y poder ser transmitida correctamente.

3.1. Estructura del sistema USRP

El sistema USRP es un dispositivo que permite diseñar radios reconfigurables por software. El dispositivo es solamente una parte de la estructura completa de un SDR. Su función principal es actuar como la sección de frecuencia intermedia de un sistema de comunicaciones.

La figura 3–1 muestra los componentes que forman un sistema de comunicaciones implementado con el USRP. Todo el procesamiento en banda base, es decir, modulaciones, codificaciones, entre otros, se lleva a cabo en una PC x86. Esta PC contiene el programa *Python* que realiza todo el procesamiento digital con la señal

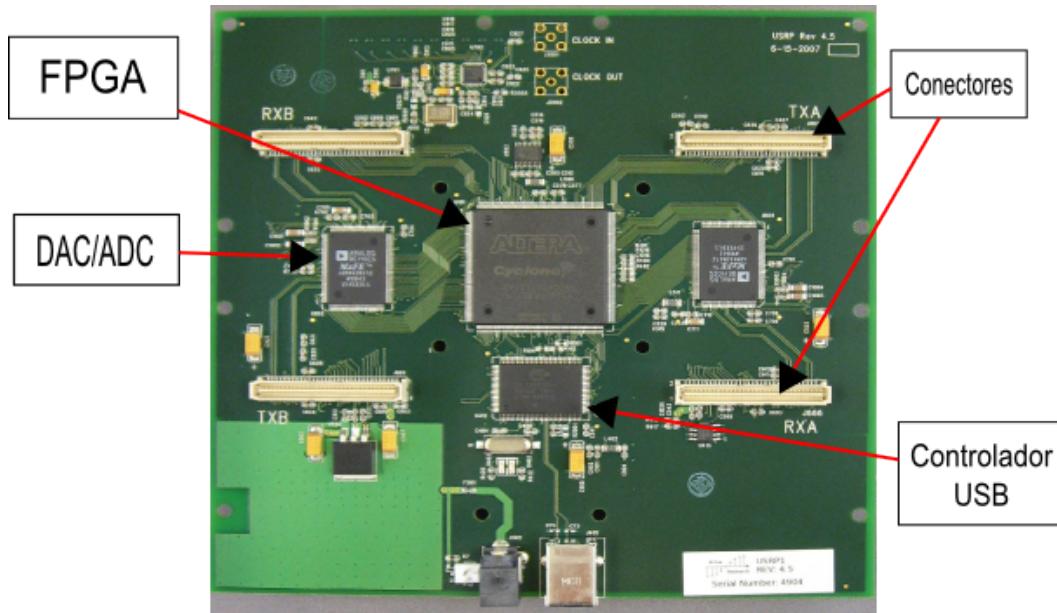


Figura 3–1: Placa principal del dispositivo USRP.

o señales que se desean trabajar. Esto se lleva a cabo utilizando las herramientas proporcionadas por *GNURadio* .

Los componentes principales del USRP son el FPGA Altera Cyclone EP1C12, los codec AD9862 de alta velocidad , las interfaces de las tarjetas auxiliares y el controlador Cypress FX2 para la comunicación por el puerto USB. La tarjeta cuenta con 4 conectores donde se pueden conectar 2 tarjetas TX y 2 RX o 2 tarjetas RFX (transmisor y receptor en una sola tarjeta auxiliar). Cada tarjeta puede acceder a dos de los 4 ADC/DACs. Si la tarjeta utiliza muestreo real (sin usar componentes en fase y cuadratura o IQ) entonces esto permitirá tener 2 secciones de RF independientes, para un total de 4. Si se utiliza muestreo complejo IQ entonces cada tarjeta tendrá una sola etapa de RF, para un total de 2 en todo el sistema. Cada tarjeta auxiliar tiene dos conectores SMA (Conector coaxial SubMiniatura versión A) para transmitir o recibir señales y una EEPROM (Memoria de solo lectura borrable y programable eléctricamente) con bus de datos I²C (Bus de comunicación en serie, *Inter-Integrated Circuit* por sus siglas en Inglés) que identifica la tarjeta al sistema.

Esto permite al software en la PC configurar el sistema basándose en la tarjeta que esté instalada. Las tarjetas que actualmente soporta el USRP se muestran en la tabla 3-1.

El controlador FX2 integra un CPU 8051 con un controlador de USB de alta velocidad que implementa tres *endpoints* lógicos para la comunicación con el FPGA y la PC como se describe en la tabla 3-2. Cada *endpoint* establece una vía de comunicación entre el dispositivo USB y el *host*. El *endpoint* 0 es el de control y es necesaria su implementación para que el dispositivo pueda ser compatible y a su vez, ser certificado por el estándar de USB ([Universal Implementers Forum, 2000](#)). Los *endpoints* 2 y 6 son de tipo Bulk, que de acuerdo a lo especificado por el estándar USB, son los que permiten el mayor rendimiento de transferencia (512 bytes por paquete). Estos son utilizados para enviar los datos de la señal al FPGA. El FX2 utiliza una interfaz de propósito genérico (GPIF por sus siglas en Inglés) para proporcionar un bus de datos al mundo exterior, esto con el propósito de facilitar la interfaz a otros dispositivos. En este caso el GPIF se conecta directamente al FPGA con una tasa de transferencia de 96 MB/sec.

El FPGA se encarga de realizar operaciones de alta velocidad y de reducir la tasa de datos a una más adecuada para que la señal pueda ser transferida por el puerto USB. Para la etapa de RX, la configuración básica contiene dos convertidores digitales hacia abajo (DDC) implementados con filtros peine integradores en cascada (CIC por sus siglas en Inglés) de 4 etapas y filtros de media banda (*Half-band* o HB) de orden 31. El propósito del DDC es transformar la señal pasa bandas a banda base. Esto se logra centrándola en 0Hz, así eliminando la portadora. La estructura de un DDC se muestra en la figura 3-2.

EL USRP, en su configuración básica, usa dos DDCs de dos entradas cada uno. La señal compleja que entregan los ADCs es multiplicada por otra señal con una

Tarjeta	Descripción	Frecuencia de operación
BasicTX y RX	Transmisor y receptor para ser utilizados con equipo externo de RF. Sus salidas están acopladas por un transformador de forma directa a los ADC/DACs con una impedancia de 5Ω .	1Mhz-250Mhz
LFTX y LFRX	Similar a las tarjetas BasicTX y RX con la distinción las salidas estan acopladas por amplificadores diferenciales en lugar de transformadores. Esto les permite trabajar con frecuencias hasta DC. Adicionalmente cuentan con un filtro pasabajas con frecuencia de corte de 30Mhz.	DC-30Mhz
TVRX	Sistema receptor de VHF y UHF basando en un sintonizador de TV con un ancho de banda de canal de 6Mhz.	50Mhz-860Mhz
DBSRX	Sistema receptor con filtro de canal controlado por software de 1Mhz a 60Mhz.	800Mhz-2.4Ghz
RFX400	TX y RX en una sola tarjeta con un ancho de banda de canal de 30Mhz.	400Mhz-500Mhz
RFX900	TX y RX con 200mW(23dBm) de potencia.	750Mhz-1050Mhz
RFX1200	TX y RX con 200mW de potencia que cubre las bandas satelitales.	1150Mhz-1450Mhz
RFX1800	TX y RX con 100mW(20dBm) de potencia	1.5Ghz-2.1Ghz
RFX2400	TX y RX con 50mW(17dBm) de potencia con un filtro pasabandas en el rango de la banda ISM(2400-2433Mhz). El filtro se puede deshabilitar para utilizar todo el rango de frecuencias.	2.3Ghz-2.9Ghz
XCVR2450	TX y RX que cubre toda la banda ISM así como algunas bandas japonesas.	2.4-2.5Ghz y 4.9-5.9Ghz
WBX	TX y RX que cubre varias bandas incluyendo televisión, comunicaciones móviles, sensores inalámbricos, etc.	50Mhz-2.2Ghz

Tabla 3–1: Tarjetas auxiliares que soporta el dispositivo USRP.

Endpoints	Descripción
0	Control/Status
2	PC → FPGA
6	FPGA → PC

Tabla 3–2: *Endpoints* implementados por el controlador FX2

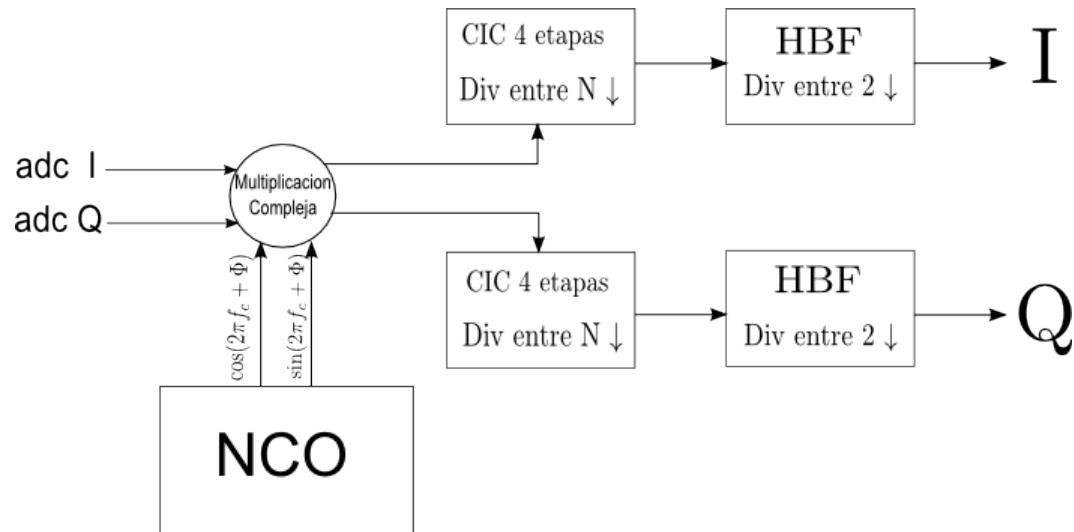


Figura 3–2: Estructura de un DDC implementado en el USRP

frecuencia intermedia constante generada por un oscilador controlado numéricamente (NCO por sus siglas en Inglés). La señal resultante se encuentra ahora centrada en 0Hz. Se prosigue a introducirla al filtro CIC para realizar una decimación por N , donde N es especificado por el usuario por medio del software. La función de transferencia del filtro CIC en el dominio Z es la siguiente (Donadio, 2000):

$$H(z) = H_I^N(z)H_C^N(z) = \frac{(1 - z^{-RM})^N}{(1 - z^{-1})^N} = \left(\sum_{k=0}^{RM-1} z^{-k} \right)^N \quad (3.1)$$

Donde: H_I^N = es la función de transferencia de la etapa integradora

H_C^N = es la función de transferencia de la etapa filtro peine

R = es la tasa de decimación o interpolación

M = número de muestras por etapa

N = número de etapas por filtro

Es de notarse que, la variable Z en la ecuación 3.1 es distinta a la utilizada en la ecuación 2.12 ya que en la ecuación 3.1 se refiere a la transformada Z y en la ecuación 2.12 se refiere a un proceso estocástico. El USRP implementa este filtro con los siguientes parámetros: $R = [0, 4]$, $M = 1$ y $N = 4$. La respuesta a la frecuencia de esta configuración se muestra en la figura 3-3.

La última etapa del DDC es el filtro HB el cual realiza una decimación de dos y ayuda a rechazar cualquier banda no deseada por las operaciones previas. La función de transferencia del filtro HB (Nguyen, 2000) es la siguiente:

$$H(z) = \sum_{n=0}^{N-1} h(n)z^{-n} \quad (3.2)$$

Estos filtros tienen las siguientes restricciones:

- $N-1$ debe ser par
- $h(n) = h(N - 1 - n)$

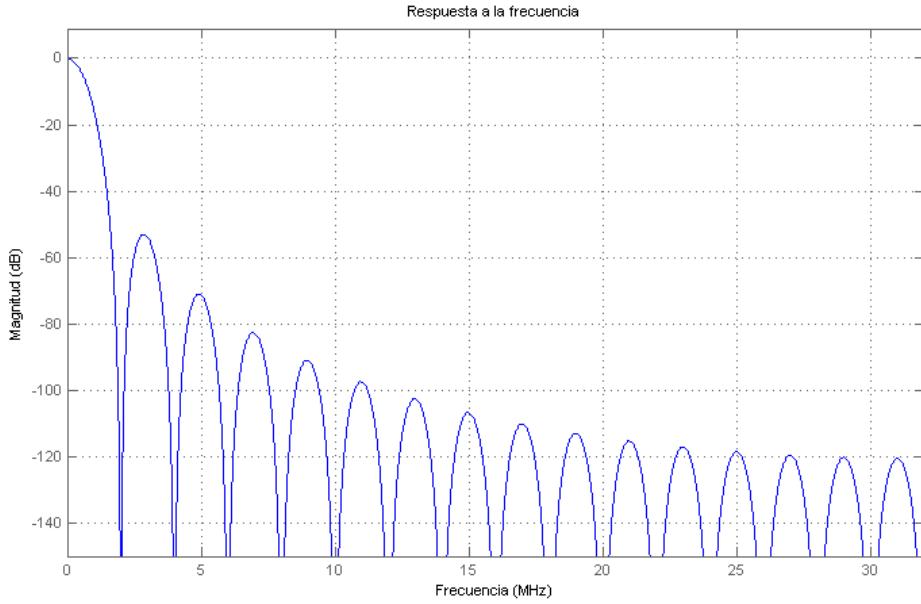


Figura 3-3: Respuesta a la frecuencia del filtro CIC de 4 etapas y $R = 4$

La respuesta a la frecuencia es

$$H(e^{j\omega}) = e^{-j\omega N-1/2} H_0(e^{-j\omega}) \quad (3.3)$$

donde $H_0(e^{-j\omega})$ representa la respuesta a la amplitud con valor real.

Las características del filtro HB de acuerdo al USRP son las siguientes:

- Orden 31
- Decimador 2
- Pasa bajas

En la figura 3-4 se muestra la respuesta a la frecuencia del filtro HB. Existe simetría con respecto a la banda $\pi/2$. Esta es una de las características de estos filtros. La respuesta al impulso es ([Nguyen, 2000](#)):

$$h(n) = \begin{cases} 0, & n - \frac{N-1}{2} = \text{par y diferente a } 0 \\ \frac{1}{2}, & n = \frac{N-1}{2} \end{cases} \quad (3.4)$$

Debido a esta simetría estos filtros son muy eficientes y fáciles de implementar ya que aproximadamente el 50 % de sus coeficientes son 0.

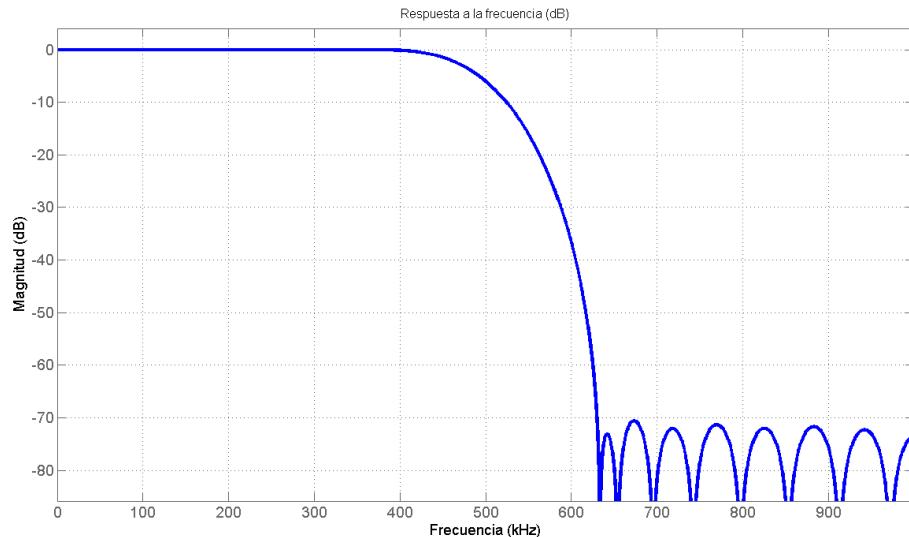


Figura 3–4: Respuesta a la frecuencia del filtro de media banda

La combinación del filtro CIC y HB dan un factor de decimación mínimo de 8. El ADC tiene una taza de muestreo de 64Ms/S, por lo tanto, el ancho de banda total es de 32Mhz. Los datos enviados son de 16 bits para I y Q, es decir, 4 bytes por muestra compleja. Esto resulta en un ancho de banda efectivo de 8Mhz (32Mhz/4bytes). El rango de decimación que soporta el USRP es [8, 256]. El ancho de banda de la señal no es afectado por la operación de decimación, únicamente su frecuencia. Conforme el valor de decimación sea mayor la tasa de transferencia será menor. El usuario puede ajustar este valor según las necesidades de su aplicación.

Para la etapa de TX el proceso es el inverso. El FPGA contiene filtros CIC interpoladores que se encargan de interpolar la señal, elevarla a la frecuencia intermedia y después enviarla a los DACs. El proceso de conversión digital hacia arriba (DUC) está contenido en el chip AD9862 y no en el FPGA como lo es en el proceso de RX. Este chip contiene ambos ADCs y DACs pero únicamente en la etapa del TX realiza algún otro procesamiento sobre la señal. La frecuencia de muestreo de los DACs es de 128MS/s de 14 bits cada uno, lo cual da una frecuencia Nyquist de 64MS/s. La salida proporciona 1V pico a una carga diferencial de 50Ω . Existe

un amplificador de ganancia programable (PGA por sus siglas en Inglés) que puede proporcionar hasta 20dB de ganancia y es programable por software. Las señales del DAC son en base a corriente y varían entre 0 y 20mA.

3.2. *GNURadio*

El dispositivo USRP fue diseñado para ser utilizado principalmente, pero no exclusivamente, por la herramienta de desarrollo *GNURadio*. Esta herramienta es abierta (*Open Source*) y consiste en un sistema de procesamiento de señales implementado a base de bloques que se ligan uno con el otro para formar un sistema completo de comunicaciones. Las aplicaciones que se desarrollan son principalmente implementadas con el lenguaje *Python*, mientras que las operaciones críticas de procesamiento de señales son implementadas en C++ utilizando las extensiones de punto flotante del CPU cuando sea posible. Cabe mencionar que aunque la función principal de esta herramienta no es realizar simulaciones, tiene la capacidad de generar sistemas de comunicaciones utilizando datos obtenidos previamente por medio de alguna simulación o generador. Esto es útil cuando no se cuenta con hardware de RF para llevar a cabo la aplicación.

GNURadio no está limitado a trabajar únicamente con el USRP. Debido a la naturaleza abierta de la herramienta, se puede diseñar bloques que encapsulen algún hardware especial. La programación del *driver* se lleva a cabo en C, es encapsulado con las clases de C++ para implementar el bloque y después es encapsulado nuevamente en *Python* para ser utilizado en la aplicación general. De esta manera el usuario no se preocupa por los detalles del funcionamiento del hardware, únicamente le proporciona los parámetros adecuados al bloque y éste se encarga de establecer la comunicación adecuada.

3.3. Características del lenguaje *Python*

El lenguaje *Python* es un lenguaje dinámico utilizado mucho en varios tipos de aplicaciones como desarrollo de aplicaciones *Web*, bases de datos, procesamiento

numérico y científico, entre otros. Se dice que es dinámico porque es un lenguaje interpretado, es decir, las instrucciones son interpretadas y ejecutadas por medio del interprete de *Python*. Esto es a diferencia de los lenguajes estáticos como C y C++ los cuales generan programas binarios con instrucciones que el CPU entiende directamente. A pesar de ser un lenguaje interpretado, *Python* es muy rápido en su ejecución de instrucciones para la mayoría de aplicaciones. Algunas de sus características principales son:

- Sintaxis clara y legible
- Soporte de programación orientada a objetos
- Soporte para generar estructuras jerárquicas de código por medio de paquetes.
- Manejo de excepciones
- Librería estándar y soporte de librerías externas para casi cualquier tipo de tarea
- Módulos y extensiones pueden ser fácilmente programados en C y C++

La página oficial de *Python* ofrece buena documentación sobre el uso del lenguaje, y su librería estándar. También ofrece tutoriales para iniciar rápidamente a desarrollar aplicaciones.

3.4. Programación en *GNURadio* con *Python*

Como se mencionó en el capítulo 3.2, la programación se lleva a cabo principalmente en el lenguaje *Python*. El lenguaje C++ también se utiliza pero únicamente si se desea desarrollar un módulo nuevo o modificar los ya existentes.

GNURadio es un conjunto de módulos de *Python* con la capacidad de realizar diversas operaciones de procesamiento digital de señales para el desarrollo de sistemas de radios configurables por software. Es posible mezclar diferentes módulos de otros *frameworks* o bibliotecas junto con las de *GNURadio* para incluir mayor soporte al que ofrece.

El concepto que se aplica para los programas realizados con esta biblioteca es el de grafos. Cada aplicación cuenta con una serie de nodos o bloques conectados entre sí para crear una cadena completa de RF llamada grafo de flujo. Los bloques se encargan de realizar alguna operación sobre la información que fluye a través del grafo los cuales pueden ser operaciones matemáticas, envío de datos a algún puerto, etc., mientras que las aristas transportan la información entre los nodos. Cada bloque tiene definido una serie de puertos de entrada y salida las cuales aceptan y entregan un tipo de dato específico. No es posible realizar una conexión entre dos puertos con tipos de datos incompatibles. Un ejemplo de una grafo de flujo sencillo se muestra en la figura 3–5.

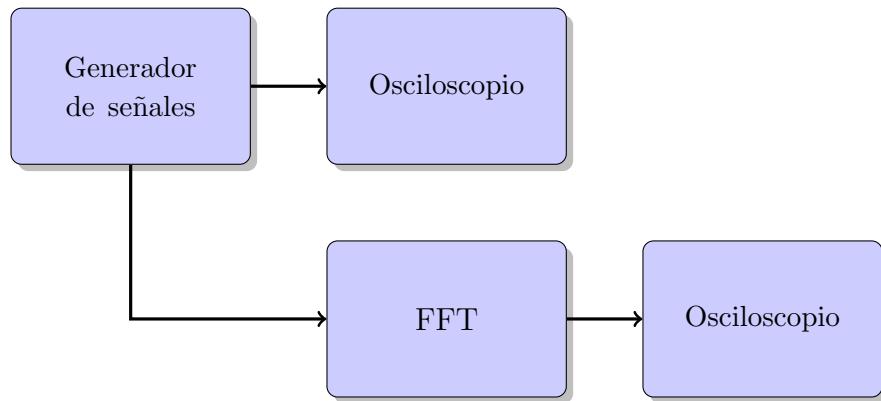


Figura 3–5: Ejemplo de un grafo de flujo en *GNURadio*

La información que viaja a través de las aristas puede provenir de una fuente externa o interna. Para el caso de las fuentes internas, estas generan señales puramente digitales a través de métodos de procesamiento digital de señales. Las fuentes externas primeramente adquieren una señal análoga por medio de alguna interfaz delantera. Esta interfaz se encarga del acondicionamiento de la señal y de su digitalización por medio de un ADC para posteriormente ser procesada por el bloque de *GNURadio*. Estos conceptos se aplican igualmente para bloques de salidas. Los

```

from gnuradio import gr
from gnuradio import audio

class my_top_block(gr.top_block):
    def __init__(self):
        gr.top_block.__init__(self)

        sample_rate = 32000
        ampl = 0.1

        src0 = gr.sig_source_f (sample_rate, gr.GR_SIN_WAVE, 350, ampl)
        src1 = gr.sig_source_f (sample_rate, gr.GR_SIN_WAVE, 440, ampl)
        dst = audio.sink (sample_rate, "")
        self.connect (src0, (dst, 0))
        self.connect (src1, (dst, 1))

if __name__ == '__main__':
    try:
        my_top_block().run()
    except KeyboardInterrupt:
        pass

```

Listado 3.1: Ejemplo de programa utilizando *GNURadio*

bloques pueden llevar datos a algún archivo, desplegarlos en forma gráfica o sacarlos a través de alguna interfaz por medio de un DAC y luego al mundo exterior. Un ejemplo de un programa que ilustra estos conceptos se muestra en el listado 3.1. Este programa utiliza dos bloques que generan dos señales senoidales independientes a diferentes frecuencias y las entrega a un bloque de audio que representa la tarjeta de sonido de la PC donde se está ejecutando el programa, esto en efecto mezcla ambas señales para producir un tono.

El programa se inicia importando los módulos necesarios para la aplicación. Aquí se importa el modulo `gnuradio` y dos de sus submódulos: `gr` y `audio`. Después se declara una clase que se deriva de `top_block`. Todas las clases de *GNURadio* se deben de derivar de `top_block` o de `hier_block2` ya que estas dos clases actúan como un contenedor para el grafo de flujo. La primera clase actúa como un contenedor para un único grafo y la segunda soporta el desarrollo de grafos jerárquicos.

Los bloques y sus conexiones se definen dentro del constructor de la clase, en este caso es la función `__init__`. Esta línea de código también manda llamar la función `__init__` de la clase de la que se deriva, pasándole los parámetros que sean necesarios (si es que los pide). Esto es necesario para llevar a cabo la inicialización de ambas clases. En las siguientes dos líneas se declaran dos variables para llevar el control del tiempo de muestro y de la amplitud. Las siguientes dos líneas demuestran la manera de cómo se declaran los bloques de ejecución. Cada bloque está representado por una clase y su inicialización consta en mandar llamar su constructor con los parámetros necesarios para su funcionamiento. Como se puede observar, ambos bloques se encuentran dentro del submodulo `gr` y su constructor se invoca con los parámetros que requieren para generar el tipo de señal que se desea. Para estos bloques los parámetros son en orden los siguientes: el tiempo de muestreo, una constante que indica el tipo de señal que se desea generar (estas constantes también se encuentran dentro del submodulo `gr`), la frecuencia de la señal y su amplitud. La siguiente línea declara un tercer bloque que es el de audio y representa la tarjeta de sonido de la PC. Los parámetros que necesita son el tiempo de muestreo y el nombre del dispositivo de audio (esto es en caso de que se tenga más de uno en la PC).

Por último las dos siguientes líneas demuestran la manera de cómo hacer las conexiones entre los bloques. Como la clase fue derivada de `top_block` ahora contamos sus funciones para nuestro uso. Una de estas funciones se llama `connect` y se utiliza para conectar cada uno de los bloques. Los parámetros que acepta son clases derivadas de los bloques principales o tuplas con dos elementos, el bloque y un entero. El entero especifica que puerto utilizar para la conexión y esto es únicamente válido para bloques que acepten más de una conexión a la vez. La numeración de los puertos inicia con cero por lo que la primera conexión lleva la primera señal al puerto 0 y la segunda conexión lleva la segunda señal al puerto 1.

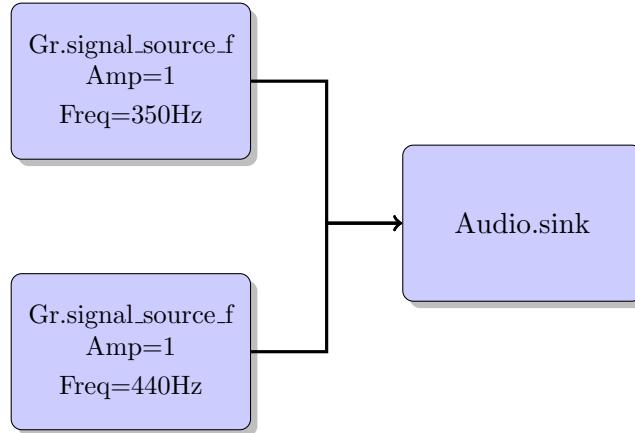


Figura 3–6: Grafo que envía dos senoidales a la tarjeta de audio de una PC.

Después de la declaración de el bloque principal, el programa se ejecuta declarando una instancia de la clase, y mandando llamar su función `run`. Esta función también es proporcionada por la clase `top_block`. Cuando este programa se ejecute el resultado debe ser un tono emitido a través de las bocinas de la PC a la frecuencia de las dos señales generadas. El grafo representativo de este programa se muestra en la figura 3–6.

3.5. Descripción del hardware y software

En esta sección se describe la manera en como se llevó a cabo el experimento para la implementación de una transmisión digital utilizando el esquema de modulación QPSK. Las instrucciones para instalar el ambiente de desarrollo se describen a detalle en el apendice A.

3.5.1. Hardware utilizado

Los componentes que forman la parte de hardware son los siguientes:

- USRP
- Tarjetas auxiliares LFTX y LFRX
- Cable coaxial SMA-SMA

- Cable USB
- Fuente de alimentación

Como se describió en la tabla 3-1, las tarjetas LFTX y LFRX tienen un ancho de banda de DC-30Mhz. Las terminales de estas tarjetas son de tipo SMA con una impedancia de 50Ω . El cable coaxial se utilizó para minimizar pérdidas en la transmisión durante la caracterización del sistema. El cable USB es un cable tipo A-B (conectar estandar A a B). Las especificaciones de la fuente de alimentación para el USRP son 6V @ 4A AC/DC. El sistema es capaz de operar en el rango de 90-260VAC @ 50/60Hz lo cual permite que funcione en diferentes países.

3.5.2. Código fuente de GNURadio

El código fuente de *GNURadio* proporciona varios ejemplos que implementan varios conceptos de programación utilizando los bloques de procesamiento que proporciona el *framework*. Algunos de estos ejemplos son: receptor de FM, decodificador de señales ATSC, captura y análisis de audio y modulación OFDM entre otros. La compilación del código fuente genera también la documentación de todos los bloques de procesamiento, lo cual es muy útil como referencia mientras se desarrolla cualquier aplicación. La estructura de los directorios que contienen los ejemplos se muestra en la figura 3-7.

El experimento está basado en el ejemplo del directorio **digital**. Este programa implementa un transmisor completo utilizando varias modulaciones digitales tales como GMSK, BPSK y QPSK y viene en dos versiones: una se maneja a través de una consola de comandos y la otra implementa una interfaz de usuario utilizando la biblioteca QT que permite a visualizar de manera gráfica lo que el sistema está recibiendo. El programa utiliza varias técnicas de programación orientada a objetos, así como también estructuración del código de diferentes módulos para separar todos los componentes importantes y poder utilizarlos en otras aplicaciones. La figura 3-8

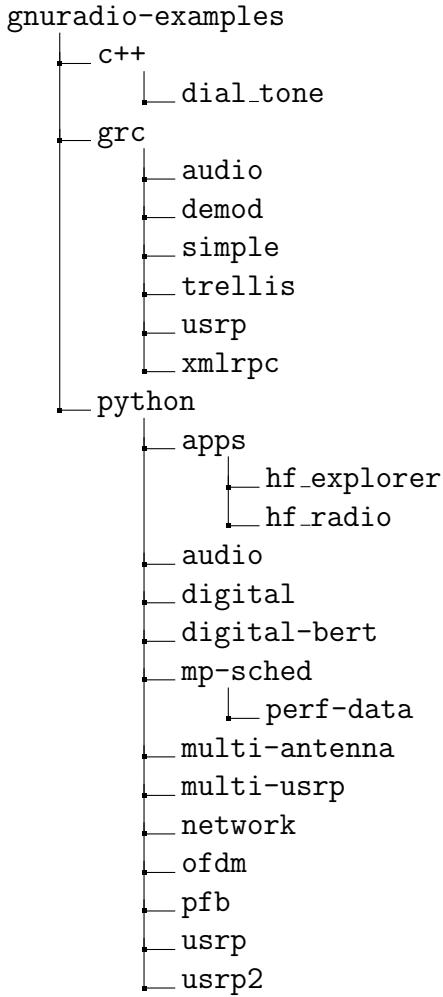


Figura 3–7: Árbol de directorios de los ejemplos que ofrece *GNURadio*

muestra una gráfica de las principales dependencias entre los módulos de Python que forman parte del ejemplo.

El módulo `benchmark_tx` es el principal y el que es ejecutado por el usuario para arrancar la aplicación. `modulation_utils` contiene una función que regresa un diccionario con todos los esquemas de modulación registrados en *GNURadio*. Este diccionario se utiliza para preparar las opciones que se le ofrecen al usuario y así impedir que no seleccione una que no exista. `usrp_transmit_path` encapsula la operación de inicializar el USRP y consiste en enviar los comandos de selección de

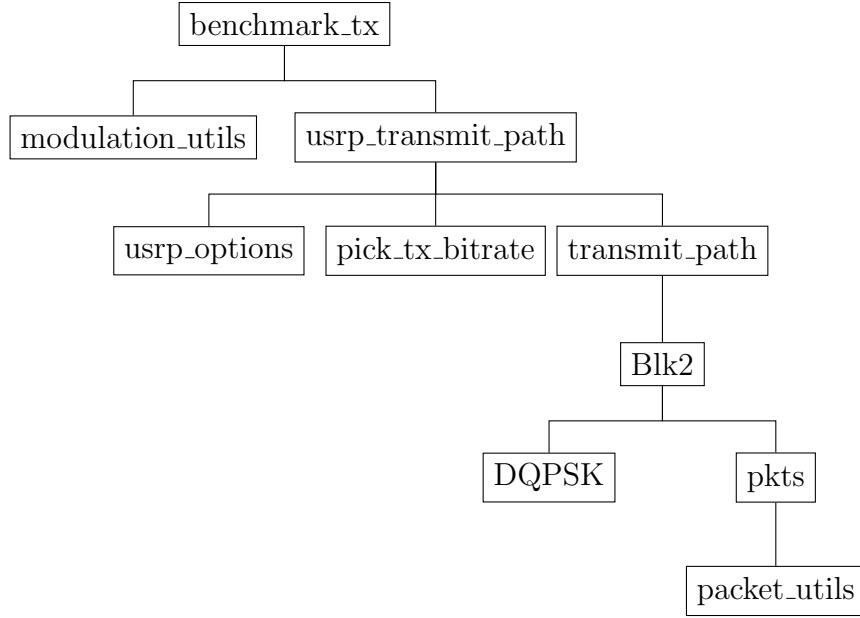


Figura 3–8: Relación de módulos del ejemplo *benchmark* de *GNURadio*

decimación / interpolación, frecuencia a la que se deben sintonizar las tarjetas auxiliares, el valor del multiplexor que controla las conexiones de los codecs al FPGA, etc. Todas estas operaciones son delegadas a una clase contenida en `usrp_options`. El resultado es un objeto de tipo `usrp_sink` configurado para enviar datos al puerto USB. El módulo `pick_tx_bitrate` se utiliza para calcular una tasa de bits óptima a partir de los parámetros muestras por símbolo, decimación / interpolación y bits por símbolo. Es posible especificar la tasa de bits también y el modulo verificará que sea valido considerando los otros parámetros especificados. De no ser así entonces se hará un ajuste a la tasa de bits para poder utilizar correctamente los otros parámetros. Si no se especifica ningún parámetro entonces el módulo trabajará con un valor default de 500kb/s. El módulo `transmit_path` realiza las operaciones de preparar y crear un objeto que encapsula el esquema de modulación seleccionado por el usuario. Los esquemas de modulación se encuentran dentro del paquete `blk2impl`, al cual se accede por medio del módulo `blk2` y se encarga de importar todo el contenido de `blk2impl` al programa principal. Los dos últimos módulos, `mod_pkts` y

`packet_utils` se encargan de realizar la operación de empaquetar los bits que se desean transmitir.

La relación que se muestra en la figura 3–8 es la base para las otras versiones del mismo ejemplo. Para el receptor la operación es la misma a excepción que utiliza los módulos `receive_path` y `usrp_receive_path`. También existen versiones que implementan una interfaz gráfica utilizando la biblioteca QT que muestra el espectro de la señal, la constelación recibida, entre otras.

3.6. Descripción del flujo del software

La estructura del programa `benchmark_tx.py` muestra un ejemplo de cómo desarrollar aplicaciones complejas que utilizan varias técnicas de programación y herramientas que se pueden incorporar al lenguaje *Python*. El usuario es libre de desarrollar aplicaciones más sencillas que constan de un solo *script* o de varios asociados unos con el otro.

3.6.1. Estructura del transmisor

El programa principal `benchmark_tx.py` genera los datos que se van a enviar de dos formas: automáticamente o a partir de un archivo proporcionado por el usuario. En el modo automático (éste es el modo default) el programa genera una secuencia consecutiva de caracteres ASCII del tamaño que el usuario especificó. Esta secuencia es enviada a una función de *Python* que a su vez la envía al grafo de RF. La estructura general del transmisor se muestra en la figura 3–9.

La declaración del grafo principal se muestra en el listado 3.2. Este grafo consta de un solo bloque que transfiere sus parámetros de entrada a un sub-bloque que encapsula las operaciones de inicialización del USRP.

Todos los bloques son clases *Python* que se derivan de la clase `top_block`. Esta clase se encuentra dentro del módulo `gr` y se accede a ella utilizando el operador punto. En el constructor se inicializa un nuevo bloque creando una nueva instancia

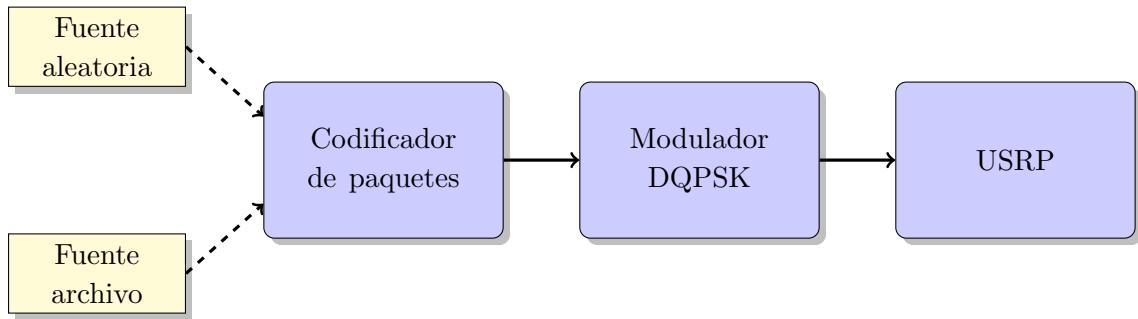


Figura 3–9: Diagrama a bloques del transmisor DQPSK.

```

class my_top_block(gr.top_block):
    def __init__(self, modulator, options):
        gr.top_block.__init__(self)

        self.txpath = usrp_transmit_path.usrp_transmit_path(modulator,
                                                              options)

        self.connect(self.txpath)
    
```

Listado 3.2: Declaración del bloque jerárquico principal.

de la clase `usrp_transmit_path`. A esta clase se le pasan los mismos parámetros que recibió el bloque principal y son el tipo de modulador que va a utilizar y las opciones del programa. La clase `top_block` define un método llamado *connect* para realizar la conexión entre los bloques de la cadena de RF. Como este grafo consta de un solo bloque, a este método se le pasa solo un parámetro. Este método acepta una cantidad variable de parámetros ya que podemos realizar conexiones entre varios bloques. Esto se mostrará más adelante.

El grafo de RF está formado por varios bloques jerárquicos, es decir, bloques que encapsulan otros bloques, que conectados entre sí, forman un sub-grafo. Esto permite crear nuevos bloques a partir de otros que ya existen. El bloque `usrp_transmit_path` es un ejemplo de un bloque jerárquico. Estas clases se derivan de `gr.hier_block2` y en su constructor se declaran las entradas y salidas que pueda tener. Es posible crear un bloque con entradas y ninguna salida o vice versa. Esto permite crear bloques que aceptan un flujo de datos para ya sea procesarse dentro de ellos únicamente o ser enviados a un puerto de salida o entrada. El constructor de este bloque se muestra en el listado 3.3.

Lo primero que se debe hacer en este tipo de bloques es definir las entradas y salidas. Esto se logra invocando el constructor de la clase base `hier_block2` y pasando los siguientes parámetros: la clase derivada por medio de la palabra clave `self`, una cadena con el nombre del bloque, la definición de entradas y la de salidas. Para definir las entradas y salidas se utiliza la función `gr.io_signature()` y se le pasan tres parámetros: número mínimo de puertos, el máximo número de puertos y el tamaño de los elementos que entran y salen por los puertos. Un ejemplo de cómo declarar un bloque con puertos de entrada tipo *float* y de salida tipo *complex* se muestra en el listado 3.4:

Dentro del constructor se crean instancias de los bloques que formarán parte del grafo y por último se realiza la conexión entre ellos. Todas las clases que se derivan de

```

class usrp_transmit_path(gr.hier_block2):
    def __init__(self, modulator_class, options):
        gr.hier_block2.__init__(self, "usrp_transmit_path",
                               gr.io_signature(0, 0, 0),
                               gr.io_signature(0, 0, 0))
    if options.tx_freq is None:
        sys.stderr.write("-f FREQ or --freq FREQ or
                         --tx-freq FREQ must be specified")
        raise SystemExit

    self._modulator_class = modulator_class
    self._setup_usrp_sink(options)

    tx_path = transmit_path.transmit_path(modulator_class, options)
    for attr in dir(tx_path):
        if not attr.startswith('_') and not hasattr(self, attr):
            setattr(self, attr, getattr(tx_path, attr))

    self.connect(tx_path, self.u)

```

Listado 3.3: Constructor del bloque transmisor del USRP.

```

class HierBlock(gr.hier_block2):
    def __init__(self, audio_rate, if_rate):
        gr.hier_block2.__init__(self, "HierBlock",
                               gr.io_signature(1, 1, gr.sizeof_float),
                               gr.io_signature(1, 2, gr.sizeof_gr_complex))

    B1 = gr.block1(...)
    B2 = gr.block2(...)

    self.connect(self, B1, B2, self)

```

Listado 3.4: Ejemplo de declaración de entradas y salidas para un bloque jerárquico.

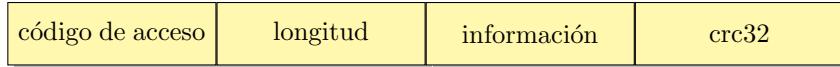


Figura 3–10: Estructura del paquete de datos.

alguno de los bloques, `top_block` o `hier_block2`, contienen un método que se llama `connect`. Este método recibe una cantidad variable de parámetros para realizar la conexión entre los bloques. En el listado 3.4, la conexión se realizó de la siguiente manera: `entradas->B1->B2->salidas`. Las entradas y salidas son representadas por el parámetro `self`. Para definir bloques que no tienen entradas o salidas es necesario pasar ceros a todos los parámetros de la función `gr.io_signature()` como se muestra en el listado 3.3. En este caso el bloque no cuenta con entradas ni salidas, únicamente encapsula un sub-grafo al cual se le delega el trabajo del transmisor.

El bloque `transmit_path`, del cual se crea una instancia en el listado 3.3, encapsula al grafo que lleva a cabo la modulación de la información que se va transmitir. El modulador en sí es un bloque que se encuentra dentro del módulo `blk2` como se muestra en la figura 3–8. Aunque es posible utilizar los bloques moduladores directamente, la aplicación `benchmark_tx.py` muestra una forma alterna de utilizarlos que consiste en codificar y formatear la información en paquetes antes de ser modulados. Este proceso se lleva a cabo utilizando la clase `mod_pkts` del módulo `pkt`. Esta clase envuelve cualquier clase moduladora que se le proporcione y se encarga de codificar y formar paquetes de bits con una estructura específica que permite sincronizar la transmisión y disminuir los errores. La estructura que tienen los paquetes generados con esta clase se muestra en la figura 3–10.

El código de acceso es una cadena de unos y ceros que se puede utilizar para aceptar mensajes que contengan únicamente el código correcto. La clase genera una secuencia pre-definida si no se especifica una. La longitud se refiere a lo largo del mensaje junto con el código CRC. La información se anexa junto con el código CRC(Comprobación de redundancia cíclica por sus siglas en Inglés) de 32 bits. Los

paquetes se envían al modulador a través de una cola que almacena lo que se envía. La clase moduladora monitorea esta cola y toma el primer mensaje que entró para iniciar con la modulación.

Después de la codificación los datos son modulados por el esquema DQPSK. La estructura de este bloque se explica más a fondo en la siguiente sección. Los datos modulados son enviados por el puerto USB hacia el USRP por medio del bloque USRP. Este bloque es representado por la clase `usrp_sink_c` y su función es tomar los datos de entrada y enviarlos al USRP con los parámetros de interpolación y frecuencia central especificada. El listado 3.3 muestra al constructor del bloque jerárquico que realiza esta configuración pero, como se muestra, la operación real es delegada a la función `_setup_usrp_sink`. Esta función se muestra en el listado 3.5.

La función calcula el factor de interpolación para lograr la tasa de bits que se está especificando por el usuario. Este factor se entrega al USRP para configurar el DUC y después se sintoniza a la frecuencia central especificada. Si el hardware no puede sintonizar la frecuencia que se especifica entonces el programa abortará con un mensaje de error. La aplicación también configura al USRP en modo de transmisión automática lo cual quiere decir que el dispositivo estará transmitiendo continuamente lo que se encuentre en sus *buffers* hasta que la aplicación termine de ejecutarse.

3.6.2. Estructura del modulador DQPSK

GNURadio contiene bloques que definen distintos esquemas de modulación digital. El estudio de este trabajo se enfoca en el bloque DQPSK. La estructura del grafo que implementa este esquema se muestra en la figura 3-11.

Los bits individuales de cada byte que entran al grafo son descompuestos en grupos de dos ya que cada símbolo en una transmision DQPSK contiene dos bits. El bloque de mapeo codifica los bits en un alfabeto de tamaño M donde en este caso es 4 para la constelación QPSK. La codificación utiliza el metodo de Gray y

```

def _setup_usrp_sink(self, options):

    self.u = usrp_options.create_usrp_sink(options)
    dac_rate = self.u.dac_rate()
    self.rs_rate = options.bitrate

    (self._bitrate, self._samples_per_symbol, self._interp) = \
        pick_tx_bitrate(options.bitrate, self.
                         _modulator_class.bits_per_symbol(),
                         options.samples_per_symbol, options
                         .interp,
                         dac_rate, self.u.get_interp_rates()
                         )

    options.interp = self._interp
    options.samples_per_symbol = self._samples_per_symbol
    options.bitrate = self._bitrate

    if options.verbose:
        print 'USRP Sink:', self.u
        print "Interpolation Rate: ", self._interp

    self.u.set_interp(self._interp)
    self.u.set_auto_tr(True)

    if not self.u.set_center_freq(options.tx_freq):
        print "Failed to set Rx frequency to %s" %(eng_notation.
            num_to_str(options.tx_freq))
        raise ValueError, eng_notation.num_to_str(options.tx_freq)

```

Listado 3.5: Función que configura el USRP como transmisor

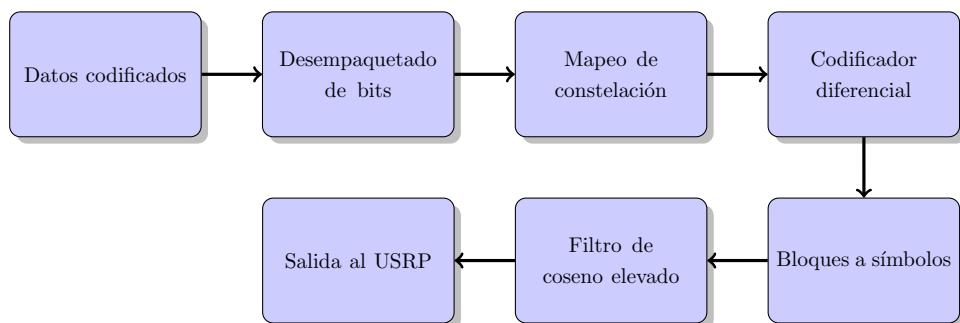


Figura 3–11: Estructura del modulador DQPSK de *GNURadio*

consiste en hacer que cada valor binario sucesivo difiera únicamente por un bit. Esta codificación es opcional y se puede deshabilitar si no se desea utilizar.

El esquema DQPSK implementa un codificador y decodificador diferencial antes de la modulación y después de la demodulación. Esta codificación tiene dos ventajas. La primera es simplificar el diseño del receptor ya que en lugar de tener una referencia exacta de la fase que se utilizó para enviar la información, ahora se utiliza la diferencia entre dos bits. Esto nos permite realizar una recuperación *no coherente* de la información. La segunda ventaja es que ofrece protección contra la inversión de polaridad. Este método nos permite recuperar la información independientemente si este fenómeno sucede o no. Como se está utilizando dos bits para la codificación, la desventaja principal es que un error causaría errores en dos bits en lugar de uno y por consecuencia aumenta la probabilidad de error de bits. La representación matemática del codificador y decodificador se muestra en las ecuaciones (3.5) y (3.6).

$$e_k = e_{k-1} \oplus b_k \quad \rightarrow \text{codificador} \quad (3.5)$$

$$b_k = e_k \oplus e_{k-1} \quad \rightarrow \text{decodificador} \quad (3.6)$$

donde e_k es el bit codificado, b_k es el bit decodificado y el operador \oplus representa la operación XOR ([Sklar, 2001](#)).

Los bits son transformados a su representación compleja dependiendo de la constelación que se especificó y posteriormente son introducidos a un filtro de coseno elevado para reducir la cantidad de ancho de banda requerida para la transmisión y eliminar el efecto de ISI. La respuesta a la frecuencia de este filtro está caracterizada por el parámetro α que especifica la cantidad de ancho de banda más allá del límite de Nyquist de $1/2T$ ([Sklar, 2001](#)). A este parámetro se le llama exceso de ancho de banda. La respuesta a la frecuencia de este filtro se muestra en la ecuación (3.7).

$$H(f) = \begin{cases} T_b & \text{para } |f| \leq \frac{1-\alpha}{2T_b} \\ \frac{T_b}{2} \left\{ 1 + \cos \left[\frac{\pi T_b}{\alpha} \left(|f| - \frac{1-\alpha}{2T_b} \right) \right] \right\} & \text{para } \frac{1-\alpha}{2T_b} \leq |f| \leq \frac{1+\alpha}{2T_b} \\ 0 & \text{de lo contrario} \end{cases} \quad (3.7)$$

donde $0 \leq \alpha \leq 1$ se le conoce como el factor de exceso de ancho de banda, T_b es el periodo de símbolos y f es la frecuencia.

La respuesta al impulso del filtro esta dado por:

$$\begin{aligned} h(t) &= \frac{\sin(\pi t/T_b)}{\pi t/T_b} \frac{\cos(\pi \alpha t/T_b)}{1 - 4\alpha^2 t^2/T_b^2} \\ &= \text{sinc}(t/T_b) \frac{\cos(\pi \alpha t/T_b)}{1 - 4\alpha^2 t^2/T_b^2} \end{aligned} \quad (3.8)$$

El punto medio de la banda de transición $1/2T_b = f_b/2$ se le conoce como la *frecuencia de Nyquist*. Cuando $\alpha = 0$ el filtro se comporta como un filtro Nyquist ideal, comúnmente conocido como *brick wall* y su respuesta al impulso tiende a decaer muy lentamente por lo cual ofrece un mayor nivel de ISI. Cuando α se acerca a 1 su respuesta al impulso decae más rápido pero a consecuencia de una mayor cantidad de ancho de banda utilizada. El exceso de ancho de banda se interpreta en términos de porcentaje con referencia a la frecuencia de Nyquist, por ejemplo, un valor de $\alpha = 0.25$ significa 25 % de exceso de ancho de banda y $\alpha = 1$ significa 100 %. La respuesta a la frecuencia y al impulso de este filtro se muestra en las figuras 3–12 y 3–13. En estas gráficas se puede observar cómo el valor de α afecta la forma del pulso. A valores cercanos de 1 el filtro comienza a tomar la forma de la función coseno mientras que a valores cercanos a 0 toma una forma al filtro ideal rectangular. Finalmente, los datos procesados por el filtro RRC se colocan en la salida del bloque jerárquico para que sean enviados al siguiente bloque del grafo.

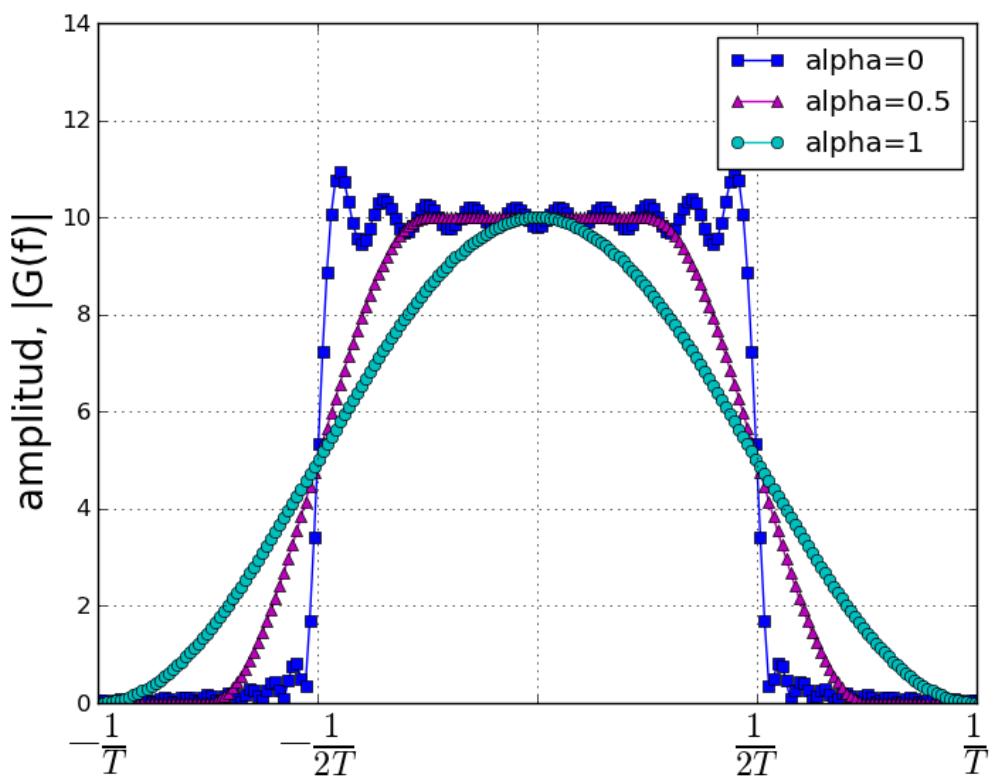


Figura 3–12: Representación en el dominio de la frecuencia de filtros coseno elevado

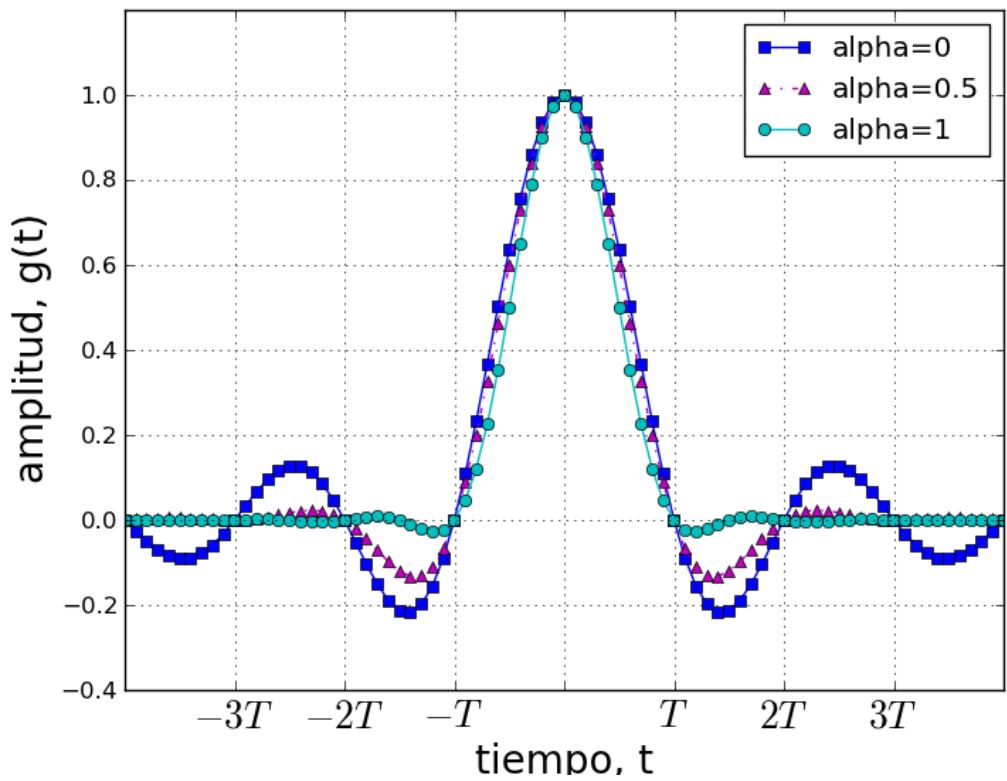


Figura 3–13: Forma de onda en el dominio del tiempo para filtros de coseno elevado

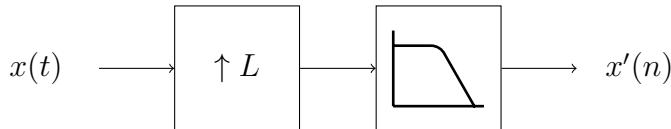
La clase `blk2.dqpsk_mod` representa el modulador DQPSK. Un ejemplo de su uso en un programa sencillo se muestra en el capítulo 4. La clase se puede configurar de acuerdo a los siguientes parámetros de entrada:

- **`samples_per_symbol`**: Especifica la tasa de muestras por símbolo. Acepta valores ≥ 2 .
- **`excess_bw`**: Especifica el porcentaje de exceso de ancho de banda que utiliza el filtro RRC.
- **`gray_code`**: Bandera booleana que activa o desactiva la codificación Gray para la generación de la constelación.
- **`verbose`**: Especifica si el modulador va imprimir su información de configuración a la consola.
- **`log`**: Activa la generación de archivos con los datos que viajan atravez de las aristas del grafo.

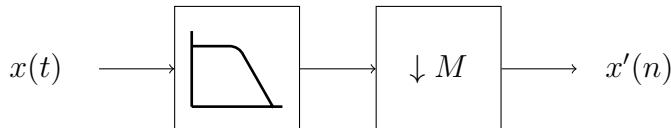
3.6.3. Modulador DQPSK con filtros polifásicos

Durante el desarrollo de este trabajo los autores de *GNURadio* incorporaron al código fuente una versión mejorada del modulador DQPSK que re-implementa algunos bloques del grafo con bancos de filtros polifásicos. Estos filtros son también llamados multi tasa (*multirate*) ya que modifican la frecuencia de muestreo (decimación e interpolación) y consisten en un filtro tradicional (pasa bajas, pasa altas, etc.) (Farhang, 2010). Un ejemplo de estos filtros se muestra en la figura 3-14.

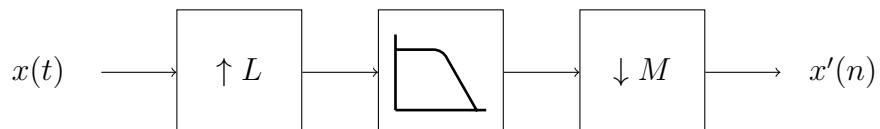
La figura 3-14a muestra la estructura de un interpolador simple con factor de interpolación L . Este filtro reconstruye una señal $x(n)$ a una tasa L veces más que la tasa de muestreo original. Esto se logra anexando $L - 1$ ceros despues de cada muestra y despues se utiliza un filtro pasa bajas para eliminar las imágenes espectrales que se generan alrededor de la señal debido a la operación de *upsampling* (Farhang, 2010).



(a) Interpolador con factor L .



(b) Decimador con factor M .



(c) Filtro factorial con factor L/M .

Figura 3–14: Estructuras básicas de filtros multi tasa.

La figura 3–14b muestra la estructura de un filtro decimador con factor M . En este caso, el filtro pasa bajas se utiliza antes del decimador para reducir el ancho de banda de la señal y prevenir que se genere el efecto alias debido a la reducción de la frecuencia de muestreo. El decimador entonces tomará cada D veces muestras e ignorará el resto.

Los filtros anteriores utilizan factores enteros y por lo tanto no se puede realizar interpolación o decimación fraccional ([Farhang, 2010](#)). Un tercer filtro combina ambas estructuras para lograr precisamente esto y se denominan filtros factoriales L/M . Su estructura se muestra en la figura 3–14c. Aquí el filtro pasa bajas se encuentre entre el bloque de *upsampling* y el bloque decimador para proporcionar interpolación a la señal y protección contra el efecto alias.

El problema principal con las estructuras anteriores es que no son muy eficientes. El filtro decimador calcula muchas muestras que eventualmente serán descartadas y el filtro interpolador procesa las muestras de la señal así como los ceros introducidos

por la operación *upsampling*. Para solucionar esto se utiliza una estructura de filtros polifásicos para lograr que el filtro decimador e interpolador procesen la señal a la tasa de muestreo original y no a la tasa modificada. La estructura del filtro decimador en su descomposición polifásica se muestra en la figura 3–15.

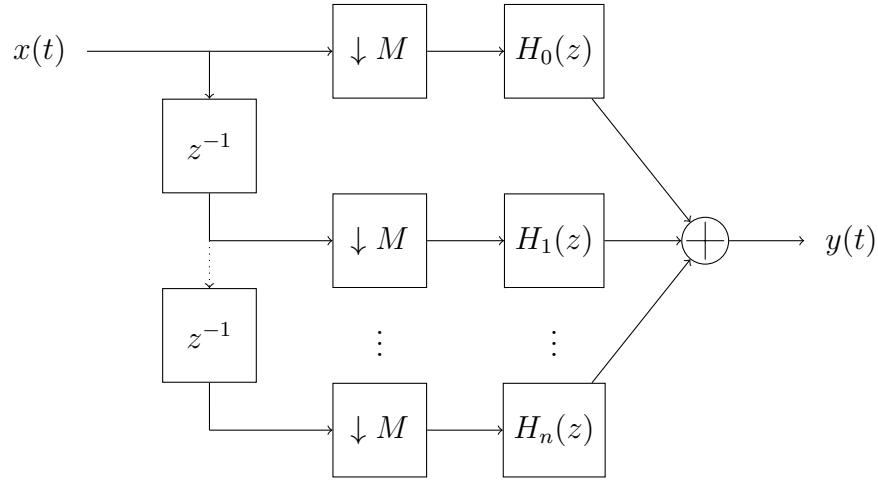


Figura 3–15: Descomposición del filtro decimador en su representación polifásica.

La estructura del modulador DQPSK fue modificada con la introducción de estos filtros al código fuente. La estructura de la segunda versión de este bloque jerárquico, llamado DQPSK2, se muestra en la figura 3–16. El bloque ilustrado de rojo representa la modificación que se realizó al grafo del bloque jerárquico original.

El bloque en rojo de la figura 3–16 implementa un re-muestreador arbitrario utilizando un banco de filtros polifásicos. La tasa de re-muestreo puede ser cualquier número real r y realiza la operación construyendo N cantidad de filtros donde N es el factor de interpolación. El factor D se calcula de la siguiente manera: $D = N/r$. Utilizando N y D el bloque realiza un muestreo arbitrario con factor N/D que representa un número racional cercano a la tasa especificada r y utilizando N cantidad de filtros que forman el banco completo de filtros polifásicos.

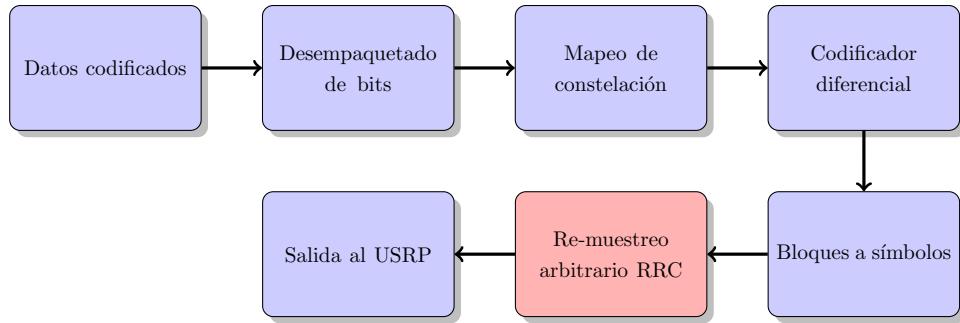


Figura 3–16: Estructura del modulador DQPSK con filtros polifásicos

Este modulador se encuentra dentro del mismo paquete `blsk2` que la versión anterior. Los parámetros que acepta la clase son los mismos que se describieron en el apartado [3.6.2](#).

3.6.4. Estructura del receptor

El programa principal del receptor se encuentra en el archivo `benchmark_rx.py`. La estructura del programa es similar a la del transmisor, la diferencia principal es en el uso de una función auxiliar que maneja el grafo RX, adicionalmente se encarga de recibir los paquetes que van recibiendo y desplegar su estatus en la consola. Esta función se puede modificar para realizar cualquier otra operación con los paquetes de datos. El código de esta función se muestra en el listado [3.6](#). La estructura del receptor se muestra en la figura [3–17](#).

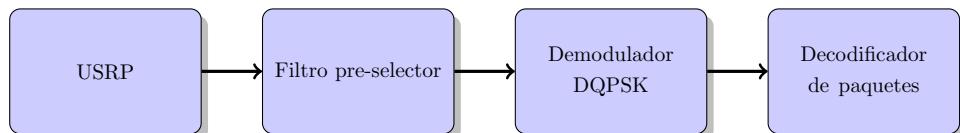


Figura 3–17: Diagrama a bloques del receptor DQPSK

```

def rx_callback(ok, payload):
    global n_rcvd, n_right
    (pktno,) = struct.unpack('!H', payload[0:2])
    n_rcvd += 1
    if ok:
        n_right += 1

    print "ok = %5s  pktno = %4d  n_rcvd = %4d  n_right = %4d" % (
        ok, pktno, n_rcvd, n_right)

```

Listado 3.6: Función auxiliar que recibe los paquetes decodificados

Al igual que el transmisor, el receptor define un grafo principal con un solo bloque jerárquico que encapsula todas las diversas operaciones del programa (configuración del USRP, demodulación y decodificación de datos) y su implementación se mantiene separado en diversos archivos, cada uno con un propósito exclusivo.

La configuración del USRP se lleva a cabo dentro del archivo `usrp_receive_path.py`. Las operaciones principales que realiza el grafo de este archivo son configurar la frecuencia central y el factor de decimación que se va utilizar dentro del DDC. Igualmente el programa realiza el cálculo de la tasa de bits con los parámetros que el usuario especificó y si no se logra la configuración el programa emite un mensaje de error. Posteriormente, el control se pasa al archivo `receive_path.py` para la generación y configuración del grafo que crea la instancia del demodulador DQPSK.

3.6.5. Estructura del demodulador DQPSK

La estructura del grafo del demodulador DQPSK se muestra en la figura 3–18.

La señal que entrega el USRP es multiplicada por una constante para escalarla de su rango completo, ± 16384 , a ± 1 . Despues se entrega a un controlador automático de ganancia no causal el cual se encarga de calcular la ganancia requerida en base al valor absoluto máximo sobre n muestras.

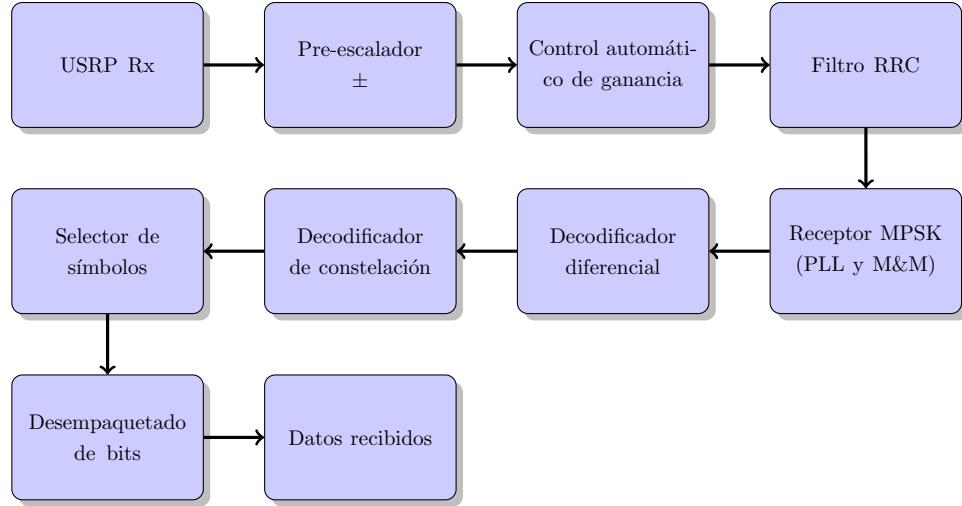


Figura 3–18: Estructura del demodulador DQPSK

Al igual que el transmisor, el demodulador cuenta con un filtro de coseno elevado con las mismas características utilizadas en la etapa de transmisión. El filtro es necesario para completar el acoplamiento ya que individualmente son filtros raíz de coseno elevado y su respuesta a la frecuencia combinada es la del filtro coseno elevado (Sklar, 2001) como se muestra en la ecuación 3.9.

$$H_{rc}(f) = H_{rrc}(f) \cdot H_{rrc}(f) \quad (3.9)$$

El bloque **Receptor MPSK** se encarga de realizar la sincronía de la fase y frecuencia de la señal y del reloj de símbolos. Esto lo hace en dos etapas las cuales son por medio de un PLL llamado Lazo de Costas y un algoritmo de sincronía de reloj llamado M&M. El Lazo de Costas encuentra el error en la señal recibida comparándola con su punto de constelación mas cercano. La frecuencia y la fase de un NCO son actualizados en base a este error. La sincronización de símbolos se realiza por medio de una versión modificada del algoritmo de Mueller y Müller descrita en el artículo técnico *Optimisation of modified Mueller and Müller algorithm* publicado el 22 de Junio de 1995. El algoritmo realiza una interpolación de una muestra (utilizando el NCO del Lazo de Costas) cada μ muestras y luego encuentra el error de muestreo

basado en el actual y los pasados símbolos y la decisión tomada sobre las muestras. Para los dos, PLL y M&M, *GNURadio* implementa algoritmos optimizados para BPSK y QPSK pero para 8PSK utiliza una solución que los autores le llaman *fuerza bruta* para calcular la distancia mínima de los puntos de la constelación.

Antes de iniciar con la decodificación de la constelación, los bits son decodificados a través de un decodificador diferencial utilizando la ecuación (3.6). El resto de los bloques se encargan de convertir los valores complejos a bits y decodificar los puntos de la constelación para luego entregar cada bit individualmente al resto de la aplicación.

3.6.6. Demodulador DQPSK con filtros polifásicos

Para la segunda versión del código que implementa el esquema DQPSK con filtros polifásicos las modificaciones se muestran en la figura 3–19. Los bloques mostrados en rojo representan las modificaciones al grafo original.

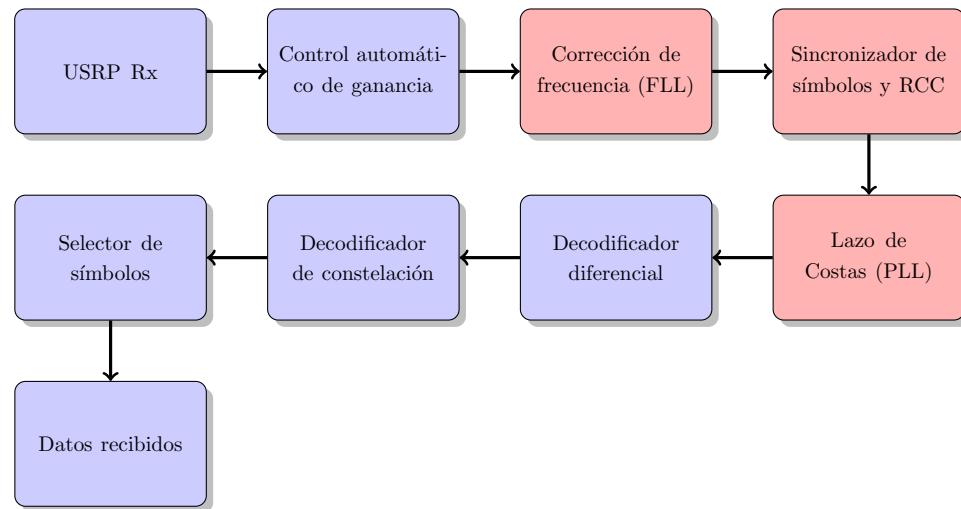


Figura 3–19: Estructura del demodulador DQPSK con filtros polifásicos.

Después de la etapa de control de ganancia el sistema inicia con un FLL (Bucle de Enganche de Frecuencia por sus siglas en Inglés) el cual se encarga de generar una señal de referencia con la frecuencia correcta de la portadora. El FLL cubre el ancho

de banda superior e inferior de la señal modulada. Este rango se determina por el factor α (exceso de ancho de banda) de la señal y las esquinas superior e inferior del ancho de banda son determinadas por la tasa de sobre-muestreo (el número de muestras por símbolo) y el factor α .

El FLL filtra las bandas superior e inferior en $x_u(t)$ y $x_l(t)$ respectivamente ([GNURadio, 2009](#)). Estas se combinan para formar

$$cc(t) = x_u(t) + x_l(t) \quad (3.10)$$

y

$$ss(t) = x_u(t) - x_l(t) \quad (3.11)$$

Combinando (3.10) y (3.11) nos da la señal

$$e(t) = Re\{cc(t) \times ss(t)^*\} \quad (3.12)$$

donde * es el operador del conjugado complejo. Esta señal error es directamente proporcional a la frecuencia de la portadora. Posteriormente se crea un lazo de segundo orden utilizando esta señal. El bloque realiza estos cálculos dentro de una función llamada `design_filter()`.

El bloque *Sincronizador de símbolos* realiza sincronización de tiempo a señales moduladas por amplitud de pulsos (PAM por sus siglas en Inglés) por medio de la minimización de la derivada de la señal filtrada, el cual a su vez maximiza el SNR y minimiza el efecto ISI ([GNURadio, 2009](#)). Para lograr esto el bloque construye dos bancos de filtros polifásicos; el primero contiene el filtro de forma de pulso (ejemplo: filtro de coseno elevado). El segundo banco contiene la derivada de los filtros del primer banco. Si se observa esto desde el dominio del tiempo, el primer banco contiene filtros que tienen una respuesta al impulso de forma sinc. El objetivo

es alinear la señal de salida de tal forma que sea muestrada justamente en el pico de esta función. La derivada de la función sinc es cero en su punto máximo, es decir, $\text{sinc}(0) = 1$, $\text{sinc}(0)' = 0$. Este hecho se utiliza para generar una señal de error $e(n)$. Esta señal entrega un valor proporcional a la distancia con respecto al punto cero. El bloque utiliza un PLL de segundo orden para lograr que el error llegue a cero.

El tercer bloque rojo implementa el Lazo de Costas para la corrección de la fase de la señal por separado (anteriormente se implementaba dentro de un solo bloque junto con la sincronización de símbolos). La clase que lo implementa se llama `gr.costas_loop_cc`. Con esto, esta versión del demodulador realiza la recuperación de la portadora en tres etapas: corrección de frecuencia, sincronización de símbolos y filtro de forma de pulso por medio de filtros polifásicos y corrección de fase por medio del Lazo de Costas. Una comparación detallada del rendimiento de ambos métodos (con y sin filtros polifásicos) de demodulación QPSK se presenta en el siguiente capítulo.

Capítulo 4

RESULTADOS

En este capítulo se muestran los resultados del estudio de la plataforma USRP. El experimento consistió en transmitir datos generados de una fuente aleatoria por software y de una fuente fija que consiste en una imagen entre las tarjetas LFTX y LFRX por medio de un cable coaxial. Estos datos son modulados con el esquema DQPSK como está implementado en *GNURadio*. La versión del *software* que se utilizó para el experimento fue la 3.3.0. Posteriormente se observó los resultados utilizando las herramientas proporcionadas por *GNURadio* y el lenguaje *Python*. El código base que se utilizó fueron los programas *benchmark_tx* y *benchmark_rx*. Estos programas son parte de los ejemplos que ofrece el código fuente de *GNURadio*. La característica principal de estos dos programas es que emplean la gran mayoría de los conceptos y técnicas de modulación que ofrece el sistema, así como también ilustra el uso de las herramientas gráficas que se pueden utilizar para crear interfaces de usuario que permitan formar aplicaciones desde sencillas hasta complejas.

Como su nombre lo indica, estos programas fueron desarrollados por los autores de *GNURadio* con el fin de poder evaluar los conceptos y todas las variantes del USRP. La aplicación consiste en un transmisor y receptor unidireccional. Ambos programas tienen acceso a todos los esquemas de modulación y demodulación del código fuente y el usuario puede elegir a través de comandos del programa cual desea utilizar para su aplicación. Los programas se pueden modificar y también se pueden utilizar como base para desarrollar otras aplicaciones y por lo tanto se eligieron para ilustrar el uso del sistema.

4.1. Material utilizado

El material que se utilizó para realizar el experimento fue el siguiente:

- 1 USRP
- Tarjetas LFTX y LFRX
- Cable coaxial con terminales SMA a SMA
- Cable USB
- Computadora laptop con las siguientes características:
 - CPU Intel Core 2 Duo T5550 a 1.83Ghz
 - 4GB memoria DDR2
 - Sistema operativo Ubuntu 10.01

Los programas `benchmark_tx.py` y `benchmark_rx.py` de la carpeta de ejemplos del código fuente fueron utilizados para realizar la transmisión, ya que estos ejemplos son programas completos y maduros que ayudan a ilustrar varias características de la programación con *GNURadio*. Los programas `benckmark_tx2.py` y `benchmark_rx2.py` tienen la misma estructura que su versión anterior pero estos implementan el modulador y demodulador utilizando la técnica de filtros polifásicos. Esta versión, incluyendo los bloques que implementan estos filtros, fueron incluidos en el código fuente el 23 de Marzo de 2010 y fueron incluidos en este estudio para comparar el rendimiento de ambas versiones.

Se optó por utilizar un cable coaxial entre las tarjetas TX y RX para observar el rendimiento del sistema lo mas aproximado posible a lo teórico, es decir, con la menor cantidad de pérdidas en la transmisión. La impedancia del cable es de 50Ω y soporta una frecuencia máxima de operación de 12.4Ghz. Los conectores SMA manejan un rango de frecuencias entre 0 y 18Ghz. La señal viaja entre el conductor central y una capa protectora, comúnmente aluminio. Este arreglo ofrece una buena protección contra el ruido externo ([Engdahl, 2009](#)).

4.2. Parámetros utilizados

Para ejecutar los ejemplos `benchmark_tx.py` y `benchmark_rx.py` es necesario moverse al directorio donde se encuentran. La ruta es:

```
<dir_del_codigo_fuente>/gnuradio-examples/python/digital
```

Los programas se ejecutan por medio de la consola de texto utilizando la notación `./` para indicar que se correrá un ejecutable. La sintaxis completa es:

```
./benchmark_tx.py <opciones>
```

donde opciones son algunas opciones que se pueden pasar al programa para controlar su funcionamiento. Para ver todas las opciones que soporta se puede utilizar el siguiente comando:

```
./benchmark_tx.py --help
```

Algunos parámetros contienen valores por defecto que se pueden utilizar como punto de partida para ajustarlos a la aplicación que se está desarrollando. Estos valores propuestos por los autores de *GNURadio* dieron resultados positivos para comprobar el correcto funcionamiento de las diversas funciones del código, aunque no siempre serán los adecuados para todas las aplicaciones pero sirven como una referencia para de ahí poder optimizarlos. Esta optimización es necesaria para lograr una transmisión eficiente con la menor cantidad de errores posibles. El análisis matemático para la obtención de estos valores está fuera del alcance de este trabajo y se deja como propuesta para una investigación a futuro. Los valores que se utilicen dependen del *hardware* (USRP, tarjetas, capacidad de la PC) que se esté utilizando para desarrollar la aplicación. En la mayoría de los parámetros de los diferentes bloques y aplicaciones que se emplearon para el experimento se utilizaron los valores por defecto ya establecidos y recomendados debido a que dieron buenos resultados y buen rendimiento. Los resultados se muestran a través de las secciones de este capítulo.

Las opciones que se utilizaron para el transmisor y receptor son las siguientes:

TRANSMISOR

- **-m:** Especifica el modulador que se va a utilizar. La opción que se utilizó fue DQPSK y DBPSK. BPSK se utilizó para realizar la comparación entre ambos esquemas.
- **-r:** Especifica la tasa de bits. Se utilizaron varios valores desde 100kbps hasta 10kbps. Este valor depende mucho de la PC que se esté utilizando ya que valores muy altos dependen del rendimiento del CPU.
- **-tx-amplitude:** Especifica la amplitud de la señal generada por el DAC en el USRP. Sus valores van de 0 a 1. Se utilizó el valor por defecto de 0.25.
- **-excess-bw:** Especifica el parámetro α del filtro acoplador de coseno elevado. Su rango de valores es de 0 a 1. Se utilizaron tres valores para el análisis: 0.35, 0.50 y 0.75.
- **-f:** Especifica la frecuencia con la cual se van a sintonizar las tarjetas auxiliares. Para este estudio se utilizó la frecuencia máxima que soportan las tarjetas LFTX y LFRX que es 30Mhz.
- **-S:** Especifica la cantidad de muestras por símbolo. El valor por defecto es de 2. Se observó que se tuvo un mejor rendimiento con un valor de 4.

RECEPTOR

- **-m:** Especifica el demodulador que se va a utilizar. Igual que en el transmisor se utilizó DQPSK y DBPSK.
- **--gain-mu:** Especifica la ganancia que se utiliza en el lazo de costas para la detección de la fase. Se utilizó un valor de 0.5.
- **-f:** Especifica la frecuencia a la que se sintoniza la tarjeta RX. Se utilizó un valor de 30Mhz.
- **-S:** Especifica la cantidad de muestras por símbolo utilizada en el demodulador. Igual que en el transmisor se utilizó un valor de 4.

- **-r**: Especifica la tasa de bits. Se utilizaron los mismos valores que en el transmisor.
- **--excess-bw**: Especifica el parámetro α del filtro acoplador de coseno elevado. Se utilizaron los mismos valores que en el transmisor.
- **--timing-alpha**: Especifica la ganancia del PLL que forma parte del banco de filtros polifásicos para la sincronización del reloj de símbolos. Se utilizó un valor de 10.
- **--phase-alpha**: Especifica la ganancia del lazo de costas en la segunda versión del demodulador DQPSK. Se utilizó un valor de 0.5.
- **--freq-alpha**: Especifica la ganancia del FLL para realizar compensación de frecuencia. Se utilizó un valor de 0.001.

Debido a que los dos programas son independientes uno del otro, es necesario igualar algunos parámetros para lograr una transmisión exitosa. Los parámetros **-f**, **-S**, **-r** y **--excess-bw** son un ejemplo de estos parámetros. Todos los parámetros tienen valores por defecto pero uno de ellos es obligatorio especificar y es el de la frecuencia de sintonización **-f**. Si se ejecutan los programas sin especificar este parámetro, aunque se especifiquen los otros, se marcará un error pidiendo que se especifique la frecuencia de operación. Los parámetros **--timing-alpha**, **--phase-alpha** y **--freq-alpha** son parte de la segunda versión del modulador DQPSK mientras que el parámetro **gain-mu** pertenece a la primera versión.

4.3. Tasa de error de bits

El análisis de la tasa de error de bits (Bit error rate o BER por sus siglas en Inglés) se realizó para ambos modelos de DQPSK proporcionados por *GNURadio*. El programa que se utilizó fue basado en un estudio generado y publicado en la página de *GNURadio* sobre una versión mejorada del modulador GMSK. El programa fue modificado para que pudiera analizar y comparar ambos esquemas DBPSK y DQPSK en sus dos versiones: costas/MM y filtros polifásicos.

La estructura del programa está dividida en 3 etapas. La primera etapa genera una señal de una fuente de números pseudo-aleatorios y despues se modula utilizando el modulador que se va a evaluar. Los datos arrojados por el modulador son enviados a archivos que se utilizarán como la entrada para las siguientes etapas. El programa define una clase llamada `SigGen` derivada de la clase `top_block` para definir un grafo de flujo de datos. La estructura del grafo se muestra en la figura 4–1.

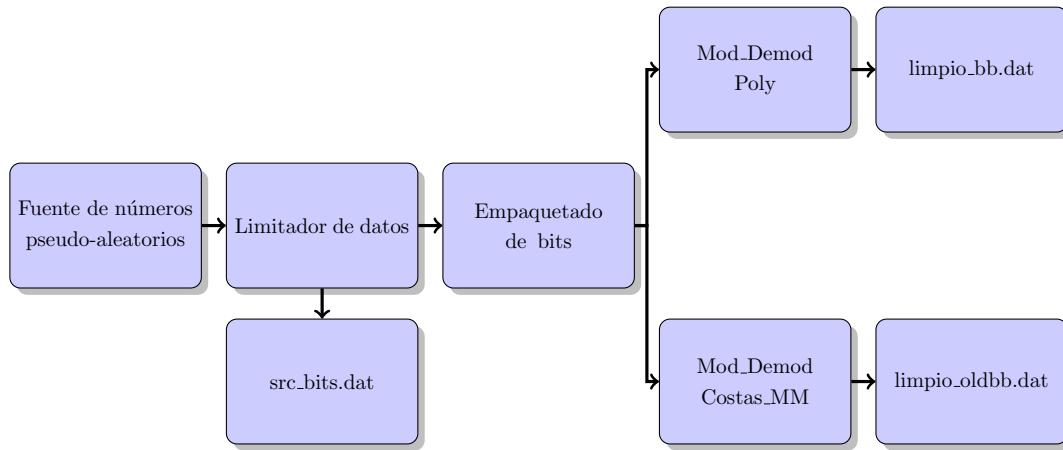


Figura 4–1: Grafo generador de datos modulados para la evaluación del BER.

Los bloques se describen de la siguiente manera:

- **Fuente de números pseudo-aleatorios:** El bloque se genera por medio de la clase `gr.glfsr_source_b`. Esta clase implementa un registro de desplazamiento con retroalimentación lineal en modo Galois. El sufijo “b” indica que el generador entrega valores binarios. El parámetro que se le especifica es el número de bits de precisión para generar una secuencia de bits de longitud 2^{N-1} , el cual expresa la máxima cantidad de valores que puede generar antes de ciclarse ([Alfke, 1996](#)).
- **Limitador de datos:** Este bloque se genera por medio de la clase `gr_head`, el cual su función es tomar una cierta cantidad de valores en su entrada y descartar todo lo demás. A esta secuencia de valores se le anexa el valor

especial *EOF* (fin de archivo por sus siglas en Inglés) que especifica el fin del flujo dentro del grafo. Esto causa que el grafo termine su ejecución una vez que este valor se propague a todos los bloques. Sus parámetros son el tipo de datos con el que va a trabajar y la cantidad de muestras que va a dejar pasar de su entrada a su salida.

- **Source_bits.dat, Limpio_bb.dat y Limpio_olddb.dat:** Estos bloques utilizan la clase `gr.file_sink` para enviar el flujo de datos a un archivo. En este grafo se generan tres archivos para guardar los bits originales y los bits modulados con los dos tipos de moduladores de *GNURadio*. Sus parámetros son el tipo de datos que se van a escribir y el nombre del archivo.
- **Empaquetado de bits:** Este bloque se implementa utilizando la clase `gr.unpacked_to_packed_bb`. Su función es agrupar los bits que entran y formar bytes completos en la salida. Estos bytes representan la información original.
- **Mod_Demod Poly y Mod_Demod Costas_MM:** Estos bloques representan los moduladores que se evaluaron durante el experimento y se implementan con las clases `blks2.dqpsk_mod` y `blks2.dqpsk2_mod` respectivamente.

El código que implementa este grafo se muestra en el listado 4.1.

La segunda etapa toma la señal modulada de los archivos y la pasa a través de un grafo que aplica ruido Gaussiano para simular un canal de transmisión AWGN. El grafo se muestra en la figura 4-2.

El grafo toma como entrada característica el valor de la relación de energía de bit a ruido E_b/N_0 que se va a utilizar en el análisis. Los bloques se implementan de la siguiente manera:

- **Fuente de datos de archivo:** Este bloque se implementa con la clase `gr.file_source`. Se utiliza para leer el archivo generado del grafo 4-1 que contiene los bits modulados.

```

class SigGen(gr.top_block):
    def __init__(self, num_bits=5000, pn_degree=23, xmit_bt=0.25,
                 samp_per_symbol=8):
        gr.top_block.__init__(self, 'SigGen')

        src = gr.glfsr_source_bb(pn_degree)
        src_limiter = gr.head(gr.sizeof_char, num_bits)

        bit_packer = gr.unpacked_to_packed_bb(1, gr.GR_MSB_FIRST)
        mod = blks2.dqpsk2_mod(samp_per_symbol, xmit_bt)

        bit_packer_old = gr.unpacked_to_packed_bb(1, gr.GR_MSB_FIRST)
        mod_old = blks2.dqpsk_mod(samp_per_symbol, xmit_bt)

        self.connect(src, src_limiter)
        self.connect(src_limiter, gr.file_sink(gr.sizeof_char, 'src_bits.
            dat'))
        self.connect(src_limiter, bit_packer, mod, gr.file_sink(gr.
            sizeof_gr_complex, 'limpio_bb.dat'))
        self.connect(src_limiter, bit_packer_old, mod_old, gr.file_sink(gr.
            sizeof_gr_complex, 'limpio_old_bb.dat'))

```

Listado 4.1: Código que implementa el grafo generador de señales.

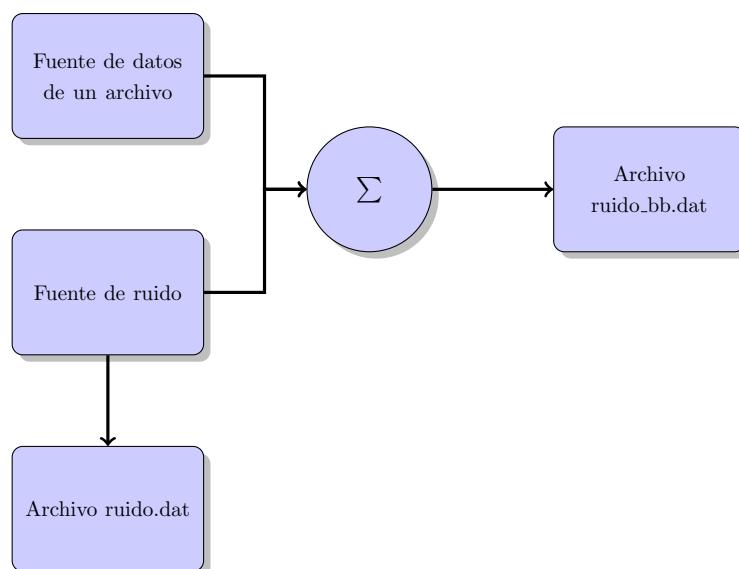


Figura 4–2: Grafo que simula un canal AWGN

- **Fuente de ruido:** Este bloque se implementa usando la clase `gr.noise_source_c`. El prefijo “c” indica que trabaja con datos complejos. La clase acepta dos parámetros de entrada: el tipo de ruido que se quiere generar y la magnitud. El tipo de ruido puede ser una de las siguientes constantes:

- `GR_UNIFORM`
- `GR_GAUSSIAN`
- `GR_LAPLACIAN`
- `GR_IMPULSE`

Para simular el canal AWGN se utilizó la constante `GR_GAUSSIAN`.

- **Archivo ruido_bb.dat y Archivo ruido.dat:** Este bloque usa la clase `file_sink_c` para guardar los datos complejos que representan la señal mezclada con el ruido. El archivo `ruido.dat` guarda los datos que representan el puro ruido sin la señal modulada.

Para determinar la magnitud del ruido el programa primero calcula la potencia promedio del archivo que contiene la señal modulada sin ruido, la energía por bit y la magnitud del ruido. La potencia se calcula utilizando la siguiente expresión:

$$P_{prom} = \frac{1}{N} \sum_{n=1}^N x^2(n) \quad (4.1)$$

La energía por bit se calcula a partir de la potencia como se muestra en la siguiente expresión:

$$E_b = \frac{P_{prom}}{f_b} \quad (4.2)$$

donde f_b es la tasa de bits en bits por segundo.

El listado 4.2 muestra dos funciones en *Python* que calculan estos dos parámetros.

El código que implementa el grafo 4–2 se muestra en el listado 4.3.

```

def get_p_avg_watts(fn , is_complex=False):

    f = open(fn)
    d = f.read()
    num_floats = (len(d)/4)
    d = struct.unpack('f'*num_floats , d)
    if is_complex:
        d = d[:2] #toma la parte real
        d_sq = [x**x for x in d]
    return sum(d_sq)/len(d_sq)

def get_eb_joules(fn , bits_per_sec , is_complex=False):

    p_avg = get_p_avg_watts(fn , is_complex=is_complex)
    return p_avg / bits_per_sec

```

Listado 4.2: Funciones en *Python* para calcular la potencia y la energía promedio de una señal

```

class NoiseGen(gr.top_block):
    def __init__(self , ebn0_dB , bits_per_sec , samp_per_sec):
        gr.top_block.__init__(<self , 'NoiseGen')

        ebn0_ratio = dB_to_ratio(ebn0_dB)

        n0_watts_per_Hz = bertool.get_eb_joules(fn = 'limpio_bb.dat' ,
            bits_per_sec = bits_per_sec) / ebn0_ratio
        n_mag = sqrt(0.5 * n0_watts_per_Hz * samp_per_sec)
        n_src = gr.noise_source_c(gr.GR_GAUSSIAN , n_mag)
        sig_src = gr.file_source(gr.sizeof_gr_complex , 'limpio_bb.dat')
        adder = gr.add_cc()

        self.connect(sig_src , adder , gr.file_sink(gr.sizeof_gr_complex , 'ruido_bb.dat'))
        self.connect(n_src , (adder , 1))
        self.connect(n_src , gr.file_sink(gr.sizeof_gr_complex , 'ruido.dat'))
    )

```

Listado 4.3: Código que implementa el grafo generador de ruido

La tercera etapa realiza la demodulación de las señales capturadas por los grafos anteriores. Esta etapa consiste en leer la señal mezclada con ruido, aplicar un filtro pasa bajas pre-selector para seleccionar la porción del canal de interés, demodular la señal recibida y guardar los datos generados en un archivo para realizar la comparación. La estructura del grafo que realiza esta etapa se muestra en la figura 4–3.

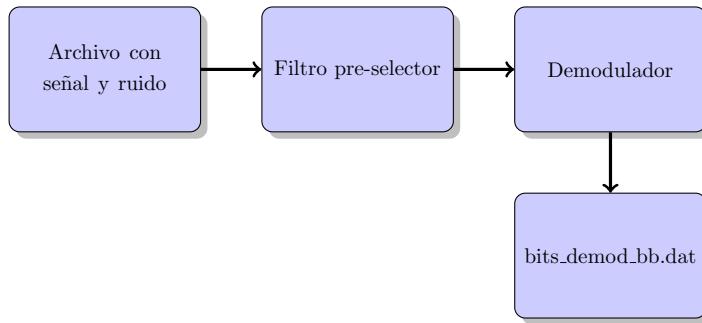


Figura 4–3: Grafo que realiza la demodulación de la señal simulada

El grafo tiene un parámetro obligatorio, el demodulador que se va a utilizar para el análisis. Los demás parámetros son opcionales y establecen las de muestras por símbolos, símbolos por segundo y los parámetros de ancho de banda para el filtro pre-selector. Para este análisis se utilizaron los valores por defecto. Los resultados del filtro se muestran en la figura 4–4.

Los bloques del grafo se implementan de la siguiente manera:

- **Archivo con señal y ruido:** La clase `gr.file_source_c` se encarga de leer el archivo generado por el grafo `NoiseGen` para enviarlo al filtro pre-selector.
- **Filtro pre-selector:** El filtro pasa bajas se implementa con una combinación de dos clases: `gr.firdes.low_pass` y `gr.fir_filter_ccf`. La primera se utiliza para generar los coeficientes del filtro a partir de los parámetros de entrada que se le especifiquen. Se utilizaron los valores por defecto del ejemplo

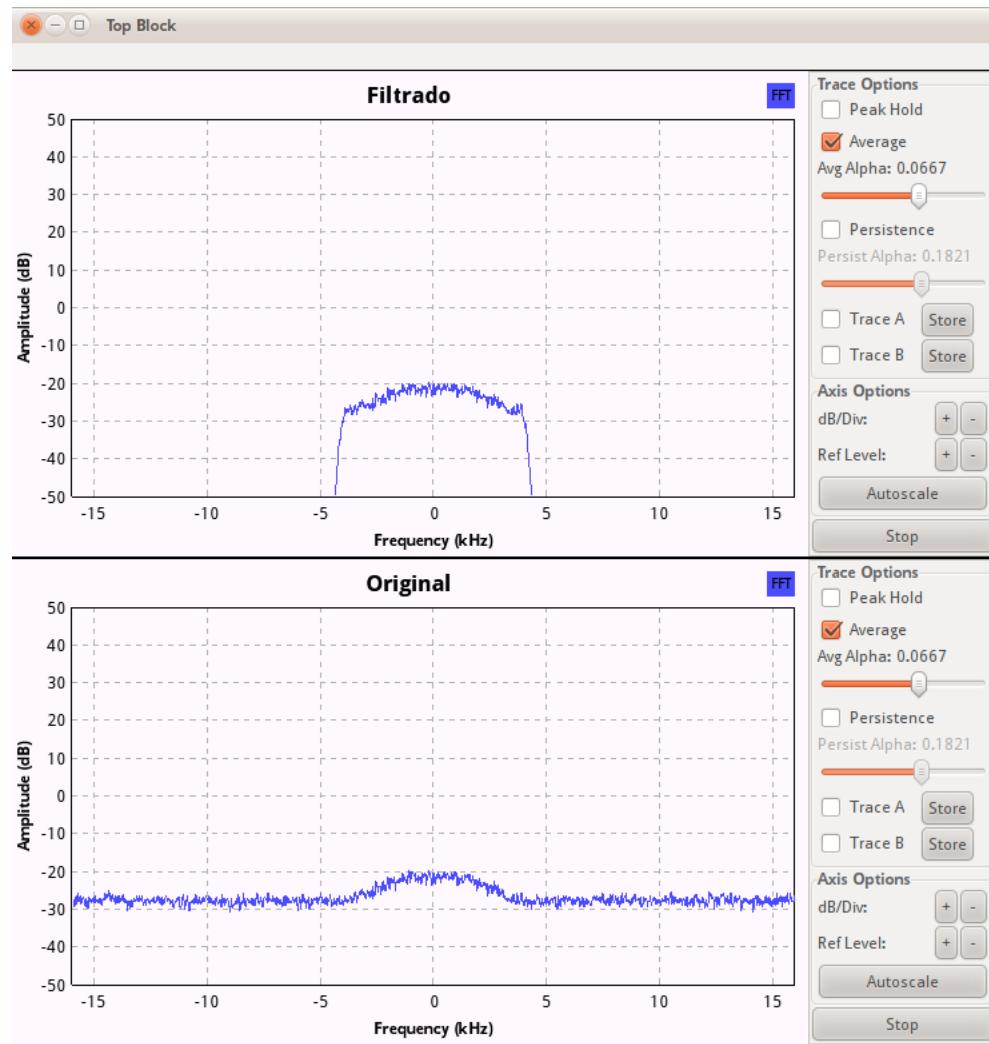


Figura 4–4: Respuesta del filtro pre-selector en el receptor para la medición del BER.

a excepción de la ventana. Originalmente utilizaba una ventana de tipo Hamming y se cambio a una ventana Kaizer ya que esta tuvo mejor rendimiento y ayudó a generar una curva de BER más suave. Los parámetros que acepta la clase son la ganancia, la frecuencia de muestreo, la frecuencia de corte, el ancho de la banda de transición y el tipo de ventana. La segunda clase implementa el filtro FIR (Respuesta finita al impulso por sus siglas en Inglés) a partir de los coeficientes generados por la clase `gr.firdes.low_pass`. Sus parámetros de entrada son el factor de decimación y los coeficientes del filtro.

- **Demodulador:** Este bloque representa el demodulador que se está evaluando. En este caso fueron ambos DQPSK y DBPSK en sus dos versiones. Las clases que los implementan se encuentran dentro del paquete `blk2s`. Su uso se muestra en el listado 4.5. La clase toma la señal filtrada e inicia el proceso de demodulación y detección. Los resultados son una serie de bits desempaquetados, es decir, bits individuales.
- **Archivo `bits_demod_bb.dat`:** Este bloque usa la clase `gr.file_sink` para escribir a un archivo los bits resultantes del bloque demodulador.

El código que implementa el tercer grafo se muestra en el listado 4.4.

El programa principal inicializa los tres grafos y realiza una serie de pruebas para generar la grafica del BER. La metodología que se siguió fue generar varios valores de E_b/N_0 , desde 12db hasta 2db en intervalos de 0.5db y ejecutar los tres grafos para generar los resultados. Estos resultados se van guardando en un arreglo y posteriormente se grafican. Los parámetros que se utilizaron para la simulación fueron los siguientes:

- **Número de bits:** 20000
- **Muestras por símbolo:** 7
- **Símbolos por segundo:** $1e6$
- **Exceso de ancho de banda:** 0.75

```

class Analyser(gr.top_block):
    def __init__(self, bb_src_fn, test_demod, samp_per_sym = 8,
                 sym_per_sec = 1e6, pre_detect_filt_bt = 0.9,
                 filt_transition_ratio = 0.1):
        gr.top_block.__init__(self, 'Analyser')

        self.bb_src_fn = bb_src_fn
        self.samp_per_sym = samp_per_sym
        samp_per_sec = samp_per_sym * sym_per_sec

        bb_src = gr.file_source(gr.sizeof_gr_complex, bb_src_fn)

        pre_detect_filt_bw = sym_per_sec * pre_detect_filt_bt
        pre_detect_filt_taps = gr.firdes.low_pass(1.0, samp_per_sec,
                                                pre_detect_filt_bw, filt_transition_ratio *
                                                samp_per_sec,
                                                gr.firdes.WIN_KAISER, 4.5)
        pre_detect_filt = gr.fir_filter_ccf(1, pre_detect_filt_taps)
        self.connect(bb_src, pre_detect_filt, test_demod)

        self.test_demod_dst_fn = 'bits_demod_bb.dat'
        self.dst = gr.file_sink(gr.sizeof_char, self.test_demod_dst_fn)
        self.connect(test_demod, self.dst)

```

Listado 4.4: Código que implementa el grafo demodulador para el análisis del BER.

El código del programa principal se muestra en el listado 4.5. El programa original se modificó para que pueda evaluar ambos esquemas a la vez.

Los resultados de la primera versión de los esquemas DQPSK y DBPSK se muestran en la figura 4–5.

Los resultados muestran el rendimiento entre los dos esquemas de modulación a diferentes niveles de SNR. Debido a que DQPSK utiliza dos bits por símbolo tiene más probabilidades de generar errores a niveles muy altos de ruido. Para poder lograr niveles confiables de transmisión se requiere más potencia que DBPSK.

La segunda versión de estos esquemas intenta mejorar el BER por medio de la implementación de bancos de filtros polifásicos. La misma metodología se siguió para su análisis y los resultados se muestran en la figura 4–6. Como se puede observar, los filtros polifásicos ofrecen una mejora en el rendimiento de ambos esquemas. A

```

if __name__ == "__main__":
    num_bits = 20000
    samp_per_sym = 7
    sym_per_sec = 1e6
    samp_per_sec = sym_per_sec * samp_per_sym
    print "Samples per second: ", samp_per_sec
    xmit_bt = 0.75
    ebn0s_dB = list(numpy.arange(12.0, 1.999, -0.5))
    recv_bt = 0.9
    recv_filt_transition_ratio = 0.1

    sg = SigGen(num_bits = num_bits, xmit_bt = xmit_bt, samp_per_symbol
                = samp_per_sym)
    sg.run()
    del sg

    dbpsk2_demod_bers = []
    dqpsk2_demod_bers = []
    dbpsk2_delay = None
    dqpsk2_delay = None

    for ebn0_dB in ebn0s_dB:
        print ebn0_dB
        ng = NoiseGen(ebn0_dB = ebn0_dB, bits_per_sec = sym_per_sec,
                      samp_per_sec = samp_per_sec)
        ng.run()
        del ng

    #
    # Analysis de DQPSK
    #
    test_dqpsk2_demod = blks2.dqpsk_demod(samples_per_symbol =
                                             samp_per_sym, excess_bw = xmit_bt)

    a = Analyser(bb_src_fn = 'noisy_qpsk2_bb.dat', test_demod =
                 test_dqpsk2_demod, samp_per_sym = samp_per_sym,
                 sym_per_sec = sym_per_sec, pre_detect_filt_bt =
                 recv_bt,
                 filt_transition_ratio = recv_filt_transition_ratio
                 )

    a.run()
    a.close()
    del a

    if dqpsk2_delay is None:
        dqpsk2_delay = bertool.get_delay('src_bits.dat', '
                                         test_demod_dst_bits.dat')
    ber_stats = bertool.get_ber_stats('src_bits.dat', '
                                         test_demod_dst_bits.dat', dqpsk2_delay)
    dqpsk2_demod_bers.append(ber_stats[0])

```

```

#=====
# Analysis de DBPSK
#=====

test_dbpsk2_demod = blks2.dbpsk_demod(samples_per_symbol =
    samp_per_sym, excess_bw = xmit_bt)

a = Analyser(bb_src_fn = 'noisy_bpsk2_bb.dat', test_demod =
    test_dbpsk2_demod, samp_per_sym = samp_per_sym,
    sym_per_sec = sym_per_sec, pre_detect_filt_bt =
    recv_bt,
    filt_transition_ratio = recv_filt_transition_ratio
    )

a.run()
a.close()
del a

if dbpsk2_delay is None:
    dbpsk2_delay = bertool.get_delay('src_bits.dat', '
        test_demod_dst_bits.dat')
ber_stats = bertool.get_ber_stats('src_bits.dat', '
    test_demod_dst_bits.dat', dbpsk2_delay)
dbpsk2_demod_bers.append(ber_stats[0])

ebn0s_dB.reverse()
dbpsk2_demod_bers.reverse()
dqpsk2_demod_bers.reverse()

fig = p.figure()
p.title(u'Desempe o de DQPSK y DBPSK')
ax = fig.add_subplot(111)
ax.semilogy(ebn0s_dB, dqpsk2_demod_bers, 'b', label = 'DQPSK')
ax.hold(True)
ax.semilogy(ebn0s_dB, dbpsk2_demod_bers, 'r', label = 'DBPSK')
ax.hold(False)

ax.yaxis.grid(True, which = 'minor')
ax.xaxis.grid(True)

p.legend()

p.ylabel('BER')
p.xlabel('Ebn0_dB')
p.show()

```

Listado 4.5: Código *Python* de la rutina principal del análisis del BER.

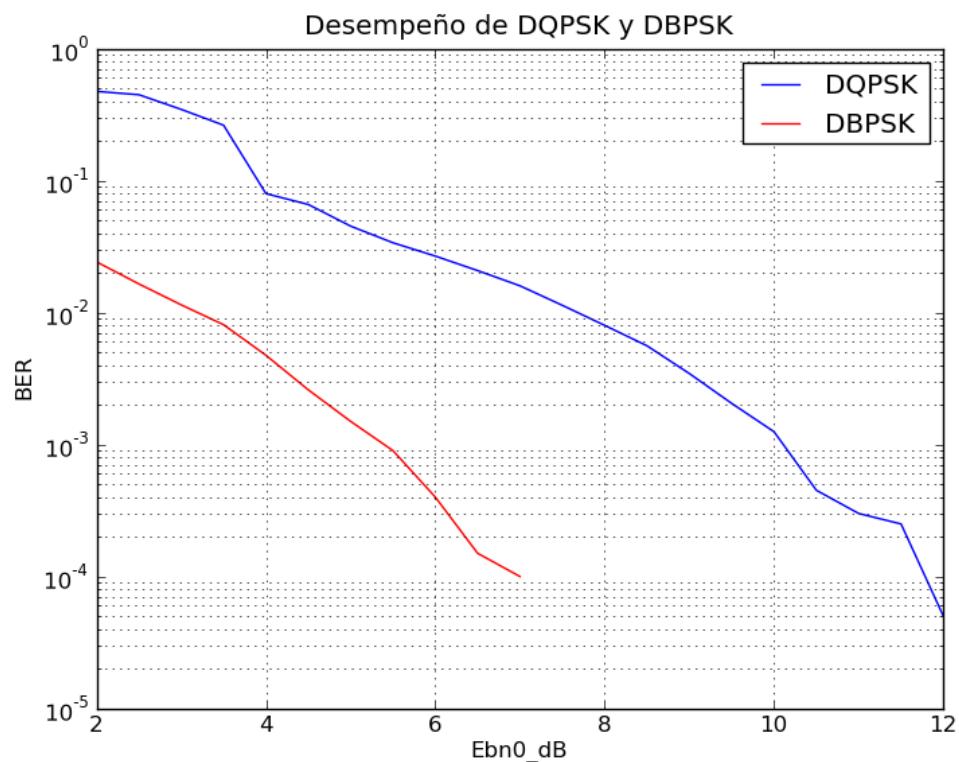


Figura 4–5: Gráfica comparativa del BER para la primera versión de los esquemas DQPSK y DBPSK.

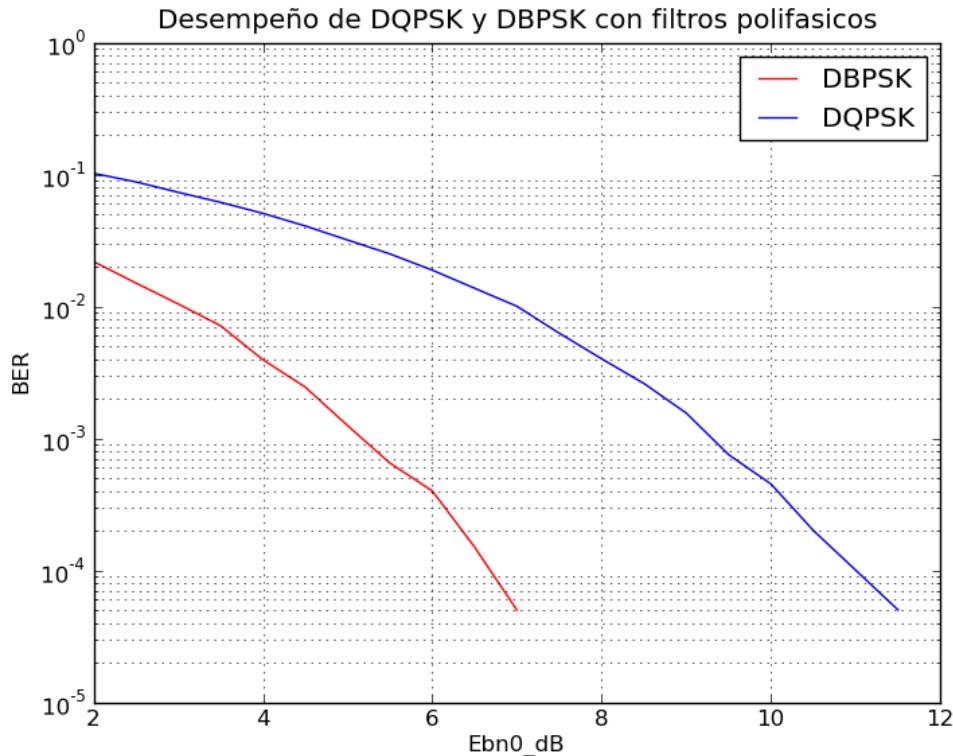


Figura 4–6: Gráfica comparativa del BER para la segunda versión de los esquemas DQPSK y DBPSK.

niveles bajos de SNR DQPSK tiene mayor tolerancia que la versión anterior para generar errores.

Por último se generó una simulación en Matlab utilizando la herramienta *BER-Tool* para generar un estudio teórico y comparar los resultados con la práctica. La herramienta fue configurada lo más cercano posible a los parámetros que se utilizaron en el experimento. Se generaron dos estudios, uno para DBPSK y otro para DQPSK, con un canal de transmisión AWGN. El resultado de esta simulación se muestra en la siguiente figura 4–7.

Al comparar los resultados de la simulación con los del experimento se observó que a menor E_bN_0 los algoritmos de *GNURadio* logran un mejor rendimiento que la teoría, sobre todo el esquema DBPSK. Esto se debe a los diversos métodos

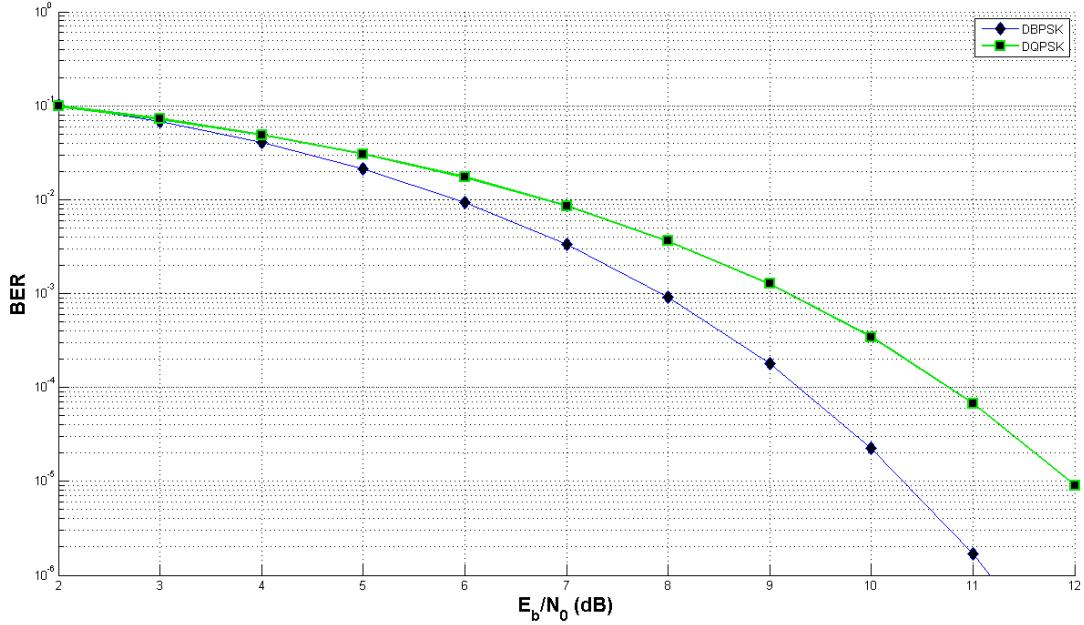


Figura 4-7: Resultados teóricos de la herramienta *BERTool* de Matlab.

de recuperación de la información que emplean ambos algoritmos ya que constantemente han estado siendo optimizados por los autores. Con esto se demuestra el rendimiento del código fuente actual de *GNURadio*.

4.4. Espectrograma de la transmisión

GNURadio contiene varias herramientas para visualizar las señales que se transmiten y se reciben. Estas herramientas son programas completos que se pueden utilizar como base para elaborar programas más complejos. Una de estas herramientas visuales es un espectrograma que permite ver la densidad espectral de potencia de una señal que se está recibiendo. La compilación del código fuente instala este programa en `/usr/bin/` para que se pueda acceder a ella de manera global. El programa se llama `usrp_fft.py` y se encuentra originalmente en la carpeta `gr-utils` del código fuente. Este programa se puede acceder desde esa carpeta o bien desde su lugar global por medio de la consola de comandos. El programa proporciona varios parámetros que permiten que se configure de diferentes maneras dependiendo del uso que se le va dar. Estos parámetros son los siguientes:

- **-w:** Selecciona el USRP que se va a utilizar en caso de haber más de uno conectado a la PC.
- **-R:** Selecciona el lado RX que se va a utilizar en el USRP: Lado A o Lado B. Esto lo determina el lugar donde esté conectada la tarjeta auxiliar.
- **-A:** Selecciona la antena que se va a utilizar. Esto solo se utiliza con las tarjetas de la serie RFX.
- **-d:** Selecciona el valor de decimación con el que se configurará el DDC del FPGA. Por defecto lo configura a 16.
- **-f:** Especifica la frecuencia con la que se sintoniza la tarjeta auxiliar.
- **-g:** Especifica la ganancia con la que se programa el amplificador programable. Los valores son en decibeles.
- **-W:** Activa la opción de utilizar el espectrograma. Por defecto la aplicación actúa como un analizador de espectros, desplegando la FFT de la señal recibida.
- **-S:** Activa la opción de utilizar el osciloscopio.
- **--fft-size:** Especifica el tamaño de muestras que utiliza la FFT. Por defecto utiliza 1024 muestras.

Este programa se utilizó durante la transmisión de los datos generados por los programas `benchmark_tx.py` y `benchmark_rx.py` para observar la densidad espectral de potencia de la portadora. Las opciones que se utilizaron para configurar el programa fueron las siguientes:

```
usrp_fft.py -f 30M -d 256 -W
```

El espectrograma se muestra en la figura 4-8.

El espectrograma consiste en una grafica de cascada (waterfall en Inglés) que se despliega en forma vertical (algunos autores la despliegan en forma horizontal). El eje Y es el tiempo medido en segundos y la graficá se desplaza de abajo hacia arriba. El eje X representa el espectro en Hz. La figura 4-8 muestra la intensidad

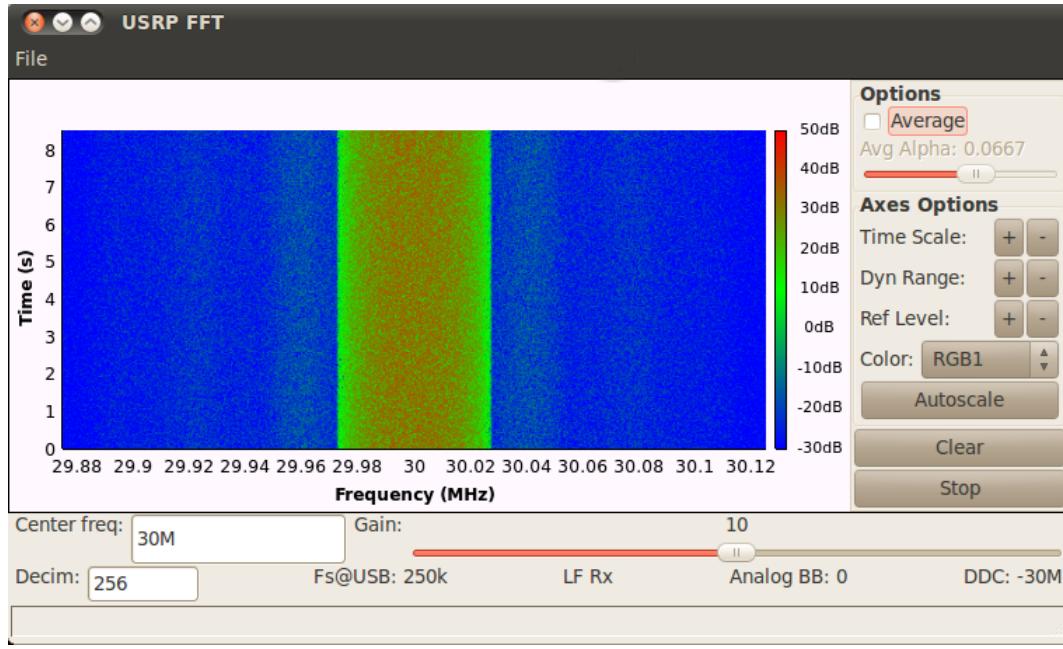


Figura 4–8: Aplicación en *Python* que muestra el espectrograma de lo que el USRP está capturando.

de la portadora de 30Mhz centrada en la grafica. Los colores tenues a un lado de la barra verde representan las bandas laterales de la portadora. Desde esta aplicación es posible modificar varios parámetros del USRP como la ganancia, la frecuencia y el valor de decimación del DDC. Los ejes también se pueden ajustar por medio de las opciones que aparecen en la derecha de la pantalla (Time Scale, Dyn Range, Ref Level). En este ejemplo la portadora se puede apreciar claramente debido a la mínima cantidad de ruido ya que se utilizó un cable coaxial para la transmisión, el cual introduce una cantidad despreciable de ruido.

4.5. Diagrama de ojo

Para observar la calidad de la señal antes de que sea demodulada se utilizó el diagrama de ojo. Esta herramienta permite ver las distorsiones que ocurren en la transmisión y ayuda a tomar mejores decisiones sobre el diseño del receptor. *GNU-Radio* no contiene esta herramienta aun por lo que se optó por crear un programa en *Python* que pueda realizar el diagrama en base a muestras capturadas por el USRP.

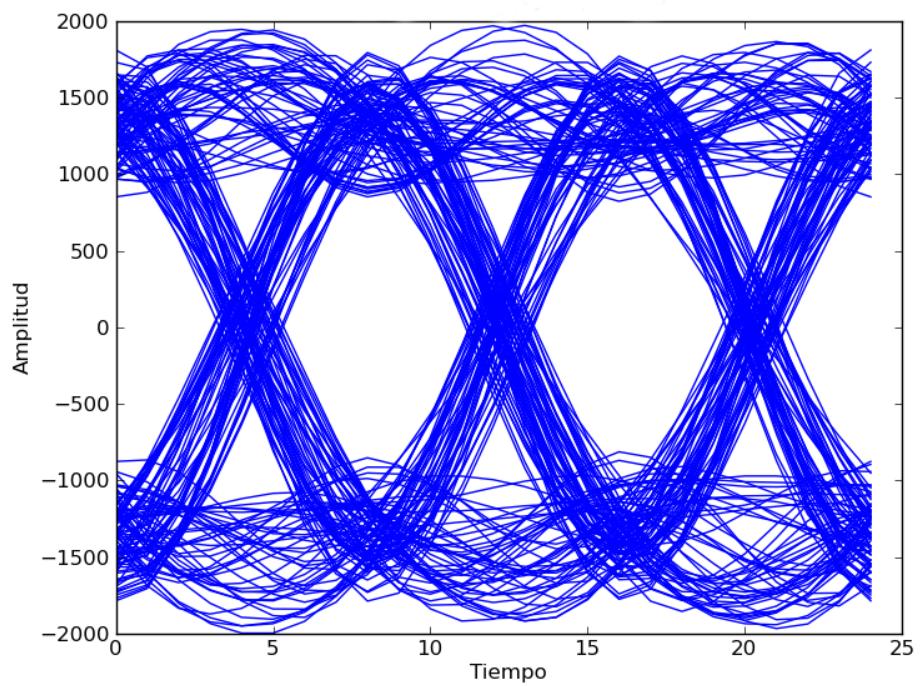
La captura de datos se realizó utilizando una de las herramientas de la carpeta `gr-utils` llamada `usrp_rx_cfile.py`. Este programa acepta como parámetro principal el nombre que se le asignará al archivo que va a almacenar las muestras. Los demás parámetros se utilizan para configurar el DDC y sintonizar la tarjeta auxiliar a la frecuencia deseada. Los parámetros que soporta el programa son los siguientes:

- **-R:** Selecciona el lado del USRP que se va a utilizar: A o B.
- **-d:** Selecciona la tasa de decimación. El valor por defecto es 16.
- **-f:** Establece la frecuencia.
- **-g:** Establece la ganancia en dB.
- **-8:** Configura el USRP para que envíe muestras de 8 bits en lugar de 16 bits.
- **-s:** Configura el USRP para que envíe muestras de tipo *short* entrelazadas en lugar de muestras complejas de tipo *float*.
- **-N:** Establece la cantidad de muestras que se van a capturar. Por defecto es infinito.

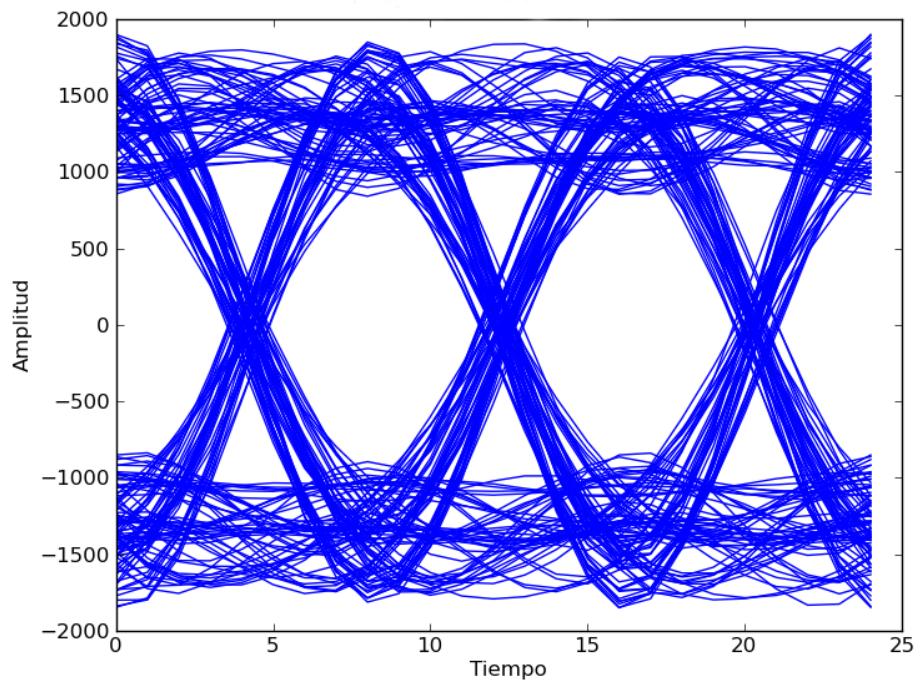
El programa se configuró para que capture muestras a una frecuencia central de 30Mhz. El tipo de muestras son números complejos de tipo *float* ya que por defecto es con el que trabaja el USRP.

El experimento consistió en capturar muestras de tres transmisiones a diferentes valores de exceso de ancho de banda (α) para observar los efectos del filtro de coseno elevado en la señal. Los resultados se muestran en la figura 4-8.

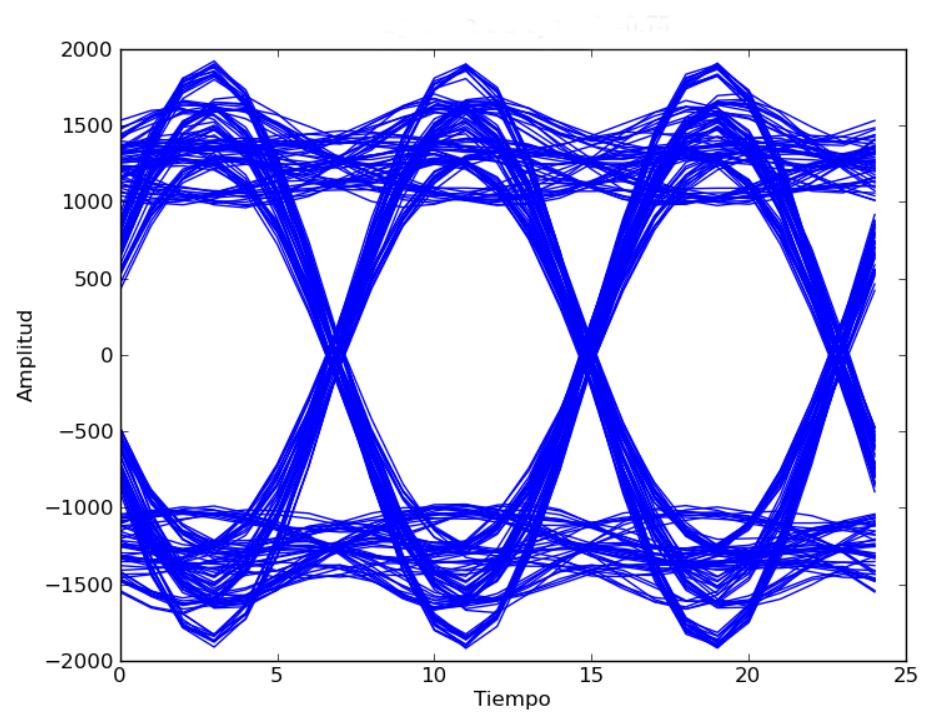
La figura 4-9a muestra que a menor α la señal se distorsiona más y esto causa mayores dificultades para detectar los símbolos correctamente pero tambien resulta en un filtro más rápido y sencillo de implementar debido a que su respuesta al impulso es más corta. La figura 4-8c muestra menos distorsión en la señal a consecuencia de un filtro más complicado de procesar eficientemente. El uso de un mayor ancho de banda también implica que se puede introducir más ruido en la etapa de captura de RF (front end) y por lo tanto es necesario considerar las compensaciones entre el



(a) $\alpha = 0.35$



(b) $\alpha = 0.50$



(c) $\alpha = 0.75$

Figura 4–8: Señales QPSK con filtrado de coseno elevado a diferentes valores de α .

```

import pylab as p
import scipy

samp_per_sym = 8
hfile = '/home/haysoos/Documents/Tesis/usrp_capture/target.dat'

def get_traces(d, num_traces, samp_per_sym):
    traces = []
    trace_len = samp_per_sym * 3
    for i in range(num_traces):
        start_ind = (samp_per_sym / 2) - 1 + (i + 3) * trace_len
        traces.append(d[start_ind:start_ind + trace_len + 1])
    return traces

iq = scipy.fromfile(hfile, dtype = scipy.complex64)

ichan = [c.real for c in iq]

traces = get_traces(ichan, 200, samp_per_sym)
p.figure()
p.hold(True)
for tr in traces:
    p.plot(tr, 'b-')

p.title('Diagrama de ojo ' + r'$\alpha=0.75$')
p.xlabel('Tiempo')
p.ylabel('Amplitud')

p.show()

```

Listado 4.6: Programa que genera el diagrama de ojo a partir de muestras tomadas del USRP.

exceso de ancho de banda, la inmunidad contra el ruido, la complejidad de la etapa de recuperación del reloj de símbolos y la eficiencia de procesamiento por parte del CPU (Lee, 2003). El CPU Core 2 Duo que se utilizó fue suficiente para procesar filtros con α de 0.75. Estos filtros son internamente implementados con una mezcla de lenguaje C y ensamblador utilizando instrucciones como SSE (*streaming SIMD extensions*) para incrementar la velocidad y eficiencia de procesamiento.

El programa que genera el diagrama de ojo a partir de un archivo con muestras del USRP se muestra en el listado 4.6.

Una observación importante es que los valores de tipo *float* en *Python* equivalen al tipo *double* en C el cual puede variar dependiendo el CPU que se esté utilizando pero en casos típicos la precisión es de 64 bits: 1 bit para el signo, 11 para el exponente y 52 para la mantisa ([Python Software Foundation, 2010](#)). El USRP configurado con el *source* o *sink* que utiliza el prefijo “c” utiliza la constante `gr_complex` la cual equivale en lenguaje C a *float*. Este tipo de dato es de 32 bits: 1 bit para el signo, 8 bits para el exponente y 23 bits para la mantisa. *Python* por defecto no tiene soporte para representar el tipo *float* de C lo cual introduce errores en la representación de las muestras del USRP. La biblioteca *Numpy* y *Scipy* de *Python* encapsula varias representaciones de números flotantes ([SciPy, 2010](#)) lo cual es necesario utilizar la correcta cuando se trabaja con muestras que genera el USRP de manera independiente de los programas de *GNURadio*. Cada muestra compleja que transmite o recive el USRP es representado por dos números flotantes de 32 bits por lo que es necesario utilizar el tipo de dato *complex64* como se muestra en el listado [4.6](#).

4.6. Constelación observada

Para observar la constelación recibida se utilizó la versión que utiliza la biblioteca QT del programa *benchmark*. El programa RX contiene dos versiones en el código fuente, una que utiliza la consola y otra que genera una interfaz de usuario para monitorear la señal de manera gráfica. Esta aplicación también es un buen ejemplo para ilustrar el uso de los controles gráficos de *GNURadio* que utilizan esta biblioteca. Su diferencia con los controles que utilizan la biblioteca *wxWidgets* es que el código de los controles QT ya está optimizado para realizar gráficas con la ayuda de la extensión *QWT* el cual es una biblioteca madura mientras que los controles que utilizan *wxWidgets* fueron creados desde cero por los autores de *GNURadio* y no están optimizados en su totalidad. *QWT* es necesario que se instale correctamente para utilizar los controles de QT. La instalación se explica a detalle en el apéndice [A](#). Los controles QT forman un solo bloque para uso en los grafos de *GNURadio*. Este

```

qtgui_make_sink_X (int fftsize , int wintype , double fc=0,
                    double bandwidth=1.0, const std::string &name="
                        Spectrum Display",
                    bool plotfreq=true , bool plotwaterfall=true , bool
                        plotwaterfall3d=true ,
                    bool plottime=true , bool plotconst=true , bool
                        use_OpenGL=true ,
QWidget *parent=NULL)

```

Listado 4.7: Constructor de los bloques qtgui_sink_x

bloque se llama `qtgui.sink_c` para datos complejos y `qtgui.sink_f` para datos de punto flotante. El constructor de estos bloques tiene la forma descrita en el listado 4.7.

Los parámetros del constructor se describen de la siguiente manera:

- **fftsize**: Tamaño inicial de la FFT.
- **wintype**: Tipo de ventana inicial para la FFT. Puede ser una de las siguientes constantes tomadas de la clase `gr.firdes`:
 - WIN_BLACKMAN
 - WIN_BLACKMAN_hARRIS
 - WIN_HAMMING
 - WIN_HANN
 - WIN_KAISER
- **fc**: Frecuencia central para el eje X.
- **bandwidth**: Especifica el rango alrededor de la frecuencia central para el eje X.
- **name**: Especifica el título del GUI.
- **plotfreq**: Despliega el analizador de espectros.
- **plotwaterfall**: Despliega el spectrograma.
- **plotwaterfall3d**: Despliega el spectrograma en 3D.
- **plottime**: Despliega el osciloscopio.
- **plotconst**: Despliega el diagrama de constelación.

- **use_OpenGL**: Utiliza extensiones para acelerar el despliegue de las gráficas.
- **parent**: Especifica un objeto el cual se utiliza como ventana principal para desplegar los controles.

El programa RX que se utilizó se llama `benchmark_qt_rx.py`. Utiliza los mismos comandos que la versión de consola para su configuración pero también agrega uno nuevo que es la opción `-G`. Esta opción habilita el GUI que consiste en una aplicación QT que muestra el rendimiento del receptor y que permite modificar los parámetros del receptor como los del lazo de costas y el sincronizador M&M. Si no se utiliza la opción `-G` el programa se comportará igual que la versión de consola.

Todos los programas `benchmark_xx.py` tienen la opción `-from-file` que especifica que los datos que se van a transmitir vienen de un archivo binario. Si esta opción no se especifica entonces el programa por defecto transmite los 8 bits menos significativos del número de paquete en una cadena de caracteres ASCII. El usuario es libre de modificar el ejemplo para transmitir algún otro tipo de secuencia.

El experimento inicial consistió en transmitir la secuencia de números de paquetes y observar la constelación recibida. Utilizando los valores por defecto de los parámetros descritos en el apartado 4.2 y anexando la opción `-G` se ejecuto primero el programa `benchmark_qt_rx.py` y despues el programa `benchmark_tx.py`. Los resultados obtenidos para una transmisión con una tasa de bits de 100kbits/s se muestran en la figura 4-9.

El GUI que muestra la figura 4-9 es una aplicación completa con la biblioteca QT. El usuario puede modificar toda la interfaz a su gusto o bien generar una totalmente diferente aunque esto requiere conocimientos de programación con esta biblioteca. Las aportaciones de *GNURadio* a esta aplicación son los dos bloques QT que muestran los controles con las graficas. Los controles de la izquierda muestran la señal antes de ser demodulada en software, es decir, después de la etapa DDC del USRP. Los controles de la derecha muestran la señal después de la etapa de

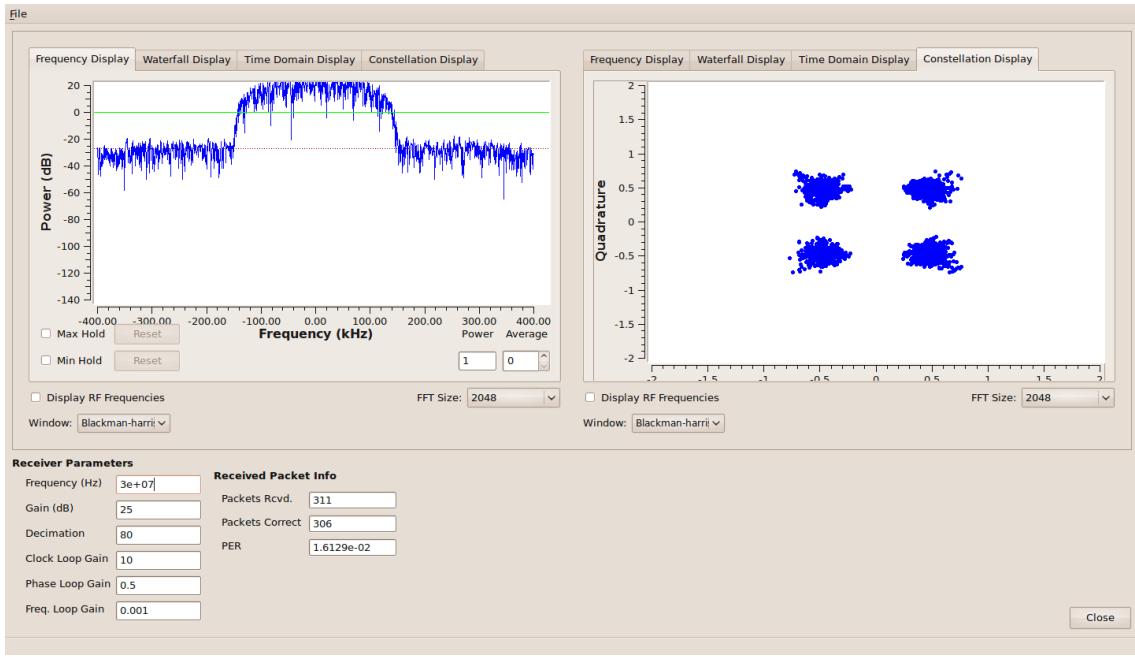


Figura 4–9: Resultados de la transmisión con parámetros por defecto del programa a 100kbits/s.

demodulación. En la parte inferior se muestran algunas opciones para modificar los parámetros del receptor como la decimación, la ganancia, y los parámetros del PLL y el sincronizador de reloj de símbolos.

Los valores por defecto que utiliza la aplicación `benchmark_tx.py` y `benchmark_rx.py` no fueron eficientes para lograr una transmisión satisfactoria. La figura 4–9 muestra los puntos de la constelación muy esparcidos y forman nubes alrededor del punto de convergencia por lo que la transmisión mostraba una probabilidad de error muy alta. Estos programas utilizan la primera versión de la modulación DQPSK (lazo de costas y sincronizador M&M).

Los parámetros se ajustaron manualmente y se realizaron varias transmisiones hasta encontrar los adecuados. Los parámetros y valores mencionados en el apartado 4.2 lograron una transmisión exitosa dando una recepción de todos los paquetes enviados. La constelación que se logró utilizando los parámetros del apartado 4.2 se

muestra en la figura 4–10. Se puede observar que los símbolos recibidos convergen correctamente en los cuatro puntos de la constelación DQPSK.

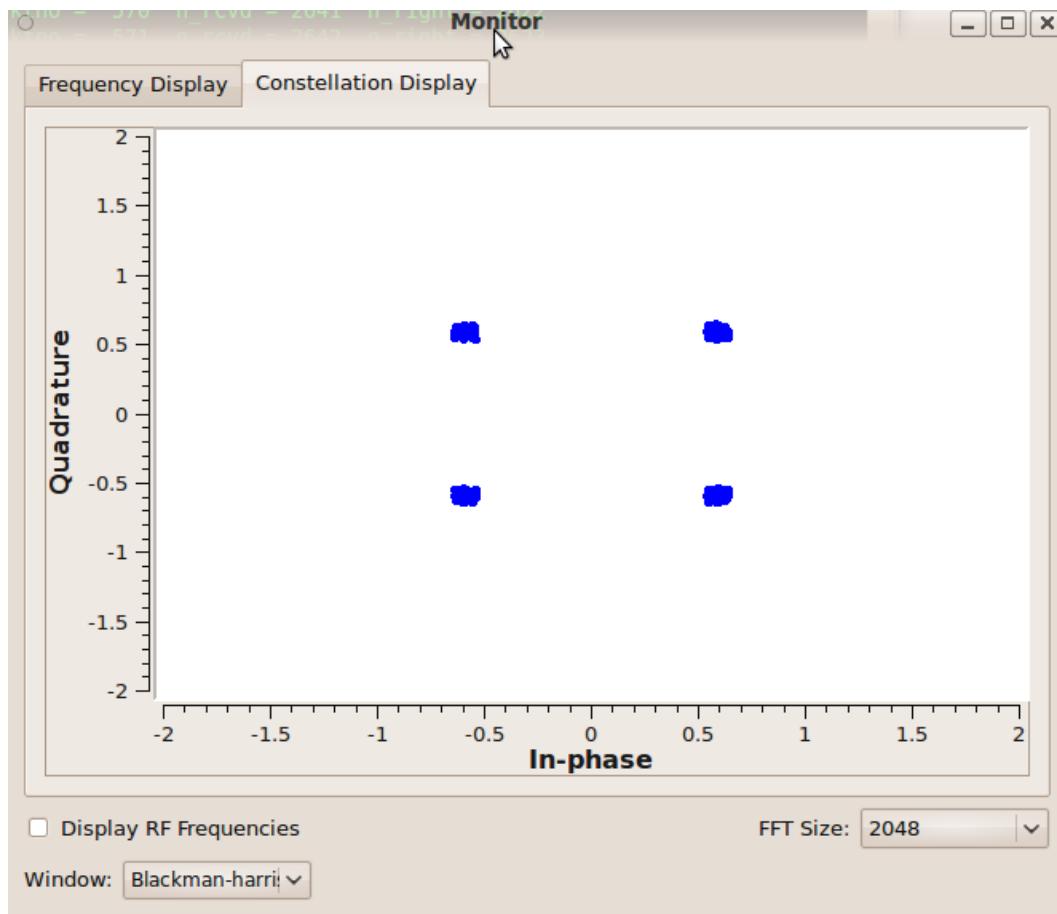


Figura 4–10: Constelación obtenida con parámetros óptimos de una transmisión a 100kbits/s

Sin la opción $-G$ el programa mostrará su actividad en la consola. Este es el comportamiento por defecto y es también el comportamiento para los programas que no implementan el GUI. La transmisión de la figura 4–10 sin utilizar el GUI se muestra en la figura 4–11.

El diseño original del programa muestra si los paquetes transmitidos son recibidos correctamente pero no despliega el contenido de ellos. Cada línea que muestra la figura 4–11b indica paquete se decodificó correctamente, el número del paquete decodificado, la cantidad total de paquetes recibidos y la cantidad total de paquetes

```
File Edit View Terminal Help
Could not find suitable rate for specified SPS and Bitrate
Using rate = 512 for bitrate of 31.25kbps
USRP Sink: A: LF Tx
Interpolation Rate: 512
>>> gr_fir_ccf: using SSE

Modulator:
bits per symbol:      2
Gray code:           True
RRS roll-off factor: 0.750000
Tx amplitude:        0.5
modulation:          dqpsk2_mod
bitrate:             31.25kb/s
samples/symbol:      4.0000
Warning: failed to enable realtime scheduling
```

(a) Transmisor

```
File Edit View Terminal Help
ok =  True  pktno =  556  n_rcvd = 2627  n_right = 2608
ok =  True  pktno =  557  n_rcvd = 2628  n_right = 2609
ok =  True  pktno =  558  n_rcvd = 2629  n_right = 2610
ok =  True  pktno =  559  n_rcvd = 2630  n_right = 2611
ok =  True  pktno =  560  n_rcvd = 2631  n_right = 2612
ok =  True  pktno =  561  n_rcvd = 2632  n_right = 2613
ok =  True  pktno =  562  n_rcvd = 2633  n_right = 2614
ok =  True  pktno =  563  n_rcvd = 2634  n_right = 2615
ok =  True  pktno =  564  n_rcvd = 2635  n_right = 2616
ok =  True  pktno =  565  n_rcvd = 2636  n_right = 2617
ok =  True  pktno =  566  n_rcvd = 2637  n_right = 2618
ok =  True  pktno =  568  n_rcvd = 2639  n_right = 2620
ok =  True  pktno =  569  n_rcvd = 2640  n_right = 2621
```

(b) Receptor

Figura 4–11: Programa *benchmark* TX y RX en modo consola.

decodificados correctamente. La figura 4-11a muestra la configuración del transmisor y los puntos en la parte inferior emulan una barra de progreso indicando que el programa está actualmente transmitiendo información al USRP.

Para demostrar el uso del bloque gráfico de QT en un programa de consola se modificó el programa original para incluir este bloque dentro del grafo y poder observar la constelación sin tener que desarrollar todo el GUI. La modificación se realizó en el archivo `usrp_receive_path2.py` dentro del constructor de la clase `usrp_receive_path` después de la etapa de configuración del USRP y en el archivo principal `benchmark_rx2.py` en la función `main()`. Esta modificación se muestra en el listado 4.8.

De acuerdo con la documentación de *GNURadio*, para poder utilizar el bloque QT es necesario seguir los siguientes pasos:

- Crear un apuntador a una aplicación de QT. Para esto se utiliza la clase `QApplication`.
- Crear el bloque sink de QT y conectarlo.
- Utilizar la utilería SIP (parte de los requerimientos de la instalación de QT) para convertir el apuntador del objeto C++ que representa el sink a un objeto *Python*. Esto se hace con el fin de poder tener acceso directo a los métodos del objeto desde *Python*.
- Mandar llamar el método `show()` del objeto creado.
- Mandar llamar el método `start()` del bloque jerárquico en lugar del método `run()`. El método `start()` no causa la aplicación que se detenga hasta que el grafo termine de ejecutarse debido a que no manda llamar el método `wait()`.
- Mandar llamar el método `_exec()` del apuntador de la aplicación QT para poder desplegar los controles gráficos.

Para llegar a los bloques del grafo del demodulador DQPSK fue necesario pasar por varias capas del código hasta llegar al bloque que se va monitorear. Esto es debido

```

class usrp_receive_path(gr.hier_block2):
    def __init__(self, demod_class, rx_callback, options):
        gr.hier_block2.__init__(self, "usrp_receive_path",
                               gr.io_signature(0, 0, 0), # Input signature
                               gr.io_signature(0, 0, 0)) # Output signature
        if options.rx_freq is None:
            sys.stderr.write("-f FREQ or --freq FREQ or --rx-freq FREQ
                             must be specified\n")
            raise SystemExit

        #setup usrp
        self._demod_class = demod_class
        self._setup_usrp_source(options)

        rx_path = receive_path.receive_path(demod_class, rx_callback,
                                             options)
        for attr in dir(rx_path): #forward the methods
            if not attr.startswith('_') and not hasattr(self, attr):
                setattr(self, attr, getattr(rx_path, attr))

    #QT sink para monitorear la recepcion en el USRP.
    fftsize = 2048
    window = gr.firdes.WIN_BLACKMAN_hARRIS
    fc = 0
    bw = options.bitrate

    self.qapp = QtGui.QApplication(sys.argv)

    self.receiver = rx_path.packet_receiver._demodulator.
                    phase_recov
    self.monitor = qtgui.sink_c(fftsize, window, fc, bw, "Monitor",
                               True, False, False, False, True)
    #connect
    self.connect(self.u, rx_path)
    self.connect(self.receiver, self.monitor)

    self.pyobj = sip.wrapinstance(self.monitor.pyqwidget(), QtGui.
                                 QWidget)
    self.pyobj.show()

```

Listado 4.8: Modificación del programa *benchmark* para desplegar el bloque gráfico de QT.

```

def main():
    global n_rcvd, n_right, dest_file

    n_rcvd = 0
    n_right = 0

        #Código que se anexo
    dest_file = open('/home/haysoos/workspace/Tesis/src/test.png', 'w')

```

Listado 4.9: Código anexo a la función `main` del programa *benchmark* para abrir un nuevo archivo destino.

al diseño modular del programa. El bloque del demodulador que se monitoreó fue el del sincronizador de fase (lazo de costas) antes del decodificador diferencial. La conexión se puede modificar para monitorear cualquier parte del grafo.

4.7. Transmisión de un archivo

Para transmitir datos de cualquier archivo es necesario utilizar la opción `--from-file` del programa *benchmark*. El programa no hace ninguna distinción del tipo de archivo que se debe utilizar, para la aplicación lo que importa son los bits contenidos en el archivo. Esto quiere decir que el archivo puede ser una imagen, un archivo de audio mp3, texto, etc. El programa original puede realizar este tipo de transmisión pero como se ilustró en la figura 4–11b solo despliega si el paquete enviado se decodificó o no. Para poder escribir los datos del paquete a un archivo destino fue necesario modificar la función que despliega las líneas en la consola del receptor.

Primeramente se inició por crear un archivo nuevo cada que se ejecuta la función `main()` como se muestra en el listado 4.9.

El programa inicia creando un nuevo archivo en modo escritura como parte de la inicialización. La segunda modificación se realizó en la función `rx_callback`. Esta función es especial ya que el grafo es quien la ejecuta de manera automática cada vez que llega un paquete. Su función es desplegar los resultados de la demodulación. La función recibe como entrada dos parámetros: una bandera indicando si el paquete es válido y la información del paquete. Por defecto el código únicamente trabaja con la

```

def rx_callback(ok, payload):
    global n_rcvd, n_right
    (pktno,) = struct.unpack('!H', payload[0:2])
    n_rcvd += 1
    if ok:
        n_right += 1
        dest_file.write(payload[2:]) #Codigo que se anexo
    else:
        print 'Dropped packet', payload[2:]

    print "ok = %5s  pktno = %4d  n_rcvd = %4d  n_right = %4d" % (
        ok, pktno, n_rcvd, n_right)

```

Listado 4.10: Función que recibe los datos demodulados y los escribe a un archivo.

```

if __name__ == '__main__':
    global dest_file
    try:
        main()
        dest_file.flush() #Codigo anexo
        dest_file.close()
    except KeyboardInterrupt:
        dest_file.flush() #Codigo anexo
        dest_file.close()
    pass

```

Listado 4.11: Código anexo al final del programa *benchmark* para cerrar el archivo destino.

bandera y los dos primeros bytes del paquete ya que ahí está contenido el número de paquete. El resto de los bytes contienen lo que se transmitió. Para poder escribir esta información al archivo destino se anexó código para que escriba los bytes a partir del segundo en adelante al archivo que se inicializó. La modificación se muestra en el listado 4.10.

Por último se agregó código después de la función *main*, antes de que termine el programa, para que cierre el archivo destino y los datos no se pierdan. Esto se muestra en el listado 4.11.

Durante el experimento se observó que el último paquete nunca llegaba por lo cual la transmisión siempre se estaba cortando. Esto fue debido a que el transmisor se cierra completamente cuando envía el último paquete al USRP. Este último paquete

```

while n < nbytes:
    if options.from_file is None:
        data = (pkt_size - 2) * chr(pktno & 0xff)
    else:
        data = source_file.read(pkt_size - 2)
        if data == '':
            break;

    payload = struct.pack('!H', pktno & 0xffff) + data
    send_pkt(payload)
    n += len(payload)
    sys.stderr.write('.')
    if options.discontinuous and pktno % 5 == 4:
        time.sleep(1)
    pktno += 1

#send_pkt eof=True) #No enviar mensaje de EOF

tb.wait()

```

Listado 4.12: Corrección al código TX para enviar el último paquete.

se queda en el USRP y no se logra transmitir ya que el control del dispositivo se perdió. Para corregir este comportamiento fue necesario realizar un cambio al código del transmisor para que no se cierre. Este cambio se muestra en el listado 4.12.

La modificación consistió en comentar la línea que envía el mensaje EOF (End Of File) al grafo. Este mensaje causa que todos los bloques terminen su ejecución después de terminar de procesar el último byte que se encuentra en sus entradas. Esta modificación tiene el efecto de que el grafo nunca terminará de ejecutarse aun después de haber enviado todos los datos al USRP. El usuario tiene que cerrar el programa manualmente.

Una observación importante es que los programas están diseñados como ejemplos ilustrativos de las capacidades de *GNURadio* y por esta razón no ofrecen alguna manera de saber si el paquete logró llegar o no. Si un paquete no llega bien el archivo no podrá escribirse correctamente. Esto se observó en la imagen que se transmitió al observar que el archivo estaba corrompido después de que terminó la transmisión

debido a que algunos paquetes no llegaban. La constelación observada de esta transmisión a 30kbits/s se muestra en la figura 4–12. Esta velocidad se utilizó para lograr una transmisión confiable y poder observar la imagen correctamente. También permitió utilizar los valores por defecto de los filtros polifásicos ya que a velocidades más altas es necesario optimizarlos.

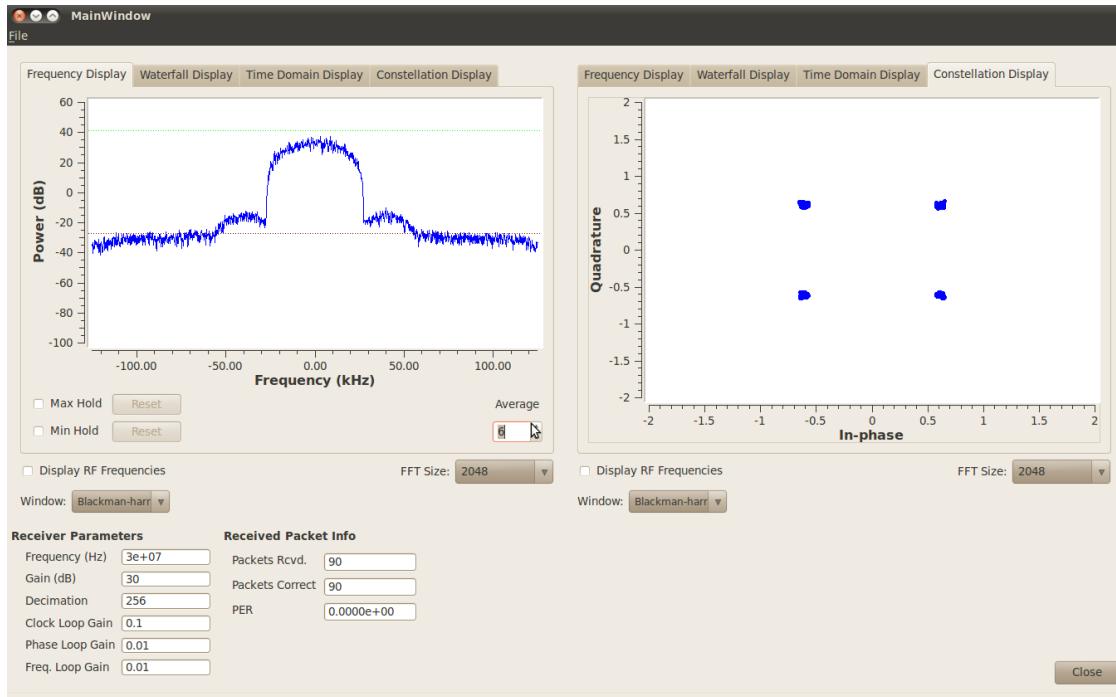


Figura 4–12: Constelación observada al transmitir un archivo a 30kbit/s.

Capítulo 5

CONCLUSIONES

Los experimentos realizados durante el desarrollo de este trabajo fueron satisfactorios e ilustraron cómo utilizar los bloques de procesamiento de *GNURadio* para transmitir datos utilizando un esquema de modulación de M estados, en este caso QPSK donde $M = 4$. Se mostró una aplicación que se puede utilizar como base para la implementación de los diferentes conceptos de telecomunicaciones como son la creación de paquetes de datos, desarrollo de protocolos de comunicación, implementación de otros esquemas de modulación (BPSK, GMSK, etc.), filtrado, etc. Por último se mostró una herramienta indispensable para la ayuda del entendimiento de los conceptos de telecomunicaciones y como apoyo para el desarrollo de aplicaciones, que es *GNURadio Companion*.

GNURadio es una plataforma flexible para el desarrollo de aplicaciones en el área de telecomunicaciones y ya que es un proyecto abierto, es constantemente desarrollado y actualizado por una comunidad activa. El foro de discusión que se encuentra en la página de *GNURadio* ([GNURadio, 2009](#)) fue de gran apoyo para el desarrollo de este trabajo debido a que la gente que se encuentra participando activamente ayuda mucho a los nuevos usuarios. Se recomienda que se utilice este recurso para cualquier persona que desee utilizar *GNURadio*.

En conclusión, *GNURadio* y el USRP son herramientas flexibles y de bajo costo que se pueden utilizar en un ambiente académico para acelerar y mejorar la comprensión de los conceptos de telecomunicaciones, ya sea a nivel licenciatura para el entendimiento de los conceptos básicos o a nivel maestría para la investigación

de conceptos más avanzados. La plataforma es capaz de ajustarse a las diferentes necesidades de los alumnos e investigadores, gracias a la amplia gama de frecuencias que soporta y su flexibilidad de implementar e incorporar las diferentes técnicas de modulación, codificación, filtrado, etc., en una plataforma que permite observar resultados reales y en tiempo real. Esto es de gran importancia para su entendimiento y comparación con la teoría que forma parte del área de telecomunicaciones.

5.1. Trabajo futuro

Utilizando los conocimientos que se presentaron en este trabajo, se puede trabajar en mejorar el código fuente con aportaciones que implementen técnicas que aun no se encuentren integradas en *GNURadio*. Un trabajo interesante es la implementación del esquema QAM. Actualmente el código fuente contiene un modulador sencillo de QAM pero el demodulador solamente contiene el esqueleto de un bloque jerárquico sin ningún funcionamiento. Los autores aun no lo han implementado pero dejaron la estructura del bloque lista para que alguien pueda desarrollarlo.

Utilizando las tarjetas auxiliares mencionadas en la tabla 3-1 es posible realizar trabajos más avanzados en diferentes rangos de frecuencias. Algunas aplicaciones en las que se puede trabajar a futuro pueden ser las siguientes:

- GPS
- Comunicación satelital
- Comunicaciones móviles
- Técnicas de acceso como TDMA, FDMA y CDMA.

APÉNDICES

Apéndice A

INSTALACIÓN DEL AMBIENTE DE DESARROLLO

El desarrollo de aplicaciones basadas en *GNURadio* en el ambiente Linux es más sencillo y más versátil ya que la mayoría de las librerías externas de las que depende esta herramienta fueron desarrolladas para Linux. Aunque *GNURadio* proporciona soporte parcial/total para otros sistemas operativos como *Windows* y *Mac OSX*, el experimento se realizó en el sistema operativo *Ubuntu*, que es una distribución del sistema operativo Linux basada en *Debian*. Se optó por esta versión de Linux ya que es amigable y fácil de usar pero principalmente se escogió por su habilidad de poder ser instalado dentro de *Windows* como una aplicación normal. Esto evita tener que hacer una partición separada para instalar el sistema operativo. Si se desea quitar es solo cuestión de ir al panel de control de *Windows* y desinstalar *Ubuntu*. Estará en la lista de aplicaciones instaladas.

Para llevar a cabo este procedimiento se utilizó un programa especial llamado *Wubi* ([Russo, 2007](#)). Esta aplicación instala el sistema operativo dentro de un solo archivo y agrega una nueva opción al menú de arranque de la PC para seleccionarlo aparte del sistema operativo primario. Cuando se ejecuta el programa *Wubi*, se deben especificar las opciones que se muestran en la figura [A-1](#). Se puede dejar el tamaño del archivo generado en su opción por defecto pero se recomienda que mínimo se le asigne 10 gigabytes para tener suficiente espacio para la instalación de librerías. Se debe especificar un nombre de usuario y la clave para esta cuenta ya que será la cuenta principal donde se realizará todo el trabajo.

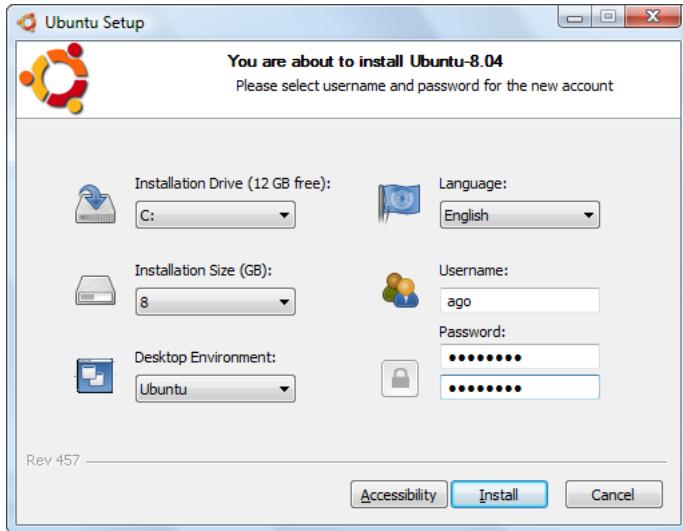


Figura A–1: Opciones del instalador *Wubi*

Una vez que se halla especificado las opciones anteriores se puede iniciar con la instalación. *Wubi* comenzará a descargar la imagen del DVD correspondiente a la distribución de *Ubuntu* que se especificó.

Cabe mencionar que la instalación puede ser de 32 o 64 bits, dependiendo del tipo de instalación de *Windows* en que se instalará *Ubuntu*. Si el sistema operativo principal es de 32 bits entonces la instalación también será de 32 bits. Esto se debe tener presente cuando se inicie la instalación de *GNURadio* ya que las librerías de las cuales depende vienen en versiones de 32 y 64 bits.

La imagen mide aproximadamente 700MB. Cuando termine la descarga la aplicación comenzará a crear el archivo del sistema operativo y agregará la opción para seleccionarlo en el menú de arranque como se muestra en la figura A–2. Una vez terminado este procedimiento la aplicación le pedirá al usuario que reinicie la PC. Se debe seleccionar la nueva opción del menú de arranque para ingresar al sistema operativo *Ubuntu*. Al terminar se le presentará al usuario que ingrese el nombre de la cuenta y la clave. Esta es la que se especificó en el instalador *Wubi*. Al ingresar se presentará el escritorio *GNOME* (o *KDE* si se especificó otra distribución como *Kubuntu*) como se muestra en la figura A–3.

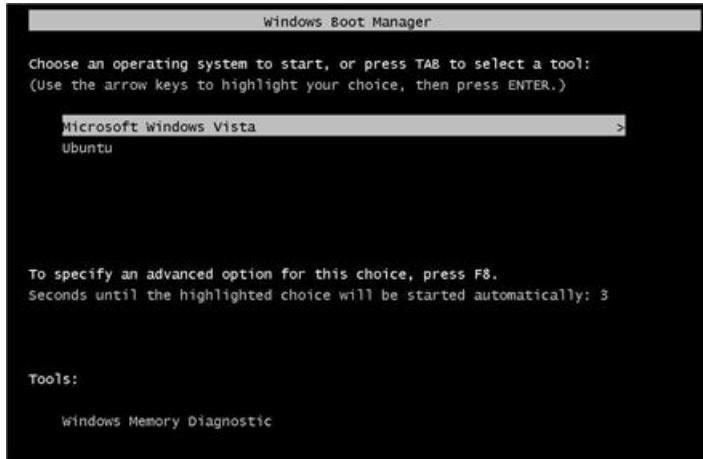


Figura A–2: Menú de arranque con la opción de *Ubuntu*

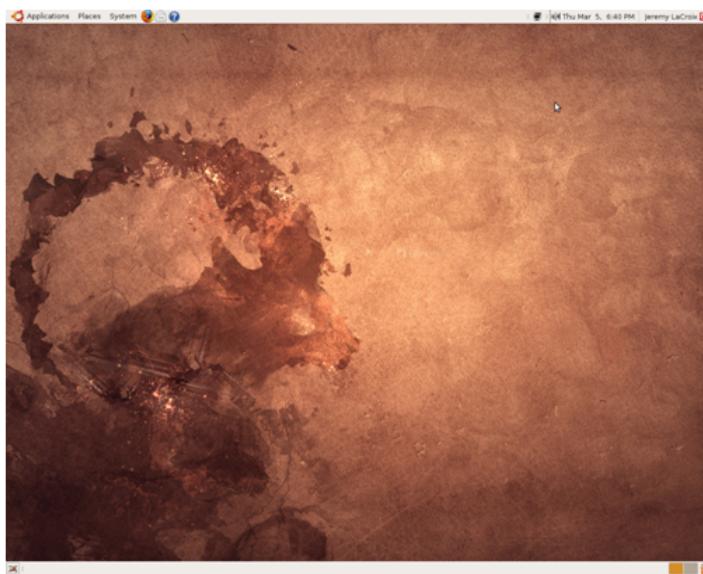


Figura A–3: Escritorio *GNOME* de la distribución *Ubuntu*

El siguiente paso es instalar *GNURadio* y sus dependencias. Todas las dependencias se pueden conseguir utilizando el sistema de instalación de software de *Ubuntu*. Utilizando el sistema Synaptic del sistema operativo se deben instalar las dependencias que se especifican en la tabla A-1.

La única librería que se recomienda que instalar manualmente es *Boost* ya que la más reciente es la versiún 1.43 y ofrece muchas mejoras contra la versión 1.38 de Ubuntu. Para instalarla es cuestión de bajar el código fuente de la página www.Boost.org y seguir los siguientes pasos:

1. Descargar el código fuente a un lugar adecuado, abrir una consola e ir al directorio donde se descargó el archivo.

2. Descomprimir el archivo con el siguiente comando:

```
tar xfz boost_1_43_0.tar.gz
```

3. Entrar al directorio nuevo que se creó al descomprimir el archivo y ejecutar el siguiente comando: ./bootstrap.sh

4. Ejecutar el siguiente comando: sudo ./bjam install

Se debe proporcionar la clave de la cuenta de administrador ya que la instalación requiere privilegios elevados.

La compilación e instalación puede demorarse entre 20 y 30 minutos dependiendo de la velocidad de la PC. Este procedimiento compilará e instalará todas las librerías que forman parte del conjunto de *Boost*. Este proceso se puede controlar detalladamente para que únicamente instale un subconjunto de librerías y así reducir el tiempo de instalación ya que *GNURadio* no utiliza el conjunto completo.

Después de instalar *Boost* se necesita utilizar el sistema *Synaptic* para instalar el resto de las dependencias como se mencionó anteriormente. Esta herramienta se encuentra en el menú *System*→*Administration*→*Synaptic Package Manager* y se muestra en la figura A-4. En la caja de texto *search* se puede escribir el nombre de las dependencias.

Tipo	Elementos
Herramientas de desarrollo que se requieren para compilar el código fuente.	<ul style="list-style-type: none"> ■ g++ ■ subversion ■ make ■ autoconf, automake, libtool ■ sdcc ■ guile ■ ccache ■ SWIG 1.3.31 o mayor ■ git
Librerías requeridas para compilación y funcionamiento de los ejecutables.	<ul style="list-style-type: none"> ■ python-dev ■ FFTW 3.X (fftw3, fftw3-dev) ■ CppUnit (libcppunit y libcppunit-dev) ■ Boost 1.35 o mayor ■ libusb y libusb-dev ■ wxWidgets (wx-common y python-wxgtk) ■ python-numpy ■ ALSA (alsa-base, libasound2, libasound2-dev) ■ Qt4 ■ SDL (libsdl-dev) ■ GSL GNU Scientific Library (libgsl0-dev) 1.10 o mayor
Elementos opcionales	<ul style="list-style-type: none"> ■ QWT 5.0 o mayor ■ QWT Plot3d para QT4 ■ Python-scipy, python-matplotlib, python-tk ■ Doxygen ■ Octave

Tabla A–1: Dependencias requeridas para la instalación de *GNURadio*

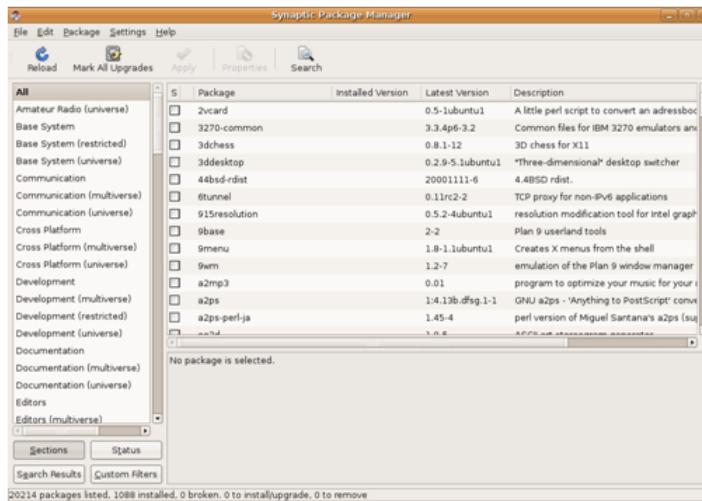


Figura A-4: Administrador de software Synaptic

Para llevar a cabo este procedimiento se deben seguir los siguientes pasos:

1. Abrir una consola e ir a un directorio adecuado para descargar el código fuente.

2. Ejecutar el siguiente comando para obtener el código fuente del repositorio git:

```
git clone http://gnuradio.org/git/gnuradio.git. Esto creará un nuevo directorio llamado gnuradio.
```

3. Entrar al nuevo directorio y ejecutar el siguiente comando:

```
./bootstrap
```

4. Ejecutar el siguiente comando para configurar la instalación:

```
./configure
```

5. Iniciar la compilación con el siguiente comando:

```
make
```

6. Realizar una serie de pruebas para asegurar que la compilación fue exitosa:

```
make check
```

7. Instalar *GNURadio*:

```
sudo make install
```

De acuerdo a la documentación de *GNURadio*, antes de realizar las pruebas que verifican la compilación es necesario ejecutar la siguiente serie de comandos:

1. Copiar el archivo *ld.so.conf* a un directorio temporal para modificarlo:

```
cp /etc/ld.so.conf /tmp/ld.so.conf
```

2. Agregar la ruta donde están instaladas las librerías:

```
echo /usr/local/lib >> /tmp/ld.so.conf
```

3. Borrar el archivo temporal y mover el nuevo a su lugar original:

```
mv /tmp/ld.so.conf /etc/ld.so.conf
```

4. Ejecutar el siguiente comando para aplicar los cambios:

```
sudo ldconfig
```

La razón del procedimiento anterior es que *Ubuntu* utiliza una implementación diferente de la herramienta *libtool* y si la ruta de las librerías no se especifica en el archivo de configuración, las pruebas de *GNURadio* fallan con errores de que no se encuentran dichas librerías ([GNURadio, 2009](#)).

Para el uso óptimo del USRP es necesario realizar una modificación en el servicio *udev* ya que únicamente permite acceso directo del dispositivo a cuentas con privilegios de administrador. Esto quiere decir que cada que se necesite ejecutar algún programa que utilice el puerto USB es necesario elevar el ejecutable con el comando *sudo*. Para evitar esto se aplican los siguientes pasos:

1. Crear un nuevo grupo de usuarios:

```
sudo addgroup usrp
```

2. Anexar la cuenta de usuario que se está utilizando a este nuevo grupo:

```
sudo usermod -G usrp -a <CUENTA_DE_USUARIO>
```

3. Crear un archivo temporal con las instrucciones que *udev* necesita para agregar este nuevo grupo a los que están permitidos acceder al puerto USB:

```
echo 'ACTION=="add", BUS=="usb", SYSFS{idVendor}=="ffff",
```

```
SYSFS{idProduct}=="0002", GROUP=="usrp", MODE=="0660",
```

```
> tmpfile
```

4. Modificar los atributos de este archivo para que sea propiedad de root:

```
sudo chown root.root tmpfile
```

5. Mover el archivo temporal al directorio de reglas de *udev*:

```
sudo mv tmpfile /etc/udev/rules.d/10-usrp.rules
```

Es posible aplicar los cambios al ejecutar el siguiente comando:

```
sudo udevadm control--reload-rules.
```

Si esto no funciona entonces se necesita reiniciar la PC para que los cambios tengan efecto.

Para poder actualizar el código fuente con los cambios más recientes del repositorio *git*, se deben seguir los siguientes pasos:

1. Desinstalar la versión actual con el siguiente comando: `make uninstall`
2. Descargar las actualizaciones e integrarlas al código fuente que ya se tiene con el siguiente comando: `git pull`. De esta manera no es necesario tener que descargar todo el código fuente ya que únicamente se integraran las diferencias entre el código que ya se tiene y el que se encuentra en el repositorio.
3. Realizar los pasos para compilar el nuevo código:

- a) `./bootstrap`
- b) `./configure`
- c) `make`
- d) `make check`
- e) `sudo make install`

Si por el proceso de actualización marca algun error del cual no se pueda recuperar, entonces es posible corregir el repositorio local (el código fuente que se descargó) sin tener que iniciar desde cero. Para lograr esto es necesario ejecutar el siguiente comando: `git clean -d -x -f`. Este comando vuelve a establecer el repositorio a su estado original desde la última vez que se descargó de la página de *GNURadio*,

aunque se debe tomar en cuenta que si se realizaron cambios en el código fuente por parte del usuario, estos se van a perder. Una vez realizado esto se puede volver a intentar el proceso de actualización.

Apéndice B

GNURADIO COMPANION

GNURadio incluye una aplicación que facilita el desarrollo de grafos y aplicaciones sencillas de manera gráfica, muy similar al software de *Simulink* de Mathworks. La aplicación genera código en *Python* apartir del diagrama de bloques. Es posible manipular y extender el código generado si se desea utilizar alguna función que no esté incluida en *GNURadio Companion*([Blum, 2009](#)) (GRC por sus siglas en *Inglés*).

Las capacidades de la aplicación son:

- Viene incluido con el código fuente de *GNURadio*. Si se instalan todas las dependencias entonces GRC se instalará junto con *GNURadio*.
- Puede generar código en *Python* para aplicaciones que utilizan bloques gráficos con la librería WxWidgets, aplicaciones que no utilizan ningún control gráfico y bloques jerárquicos.
- Se pueden definir bloques que actuan como variables o constantes. Otros bloques pueden hacer referencia a ellas e incluir su valor en algún parámetro. Tambien es posible usar bloques variables que definen un control gráfico como barras deslizadoras, botones, etc.
- Cada bloque está definido por un archivo XML que contiene todas sus características (número de puertos, tipo de datos con los que trabaja, parámetros de entrada, etc.). Es posible crear un archivo XML para incluir algún bloque creado por el usuario.
- Los bloques pueden ser deshabilitados dentro de la gráfica sin tener que borrarlos de la aplicación. Estos bloques son ignorados por el generador de

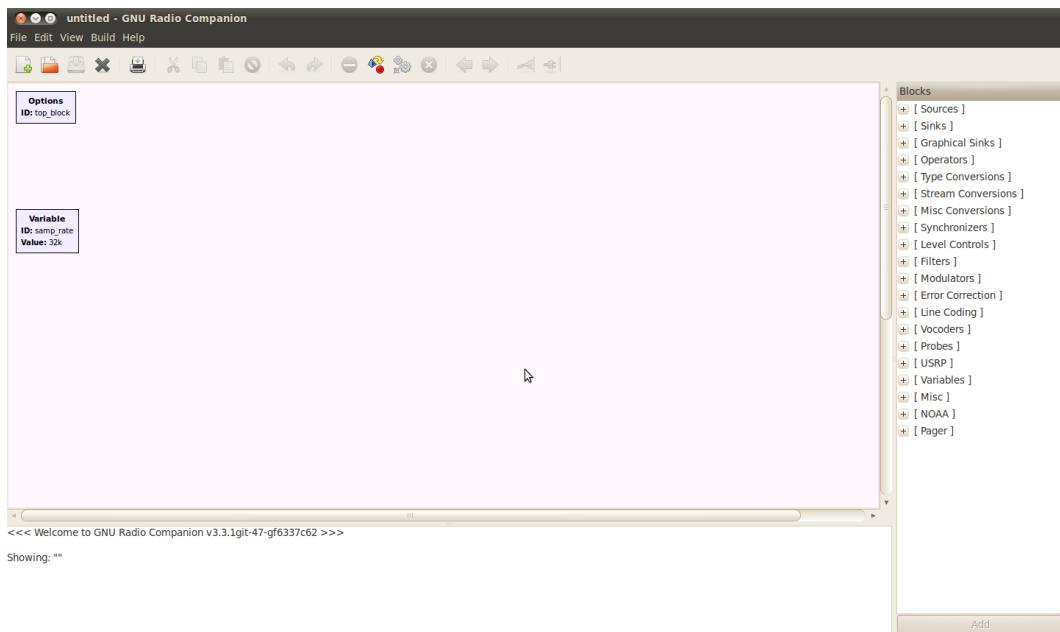


Figura B–1: Pantalla principal de *GNURadio Companion*

código, permitiendo así una manera más rápida de realizar cambios en la gráfica. También se pueden cortar, copiar y pegar bloques.

Para instalar GRC se debe tener las siguientes dependencias instaladas antes de iniciar la compilación:

- Python 2.5 o mayor (<http://www.python.org/download/>)
- Python-LXML 2.0 o mayor (<http://codespeak.net/lxml/installation.html>)
- Cheetah Template Engine 2.0 o mayor (<http://www.cheetahtemplate.org/download.html>)
- Python-GTK 2.10 o mayor (<http://www.pygtk.org/downloads.html>)

Si estas dependencias se instalan antes de iniciar con la compilación de *GNURadio* entonces GRC se instalará automáticamente.

Para iniciar la aplicación se necesita abrir una ventana de consola y escribir el comando `gnuradio-companion`. Inmediatamente aparecerá la ventana que se muestra en la figura B–1.

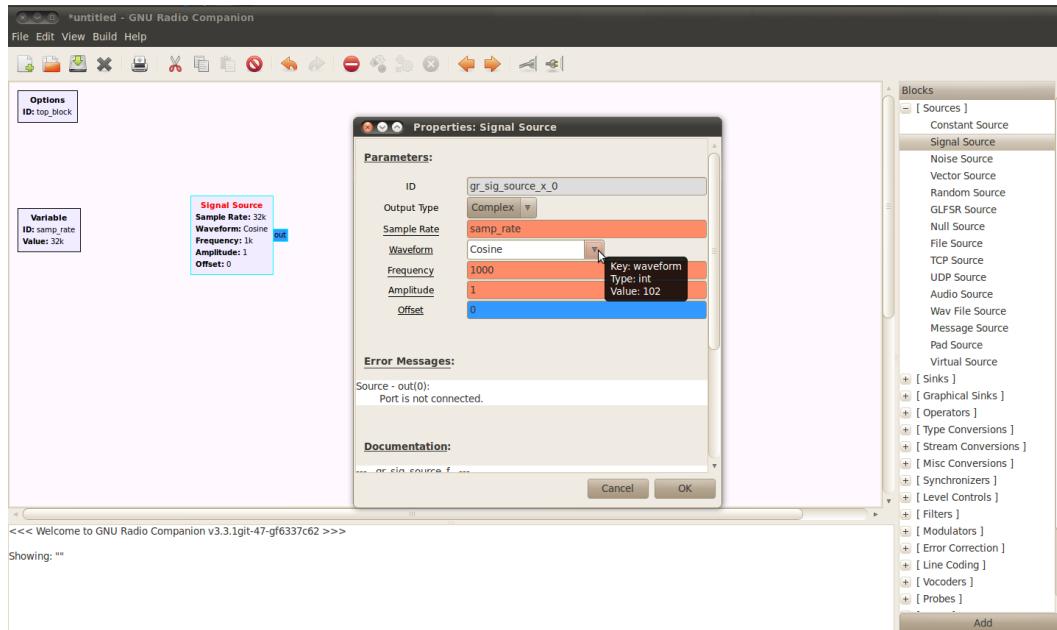


Figura B–2: Pantalla de propiedades del bloque signal_source

El área de trabajo inicia con dos bloques por default, uno que define las opciones generales del grafo y un bloque que define una variable llamada `sample_time`. Esta variable se puede modificar o borrar según las necesidades del usuario. Del lado derecho se encuentra la lista de las categorías de bloques que soporta GRC. Cada categoría se puede expandir para revelar los diferentes bloques que contiene. Para colocar bloques en el área de trabajo es necesario seleccionar el de interés y arrastrarlo con el mouse a la area de trabajo. De ahí se puede seguir arrastrando para colocarlo en el área que se deseé.

Las propiedades de los bloques se pueden obtener haciendo doble click sobre el bloque como se muestra en la figura B–2.

Todos los bloques comparten un parámetro en común y es el ID. Este parámetro indentifica al bloque para que otros puedan hacer referencia a él. Se puede aceptar el nombre por defecto que GRC sugiere o bien se puede especificar otro. Los otros parámetros son específicos del bloque con el que se está trabajando. El bloque

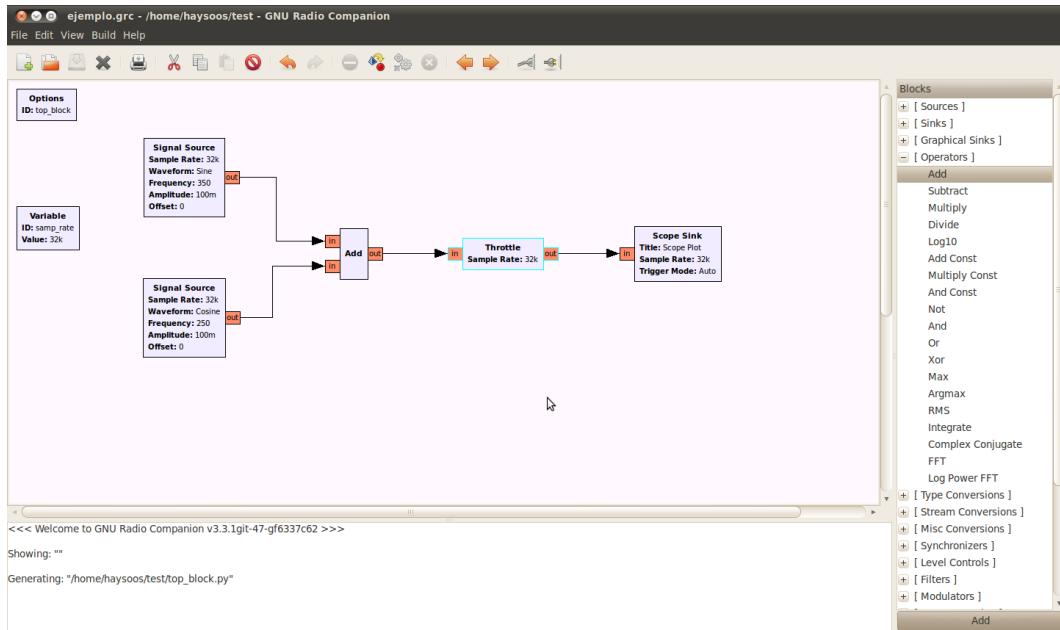


Figura B–3: Ejemplo de un grafo desarrollado en GRC

`signal_source` de la figura B–2 es un generador de señales que puede generar varios tipos de formas de onda como senoidales, cuadradas, dientes de sierra, etc. Los parámetros se describen como sigue:

Output Type Especifica el tipo de datos que utiliza para generar la forma de onda:

Real o Complejo.

Sample Rate La tasa de muestreo con la que se generará la señal. Aquí se muestra haciendo referencia al bloque que declara la variable `sample_rate`. Esta variable tiene un valor de 32k.

Waveform Aquí se define el tipo de señal que se quiere generar.

Frequency Este parámetro especifica la frecuencia de la señal.

Amplitude Este parámetro especifica la amplitud de la señal.

Offset Especifica un retraso de la señal generada.

En la figura B–3 se muestra un ejemplo completo de un grafo que suma dos señales de diferentes frecuencias y muestra el resultado en un osciloscopio.

El ejemplo muestra un requerimiento importante de los grafos que no utilizan fuentes o sinks de hardware como el USRP o la tarjeta de sonido. Los bloques que

hacen referencia a algún hardware limitan automáticamente la velocidad de ejecución del grafo a la velocidad de muestreo del hardware. Como el ejemplo de la figura B-3 no contiene este tipo de bloques es necesario incluir el bloque **Throttle** para limitar la velocidad de ejecución, de lo contrario el grafo se ejecutará a la velocidad máxima del CPU y esto causara que la PC no responda correctamente. Para este ejemplo el bloque hace referencia a la variable `sample_rate` declarada y con esto limita la velocidad de ejecución a la tasa de muestreo que se está utilizando.

El bloque **Scope Sink** es uno de los diversos visualizadores gráficos que permiten observar las señales en diversos puntos del grafo. Este bloque actua como un osciloscopio con opciones de disparo y modo XY. Otros bloques con los que cuenta GRC para visualizar señales son: analizador de espectros, espectrograma en 2D, histograma y diagrama de constelación.

Si existe un error en alguna de las conexiones, ya sea porque los tipos de datos son incompatibles entre bloques, algunas de las opciones no están correctamente configuradas o falta algún puerto por conectar, estos serán enfatizados de color rojo. Es necesario realizar las correcciones que se piden antes de generar el código y poder ejecutar el grafo.

Para ejecutar el grafo es necesario hacer click en el botón de generar código o en el de ejecutar grafo. Si no se ha grabado el trabajo GRC pedirá que se grabe antes de la ejecución como se muestra en la figura B-4. Una vez realizado esto, GRC genera un archivo con código *Python* y este es el que se ejecuta. La figura B-5 muestra el grafo de la suma de las dos señales ejecutándose y mostrando los resultados en un osciloscopio.

Conforme se vallan anexando bloques visualizadores, estos se van agrupando en la misma pantalla de forma vertical. Esto permite visualizar los resultados de una manera más eficiente sin tener que estar cambiando entre diferentes ventanas.

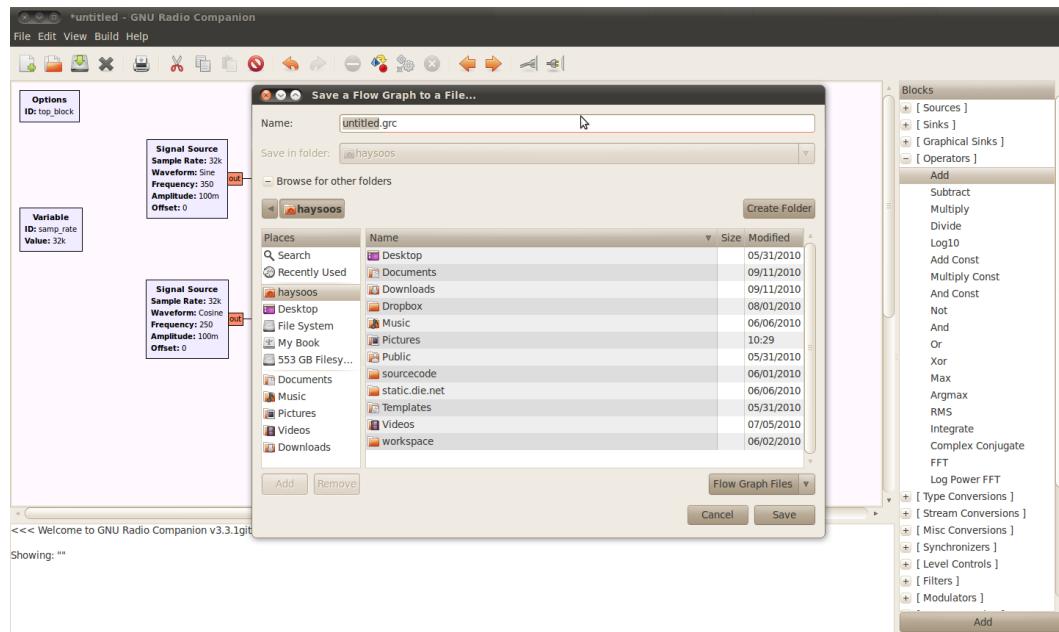


Figura B–4: Pantalla para guardar el trabajo en GRC

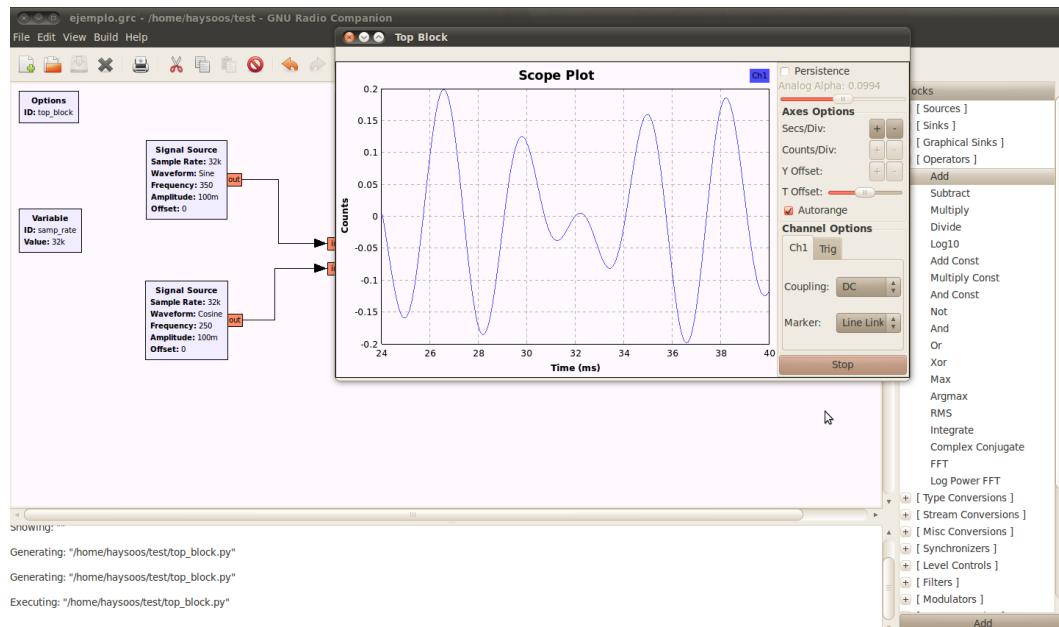


Figura B–5: Ejemplo de un grafo en ejecución

Agregando el bloque FFT al ejemplo obtenemos la pantalla que se muestra en la figura B-6.

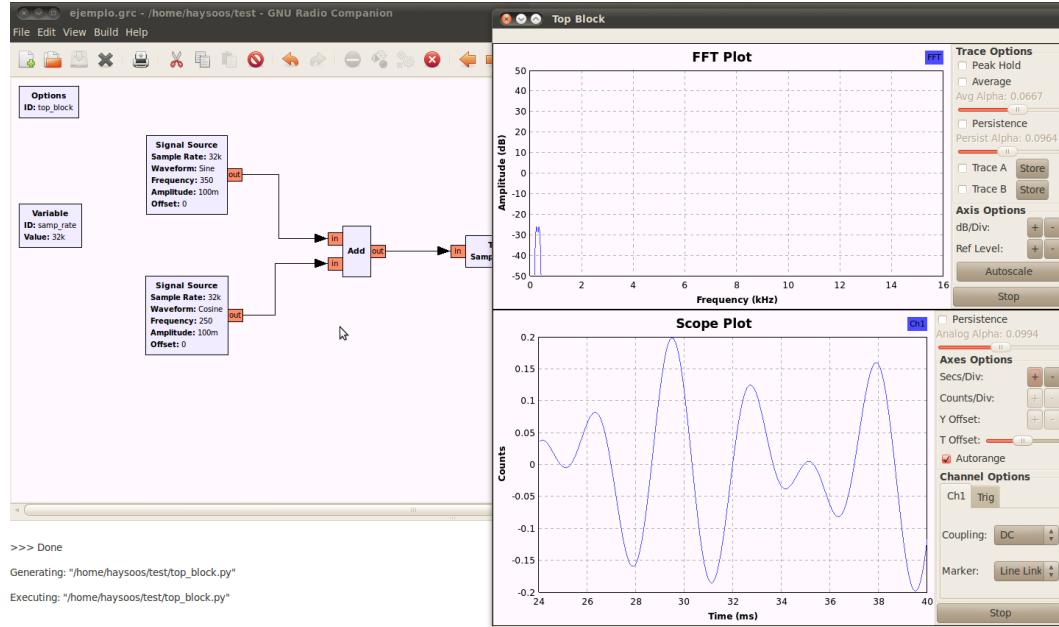


Figura B-6: Bloque FFT en conjunto con el osciloscopio en GRC

Como se mencionó al inicio de este apéndice, una de las habilidades de GRC es poder especificar bloques que implementan algún tipo de control gráfico como barras deslizadoras. Estos controles se anexan automáticamente junto con los otros controles gráficos y actúan como un bloque con algún valor variable que puede modificar cualquier parámetro del grafo. Para ilustrar esto al ejemplo se anexa un bloque de este tipo con el nombre de amplitud. Al bloque se le especifica un valor por defecto, valor máximo y mínimo y el tipo de dato con el que trabaja. Asignando esta variable a los campos de amplitud de los dos bloques **Signal Source** se puede modificar la amplitud de ambos moviendo la barra deslizadora que aparece en la ventana de visualización como se muestra en la figura B-7.

Todo el trabajo realizado dentro de GRC se guarda dentro de una archivo con extensión **.grc** y contiene la estructura del diagrama realizado en formato XML. Una porción del archivo generado por el ejemplo de este apéndice se muestra en el listado B.1.

```

<?xml version='1.0' encoding='ASCII'?>
<flow_graph>
  <timestamp>Sat Oct 23 11:19:29 2010</timestamp>
  <block>
    <key>variable</key>
    <param>
      <key>id</key>
      <value>samp_rate</value>
    </param>
    <param>
      <key>_enabled</key>
      <value>True</value>
    </param>
    <param>
      <key>value</key>
      <value>32000</value>
    </param>
    <param>
      <key>_coordinate</key>
      <value>(10, 170)</value>
    </param>
    <param>
      <key>_rotation</key>
      <value>0</value>
    </param>
  </block>
  .
  .
  .
  <connection>
    <source_block_id>gr_sig_source_x_0</source_block_id>
    <sink_block_id>gr_add_xx_0</sink_block_id>
    <source_key>0</source_key>
    <sink_key>0</sink_key>
  </connection>
</flow_graph>

```

Listado B.1: Código XML de un proyecto de GRC

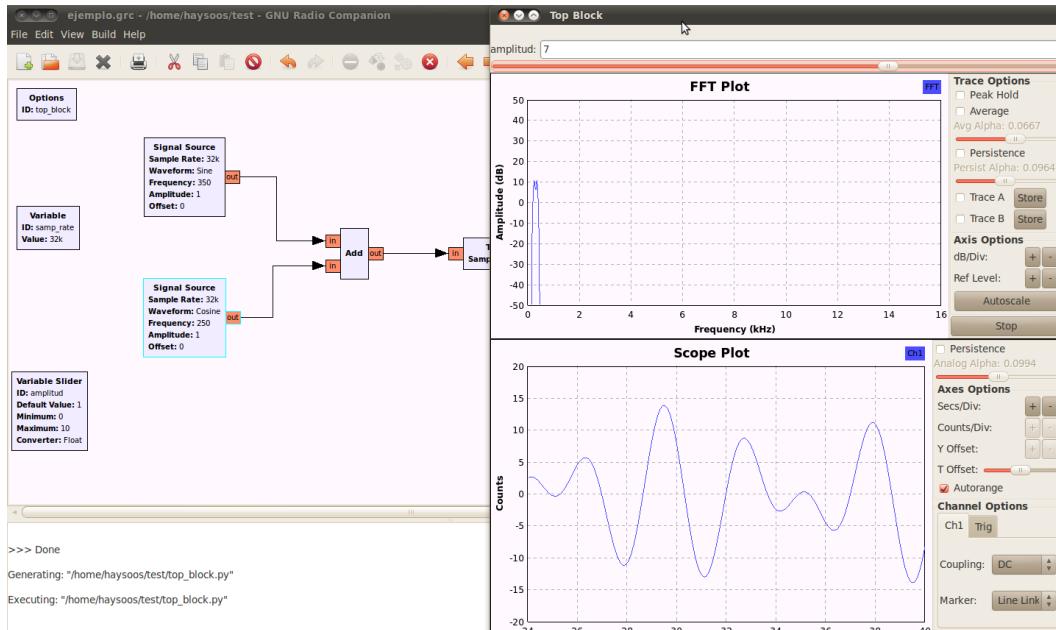


Figura B-7: Modificación de parámetros por medio de bloques variables en GRC

Cada uno de los bloques que forman parte del grafo se definen entre las llaves `<block>` `</block>` y dentro de estas llaves se definen los parámetros del bloque en cuestión. Las conexiones se definen entre las llaves `<connection>` `</connection>`. Cuando se ejecuta el programa, GRC traduce el archivo XML en un archivo *Python* que lleva a cabo el funcionamiento del grafo que se definió. Su estructura se basa en un esqueleto pre-definido para todos los programas que genera, ya sea si son gráficos o por medio de la consola. El programa *Python* generado se muestra en el listado B.2.

El programa generado se puede modificar manualmente y expandirse sin utilizar GRC pero el usuario debe tener en consideración de que si se vuelve a generar a través de GRC los cambios que se realizaron a mano se perderán.

El ejemplo ilustrado en este apéndice es un solo fragmento de las capacidades de *GNURadio Companion*. Los bloques de los generadores de señales se pueden reemplazar por bloques que encapsulen algún hardware como tarjetas de audio, el USRP o incluso tarjetas de adquisición de datos, aunque para estas es necesario crear el bloque desde cero y proporcionar los drivers del hardware adecuados. También

```

#!/usr/bin/env python
#####
# Gnuradio Python Flow Graph
# Title: Ejemplo
# Author: Jesus Espinoza Hernandez
# Description: Suma de dos senales
# Generated: Sat Oct 23 11:19:35 2010
#####

from gnuradio import eng_notation
from gnuradio import gr
from gnuradio import window
from gnuradio.eng_option import eng_option
from gnuradio.gr import firdes
from gnuradio.wxgui import fftsink2
from gnuradio.wxgui import scopesink2
from grc_gnuradio import wxgui as grc_wxgui
from optparse import OptionParser
import wx

class top_block(grc_wxgui.top_block_gui):

    def __init__(self):
        grc_wxgui.top_block_gui.__init__(self, title="Ejemplo")
        _icon_path = "/usr/share/icons/hicolor/32x32/apps/
                      gnuradio-grc.png"
        self.SetIcon(wx.Icon(_icon_path, wx.BITMAP_TYPE_ANY))

        #####
        # Variables
        #####
        self.samp_rate = samp_rate = 32000

        #####
        # Blocks
        #####
        self.gr_add_xx_0 = gr.add_vff(1)
        self.gr_sig_source_x_0 = gr.sig_source_f(samp_rate, gr.
                                                GR_COS_WAVE, 1000, 1, 0)
        self.gr_sig_source_x_1 = gr.sig_source_f(samp_rate, gr.
                                                GR_COS_WAVE, 2000, 1, 0)
        self.gr_throttle_0 = gr.throttle(gr.sizeof_float*1,
                                         samp_rate)

```

```

        self.wxgui_fftsink2_0 = fftsink2.fft_sink_f(
            self.GetWin(),
            baseband_freq=0,
            y_per_div=10,
            y_divs=10,
            ref_level=50,
            ref_scale=2.0,
            sample_rate=samp_rate,
            fft_size=1024,
            fft_rate=30,
            average=True,
            avg_alpha=None,
            title="Espectro",
            peak_hold=False,
        )
        self.Add(self.wxgui_fftsink2_0.win)
        self.wxgui_scopesink2_0 = scopesink2.scope_sink_f(
            self.GetWin(),
            title="Senal Original",
            sample_rate=samp_rate,
            v_scale=0,
            v_offset=0,
            t_scale=0,
            ac_couple=False,
            xy_mode=False,
            num_inputs=1,
            trig_mode=gr.gr_TRIGGER_MODE_AUTO,
        )
        self.Add(self.wxgui_scopesink2_0.win)

#####
# Connections
#####
self.connect((self.gr_sig_source_x_0, 0), (self.
    gr_add_xx_0, 0))
self.connect((self.gr_sig_source_x_1, 0), (self.
    gr_add_xx_0, 1))
self.connect((self.gr_add_xx_0, 0), (self.gr_throttle_0
    , 0))
self.connect((self.gr_throttle_0, 0), (self.
    wxgui_scopesink2_0, 0))
self.connect((self.gr_throttle_0, 0), (self.
    wxgui_fftsink2_0, 0))

```

es posible trabajar con varios filtros como pasa bajas, hilbert y ventanas. GRC en sí actúa como una interfaz gráfica a los bloques de *GNURadio* facilitando así su uso de una manera más intuitiva. Esto permite una mejor interacción con el software para aplicaciones académicas donde la experiencia con programación no es necesaria.

```

def set_samp_rate(self, samp_rate):
    self.samp_rate = samp_rate
    self.gr_sig_source_x_0.set_sampling_freq(self.samp_rate
        )
    self.wxgui_fftsink2_0.set_sample_rate(self.samp_rate)
    self.gr_sig_source_x_1.set_sampling_freq(self.samp_rate
        )
    self.wxgui_scopesink2_0.set_sample_rate(self.samp_rate)

if __name__ == '__main__':
    parser = OptionParser(option_class=eng_option, usage="%prog: ["
        options]")
    (options, args) = parser.parse_args()
    tb = top_block()
    tb.Run(True)

```

Listado B.2: Código *Python* generado por GRC

Referencias

- Abendroth, S. (2003).
En *A software defined radio front-end implementation*. IEEE DSPEnabled Radio Conference.
- Alfke, P. (1996). *Efficient shift registers, lfsr counters, and long pseudo-random sequence generators* (Artículo Técnico n.º XAPP 052). Xilinx.
- Wireless Networking Communications Group. (2007). *Hydra - a mimo-ofdm multi-hop testbed*. Descargado de <http://hydra.ece.utexas.edu/>
- Blum, J. (2009). *Gnuradio companion*. Descargado de <http://gnuradio.org/redmine/wiki/gnuradio/GNURadioCompanion>
- Breed, G. (2005). *Analysing signals using the eye diagram* (Artículo Técnico). Summit Technical Media.
- Donadio, M. P. (2000). *Cic filter introduction* (Artículo Técnico). IEEE.
- Engdahl, T. (2009). *Coaxial cables*. Descargado de <http://www.epanorama.net/documents/wiring/coaxcable.html>
- Farhang, B. (2010). *Signal processing techniques for software radios* (2.^a ed.). Lulu Publishing House.
- Foster, G. (2004). *Anatomy of an eye diagram* (Artículo Técnico). SyntheSys Reseach Inc.
- GNURadio. (2009). *Gnuradio wiki*. Descargado de <http://gnuradio.org/redmine/wiki/gnuradio>
- Lee, E. A. (2003). *Digital communication*. Kluwer Academic Publishers.
- Litwin, L. (2011). *Matched filtering and timing recovery in digital receivers* (Artículo Técnico). RF Design.

- LLC, E. R. (2009). *Ettus research llc*. Descargado de <http://www.ettus.com/>
- Lyrtech. (2009). *Small form factor sdr development platforms*. Descargado de <http://www.lyrtech.com/Products/Families/sdr.php?m=2>
- McHale, J. (2004). *Software-defined radio and jtrs*. Descargado de <http://www.militaryaerospace.com/index/display/article-display/217738/articles/military-aerospace-electronics/volume-15/issue-12/features/special-report/software-defined-radio-and-jtrs.html>
- Meyr, H. (1998). *Digital communication receivers*. John Wiley and Sons.
- Mitola, J. (1992). *The software radio* (Artículo Técnico). IEEE National Telesystems Conference.
- Nguyen, T. Q. (2000). *A trick for the design of fir half-band filters* (Artículo Técnico). IEEE.
- Pentek Inc. (2009). *Pentek inc - software defined radio*. Descargado de <http://www.pentek.com/sftradcentral/SFTRadCntrlPrd.cfm#Overview>
- Pérez, E. H. (2004). *Comunicaciones ii: Comunicación digital y ruido*. Limusa S.A de C.V.
- Python Software Foundation. (2010). *Python programming language - official web-site*. Descargado de <http://www.python.org>
- Russo, A. (2007). *Wubi - ubuntu installer for Windows*. Descargado de <http://wubi-installer.org/>
- SciPy. (2010). *Scipy*. Descargado de <http://www.scipy.org>
- Sklar, B. (2001). *Digital communications fundamentals and applications*. Prentice Hall.
- Texas Instruments. (2008). *Block diagram (sbd) - software defined radio - ti.com*. Descargado de <http://focus.ti.com/docs/solution/folders/print/357.html>

Universal Implementers Forum. (2000). *Universal serial bus 2.0 specification* (Especificación Técnica). Universal Implementers Forum.