UNIVERSIDAD NACIONAL DE TUCUMÁN

FACULTAD DE CIENCIAS EXACTAS Y TECNOLOGÍA

Laboratorio de Telecomunicaciones

Sistemas de Comunicaciones Digitales I (15_ESC)

Proyecto Integrador

Docente: Dr. Ing. Bonomi Miranda, Fernando Alberto

Alumnos: Arnedo, Emiliano

Diaz, Leonardo Leandro

Fecha: 08/07/24

Introducción

El proyecto consiste en el diseño y la implementación de un sistema de transmisión y recepción de audio, particularmente optimizado para la transmisión de voz, utilizando la herramienta de software libre *GNU Radio*. El proyecto aborda cada etapa del proceso de comunicación, desde la codificación de la fuente pasando por la modulación hasta la demodulación terminando con la recuperación de la señal, siempre con el objetivo de proporcionar una comunicación eficiente y robusta.

Se presentará un diagrama de bloques completo del sistema que servirá como guía de las diferentes etapas y los distintos procesamientos por los que pasa la señal hasta su correcta recepción. A continuación, se profundizará en cada una de estas describiendo los bloques utilizados, sus correspondientes configuraciones y mostrando las respuestas obtenidas a su salida. Finalmente, se sacarán conclusiones sobre los resultados obtenidos.

Diseño

Se partió de un diagrama de bloques completo del sistema, que se puede observar en la figura 1. Este diagrama muestra cada una de las etapas clave del proceso de comunicación, además de proporcionar una visión clara de cómo se implementa y opera el sistema, asegurando una comunicación fiable y de buena calidad.

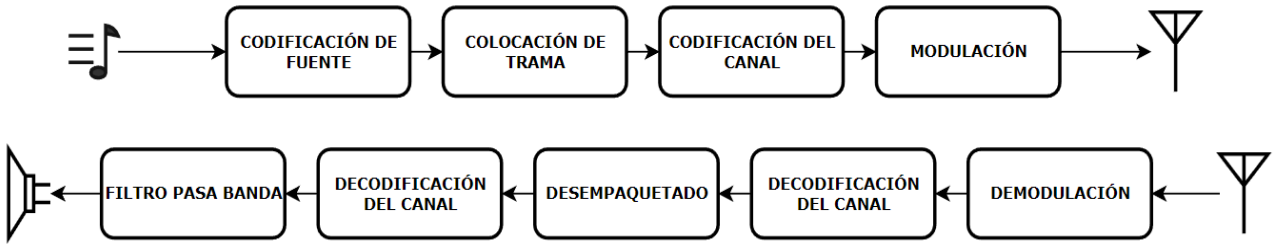

Figura 1: Diagrama de bloques del transmisor y receptor

TRANSMISOR

Codificación de fuente

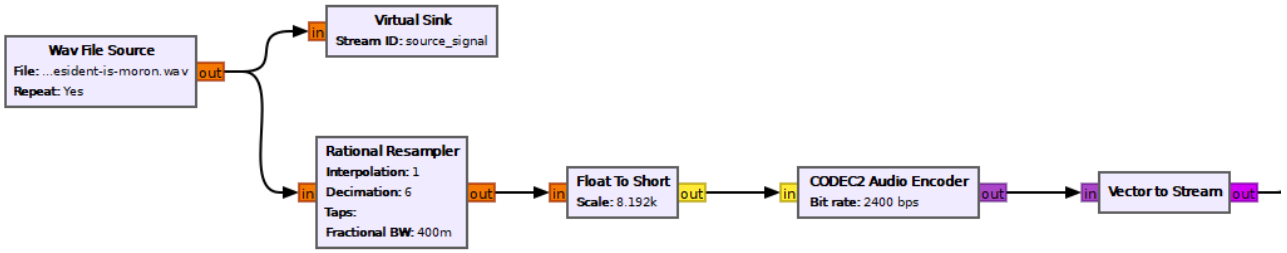

Figura 2: Diagrama de bloques del codificador de fuente

Se comienza con el bloque **Wav File Source** que es utilizado para leer archivos de audio en formato WAV (sin compresión) y convertirlos en una secuencia de muestras que pueden ser procesadas en el flujo de datos. Su salida es una trama de datos en formato flotante (floats), donde cada muestra representa la amplitud del sonido en ese punto específico del tiempo (ver Figura 3).

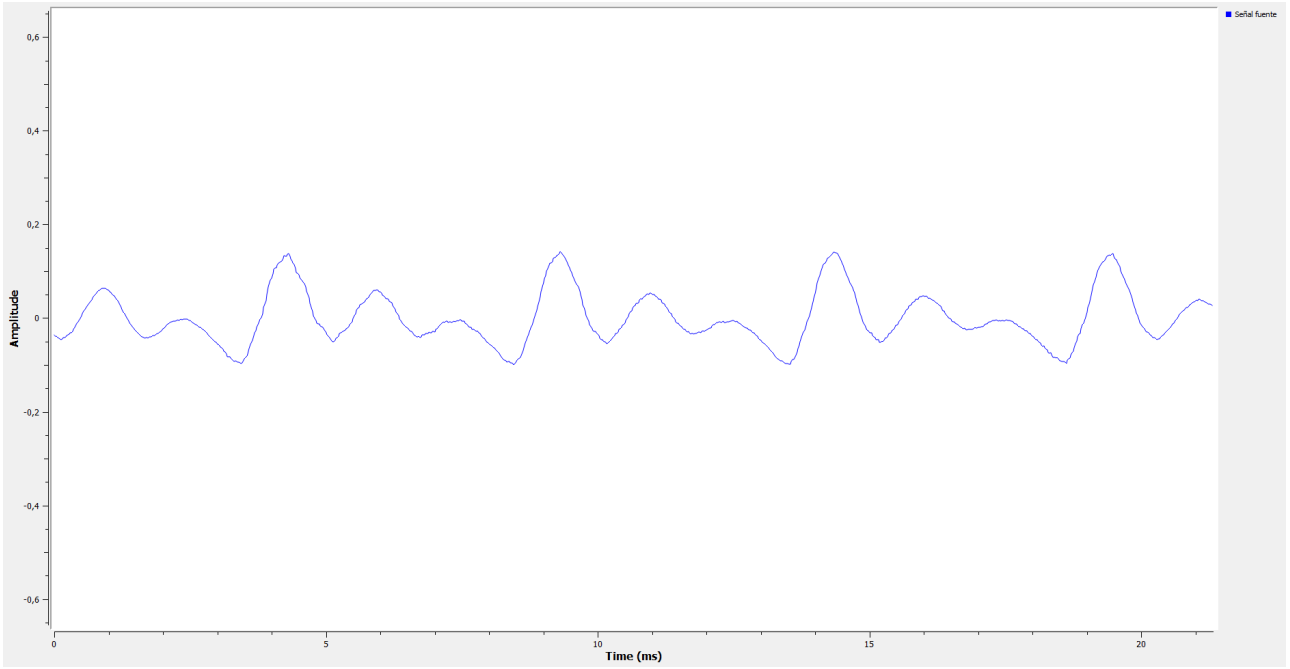

Figura 3: Señal fuente proveniente de una voz humana en formato Wav.

El bloque **Rational Resampler** es utilizado para cambiar la tasa de muestreo de la señal mediante interpolación y decimación. Este bloque es crucial para adaptar la tasa de muestreo de la señal de entrada a los requerimientos del **CODEC2 Audio Encoder**. Está configurado con una interpolación de 1 y una decimación de 6. Esto significa que la tasa de muestreo de la señal se reduce por un factor de 6. La tasa de muestreo original del archivo Wav es de 48 kHz, por lo tanto,

la salida del *resampler* será de 8 kHz (48000 / 6). El módulo **CODEC2 Audio Encoder** es un codificador de voz, con pérdida, de baja tasa de bits utilizado comúnmente para comunicaciones de voz en radioaficionados y otras aplicaciones donde el ancho de banda es limitado. Este, entrega la señal de audio codificada a una tasa de 2400 bps, lista para ser encapsulada en paquetes (figura 4).

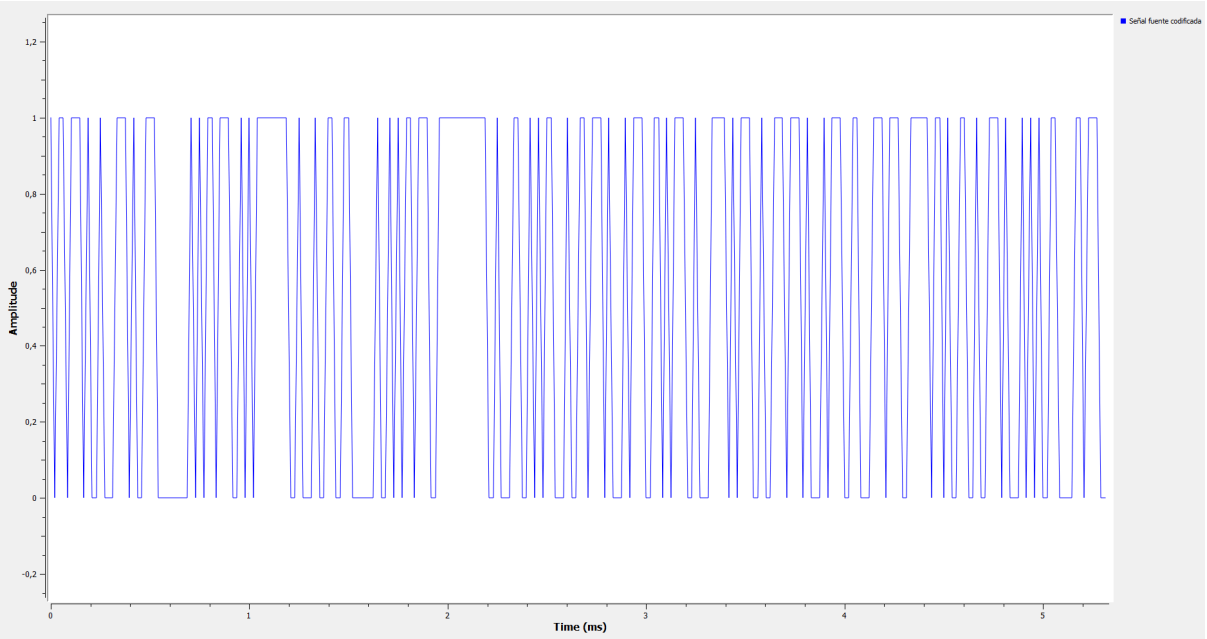

Figura 4: Señal fuente codificada usando CODEC2 Audio Encoder

Como la salida del *Codec2* es un entero complejo de 8 bits en cada parte, el bloque **Vector to Stream** los convierte en un flujo continuo de datos para que con el bloque **Unpack to Packed** se ajusten los datos en un formato más compacto para el posterior procesamiento.

Colocación de la trama

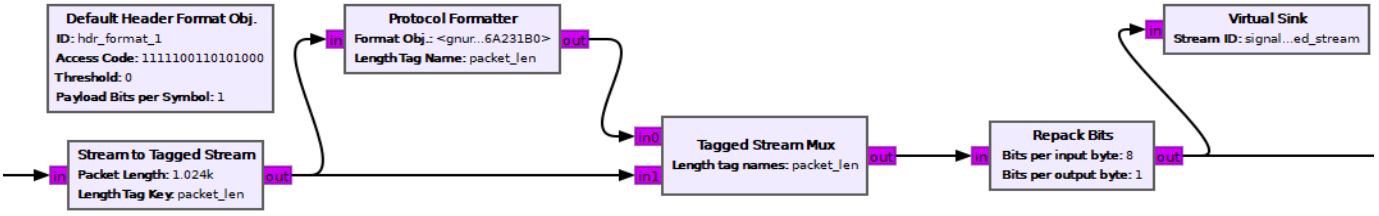

Figura 5: Diagrama de bloques de la etapa de colocación de trama

La etapa que se visualiza en la figura 5, implica el proceso de encapsular los datos codificados en segmentos o paquetes, cada uno con información adicional de encabezado que facilita la reconstrucción precisa de la señal en el receptor. Este encabezado incluye datos críticos que permiten al receptor identificar y procesar correctamente cada trama recibida.

El **Stream to Tagged Stream** convierte un flujo de datos normal en uno etiquetado añadiendo banderas cada una cantidad de bits especificada en Packet Length. De esta forma generamos etiquetas para posteriormente agregar el encabezado (*header*). A continuación, colocamos el bloque **Protocol Formatter** que recibe un flujo etiquetado y crea un encabezado cada cierta cantidad de bits (necesario que se agregue en cada paquete y por eso se coloca la variable *packet_len* generada por *Stream To Tagged Stream*) utilizando el bloque **Default *Header Format Object*.**

En Format Obj se coloca la ID del **Default Header Format Obj** que es el que este usa para crear el encabezado. El encabezado predeterminado creado consiste en un código de acceso escrito en el parámetro *Access Code* y la longitud del paquete. La longitud se codifica como un valor de 16 bits repetido dos veces. El parámetro *threshold* especifica cuántos bits del código de acceso pueden ser incorrectos antes de rechazar el paquete, en nuestro caso, lo dejamos en su valor por defecto '0'. Por último, la variable *Payload Bits per Symbol* es la cantidad de bits por símbolos, en este caso es 1 debido a que es el formato del flujo de datos que tenemos de entrada.

El bloque **Tagged Stream Mux** multiplexa los datos de entrada, dando cada bloque de datos con su correspondiente encabezado en la salida (figura 6)

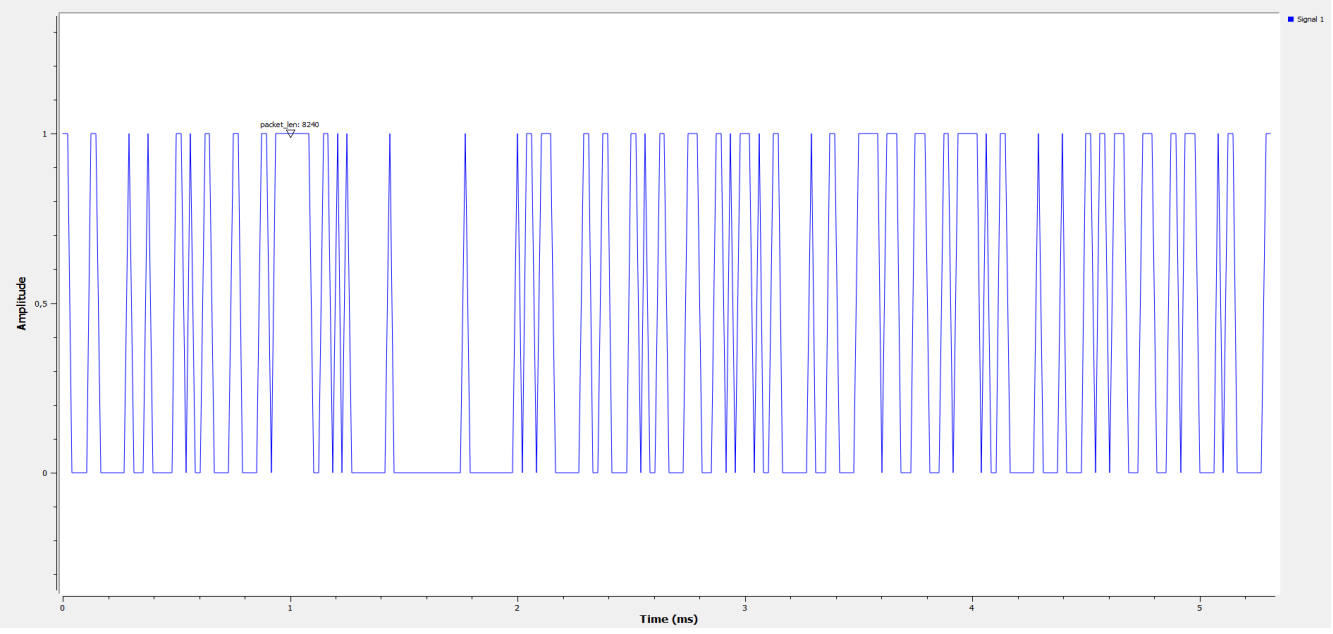

Figura 6: Señal con la colocación de trama

Codificación de Canal

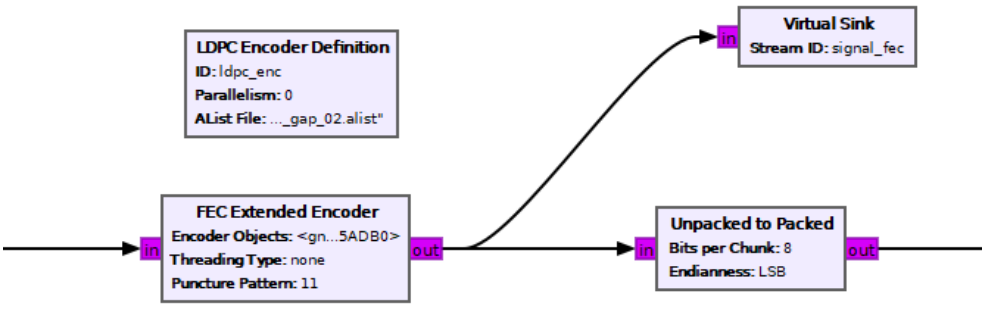

Figura 7: Diagrama de bloques de la codificación de canal

Esta etapa sirve para mejorar la fiabilidad de la transmisión de datos a partir de utilizar códigos de corrección de errores. En nuestro caso optamos por utilizar el código LDPC (*Low-Density Parity-Check*) que utiliza una matriz de paridad de baja densidad (con pocos elementos distintos de cero) donde cada fila y columna representa una ecuación de paridad que relaciona los bits de datos. En la figura 7, se pueden observar los distintos bloques que conforman esta etapa.

Como se puede observar, se utilizó el bloque **Fec Extended Encoder**, que utiliza a su vez a **LDPC Encoder Definition**, para implementar el código de corrección de errores propuesto a través de su parámetro *Encoder Objects.* Se deja en 'none' el parámetro *Threading Type* para no usar hilos adicionales en el procesamiento de datos y se establece en '11' al parámetro *Puncture Pattern* para especificar los bits de salida omitidos y transmitidos. Finalmente se empaquetan los bits de salida usando el bloque **Unpacked to Packed** por los requisitos de la siguiente etapa. Los datos de salida se pueden visualizar en la figura 8.

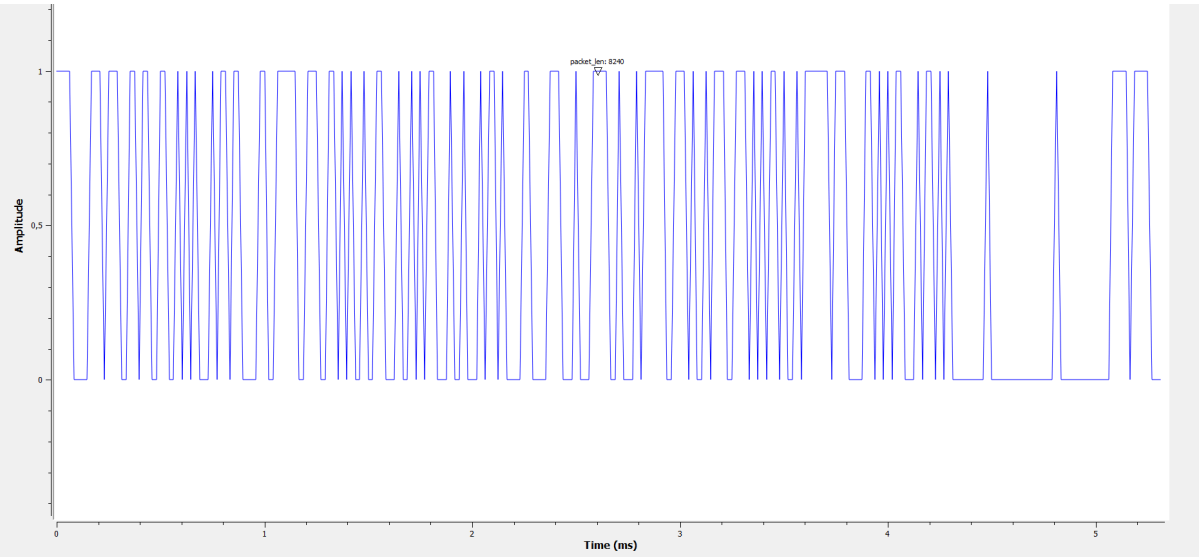

Figura 8: Señal luego de la codificación de canal

Modulación

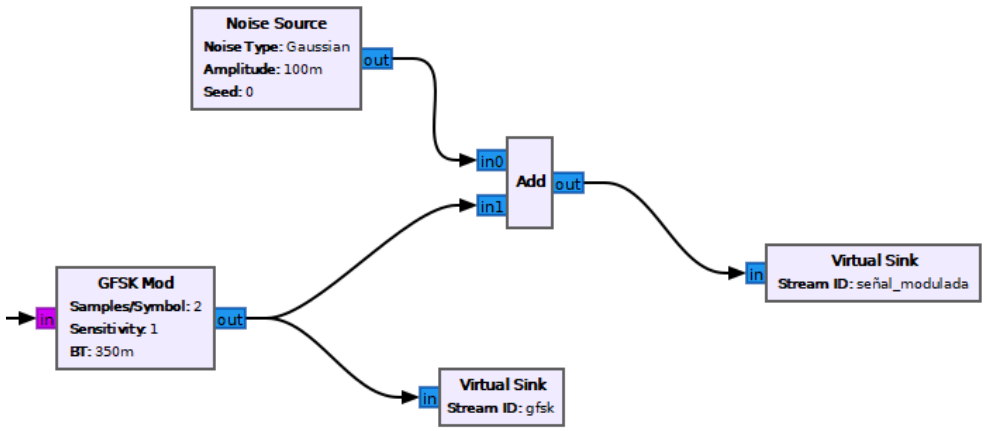

Figura 9: Diagrama de bloques del modulador

Se optó por la modulación GFSK (*Gaussian Frequency Shift Keying*), que como en la FSK, los datos binarios modulan la frecuencia de la señal portadora. La diferencia es que en GFSK se utiliza un filtro pasobajo gaussiano que suaviza las transiciones. De esta forma se reducen los cambios abruptos de frecuencia mejorando notoriamente el ancho de banda.

El bloque **GFSK Mod** es el que lleva a cabo la modulación. Se tomó un valor de 2 para el parámetro Samples/Symbols, que son la cantidad de muestras por baudio. El parámetro Sensitivity es para el bloque Frequency Mod (uno de los bloques que componen GFSK Mod). La sensibilidad especifica cuánto cambia la fase de la señal generada según la nueva muestra de entrada y para este caso se tomó el valor por defecto: 1. El parámetro *BT* es el ancho de banda del filtro interno por el tiempo de símbolo en los datos de entrada, siendo recomendado dejarlo en su valor por defecto 0.35. Finalmente, para simular el ruido introducido por el canal, se suman los datos con la salida de una fuente de ruido Gaussiano (figura 9).

En la figura 10, podemos observar en azul la parte real que es la componente en fase de la señal y en rojo la parte imaginaria que representa la componente en cuadratura de la señal. En la misma figura, se distingue el espectro de frecuencias de la señal modulada, donde se muestra cómo la energía de la señal se distribuye a lo largo de diferentes frecuencias. Esto es útil para ver la eficiencia del filtrado gaussiano y la calidad de la modulación GFSK.

En la figura 11, vemos que el diagrama de constelación muestra una forma circular que indica el radio y la posición de los puntos a lo largo de estas trayectorias que están influenciados por la frecuencia instantánea de la señal. En presencia de ruido, estas trayectorias aparecen más dispersas o distorsionadas, como se aprecia en la figura 12. Ahora bien, en ambas simulaciones se puede observar como estos puntos van rotando constantemente, y esto se debe a que en la modulación GFSK no tenemos una única frecuencia. Cada vector IQ (donde I es la amplitud momentánea de la señal y Q es la amplitud momentánea de la señal desplazada en fase -90°) rota en un círculo por cualquier desviación desde la frecuencia central, con la velocidad de rotación siendo proporcional a la desviación de frecuencia. Esto se explica porque hay dos o más ondas de frecuencia diferente, lo cual implica que se están **desfasando continuamente** en el tiempo. De esta forma el desfase relativo de la portadora también cambia continuamente con el tiempo manteniendo la amplitud constante. Así que FSK (o GFSK) modula la velocidad de

rotación del vector IQ. Para 4-FSK, por ejemplo, habría 4 velocidades de rotación diferentes que se observarían a lo largo del tiempo.

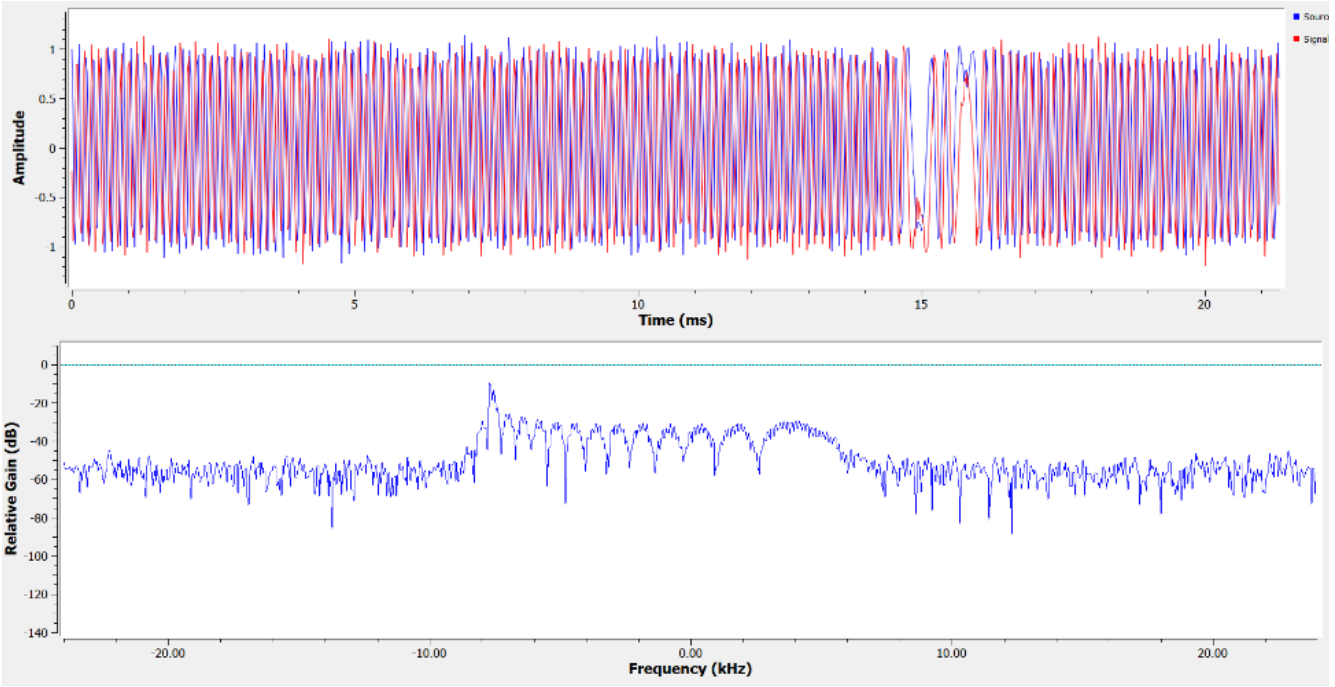

Figura 10: Señal a la salida del modulador GFSK con su espectro de frecuencias

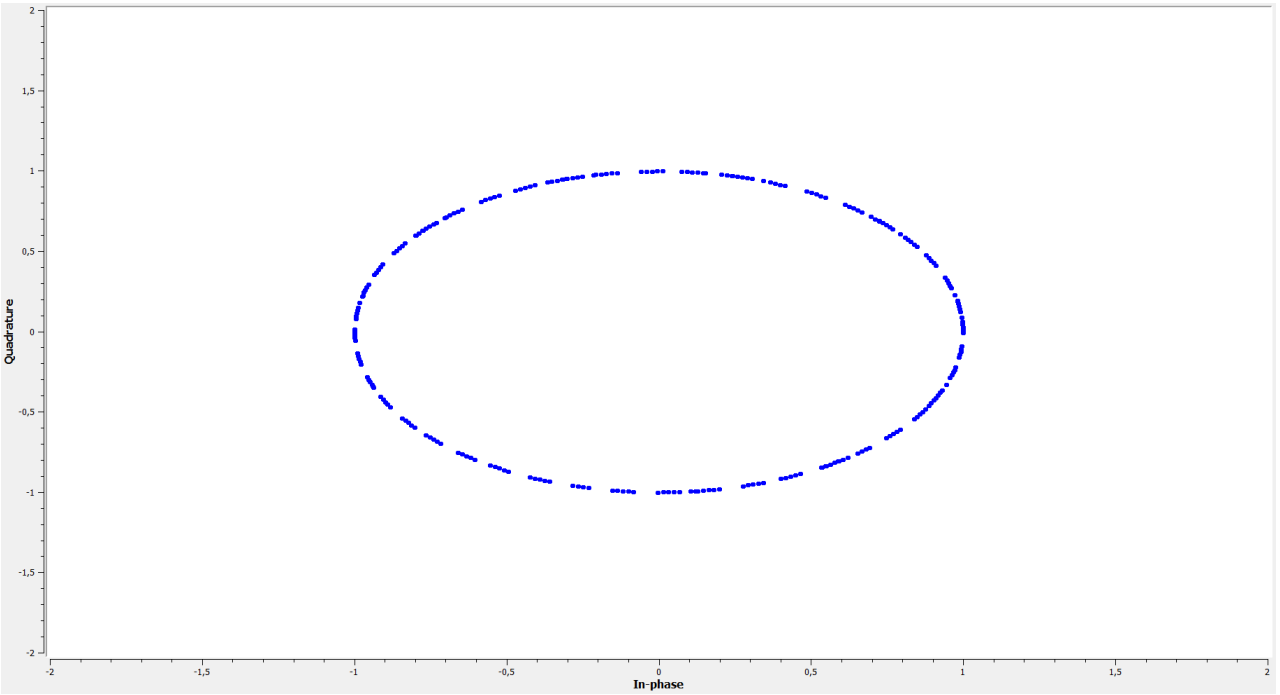

Figura 11: Diagrama de constelación de la señal a la salida del modulador GFSK

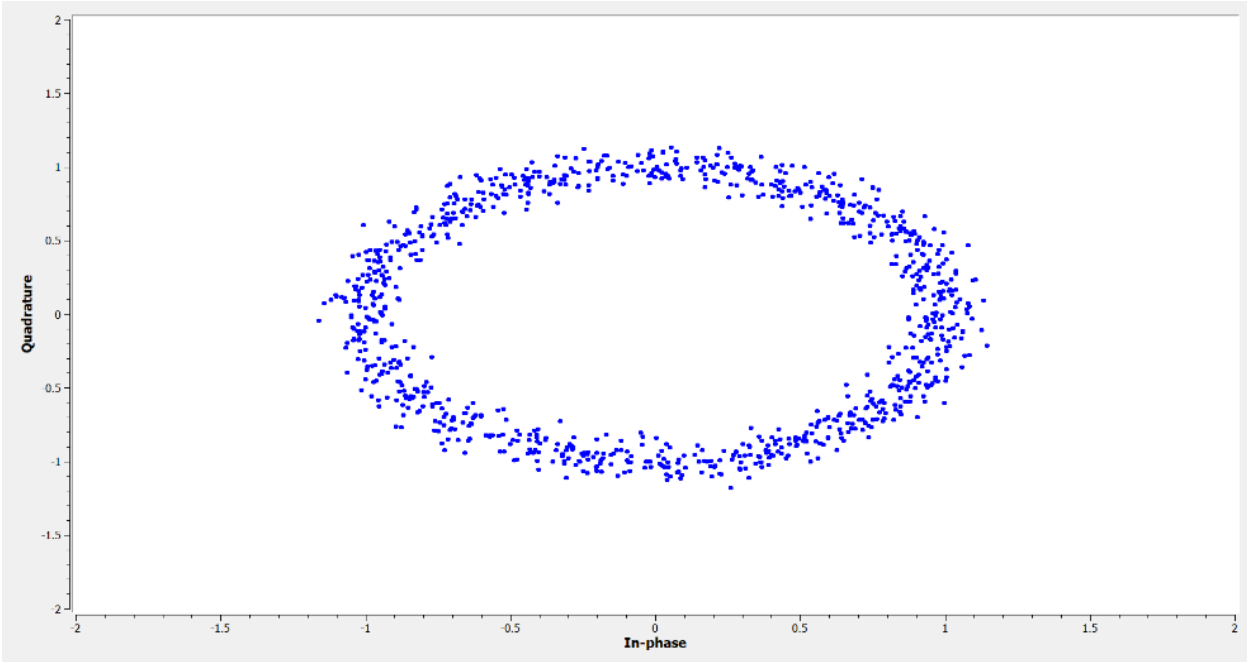

Figura 12: Diagrama de constelación de la señal a la salida del modulador GFSK con ruido Gaussiano

RECEPTOR

Demodulación de datos

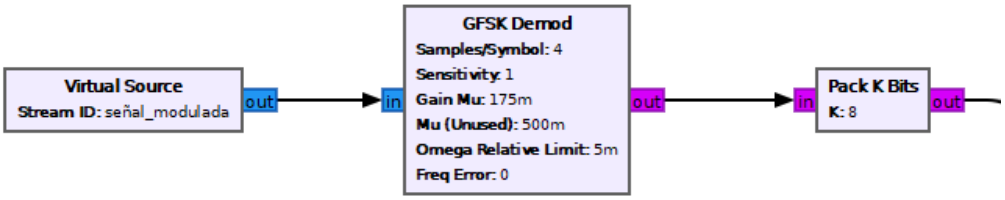

Figura 13: Diagrama de bloques del demodulador

A partir de los datos recibidos del transmisor procedemos a demodular con el bloque **GFSK Demod**, este proceso implica detectar cambios en la frecuencia de la señal dentro de cada ventana de símbolo en el que se define el número de 4 muestras por símbolo, lo que significa que cada símbolo en la señal modulada está representado por 4 muestras.

El parámetro *Gain Mu* controla la ganancia del filtro adaptativo que se usa para afinar la sincronización de símbolos, se eligió un valor moderado de 175m para tener una capacidad de ajuste dinámico en respuesta a variaciones en la señal recibida. El parámetro *Omega Relative Limit*, limita el rango de variación de la frecuencia de muestreo estimada durante la demodulación y el parámetro *Freq Error* es un parámetro para ajustar errores en la estimación de la frecuencia que en nuestro caso no será considerado. Como la salida es un flujo de datos desempaquetados, se utiliza el bloque **Pack K Bits** ya que en la siguiente etapa necesitaremos paquetes de 8 bits.

Decodificador de Canal

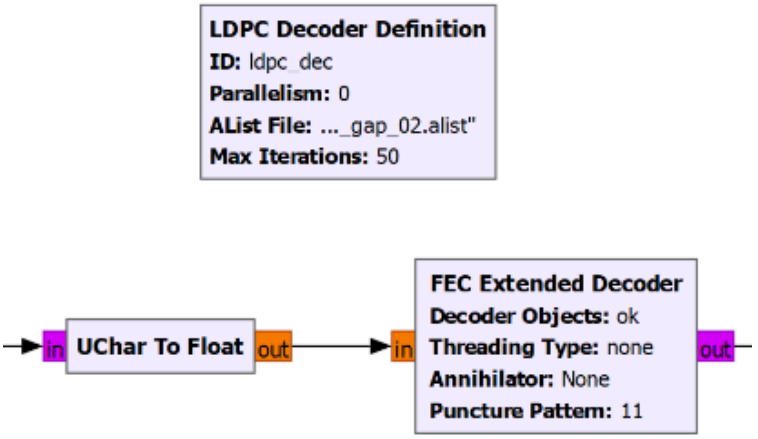

Figura 14: Diagrama de bloques del decodificador de canal

Primero se transforman los datos de tipo entero sin signo a datos de tipo flotante mediante **UChar To Float**. Luego, los datos convertidos son procesados por el **FEC Extended Decoder** que utiliza la definición **LDPC Decoder Definition**. Este utiliza el método iterativo para intentar corregir los errores detectados en la señal recibida basándose en la matriz de paridad especificada en el archivo AList.

Detección de la trama

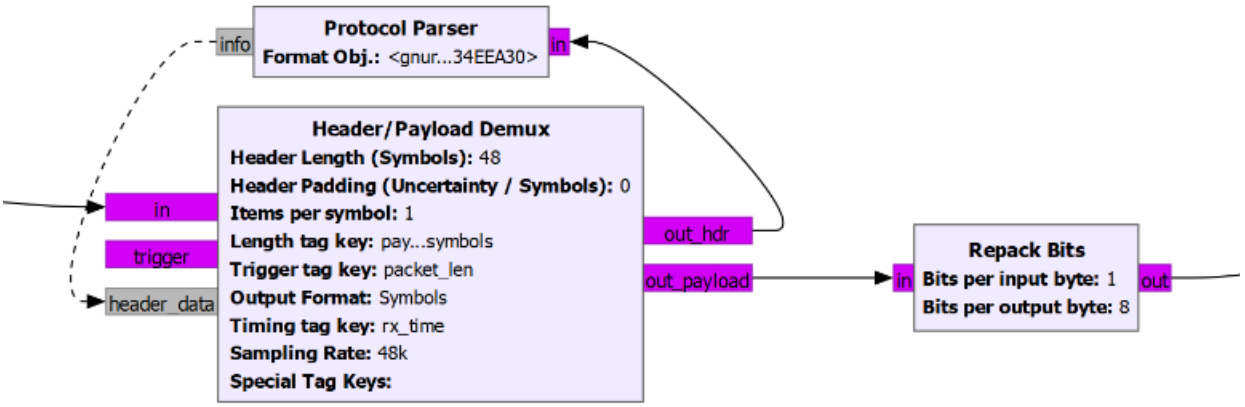

Figura 15: Diagrama de bloques para detectar la trama

Los datos que se agregaron en la trama aseguran la correcta transmisión y ahora se debe separar la trama de la información, por lo que se utiliza el bloque **Header/ Payload Demux**.

La entrada *Trigger* funciona como un disparador que indica cuando comienza un encabezado. En este caso no se la utiliza ya que se necesitaría un algoritmo de sincronización de tiempos que no suele ser óptimo. En su defecto, se utiliza el parámetro *Trigger Tag Key* para indicar cuando comienza el encabezado ya que es un valor que conocemos y es igual a la variable *packet_len* (1024). Una vez que se detecta el disparo, un número igual a *Header Length* de elementos de *In* se copia a la salida *out_hdr*. El bloque se detiene hasta que recibe un mensaje en el puerto de entrada *header_data*. Este mensaje es un diccionario PMT y lo proporciona el bloque **Protocol Parser**. Entonces todos los pares clave valor del diccionario se copian como etiquetas al primer elemento del payload o carga útil (información a recuperar). El valor correspondiente a Length Tag Key se lee y se toma como la longitud de la carga útil (ver nota). Finalmente, la información recuperada, junto con los datos del encabezado como etiquetas, se copian a la salida *out_payload*.

Nota: Aunque no se corresponda con ninguna variable definida previamente, el bloque solo funciona correctamente cuando se coloca en el parámetro 'Length Tag Key' la clave "payload symbols". Agradecemos a nuestros compañeros por brindar los resultados de su proyecto en el que se descubre este detalle (ver referencias).

Decodificación de fuente y recuperación de la señal

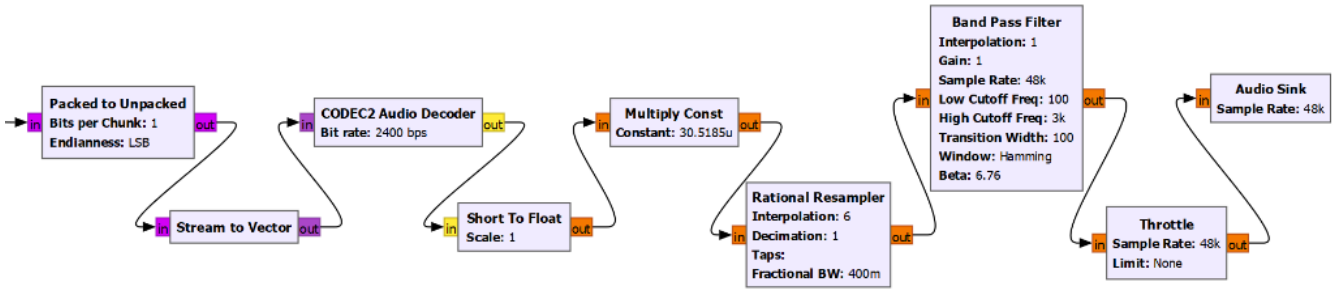

Figura 16: Diagrama de bloques de decodificacion de fuente

En esta etapa se decodifica audio comprimido utilizando el bloque **CODEC2 Audio Decoder**, la tasa de bits es de 2400 bps, lo que indica una compresión de audio de baja tasa de bits. Previo a esto se desempaqueta los datos en un bit por byte y se agrupa el flujo de datos en vectores de tamaño fijo. Luego se procede a convertir a los datos en tipo flotante y se multiplica por una constante de 1/8192, usada para ajustar el nivel de señal al rango esperado por el **Rational Resampler.** Este a su vez, cambia la tasa de muestreo de la señal mediante una interpolación de 6 y decimación de 1 para volver a la tasa de muestreo original. Se filtra la señal para limitar el ancho de banda entre una frecuencia baja de 100 Hz y una alta de 3 Khz, eliminando frecuencias fuera de este rango que pueden ser ruido o interferencias no deseadas. Se usa una ventana de Hamming para reducir la amplitud de los extremos del bloque, disminuyendo así la energía de ciertas frecuencias que se 'filtran' hacia otras frecuencias en el espectro

resultante y además disminuir la distorsión de la señal. Se regula la velocidad de procesamiento de los datos para simular el tiempo real y evitar el consumo excesivo de CPU a la frecuencia de muestreo utilizada con el bloque **Throttle** y finalmente se reproduce la señal con el **Audio Sink**. En la figura 17 se puede observar la señal de salida.

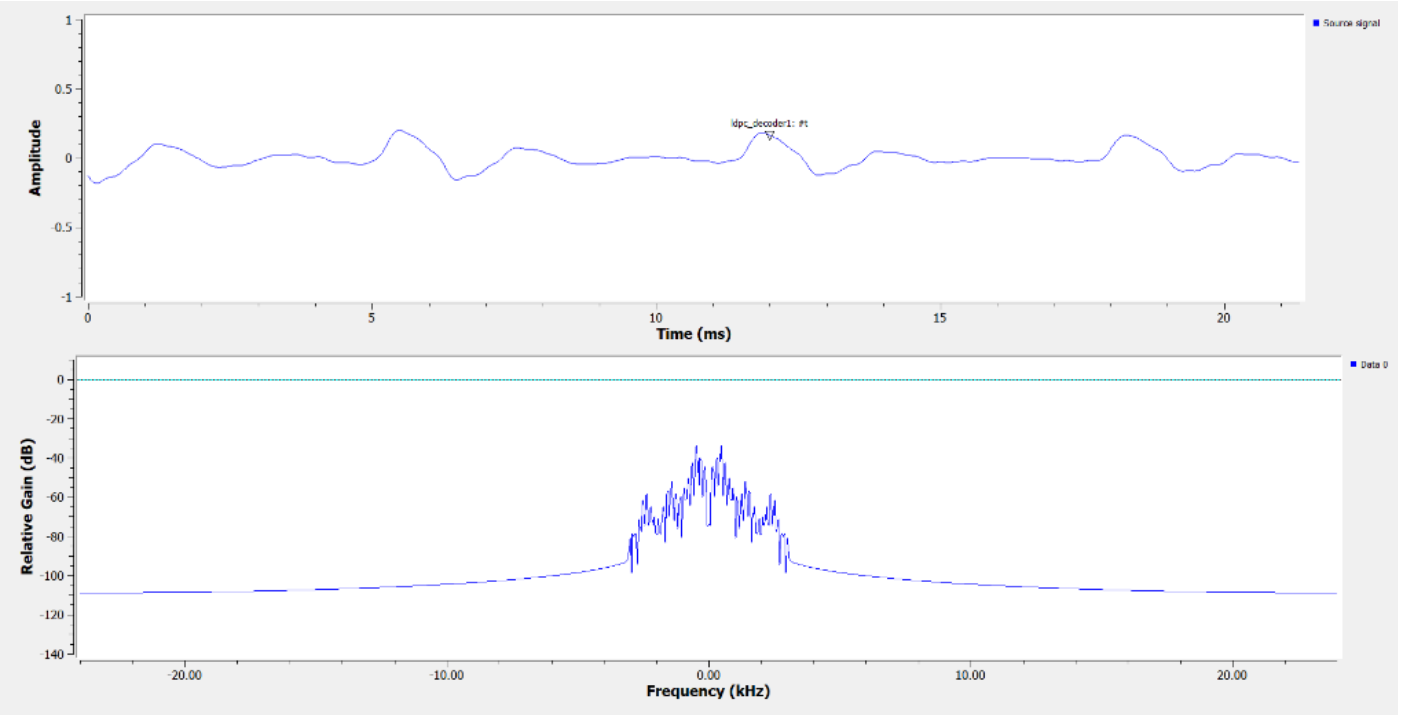

Figura 17: Señal obtenida a la salida del receptor con su espectro de frecuencias

CONCLUSIÓN

Analizando y probando cada bloque y entendiendo su funcionamiento y aplicación, logramos transformar con éxito una señal de audio de voz a través de varias etapas de procesamiento, desde su codificación y modulación hasta la demodulación y decodificación final. El resultado fue la recuperación satisfactoria de la señal de audio en el punto de recepción, con una calidad que permitió su clara apreciación. Si bien la calidad de la señal recuperada no es la misma que la de entrada, se tenía esto en consideración desde el principio ya que *CODEC2* es un código de compresión con pérdida.

El resultado de este proyecto no solo demuestra la viabilidad de diseñar sistemas de comunicación digital complejos con GNU Radio, sino que también subraya la importancia de una comprensión profunda de cada componente dentro del sistema.

Referencias:

- https://github.com/gnuradio/gnuradio/blob/master/gr-fec/examples/fecapi_ldpc_decoders.grc
- GitHub - adanlema/tx-rx-gnuradio: Implementación de Transmisión y Recepción Digital con GNU Radio
- https://wiki.gnuradio.org/
- http://whiteboard.ping.se/SDR/IQ