



Laboratorio 1

Demodulación de señales con Dongle

SDR

Comunicaciones (E0311)

Ternouski Sebastian Nahuel 756/6

La Plata, Junio de 2018

Índice

| | |
|---|-----------|
| 1. Introducción | 1 |
| 2. Desarrollo | 1 |
| 2.1. Generador de una muestra de FM | 1 |
| 2.1.1. Demodulación | 5 |
| 2.1.2. Densidad espectral de potencia | 5 |
| 2.1.3. Deénfasis | 6 |
| 2.2. Radio Streaming | 6 |
| 3. Conclusiones | 7 |
| Referencias | 9 |
| Anexo | 10 |
| A. Guías | 10 |
| A.1. Instalación | 10 |
| A.2. Comprobando resultados | 10 |
| B. Código | 11 |
| B.1. sample.py | 11 |
| B.2. utils.py | 13 |
| B.3. radio.py | 19 |

1. Introducción

En el presente laboratorio se espera dar las bases de la teoría de las comunicaciones para la demodulación de una radio FM y aplicarlas sobre un dispositivo físico. Para ello se toma como referencia el libro *Principles of Communications* [Ziemer-Tranter, 2015].

La modulación de FM se trata de una modulación angular no lineal y idealmente se basa en transmitir el mensaje en la fase de la portadora (logrando una desviación de fase según el mensaje) y la amplitud de la portadora mantenerla constante. Esto es, la señal modulada que sera transmitida por el canal se traduce en:

$$x_c(t) = A_c \cos \left(2\pi f_{ct} t + 2\pi k_f \int_{-\infty}^t m(\lambda) d\lambda \right)$$

Donde k_f es la constante desviación de frecuencia y es definido como:

$$k_f = \frac{\Delta f}{\max|m(t)|}$$

2. Desarrollo

En la demodulación se trata de reconstruir el mensaje $m(t)$ a partir de la señal que recibe la antena, que como se mencionó anteriormente es $x_c(t)$. Para ello se utilizó el lenguaje python con las siguientes librerías para la implementación digital del circuito:

- **rtlsdr**: Contiene funciones para poder controlar y usar el dongle SDR, véase [Roger, 2013].
- **numpy**: Biblioteca de funciones matemáticas de alto nivel para operar con vectores o matrices
- **scipy**: También es una librería pero se basa más en operaciones que se utilizan en el procesamiento de señales y puntualmente con la sub librería io es posible realizar operaciones como guardar o abrir un archivo.
- **matplotlib**: Es utilizado para generar gráficos que serán exportados a formato pdf.
- **sounddevice**: Es módulo de reproducción de audio, que sirve también para streaming.

Se facilita una guía de instalación y uso de los scripts en el apéndice A donde se pueden encontrar los pasos para recrear lo estudiado en el presente informe.

2.1. Generador de una muestra de FM

El archivo `sample.py` contiene el algoritmo para generar una muestra de la estación de radio 105.3MHz, el cual utiliza las funciones de `utils.py` para demodular la señal que se obtiene del dongle.

El primer paso es tomar las muestras, para ello se llama a la función *GetSample*, la cual se inicializa los parámetros necesarios para el `rtlsdr` y toma N muestras con una cierta ganancia *gain*.

La estación en cuestión se encuentra centrada en *f_offset* de tal manera de evitar interferencias en continua es decir en frecuencia cero. Esto ocurre porque al muestrear la señal provoca

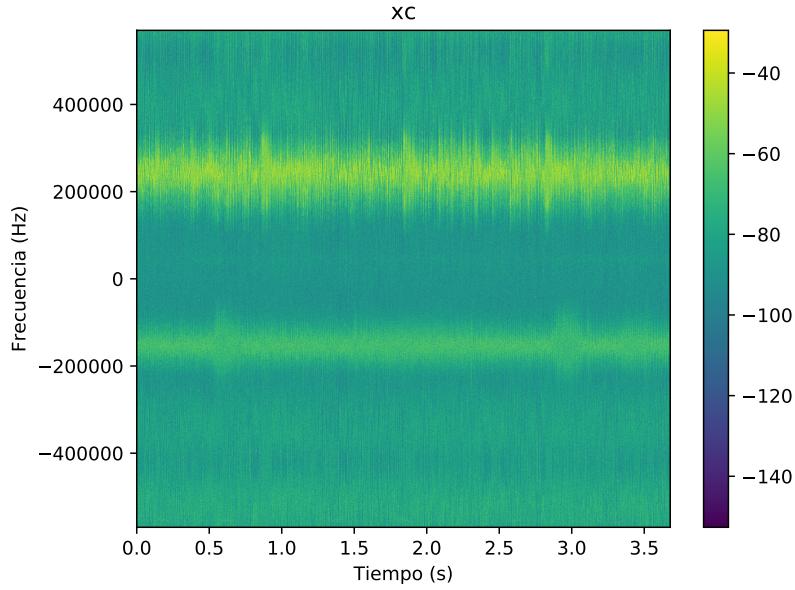


Figura 1: Espectro de la muestra capturada.

que la misma no tenga exactamente media cero. La figura 1 muestra el espectro de la señal obtenida del dispositivo.

Posteriormente se pasa a banda base usando la función *ToBaseBand*, esta función lo que hace es multiplicar la señal por una exponencial, lo que se traduce en el espectro como un desplazamiento en frecuencia, quedando la señal x_c centrada en frecuencia cero. Es decir:

$$x_b = x_c \cdot e^{-j2\pi f_{os}} \quad f_{os} = \frac{f_{offset}}{f_s}$$

Se puede observar el cambio en la figura 2, y su correspondiente DEP en la figura 3.

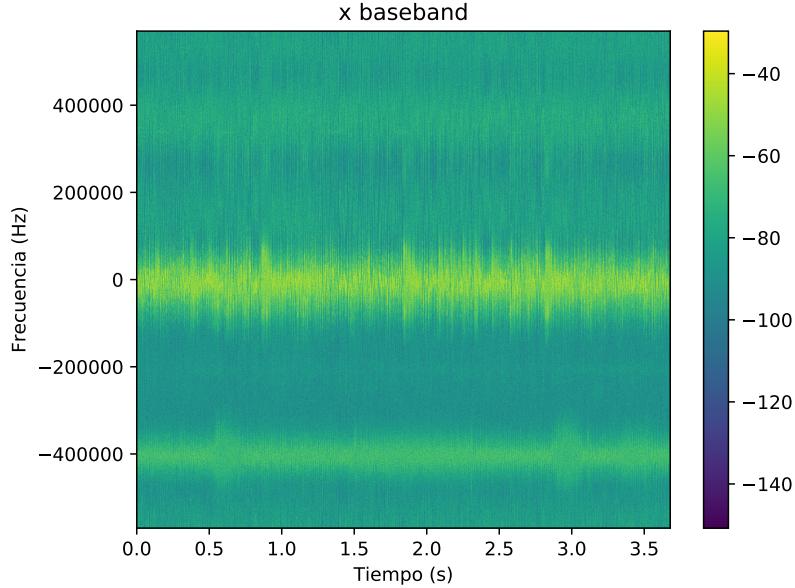


Figura 2: Señal x_c en banda base, es decir x_b .

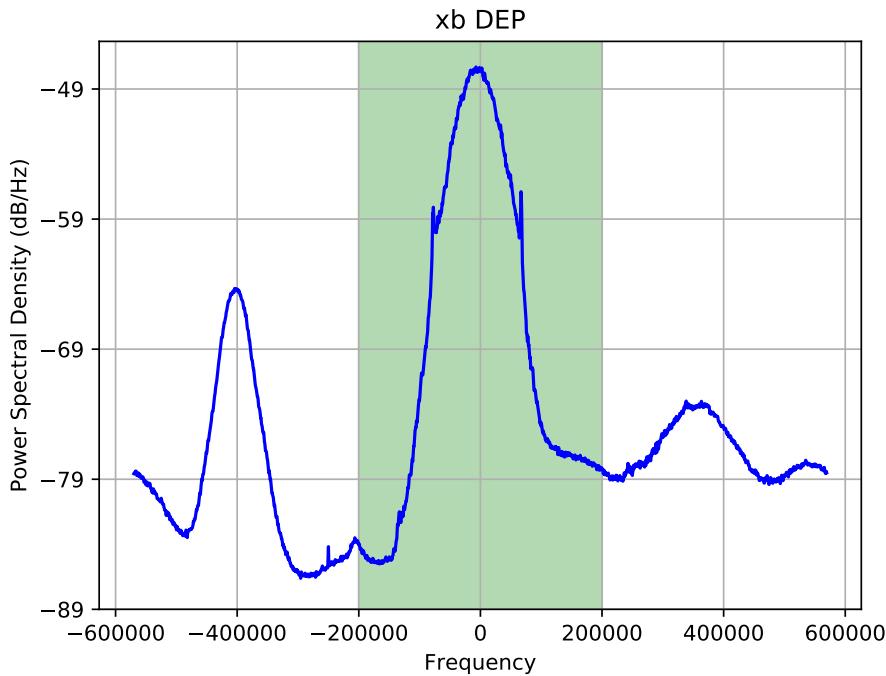


Figura 3: Densidad espectral de potencia la señal x_b .

Ahora que se tiene la señal x_b , señal en banda base, se procede a realizar el filtrado y diezmado para que entre otras cosas no se sume ruido ni parte de otros canales al siguiente paso que es demodular. La función *FilterAndDownSample* en el archivo *utils.py* tiene este propósito, en ella se calculan los coeficiente adecuados para armar un filtro con las características mostradas en la figura 4, sabiendo que el ancho de banda de los canales de radios comerciales es de $f_{bw} = 200\text{kHz}$, el área verde de la figura 3 indica lo que deja pasar dicho filtro.

Se puede observar en el apéndice B.2 (línea 71) la función *remez*¹ que es utilizada para la creación del filtro, el cual calcula los coeficientes del mismo de una respuesta de impulso finito (FIR) cuya función de transferencia minimiza el error máximo entre la ganancia deseada y la ganancia realizada en las bandas de frecuencia especificadas utilizando el algoritmo de intercambio Remez. En resumen, se detallan a continuación los cuatro parámetros que recibe dicha función para confeccionarlos:

- 1^{er} parámetro: *numtaps*, *traps* es el número de términos en el filtro, o el orden de filtro más uno.
- 2^{do} parámetro: es una secuencia que contiene los bordes de la banda. En el ejemplo se tomaron intervalo de cero a el ancho de banda deseado y una caída del filtro que posee un 40% del ancho de pasa bajos.
- 3^{er} parámetro: Una ponderación relativa para dar a cada región de banda. La longitud del peso tiene que ser la mitad de la longitud segundo parámetro. Es decir en el ejemplo se le multiplica por la unidad a las frecuencias que van de cero a f_{bw} y un valor muy chico a las frecuencias que van de $f_{bw} + 40\%$ a $f_s/2$.
- 4^{to} parámetro: Frecuencia de muestreo.

¹De la librería *scipy.signal*, véase [línk](#)

La figura 5 muestra el resultado de filtrar utilizando los coeficiente de Remez de orden 50 y de diezmara la señal x_b .

Antes de demodular, se puede observar en la figura 6 el espacio de señal de fm.

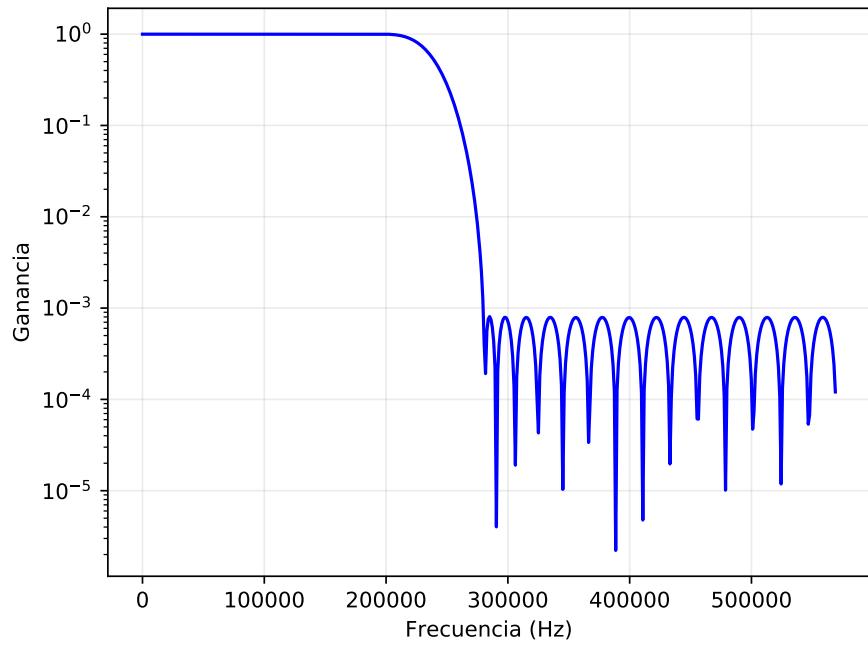


Figura 4: Respuesta en frecuencia, que caracteriza el filtro pre-demodulación.

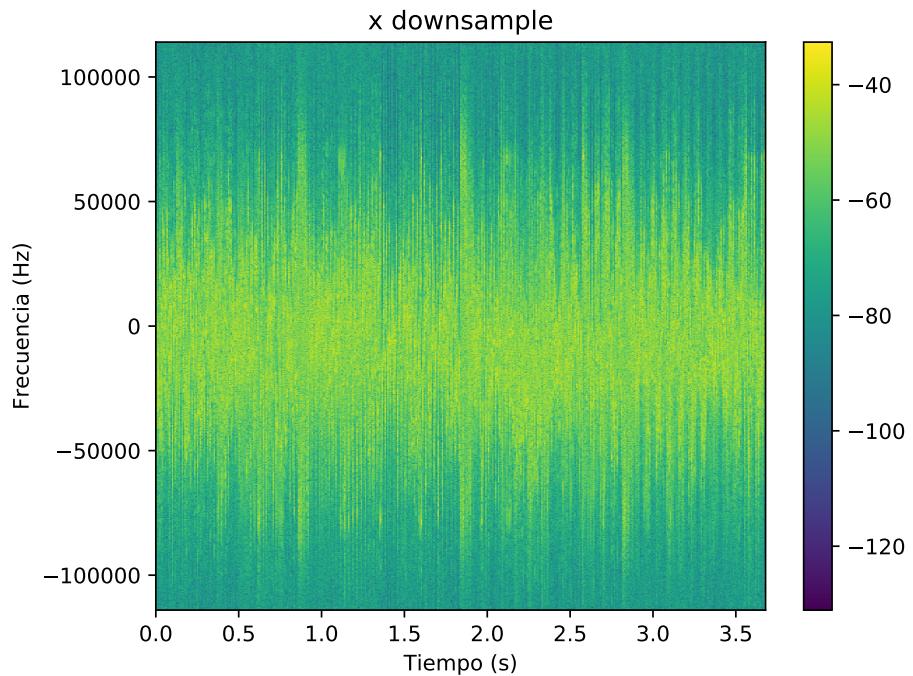


Figura 5: Señal filtrada y diezmada.

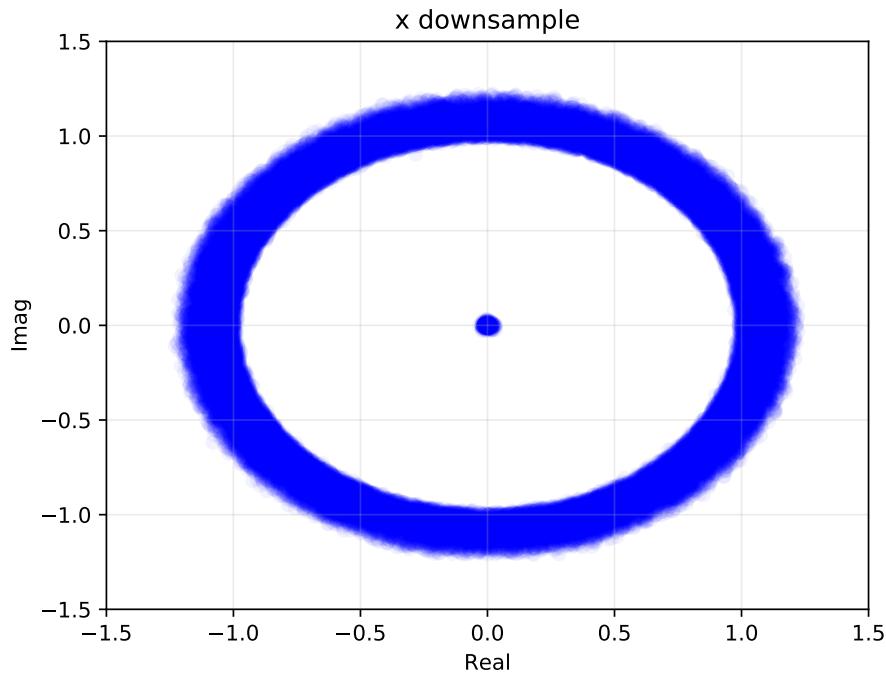


Figura 6: Gráfico de constelación de señal filtrada y diezmada.

2.1.1. Demodulación

La señal resultante de los pasos anteriores todavía no está demodulada sino que se acondiciona la señal ya que es posible que anteriormente contenga parte de otros canales, entre otros aspectos.

Para demodular la señal se optó por un tipo de discriminador de frecuencia llamado discriminador polar. Este mide la diferencia de fase entre muestras consecutivas de una señal FM muestreada de forma compleja. Esto se traduce en realizar la siguiente operación (línea 101 del apéndice B.2):

$$x_d[n] = x[n] \cdot \bar{x}[n-1]$$

Donde \bar{x} es el conjugado de x y luego tomar el angulo de $x_d[n]$ (línea 102 del apéndice B.2):

$$y_d[n] = \arctan\left(\frac{\text{Imag}}{\text{Real}}\right)$$

Más específicamente, toma muestras sucesivas de valores complejos y multiplica la nueva muestra por el conjugado de la muestra anterior. Luego toma el ángulo de este valor complejo. Esta resulta ser la frecuencia instantánea de la señal de FM muestreada.

2.1.2. Densidad espectral de potencia

El resultado de todos los procesos anteriormente mencionados, se puede observar en la figura 7. Se puede notar cuatro zonas de interés con diferentes colores²:

- Color rojo: Señal mono, es el de mayor interés, va de 0 a 15kHz.

²Para más información ingrese al siguiente [link](#)

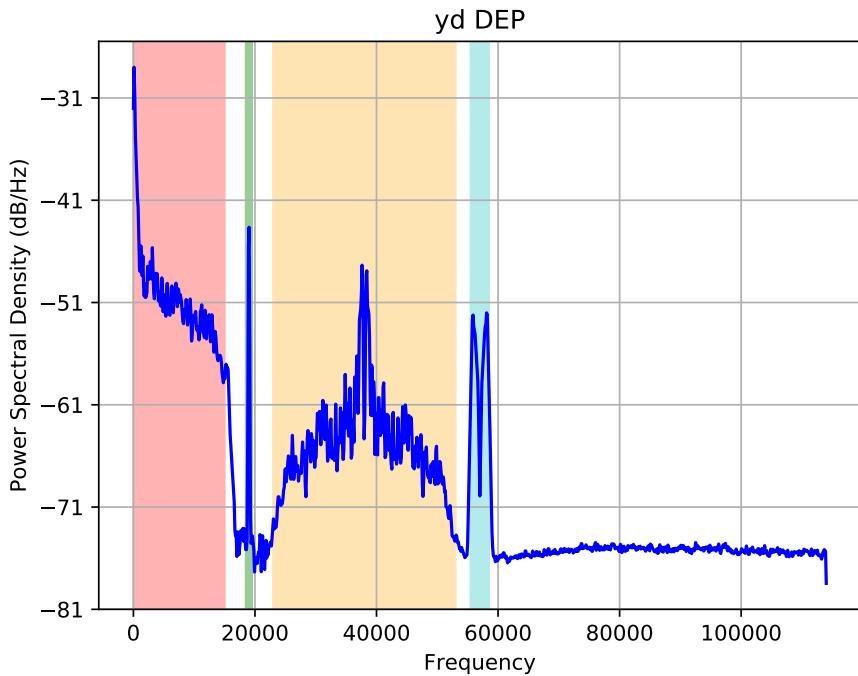


Figura 7: Densidad espectral de potencia la señal (yd) resultante diferenciando las distintas partes.

- Color verde: En la transmisión FM estéreo, el tono piloto centrada en 19kHz, indica que hay una información estereofónica. El receptor duplica la frecuencia del tono piloto (38kHz) y lo usa como frecuencia de referencia para demodular la información estéreo.
- Color naranja: Señal stereo, la banda de 23k a 38k Hz se encuentra el canal izquierdo y de 38k a 53kHz el canal derecho.
- Color cyan: Una subportadora centrada en 57kHz, se usa para transmitir una señal de sistema de datos de radio digital de ancho de banda angosta, que proporciona características adicionales de la estación de radio.

2.1.3. Deénfasis

El deénfasis (o de-emphasis en inglés³) consiste en disminuir las altas frecuencias en el receptor ya que el transmisor envía la señal con una ganancia elevada en comparación a las bajas frecuencias, esto se debe a que el ruido afecta más a las altas frecuencias en FM⁴. Para ellos físicamente se utiliza un circuito RC y en América Latina se utiliza una constante $\tau = 75$ para dicho propósito.

El código que realiza esta tarea se encuentra en el apéndice B.2, línea 106.

2.2. Radio Streaming

El objetivo de esta sección es tratar de reproducir la señal recibida por alguna emisora en tiempo real, es decir como una radio comercial. Cabe destacar que se utilizó la librería gráfica

³Para más información véase [link](#).

⁴Si deseas saber más sobre cuánto afecta el ruido en la señal FM puedes encontrar en la sección 8.8.3 del libro [Ziemer-Tranter, 2015].

Qt versión 4 para el desarrollo de la GUI.

Para ello se tienen algunas consideraciones respecto de la sección anterior, los cuales se mencionarán a continuación. Las librerías usadas en python como el lenguaje en sí no tienen paralelismo implícito, esto es, si se quiere realizar dos tareas a la vez (en simultaneo) se debe usar la librería *threading*, que contiene algoritmo de bajo nivel para el aprovechamiento del hardware. Se destacan dos tareas que deben ser ejecutadas por separado de forma simultanea y sin interrupción, capturar las muestras con el dongle y reproducir el audio, por lo tanto se optó por crear tres threads o hilos (el tercero es para la demodulación y procesamiento de la señal), teniendo en cuenta esto último el tiempo de procesamiento T_p tiene que ser considerablemente menor a los tiempos de captura T_c y de reproducción T_r , es decir:

$$T_p \ll T_c = \frac{N}{f_s}$$

Donde f_s es la frecuencia de muestreo; si esto no se cumple se obtendrá interrupciones al escuchar ya que el buffer de reproducción de audio estará vacía.

En el archivo llamado `radio.py` (véase apéndice B.3) se implementa la radio en cuestión. Se pueden observar los tres hilos de ejecución:

- Capturadora: en línea 148, a diferencia de como se captura las muestras en la sección anterior, en primer lugar se sustituyo la función *read_samples* por lo que hace realmente⁵. Por otro, se toman de a muestras mas pequeñas por limitaciones del hardware (USB y dongle) y se concatena las muestras para procesar el conjunto.
- Procesamiento: en línea 158, realiza lo mismo que en la sección anterior salvo por deénfasis que no se realiza, ya este caso se trata de optimizar el uso de variables y uso del procesador.
- Reproducción: en línea 176, este hilo tiene la particularidad de usar la clase *OutputStream* de la librería sounddevice en la que es posible almacenar muestras de audio en un buffer y reproducirlas rápidamente.

Por ultimo, los archivos de la GUI que usa las funciones de la radio son: `qt.py` en el que se ejecuta la ventana de control. El archivo `windows.py` configuran las variables de la interfaz (tamaño de ventana, título de ventana, barra de estado, etc) y cumple la función de nexo entre la radio en sí y los botones que se encuentran en el panel central. Y el archivo `centerWidget.py` crea los botones junto con el deslizable de frecuencia los agrupa de forma tal como se muestra en la figura 8.

3. Conclusiones

Se puede llegar a la conclusión de que con el procesamiento digital genera resultados muchos mas rápidos y de menor costo si se hubiera hecho de forma analógica. Sin embargo, el lenguaje python, que es un lenguaje interpretado, no tiene suficiente rendimiento para optimizar los cálculos, lo que conlleva a un mal desempeño para el procesamiento de señales en tiempo real. En contrapartida existe lenguajes capaces de lograr el objetivo como Matlab, que a pesar de que es interpretado posee esta optimizado o de mas bajo nivel como C entre otros.

Existen módulos pendientes o en fase beta para la radio streaming que son: volumen (fase beta), cascada de la potencia promedio o waterfall FM, entre otros. Para futuras actualizaciones se podran ver en el repositorio de [github](#).

⁵Puede verlo en el siguiente [link](#).



Figura 8: Diseño de la GUI con Qt4

Referencias

- Roger. [2013]. *Pyrtlsdr*. (A Python wrapper for librtlsdr <https://nocarryr.github.io/pyrtlsdr>)
- Ziemer-Tranter. [2015]. *Principles of communications: systems modulation and noise* (7ma ed.). Wiley.

Anexo

A. Guías

A.1. Instalación

Se usaron algunas librerías que no se instalan por defecto con el lenguaje python por lo que son necesarias instalarlas manualmente con comandos en la terminal, solo desde una distribución ubuntu o similares como linux mint. El fragmento 1 indica los comandos iniciales para la instalación:

```
1 # Refresca el repositorio
2 sudo apt update
3 # Actualiza apt
4 sudo apt upgrade
5 # Instala python 2.7 y pip
6 sudo apt install python2.7 python-pip
7 # Posicionarse en la carpeta raíz
8 # Instala las librerías necesarias de python
9 # Importante: Es necesario estar en la carpeta que contiene el
    ↪ archivo requirements.txt (carpeta raíz)
10 sudo pip install -r requirements.txt
```

Fragmento 1: Script de instalación

Con estos pasos es posible simular lo mencionado en la sección 2.1. Para la radio en tiempo real es necesario instalar Qt.

Para que no sea muy extenso la guía se deja al lector instalar Qt con la versión cuatro. Tenga en cuenta que dicha instalación dura un par de horas. Puede servir de mucha utilidad los siguientes links: [sourceforge](#) y [github](#), nótese que es para la versión cinco pero al final quedará instalada la versión cuatro.

A.2. Comprobando resultados

Existen tres scripts en la carpeta raíz. Cabe destacar que deberá permitir permisos de ejecución con el comando chmod para correrlos.

El script `script-sample.sh` es para ejecutar lo de la sección 2.1, debe estar conectado el dongle previamente. Al ejecutar a través del script dado, se debe pasar un parámetro que determina si se desea crear una nueva muestra del dongle y guardarla en un archivo llamado `FMcapture.npy`, ingresar 0 (cero), o si se quiere demodular a través de un archivo guardado previamente ingresar 1. Por ejemplo: `./script.sh 1` ejecutara la demodulación de el archivo fuente y se escuchara el sonido resultante. Cabe destacar que el script pedirá permisos de administrador para poder usar las librerías de `rtlsdr`. Nótese que si se quiere generar los gráficos, hay una constante llamada `PLOT` que define si generan o no.

El script `script-radio.sh` ejecutara la radio en tiempo real sin interfaz gráfica por 14 segundos. Conectar el dongle previamente.

El script `script-qt-run.sh` ejecutara la radio pero esta vez con interfaz gráfica. Por defecto la radio tiene configurada una frecuencia de estación a 99.1MHz y no se reproducirá hasta que se haga click en el botón de play.

B. Código

B.1. sample.py

```
1 # -*- coding: utf-8 -*-
2
3 from utils import *
4 import sys
5 import time
6
7 FILE_CAPTURE = "FMcapture.npy"
8 FILE_SOUND = "soundFM.wav"
9
10 def main():
11     f_station = int(105.3e6) # Frecuencia de radio
12     f_offset = 250000 # Desplazamiento para capturar
13     fs = int(1140000) # Frecuencia de muestreo
14     N = int(2**22) # Numero de muestras
15     # N = int(332800) # Numero de muestras
16
17     if not (len(sys.argv) == 2):
18         print("Ingrese un numero: 0 o 1..")
19         sys.exit(1)
20
21     print("El parametro ingresado es: ", sys.argv[1])
22     if int(sys.argv[1]) == 0:
23         print("\tSe tomaran muestras del dounle..\n")
24         samples = GetSample(f_station, f_offset, fs, N, 'auto')
25         np.save(FILE_CAPTURE, samples)
26     elif int(sys.argv[1]) == 1:
27         print("\tSe leerá del archivo..", FILE_CAPTURE, "..\n")
28         samples = np.load(FILE_CAPTURE)
29
30     print("\tSe mide el tiempo para N = ", N, '(%f %%(N/fs), "s'
31         ↪ ..\n')
32     start_time = time.time()
33     # Convierte las muestras en numpy array
34     xc = np.array(samples).astype("complex64")
35     x_b = ToBaseBand(xc, f_offset, fs)
36     DEP(x_b, "xb_DEP", "xb_DEP.pdf", fs)
37     x_filter, fs_y = FilterAndDownSample(x_b, fs)
38     yd = Demodulation(x_filter, fs_y)
39     DEP(yd, "yd_DEP", "yd_DEP.pdf", fs_y, True)
40     yd = FilterDeEmphasis(fs_y, yd)
41     finish_time = time.time() - start_time
42     print("\tTiempo de computo: ", '%.3f %%finish_time, "seg'
43         ↪ --")
43     SaveToFile(yd, FILE_SOUND, fs_y)
```

```
43     PlaySound(yd, fs_y)  
44  
45 main()
```

B.2. utils.py

```
1 # -*- coding: utf-8 -*-
2
3 import rtlsdr
4 import numpy as np
5 from scipy.signal import hilbert, remez, lfilter, decimate, freqz
6 from scipy.io import wavfile
7 import sounddevice as sd
8 import matplotlib
9 matplotlib.use('Agg') # http://stackoverflow.com/a/3054314/3524528
10 import matplotlib.pyplot as plt
11
12
13 # Define si se quiere que se hagan los gráficos o no:
14 # PLOT = 0 # No se calculan los gráficos
15 PLOT = 1 # Se calcula los gráficos
16
17 F_BW = 200e3 # FM tiene un ancho de banda teorico de BW
    ↪ =180/200 kHz
18
19
20 def GetSample(f_station, f_offset, fs, N, gain='auto'):
21     """
22         Genera una muestra de radio FM:
23
24         Parametros:
25             f_station: Frecuencia de la Estación
26             f_offset: Frecuencia corrida
27             fs: Frecuencia de muestreo
28             N: Cantidad de muestras
29             gain: Ganancia de la señal
30     """
31
32     sdr = rtlsdr.RtlSdr()
33     fc = f_station - f_offset # Frecuencia del centro de captura
34     # Se configura los parametros
35     sdr.sample_rate = fs
36     sdr.center_freq = fc
37     sdr.gain = gain
38     # Lee las N muestras
39     samples = sdr.read_samples(N)
34     # Limpiar el dispositivo SDR
40     sdr.close()
41     del(sdr)
42
43     return samples
44
45
```

```

46 def ToBaseBand(xc, f_offset, fs):
47     """
48     Parametros:
49         xc: Señal a mandar a banda base
50         f_offset: Frecuencia que esta corrido
51         fs: Frecuencia de muestreo
52     """
53     if PLOT:
54         PlotSpectrum(xc, "xc", "xc_offset_spectrum.pdf", fs)
55         # Se lo vuelve a banda base, multiplicando por una
56         # → exponencial con fase f_offset / fs
57         x_baseband = xc * np.exp((-1.0j * 2.0 * np.pi * f_offset/fs) * np.
58             # → arange(len(xc)))
59     if PLOT:
60         PlotSpectrum(x_baseband, "x_baseband", "
61             # → x_baseband_spectrum.pdf", fs)
62     return x_baseband
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
def FilterAndDownSample(x, fs):
    """
    Parametros:
        x: Señal a filtrar y
        fs: Frecuencia de muestreo
    """
    # traps es el número de términos en el filtro, o el orden de
    # → filtro más uno.
    n_taps = 50
    # Calcule el filtro óptimo minimax utilizando el algoritmo de
    # → intercambio Remez.
    coef = remez(n_taps, [0, F_BW, F_BW*1.4, fs/2], [1, 0], Hz=fs)
    # x_filter = lfilter(coef, 1.0, x)
    if PLOT:
        PlotFilterCharacteristic(coef, fs)
    dec_rate = int(fs / F_BW)
    x_downsample = x_filter[0::dec_rate]
    # Se calcula la nueva frecuencia de muestreo
    fs_y = fs / dec_rate
    if PLOT:
        PlotSpectrum(x_downsample, "x_downsample", "
            # → x_downsample_spectrum.pdf", fs_y)
        PlotConstellation(x_downsample, "x_downsample", "
            # → x_downsample_constellation.pdf")
    return x_downsample, fs_y

```

```

87 def Demodulation(x, fs):
88     """
89     Demodula con discriminador polar
90
91     Parametros:
92         xc: Señal modulada.
93         fs: Frecuencia de muestreo
94     """
95     # Derivador y detector de envolvente.
96     # xd = [0, np.diff(x) * fs]
97     # xd = hilbert(xd)
98     # yd = xd - np.mean(xd)
99     # yd = yd/max(abs(yd))
100
101    xd = x[1:] * np.conj(x[:-1]) #
102    yd = np.angle(xd) #
103    return yd
104
105
106 def FilterDeEmphasis(fs, yd): #
107     """
108     Parametros:
109         fs: Frecuencia de muestreo
110         yd: Señal resultante
111     """
112     fs_tau = fs * 75e-6 # Constante de tiempo tau
113     x = np.exp(-1/fs_tau) # Calcula decaimiento del filtro
114     b = [1-x] # Crea el filtro de coeficientes
115     a = [1, -x]
116     yd_filter = lfilter(b, a, yd)
117     return yd_filter
118
119
120 def PlotSpectrum(x, x_name, file_name, fs):
121     """
122     Parametros:
123         x: Señal a realizar el gráfico del espectro
124         x_name: Nombre de la variable, sera mostrada como
125             ↪ label
126         file_name: Nombre del archivo a guardar por defecto
127         fs: Frecuencia de muestreo
128     """
129     # viridis, plasma, inferno, magma
130     plt.specgram(x, NFFT=2048, Fs=fs, cmap='viridis')
131     plt.title(x_name)
132     plt.xlabel("Tiempo_(s)")
133     plt.ylabel("Frecuencia_(Hz)")
134     plt.ylim(-fs/2, fs/2)
135     plt.xlim(0, len(x)/fs)

```

```

135     plt.margins(0.1)
136     plt.ticklabel_format(style='plain', axis='y')
137     plt.colorbar()
138     plt.savefig(file_name, dpi=300, bbox_inches='tight', pad_inches
139             ↪ =0)
140     plt.close()
141
142 def PlotConstelation(x, x_name, file_name):
143     """
144     Parametros:
145         x: Señal a realizar el gráfico de constelación
146         x_name: Nombre de la variable, sera mostrada como
147             ↪ label
148         file_name: Nombre del archivo a guardar
149     """
150     limD = -1.5
151     limU = 1.5
152     plt.scatter(np.real(x[0:50000]), np.imag(
153         x[0:50000]), color="blue", alpha=0.05)
154     plt.grid(alpha=0.25)
155     plt.title(x_name)
156     plt.xlabel("Real")
157     plt.xlim(limD, limU)
158     plt.ylabel("Imag")
159     plt.ylim(limD, limU)
160     plt.savefig(file_name, dpi=300, bbox_inches='tight', pad_inches
161             ↪ =0)
162     plt.close()
163
164 def PlotFilterCharacteristic(coef, fs):
165     freq, response = freqz(coef)
166     plt.semilogy(0.5 * fs * freq / np.pi, np.abs(response), 'b-')
167     plt.grid(alpha=0.25)
168     plt.xlabel('Frecuencia_(Hz)')
169     plt.ylabel('Ganancia')
170     plt.savefig("filter_charac.pdf", bbox_inches='tight', pad_inches
171             ↪ =0)
172     plt.close()
173
174 def DEP(x, x_name, file_name, fs, FM_signal=False):
175     """
176     Parametros:
177         x: Señal a realizar el gráfico DEP
178         x_name: Nombre de la variable, sera mostrada como
179             ↪ label
180         file_name: Nombre del archivo a guardar

```

```

179         fs: Frecuencia de muestreo
180
181     Ver mas:
182         https://matplotlib.org/examples/color/named\_colors.html
183         """
184         plt.psd(x, NFFT=2048, Fs=fs, color="blue")
185         plt.title(x_name)
186         if FM_signal:
187             plt.axvspan(0, 15000, color="red", alpha=0.3)
188             plt.axvspan(19000-500, 19000+500, color="green",
189                         ↪ alpha=0.4)
190             plt.axvspan(19000*2-15000, 19000*2+15000, color="
191                         ↪ orange", alpha=0.3)
192             plt.axvspan(19000*3-1500, 19000*3+1500, color="c",
193                         ↪ alpha=0.3)
194         else:
195             plt.axvspan(-F_BW, F_BW, color="green", alpha=0.3)
196             plt.ticklabel_format(style='plain', axis='y')
197             plt.savefig(file_name, dpi=300, bbox_inches='tight', pad_inches
198                         ↪ =0)
199             plt.close()
200
201     def SaveToFile(yd, file_name, fs):
202         """
203         Parametros:
204             yd: Señal que ya esta adaptada para escuchar
205             file_name: Nombre del archivo a guardar
206             fs: Frecuencia de muestreo
207             """
208         # Acondiciona el sonido para que tenga una frecuencia de
209             ↪ entre 44–48 kHz
210         audio_freq = 44100.0
211         dec_audio = int(fs / audio_freq)
212         fs_audio = int(fs / dec_audio)
213         sound = decimate(yd, dec_audio)
214         print("Frec_audio:", fs_audio)
215
216         # Se escala el volumen del audio
217         sound *= 10000 / np.max(np.abs(sound))
218         # Guarda el sonido en 16-bit con signo
219         wavfile.write(file_name, fs_audio, sound.astype("int16"))
220
221     def PlaySound(yd, fs):
222         """
223         Reproduce el audio que fue grabado previamente.
224
225         Parametros:
```

```
223      yd: señal de audio
224      fs: frecuencia en la que esta muestreada
225      ”””
226      audio_freq = 44100.0
227      dec_audio = int(fs / audio_freq)
228      fs_audio = fs / dec_audio
229      sound = decimate(yd, dec_audio)
230      sd.default.samplerate = fs_audio
231      # print("Frec audio el reproduc: ", sd.default.samplerate)
232      sd.play(sound)
233      sd.wait()
234      return sd
```

B.3. radio.py

```
1 # -*- coding: utf-8 -*-
2
3 import sys
4 import time
5 import rtlsdr
6 import numpy as np
7 from scipy.signal import remez, lfilter, decimate, freqz
8 import sounddevice as sd
9 from threading import Semaphore, Thread, Lock
10 import Queue as queue
11
12 FS = 1140000 # Frecuencia de muestreo
13 # FS = 2.04e6
14 N_SAMPLE = 332800 # Cantidad de muestras
15 F_BW = 180e3
16 AUDIO_FREC = 44100.0
17 # AUDIO_FREC = 22050.0
18 N_TRAPS = 10
19 TIME_BUFFER = 7 # En segundos
20 FRAMES = 3 # Frames contatenado por muestra
21
22 class Streaming:
23     def __init__(self, stationMHz):
24         self.sdr = rtlsdr.RtlSdr()
25         self.validsGains = self.sdr.get_gains()
26         self.indexGain = 10
27
28         self.f_offset = 250000 # Desplazamiento para
29             # capturar
29         self.f_station = stationMHz # Frecuencia de radio
30
31         self.dec_rate = int(FS / F_BW)
32         self.fs_y = FS / (self.dec_rate)
33         self.coef = remez(N_TRAPS, [0, F_BW, F_BW*1.4, FS
34             # /2], [1, 0], Hz=FS)
34         self.expShift = (-1.0j * 2.0 * np.pi * self.f_offset/FS)
35
36     # Se configura los parametros
37     self.dec_audio = int(self.fs_y / AUDIO_FREC)
38     self.stream = sd.OutputStream(device='default',
39         samplerate=int(self.fs_y / self.dec_audio),
40         channels=1, dtype='float32')
41     self.beginListening = 0
42
43     self.soundQueue = queue.Queue()
44     self.samplesQueue = queue.Queue()
```

```

43
44
45     def __del__(self):
46         print("Destructor del Streaming")
47         self.sdr.close()
48         del(self.sdr)
49
50
51     def start(self):
52         self.semaphorePlay = Semaphore(1)
53         self.lockCapture = Lock()
54         self.lockProcessing = Lock()
55         self.lockReproduce = Lock()
56         self.stopStreaming = False
57         self.pauseStreaming = False
58
59         self.setupSDR(FS, self.f_station - self.f_offset, self.
60                         ↪ validsGains[self.indexGain])
61
62         self.cThread = Thread(target=self.captureSamples,
63                               ↪ name='Capturadora de muestras')
64         self.pThread = Thread(target=self.processingSamples,
65                               ↪ name='Procesamiento de muestras')
66         self.rThread = Thread(target=self.reproduce, name='
67                               ↪ Escucha')
68         # Adquiero el semaforo antes de que empiece los hilos,
69         # para que demodListen tenga datos para procesar
70         self.semaphorePlay.acquire()
71         self.stream.start()
72         self.cThread.start()
73         self.pThread.start()
74         self.rThread.start()
75
76     def play(self):
77         """
78             Se llama solo una vez
79         """
80         time.sleep(TIME_BUFFER)
81         print("Reproducindo..")
82         self.semaphorePlay.release()
83
84     def stop(self):
85         self.stopStreaming = True
86         if (self.pauseStreaming == True):
87             self.semaphorePlay.release()

```

```

88         self.lockReproduce.release()
89         self.cThread.join()
90         self.pThread.join()
91         self.rThread.join()
92         print("Stop_Streaming")
93
94
95     def pauseOrPlay(self, station):
96         """
97             Pausa o reanuda, no llamarlo sin antes hacer start()
98
99             Devuelve: False si se reanuda, o True si se pausó
100        """
101    if (self.pauseStreaming == True):
102        # Si esta pausado lo saco de la pausa
103        # y despierto los hilos con la variable condición
104        self.pauseStreaming = False
105        self.setupSDR(FS, station - self.f_offset, self.
106                      → validsGains[self.indexGain])
107        self.lockCapture.release()
108        self.lockProcessing.release()
109        self.lockReproduce.release()
110        self.play()
111        return False
112    else:
113        self.pauseStreaming = True
114        print("Pause_Streaming")
115        return True
116
117    def setupSDR(self, fs, fc, gain):
118        """
119            Parametros:
120                fs: Frecuencia de muestreo a captar
121                fc: Frecuencia del centro de captura
122                gain: Ganancia en db
123
124                self.sdr.sample_rate = fs
125                self.sdr.center_freq = fc
126                self.sdr.gain = gain
127
128
129    def changeGain(self, gain):
130        """
131            Parametros:
132                gain: solo puede valer +1 o -1
133
134            Retorna: True en caso que se pudo cambiar (éxito) y su
135                    → ganancia, False y None en caso de sobrepasar

```

```

    ↪ los límites
135
136     """
137     if (len(self.validsGains) > (self.indexGain + gain)) and ((
138         ↪ self.indexGain + gain) > 0):
139         self.indexGain += gain
140     else:
141         return False, None
142     self.sdr.set_gain(self.validsGains[self.indexGain])
143     return True, self.validsGains[self.indexGain]
144
145
146
147
148     def changeStation(self, MHz):
149         self.f_station = MHz
150
151
152
153
154     def captureSamples(self): #
155         while not (self.stopStreaming):
156             frame = np.array(self.sdr.packed_bytes_to_iq(self.
157                 ↪ sdr.read_bytes(2*N_SAMPLE)))
158             for i in range(FRAMES-1):
159                 frame = np.concatenate((frame, np.array(
160                     ↪ self.sdr.packed_bytes_to_iq(self.sdr.
161                         ↪ read_bytes(2*N_SAMPLE)))))
162             self.samplesQueue.put(frame)
163             if self.pauseStreaming:
164                 self.lockCapture.acquire()
165
166
167     def processingSamples(self): #
168         # Proceso las muestras
169         while not (self.stopStreaming):
170             xc = self.samplesQueue.get().astype("complex64"
171                 ↪ )
172             # Paso a banda base y filtro
173             x_filter = lfilter(self.coef, 1.0, xc * np.exp(self.
174                 ↪ expShift * np.arange(len(xc))))
175             x_downsample = x_filter[0::self.dec_rate]
176             # Demodulo
177             yd_demod = np.angle(x_downsample[1:] * np.conj(
178                 ↪ x_downsample[:-1]))
179             # Deemfasis
180             #x_deemfasis = np.exp(-1/(self.fs_y * 75e-6))
181             #yd = lfilter([1-x_deemfasis], [1, -x_deemfasis],
182             #    ↪ yd_demod)
183             # Diezmo para que pueda reproducir la placa de
184             #    ↪ sonido
185             self.soundQueue.put(decimate(yd_demod, self.
186                 ↪ dec_audio))
187             if self.pauseStreaming:

```

```
173                     self.lockProcessing.acquire()
174
175
176     def reproduce(self): #
177         self.semaphorePlay.acquire()
178         while not (self.stopStreaming):
179             # Cambio de float64 a 32 para stream.write
180             self.stream.write(self.soundQueue.get().astype(
181                 ↪ float32'))
182             if self.pauseStreaming:
183                 self.lockReproduce.acquire()
184                 self.semaphorePlay.acquire()
185
186     if __name__ == '__main__':
187         s = Streaming(99.1e6)
188         s.start()
189         s.play()
190         time.sleep(14)
191         s.stop()
```
