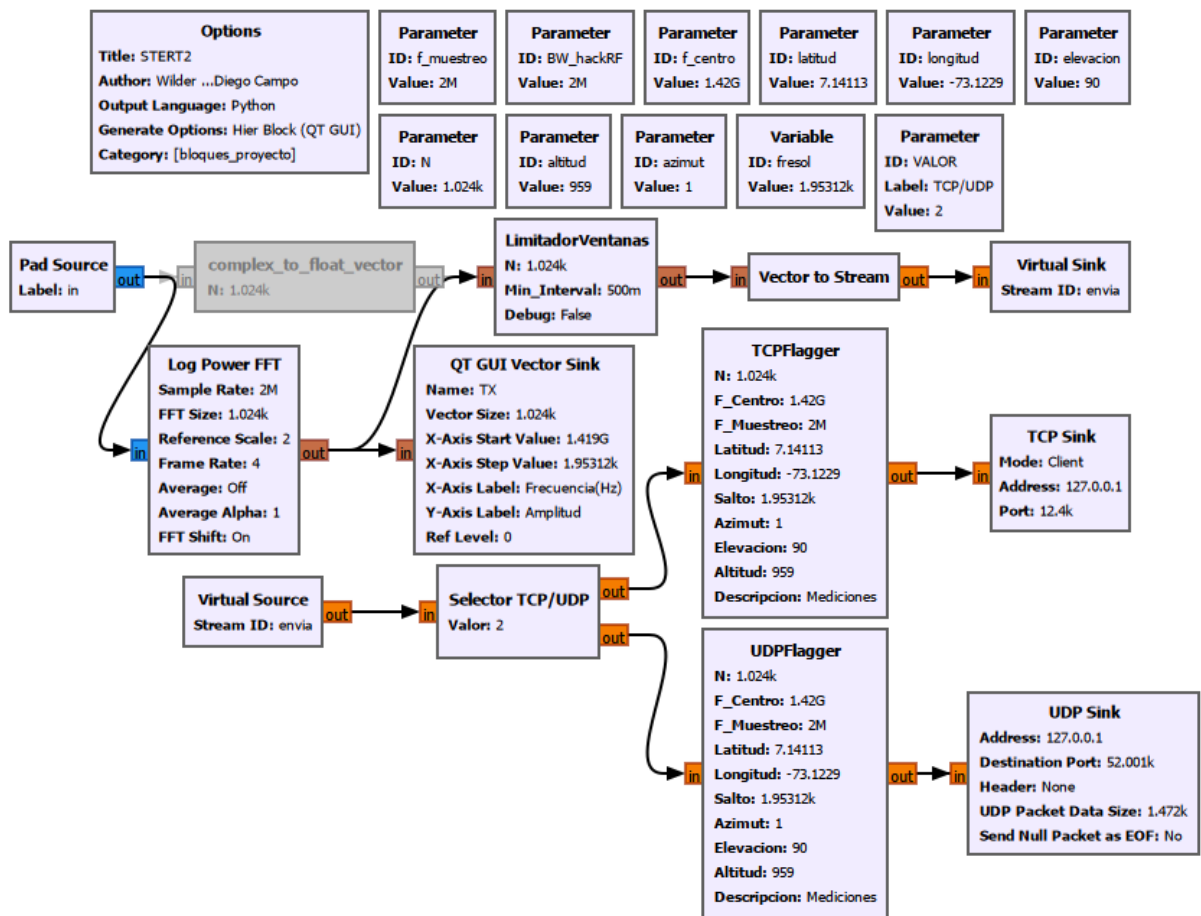


## Apéndice E. Guía en la Conexión de los Bloques en GNU Radio de la Arquitectura

### STERT2

**Figura 1**

*Bloques en GNU Radio para la medición y transmisión del espectro.*



*Nota.* Esta imagen muestra que el Pad Source encapsula varios bloques, agrupándolos en un solo elemento visual en GNU Radio. Fuente: Elaboración propia.

### 1. Creación de Bloques Parameter y Variable

Estos parámetros están definidos por el grupo de investigación RadioGIS y CEMOS para mediciones.

### **1.1 $f_{\text{muestreo}}$**

En ID con  $f_{\text{muestreo}}$ , se define la frecuencia de muestreo del sistema, establecida en 2 MS/s, Value con 2e6, este valor determina cuántas muestras por segundo se están adquiriendo del archivo de audio.

### **1.2 $BW_{\text{hackRF}}$**

En ID con  $BW_{\text{hackRF}}$ , se define el ancho de banda del sistema, establecida en 2 MS/s, Value con 2e6.

### **1.3 $f_{\text{centro}}$**

En ID con  $f_{\text{centro}}$ , representa la frecuencia central del espectro que se desea visualizar. En este caso, se ha fijado en 1.42 GHz, Value con 1420e6 una frecuencia de interés en RadioGis y CEMOS.

### **1.4 *Parameter***

ID = latitud

Value = 7.14113

ID = longitud

Value = -73.1229

ID = elevación

Value = 90

ID = altitud

Value = 959

ID = azimuth

Value = 1

Son variables que no participan directamente en el flujo de señal en GNU Radio, pero acompañan los datos enviados para contextualizar la ubicación y el ángulo de captura del espectro.

Se usan en la tabla de visualización remota en Node-RED.

### ***1.5 fresol***

En ID con fresol, calculado en Value como  $f_{\text{muestreo}} / N$ , este valor representa el espaciamiento entre cada punto en el eje de frecuencias del espectro.

### **1.6 N**

Define el tamaño de la FFT y del vector de muestras que se usan tanto para visualización como para el análisis espectral. Una mayor N brinda mayor resolución en frecuencia, pero también exige más procesamiento.

### **1.7 VALOR**

ID = VALOR, label = TCP/UDP, Value = 2, se usa para dar una orden al bloque Selector TCP/UDP, solo se puede colocar estos valores, Value = 0, Value = 1, Value = 2.

## **2. Flujograma STERT2\_Pad**

### **2.1 Pad Source**

En Label = in, tipo float, vector Length=N, num Streams=1, optional=Required, todo lo que esté conectado a este bloque, lo convierte en un solo bloque que es QT GUI Enviar Datos a la

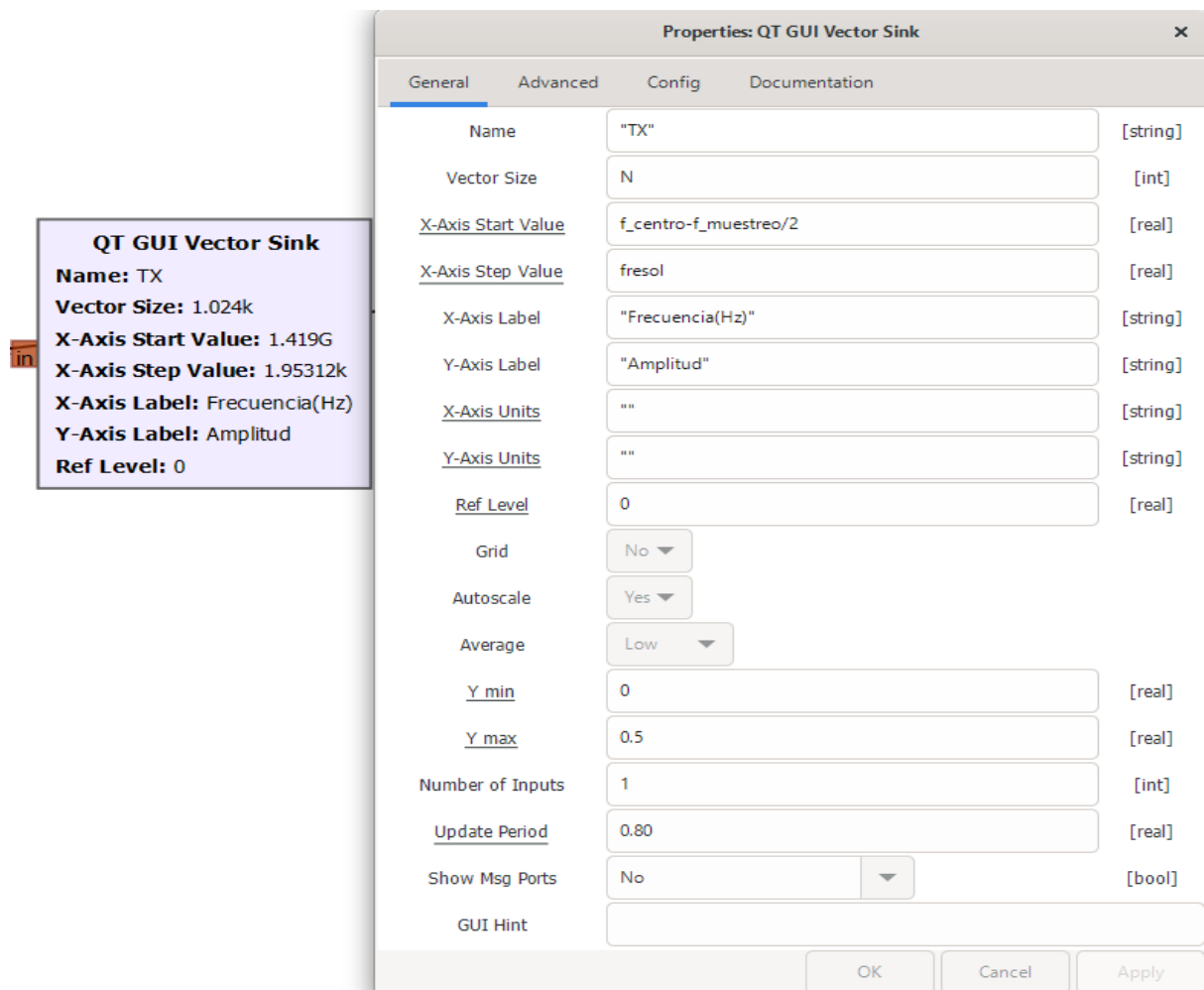
Nube en Node-RED, el cual se ve en el archivo STERT2.grc, el Pad Source se conecta en paralelo a los bloques QT GUI Vector Sink y *LimitadorVentanas* (*Pad Source*, 2022).

## 2.2 QT GUI Vector Sink

El bloque QT GUI Vector Sink recibe vectores de longitud N y los representa instantáneamente en un plano cartesiano X–Y. Cada vector entrante se dibuja como una curva completa, sin necesidad de realizar transformaciones internas (por ejemplo, FFT), pues se espera que el flujo upstream ya haya producido los datos en la forma deseada (p. ej., salida de un bloque Stream to Vector o de un FFT seguido por Complex to Mag<sup>2</sup>), (*QT GUI Vector Sink - GNU Radio*, 2020).

### Figura 2

*Configuración del bloque QT GUI Vector Sink*



*Nota.* Se muestran los parámetros principales.

### 2.2.1 Parámetros Principales

Vector Size, asocia la longitud de cada vector entrante a la variable N. De este modo, si se define  $N = 1024$ , el bloque esperará vectores de 1024 elementos; si cambia a 256, el gráfico se ajustará automáticamente al nuevo tamaño.

X-Axis Start Value, indica el valor inicial del eje X para el primer elemento del vector. En aplicaciones espectrales, se utiliza normalmente

$$Start = f_{centro} - \frac{f_{muestreo}}{2},$$

de manera que el índice 0 del vector corresponda a la frecuencia más baja ( $f_{\text{centro}} - f_{\text{muestreo}}/2$ ).

X-Axis Step Value, define la separación entre puntos sucesivos en el eje X. Habitualmente se calcula como

$$\text{Step} = \frac{f_{\text{muestreo}}}{N} = f_{\text{resol}}$$

de modo que el vector de tamaño N cubra todo el ancho de banda  $f_{\text{muestreo}}$  con resolución constante.

X-Axis Label y Y-Axis Label

Son las etiquetas textuales que aparecen bajo cada eje. En el ejemplo de la Figura 2 se emplea “Frecuencia (Hz)” en X y “Amplitud” en Y, para clarificar que el gráfico muestra la magnitud espectral en función de la frecuencia medida en hertz.

Autoscale, controla si el rango vertical se ajusta automáticamente al vector recibido.

Yes: cada vez que llega un nuevo vector, el eje Y se reescalan según los valores mínimo y máximo del vector.

No: se utilizan los campos Y min y Y max fijos para mantener una escala constante entre múltiples ejecuciones o comparaciones.

Average, determina el grado de suavizado aplicado a la señal mediante un promedio de vectores anteriores.

None: sin suavizado (cada vector se grafica tal cual).

Low/Medium/High: se promedian 2, 3 o más vectores previos para reducir fluctuaciones y ruido.

Update Period, fija el intervalo de tiempo (en segundos) entre cada refresco de la gráfica, independiente de la llegada de datos. Por ejemplo, un valor de 0.80 significa que la ventana se redibuja cada 0.8 s con el último vector recibido, lo cual evita sobrecargar la CPU si el flujo upstream es muy rápido.

Number of Inputs, especifica cuántas series simultáneas puede recibir el bloque. Si se configura en 2, por ejemplo, se pueden conectar dos salidas upstream (p. ej., espectro TCP vs. espectro UDP) y ambas curvas se trazarán en el mismo gráfico con colores o estilos distintos.

### 2.3 LimitadorVentanas

El bloque LimitadorVentanas (clase RateLimiterDiscardVentanas) deja pasar una única ventana cada vez que ha transcurrido el tiempo mínimo definido por min\_interval. Si llegan otras ventanas antes de que se cumpla ese intervalo, se descartan completamente. Esto permite controlar la frecuencia de salida hacia bloques posteriores, como los de visualización, y evita que se saturen cuando los datos entran a alta velocidad (*Embedded Python Block - GNU Radio*, 2019)

### Figura 3

*Código del bloque LimitadorVentanas*

---

```

import time
import numpy as np
from gnuradio import gr

class RateLimiterDiscardVentanas(gr.sync_block):
    """
    Limita la tasa de salida por ventanas completas descartando ventanas si
    llegan antes de `min_interval` segundos desde la última salida.
    """

    def __init__(self, N=1024, min_interval=0.05, debug=False):
        gr.sync_block.__init__(
            self,
            name="LimitadorVentanas",
            in_sig=[(np.float32, int(N))],
            out_sig=[(np.float32, int(N))]
        )
        self.N = int(N)
        self.min_interval = min_interval
        self.last_send_time = 0
        self.debug = debug

    def work(self, input_items, output_items):
        in_vecs = input_items[0]
        out_vecs = output_items[0]

        out_idx = 0
        for vec in in_vecs:
            now = time.time()
            if now - self.last_send_time >= self.min_interval:
                # Pasar ventana completa
                out_vecs[out_idx][:] = vec
                out_idx += 1
                self.last_send_time = now
                if self.debug:
                    print(f"[RateLimiter] ventana pasada a t={now:.3f}")
            else:
                # Descarta ventana entera
                if self.debug:
                    print(f"[RateLimiter] ventana descartada a t={now:.3f}")

        return out_idx # cantidad de ventanas pasadas

```



*Nota.* El bloque extiende `gr.sync_block` de GNU Radio para controlar la tasa de salida de ventanas (vectores), dejando pasar solo una ventana cada vez que se cumple el intervalo mínimo (`min_interval`), y descartando las demás si llegan demasiado pronto.

### 2.3.1 Constructor y Parámetros

El bloque recibe tres parámetros principales:

`N`: tamaño del vector de entrada y salida.

`min_interval`: intervalo mínimo de tiempo (en segundos) que debe transcurrir para permitir el paso de una nueva ventana.

`debug`: valor booleano que, si se activa, imprime en consola si una ventana fue pasada o descartada, junto con la marca de tiempo.

## Figura 4

*Configuración de entradas y salidas*

```
gr.sync_block.__init__(
    self,
    name="LimitadorVentanas",
    in_sig=[(np.float32, int(N))],
    out_sig=[(np.float32, int(N))]
)
```

*Nota.* En el constructor, se definen las señales de entrada y salida como vectores de tipo `float32` con longitud `N`.

### 2.3.3 Lógica del Método `work()`

El método `work()` recorre las ventanas entrantes (`input_items[0]`), y por cada una verifica si ya ha pasado el tiempo necesario desde la última enviada (`self.last_send_time`).

## Figura 5

*Comportamiento básico.*

```

    if now - self.last_send_time >= self.min_interval:
        # Pasar ventana completa
        out_vecs[out_idx][:] = vec
        out_idx += 1
        self.last_send_time = now
        if self.debug:
            print(f"[RateLimiter] ventana pasada a t={now:.3f}")
    else:
        # Descarta ventana entera
        if self.debug:
            print(f"[RateLimiter] ventana descartada a t={now:.3f}")

    return out_idx # cantidad de ventanas pasadas

```

*Nota.* Si se cumple ese tiempo ( $\text{now} - \text{self.last\_send\_time} \geq \text{self.min\_interval}$ ), la ventana se copia al vector de salida; si no, se ignora.

Al final, el bloque retorna cuántas ventanas fueron pasadas (`out_idx`), para que GNU Radio propague correctamente los datos, una vez validada, su salida se conecta al bloque Vector to Stream.

**Tabla 1**

*Ritmo de actualización según el valor de `Min_Interval`.*

<b>Min_Interval (s)</b>	<b>Ventanas por segunda recomendación</b>
0.05	20 Hz · apariencia casi continua
0.10	10 Hz · fluido, ligera intermitencia
0.20	5 Hz · cambios evidentes
0.30	3.3 Hz · transición lenta

<b>0.50</b>	<b>2 Hz · valor adoptado</b>
0.80	1.25 Hz · casi estático
1.00	1 Hz · actualización puntual

---

*Nota.* El usuario puede ajustar Min\_Interval según la fluidez deseada, los valores menores aportan dinamismo a costa de mayor carga de datos; valores mayores reducen esa carga, pero hacen que la gráfica se actualice con menor frecuencia.

## 2.4 Vector to Stream

El Vector to Stream toma un vector de N muestras y lo convierte en N salidas sucesivas, liberando cada muestra una a una en el orden original. Así, adapta la salida de bloques que generan vectores (por ejemplo, FFT o Stream to Vector) a la entrada de bloques que procesan muestras individuales (*Vector to Stream - GNU Radio*, 2020).

### 2.4.1 Parámetros Principales

Vector Length (N): número de muestras que contiene cada vector de entrada.

Output Type: tipo de dato de las muestras emitidas.

Este bloque se conecta al bloque Selector TCP/UDP.

## 2.5 Selector TCP/UDP

El Selector TCP/UDP decide por dónde enviar cada muestra de entrada:

Si valor = 0, solo pasa los datos por la salida TCP (out0).

Si valor = 1, solo pasa los datos por la salida UDP (out1).

Si valor = 2, pasa los datos por ambas salidas (*Embedded Python Block - GNU Radio*, 2019).

**Figura 6***Configuración del bloque Selector TCP/UDP*


---

```

import numpy as np
from gnuradio import gr

class SelectorTCPUDP(gr.sync_block):
    """
    Selector TCP/UDP simple que pasa datos solo por las salidas indicadas:
    valor=0 → sólo TCP (out0)
    valor=1 → sólo UDP (out1)
    valor=2 → TCP + UDP (out0 + out1)
    """

    def __init__(self, valor=0):
        if valor not in (0, 1, 2):
            raise ValueError("Parámetro 'valor' debe ser 0, 1 o 2")
        gr.sync_block.__init__(
            self,
            name='Selector TCP/UDP',
            in_sig=[np.float32],
            out_sig=[np.float32, np.float32]
        )
        self.valor = int(valor)

    def work(self, input_items, output_items):
        in0 = input_items[0]
        out_tcp = output_items[0]
        out_udp = output_items[1]
        n = len(in0)

        if self.valor in (0, 2):
            out_tcp[:n] = in0
            # Si no pasa, no toca out_tcp

        if self.valor in (1, 2):
            out_udp[:n] = in0
            # Si no pasa, no toca out_udp

        return n

```

*Nota.* Este bloque enruta las muestras entrantes a la salida TCP (out0), a la salida UDP (out1) o a ambas, según el parámetro valor.

### 2.5.1 Constructor y Parámetros

En el constructor se comprueba que valor sea 0, 1 o 2, y se inicializa el bloque con una entrada (in\_sig=[np.float32]) y dos salidas (out\_sig=[np.float32, np.float32]).

#### Figura 7

*Indicación de valor*

```
def __init__(self, valor=0):
    if valor not in (0, 1, 2):
        raise ValueError("Parámetro 'valor' debe ser 0, 1 o 2")
    gr.sync_block.__init__(
        self,
        name='Selector TCP/UDP',
        in_sig=[np.float32],
        out_sig=[np.float32, np.float32]
    )
    self.valor = int(valor)
```

*Nota.* Dentro de work(), el bloque copia todas las muestras de input\_items[0] a output\_items[0] (TCP) si valor es 0 o 2, y a output\_items[1] (UDP) si valor es 1 o 2. Al final retorna n, el número de muestras procesadas.

La salida del Selector TCP/UDP se conecta a los bloques TCPFlagger (out0) y UDPFlagger (out1).

### 2.6 TCPFlagger

Agrupar las N muestras de entrada.

Cuando acumula N, crea un paquete de espectro:

-1.0 | f\_inicial | f\_paso | f\_centro | N muestras | f\_final | -2.0.

A continuación, agrega un paquete de constantes con latitud, longitud, salto, azimut, elevación, altitud y la descripción en ASCII, delimitado por  $-3.0 \dots -4.0$ .

Ambos paquetes se almacenan en una cola y se van entregando al flujo de salida (*Embedded Python Block - GNU Radio*, 2019).

### Figura 7

*Código del bloque TCPFlagger Parte A.*

```

1  #####
2  # TCPFlaggerHiLo32                                     #
3  # -----                                             #
4  # • Recibe N muestras de espectro en float32          #
5  # • Inserta un timestamp absoluto en ms (64 bit)      #
6  #   empaquetado en dos float32 (hi, lo) antes de     #
7  #   los metadatos (-3 -4).                            #
8  # • Inserta también el tamaño de ventana N en         #
9  #   metadatos.                                         #
10 # • Conserva la estructura de banderas y muestras.    #
11 # • Todo sale en float32 compatible con sinks estándar. #
12 #####
13 import time
14 import numpy as np
15 from gnuradio import gr
16
```

*Nota.* Este bloque corresponde a la parte A del sistema.

### Figura 8

*Código del bloque TCPFlagger Parte B.*

```

17 class blk(gr.sync_block):
18     """
19     Bloque TCPFlagger:
20     - Acumula N muestras de espectro (float32)
21     - Inserta banderas FS/FE para espectro
22     - Captura timestamp t0 en ms dividido en hi/lo
23     - Inserta banderas MS/ME para metadatos, incluyendo:
24         • hi, lo
25         • tamaño de ventana N
26         • metadatos fijos (geolocalización y descripción)
27     - Emite todo por TCP en un único paquete float32[]
28     """
29     def __init__(self,
30                 N=1024,
31                 f_centro=1.42e9,
32                 f_muestreo=2e6,
33                 latitud=7.141879,
34                 longitud=-73.122193,
35                 salto=4000,
36                 azimut=0,
37                 elevacion=90,
38                 altitud=959,
39                 descripcion="UIS"):
40         gr.sync_block.__init__(
41             self,
42             name="TCPFlagger",
43             in_sig=[np.float32],
44             out_sig=[np.float32]
45         )
46
47         # Parámetros de espectro
48         self.N = int(N)
49         self.fc = float(f_centro)
50         self.fs = float(f_muestreo)
51         self.fp = self.fs / self.N
52         self.fi = self.fc - self.fs/2
53         self.ff = self.fc + self.fs/2
54
55         # Metadatos fijos: lat, lon, salto, azimut, elev, alt y descripción
56         self.meta_fixed = np.array([
57             float(latitud), float(longitud),
58             float(salto), float(azimut),
59             float(elevacion), float(altitud),
60             *[float(b) for b in descripcion.encode('utf-8')]
61         ], dtype=np.float32)
62
63         # Banderas especiales: FS/FE (espectro), MS/ME (metadatos)
64         self.FS, self.FE = -1.0, -2.0
65         self.MS, self.ME = -3.0, -4.0
66
67         # Buffer temporal de muestras
68         self._buf = []
69
70     def _split_hi_lo(self, t_ms):
71         """ Divide t_ms (float64) en dos float32: hi = t_ms//65536, lo = resto """
72         hi = np.float32(np.floor(t_ms / 65536.0))
73         lo = np.float32(t_ms - hi * 65536.0)
74         return hi, lo

```

*Nota.* Este bloque corresponde a la parte B del sistema.

**Figura 9***Código del bloque TCPFlagger Parte C.*

```

75
76     def work(self, input_items, output_items):
77         inp = input_items[0]
78         out = output_items[0]
79         p_out = 0
80
81         for sample in inp:
82             self._buf.append(sample)
83
84             if len(self._buf) == self.N:
85                 # 1) Construir bloque espectral: [FS, fi, fp, fc] + muestras + [ff, FE]
86                 spec = np.concatenate([
87                     np.array([self.FS, self.fi, self.fp, self.fc], dtype=np.float32),
88                     np.array(self._buf, dtype=np.float32),
89                     np.array([self.ff, self.FE], dtype=np.float32)
90                 ])
91
92                 # Limpiar buffer para próxima ventana
93                 self._buf.clear()
94
95                 # 2) Timestamp actual en ms (float64) y dividirlo
96                 t0_ms = time.time() * 1000.0
97                 hi, lo = self._split_hi_lo(t0_ms)
98
99                 # 3) Construir bloque de metadatos:
100                 # [MS, hi, lo, N] + meta_fixed + [ME]
101                 meta = np.concatenate([
102                     np.array([self.MS, hi, lo, np.float32(self.N)], dtype=np.float32),
103                     self.meta_fixed,
104                     np.array([self.ME], dtype=np.float32)
105                 ])
106
107                 # 4) Empaquetado final y envío
108                 pkt = np.concatenate((spec, meta))
109                 n = len(pkt)
110                 if p_out + n > len(out):
111                     break
112                 out[p_out:p_out+n] = pkt
113                 p_out += n
114
115         return p_out
116

```



*Nota.* Este bloque enruta las muestras entrantes a la salida TCP (out0), a la salida UDP (out1) o a ambas, según el parámetro valor.

### **2.6.1 Parámetros Clave**

N, son las Muestras por ventana (p. ej., 1024).

f\_centro, es la frecuencia central (Hz).

f\_muestreo, es el ancho de banda total (Hz).

Latitud, longitud, salto, azimut, elevación, altitud, son los Metadatos que acompañan cada espectro.

### **2.6.2 Lógica del Método *general\_work()***

Vacía primero la cola pendiente en el búfer de salida.

Consume muestras hasta completar N.

Cuando N está lleno, arma y encola los dos paquetes descritos.

Copia al búfer de salida todo lo que quepa y devuelve la cantidad producida.

La salida del **TCPFlagger** se conecta al bloque **TCP Sink**.

## **2.7 UDFlagger**

Opera de forma idéntica al TCPFlagger: acumula N muestras, construye el paquete de espectro seguido del paquete de constantes y los envía en orden al flujo de salida (*Embedded Python Block - GNU Radio*, 2019).

### **Figura 10**

*Código del bloque UDPFlagger parte A.*

```

1  # UDPFlagger: inserta banderas, timestamp y tamaño de ventana en cada
2  # ventana espectral antes de enviarla por UDP
3
4  import time                # Para obtener tiempo actual en segundos
5  import numpy as np        # Para manipulación de arrays numéricos
6  from gnuradio import gr   # Base de bloques GNU Radio
7
8  class blk(gr.sync_block):
9      """
10     Bloque UDPFlagger:
11     - Acumula N muestras de espectro (float32)
12     - Inserta banderas de inicio/fin de espectro
13     - Captura timestamp t0 en ms dividido en hi/lo
14     - Inserta bandera de inicio/fin de metadatos, incluye:
15         • timestamp hi/lo
16         • tamaño de ventana N
17         • metadatos fijos (geolocalización y descripción)
18     - Envía todo por UDP en un único paquete float32[]
19     """
20     def __init__(self,
21                 N=1024,
22                 f_centro=1.42e9,
23                 f_muestreo=2e6,
24                 latitud=7.141879,
25                 longitud=-73.122193,
26                 salto=4000,
27                 azimut=0,
28                 elevacion=90,
29                 altitud=959,
30                 descripcion="UIS"):
31         # 1) Inicializar bloque: 1 entrada float32, 1 salida float32
32         gr.sync_block.__init__(
33             self,
34             name="UDPFlagger",
35             in_sig=[np.float32],
36             out_sig=[np.float32]
37         )
38
39         # 2) Parámetros de configuración básicos
40         self.N = int(N)                # Número de muestras por ventana

```

*Nota.* Imagen correspondiente a la parte A del sistema UDPFlagger.

**Figura 11**

*Código del bloque UDPFlagger parte B.*

```

41     self.fc = float(f_centro)          # Frecuencia central (Hz)
42     self.fs = float(f_muestreo)       # Frecuencia de muestreo (Hz)
43     self.fp = self.fs / self.N        # Paso de frecuencia (Hz)
44     self.fi = self.fc - self.fs/2     # Frecuencia inicial del espectro
45     self.ff = self.fc + self.fs/2     # Frecuencia final del espectro
46
47     # 3) Metadatos fijos: latitud, longitud, salto, azimut, elevación,
48     #     altitud y cada byte de la descripción como float32
49     self.meta_fixed = np.array(
50         [latitud, longitud, salto, azimut, elevacion, altitud] +
51         [float(b) for b in descripcion.encode('utf-8')],
52         dtype=np.float32
53     )
54
55     # 4) Definir banderas especiales
56     self.FS, self.FE = -1.0, -2.0     # Start/End espectro
57     self.MS, self.ME = -3.0, -4.0     # Start/End metadatos
58
59     # 5) Buffer temporal para acumular muestras hasta N
60     self._buf = []
61
62     def _split(self, t_ms):
63         """
64         Divide un timestamp en milisegundos (t_ms) en dos floats 32-bit:
65         - hi = floor(t_ms / 65536)
66         - lo = t_ms mod 65536
67         Evita pérdida de precisión al enviar por UDP como float32.
68         """
69         hi = np.float32(np.floor(t_ms / 65536.0))
70         lo = np.float32(t_ms - hi * 65536.0)
71         return hi, lo
72
73     def work(self, ins, outs):
74         """
75         work: llamado con cada bloque de muestras entrantes.
76         - ins[0]: array de entrada (float32)
77         - outs[0]: array de salida donde escribimos el paquete UDP
78         Devuelve número de floats escritos en outs[0].
79         """
80         inp = ins[0]          # Muestras que llegan desde el flujo
81         out = outs[0]         # Buffer donde construiremos el paquete
82         p = 0                 # Índice de escritura en 'out'
83
84         # 6) Procesar cada muestra entrante
85         for s in inp:
86             self._buf.append(s)          # Añadir muestra al buffer
87             # 7) Cuando acumulamos N muestras, construimos y enviamos paquete
88             if len(self._buf) == self.N:
89                 # 7.1) Bloque espectral:
90                 #     [FS, fi, fp, fc] + N muestras + [ff, FE]
91                 spec = np.concatenate([
92                     np.array([self.FS, self.fi, self.fp, self.fc], dtype=np.float32),
93                     np.array(self._buf, dtype=np.float32),
94                     np.array([self.ff, self.FE], dtype=np.float32)
95                 ], dtype=np.float32)
96
97                 # 7.2) Limpiar buffer interno para próxima ventana
98                 self._buf.clear()
99

```

*Nota.* Imagen correspondiente a la parte B del sistema UDPFlagger.

**Figura 12***Código del bloque UDPFlagger parte C.*

```

100 # 7.3) Capturar timestamp actual en ms y dividirlo
101 t0_ms = time.time() * 1000.0 # Tiempo en ms (float64)
102 hi, lo = self._split(t0_ms) # Partes hi/lo en float32
103
104 # 7.4) Construir bloque de metadatos:
105 #     [MS, hi, lo, N] + meta_fixed + [ME]
106 #     Insertamos self.N para que Node-RED lo reciba
107 meta = np.concatenate([
108     np.array([self.MS, hi, lo, np.float32(self.N)], dtype=np.float32),
109     self.meta_fixed,
110     np.array([self.ME], dtype=np.float32)
111 ], dtype=np.float32)
112
113 # 7.5) Empaquetado final: espectro + metadatos
114 pkt = np.concatenate([spec, meta], dtype=np.float32)
115 n = len(pkt) # Número total de floats
116
117 # 7.6) Verificar que quepa en el buffer de salida
118 if p + n > len(out):
119     break # Si no cabe, salimos
120
121 # 7.7) Copiar el paquete completo en 'out'
122 out[p:p+n] = pkt
123 p += n # Avanzar el puntero
124
125 # 8) Retornar la cantidad de floats escritos
126 return p
127

```

*Nota.* Inserta las mismas banderas y metadatos que el TCPFlagger, pero prepara los datos para envío por UDP.

### 2.7.1 Parámetros Clave

Hereda los mismos argumentos (N, f\_centro, f\_muestreo, metadatos de localización y descripción).

### 2.7.2 Lógica del Método *general\_work()*

La secuencia de pasos es la misma que en el TCPFlagger, garantizando que la estructura de datos sea compatible tanto para UDP como para TCP.

La salida del UDPFlagger se conecta al bloque UDP Sink.

## 2.8 TCP Sink

Recibe la salida del TCPFlagger, abre un socket en modo cliente TCP y envía la secuencia de muestras en el mismo orden en que las recibe al visor de Node-RED a través de la conexión TCP (*TCP Sink*, 2020).

### 2.8.1 Parámetros Principales

Dirección IP ejemplo 127.0.0.1

Puerto de destino 12400.

## 2.9 UDP Sink

Recibe la salida del **UDPFlagger** y transmite los datos hacia Node-RED en tiempo real.

Dirección IP ejemplo 127.0.0.1

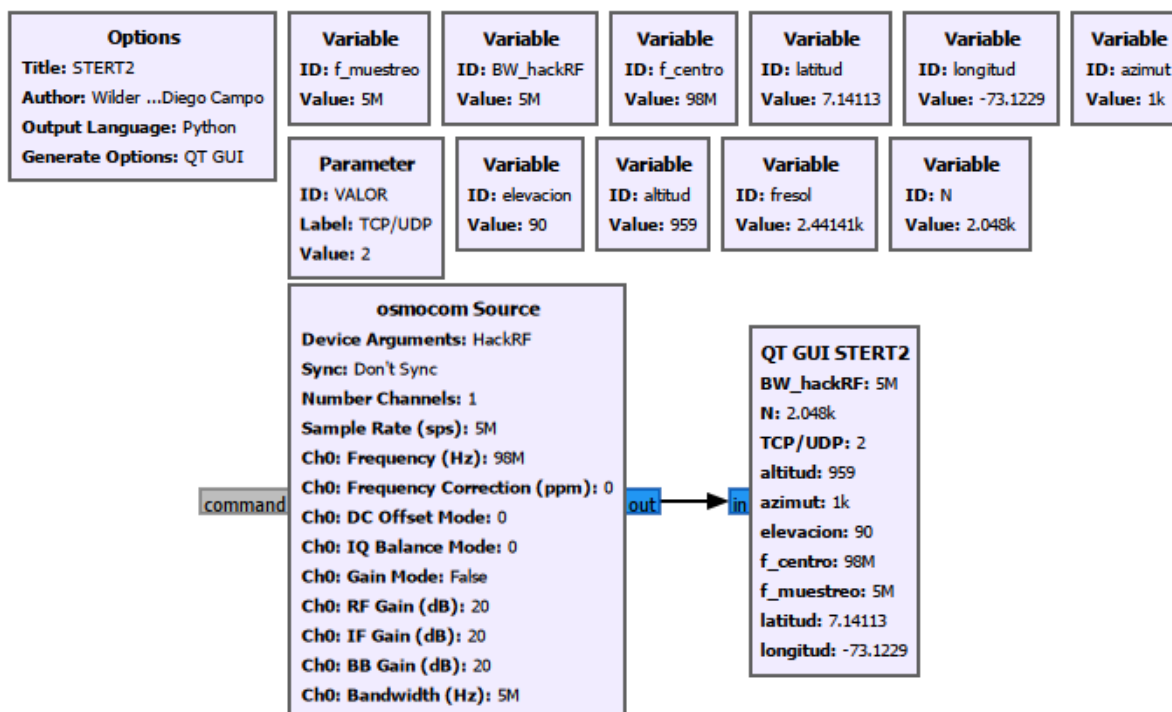
Puerto de destino 52001

Se conecta a la red mediante **UDP** sin hand-shake (*UDP Sink*, 2023).

## 3. Flujograma STERT2

### Figura 13

*Ruta de señal a través de Virtual Sink / Virtual Source en el Flujograma STERT2 en un bloque.*



*Nota.* El bloque QT GUI Enviar Datos a la Nube en Node-RED envía cualquier señal que se conecte al Virtual Sink. El Virtual Source funciona como la señal  $x(t)$  que alimenta el sistema, enviando esa señal al bloque para que la mande a la nube. Así, puedes cambiar la señal fácilmente sin tocar nada más

### 3.1 Creación de Bloques Parameter y Variable

Son los mismos que en el punto 1.

### 3.2 Osmocom Source

El Osmocom Source recibe el espectro radioeléctrico captado por el SDR, lo digitaliza en muestras IQ complejas según la frecuencia central y la tasa de muestreo configuradas, y entrega ese flujo baseband al resto del diagrama; su salida se conecta inmediatamente al bloque Log Power FFT (o al bloque Complex to Float/Vector, según la ruta de procesamiento elegida).

**Figura 14**

*Configuración del bloque Osmocom Source*



Properties: osmocom Source

General

Advanced

Documentation

Output Type

Complex Float32

Device Arguments

"HackRF"

[string]

Sync

Don't Sync

Number MBoards

1

[int]

MB0: Clock Source

Default

[string]

MB0: Time Source

Default

[string]

Number Channels

1

[int]

Sample Rate (sps)

f\_muestreo

[real]

Ch0: Frequency (Hz)

f\_centro

[real]

Ch0: Frequency Correction (ppm)

0

[real]

Ch0: DC Offset Mode

0

[int]

Ch0: IQ Balance Mode

0

[int]

Ch0: Gain Mode

False

[bool]

Ch0: RF Gain (dB)

20

[real]

Ch0: IF Gain (dB)

20

[real]

Ch0: BB Gain (dB)

20

[real]

Ch0: Antenna

[string]

Ch0: Bandwidth (Hz)

BW\_hackRF

[real]

OK

Cancel

Apply

*Nota.* Captura muestras IQ directamente desde un dispositivo SDR (HackRF One).

3.3 Log Power FFT

El bloque Log Power FFT convierte una secuencia de muestras complejas (la envolvente que entrega el *osmocom source* de la SDR) en un espectro de potencia en dB. De esta forma se vuelven evidentes las variaciones de nivel a lo largo de la banda de interés y se facilita la comparación entre protocolos de transmisión (*Log Power FFT*, 2025).

3.3.1 Trayectos de Datos

Tabla 2

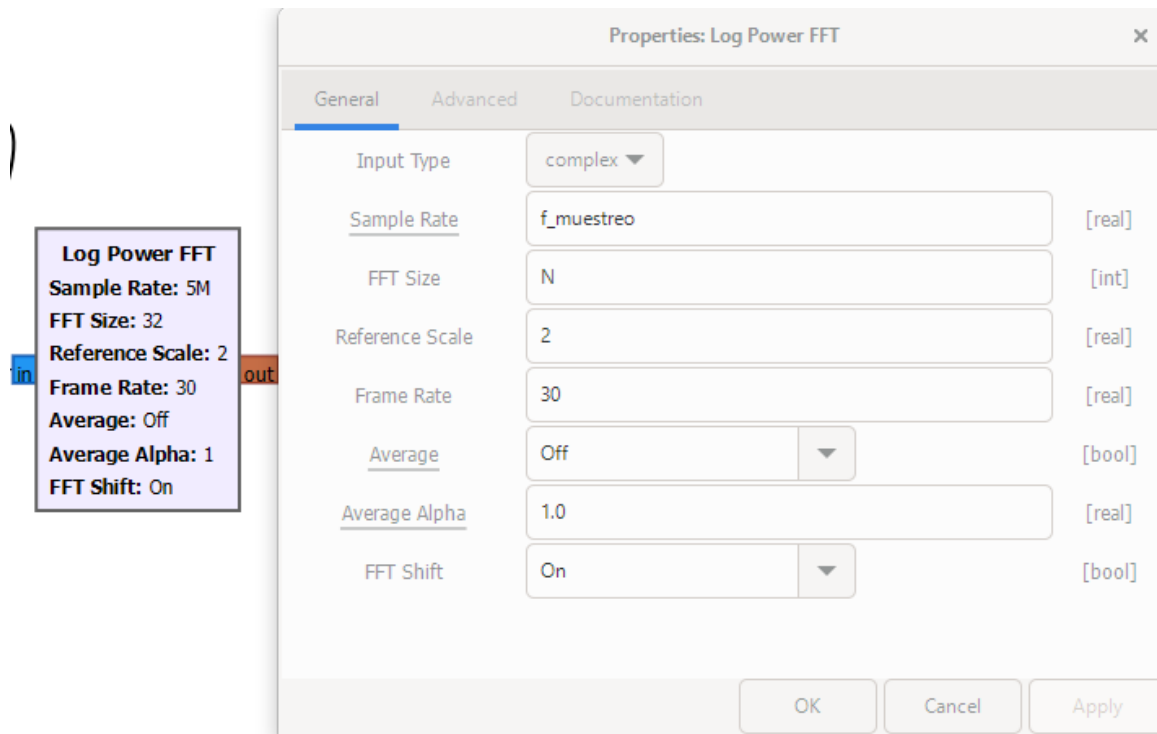
Puertos de Entrada y Salida del Bloque Log Power FFT

Puerto	Nombre	Firma oficial	Tipo	Descripción
Entrada	Muestras complejas (real or complex input)	real or complex input	complex64	Salida del bloque SDR; corresponde a la señal en el dominio temporal antes del análisis espectral.
Salida	Potencia espectral (dB) ( $\log_{10}( fft ^2)$ )	fft	float 32 (vector)	Resultado $\log_{10}( fft ^2)$

*Nota.* La “firma oficial” y los tipos de datos se han extraído de la documentación del bloque Log Power FFT de GNU Radio (*Log Power FFT*, 2025).

Figura 15

Configuración del bloque Log Power FFT



*Nota.* Calcula la FFT sobre ventanas de  $N$  muestras, convierte cada bin a potencia en dB (escala logarítmica) y reordena el espectro con FFT Shift (*Frequency Bin*, 2025).

### 3.3.2 Etapas de Procesamiento Internas

#### 3.3.2.1 Ventaneo. $x_w[n] = w[n] x[n]$

Se aplica una función de ventana (Hann por defecto) para reducir la fuga espectral al procesar bloques de longitud  $N$  (*Antoniou, Andreas*, 2018).

#### 3.3.2.2 Transformada Rápida de Fourier.

$$X[k] = \sum_{n=0}^{N-1} x_w[n] e^{-j 2\pi kn/N}$$

Calcula los coeficientes complejos  $X[k]$  en el dominio de la frecuencia (*Muthuswamy*, 2021).

**3.3.2.3 Potencia Lineal.**

$$P_{lin}[k] = |X[k]|^2$$

Obtiene la densidad espectral de potencia en escala lineal (Hsu, Hwei P, 2020).

**3.3.2.4 Escala Logarítmica.**

$$P_{dB}[k] = 10\log_{10}(P_{lin}[k]) = 20\log_{10}|X[k]|$$

**3.3.2.5 Resolución en Frecuencia.**

$\Delta f$  es la resolución espectral (separación en Hz entre puntos contiguos del eje de frecuencia):

$$\Delta f = \frac{f_{muestreo}}{N} \text{ (Hz por bin)}$$

Cada “bin” corresponde a un segmento de ancho  $\Delta f$  donde  $f_{muestreo}$  es la tasa de muestreo del SDR y  $N$  el tamaño de la FFT (*Frequency Bin*, 2025).

No debe confundirse con el Frame Rate (tramas por segundo), que indica cuántas FFT completas de  $N$  bins se generan por segundo y se ajusta mediante el parámetro Frame Rate del bloque Log Power FFT.

Parámetro Frame Rate (Output frame rate) controla cuántos vectores completos de  $N$  bins entrega el bloque por segundo.

**3.3.2.6 Frame Rate (tramas/s).**

$$FrameRate = \frac{f_{muestreo}}{DN}$$

$D$  es el factor de decimación (parámetro *Decimation*), que reduce la velocidad de salida del bloque y  $N$  el número de puntos de la FFT.

Con ello, para un muestreo de 2 MS/s y  $N = 1024$ , al fijar Frame Rate en 30 frames/s, queda.

$$D = \frac{f_{\text{muestreo}}}{\text{FrameRate} \times N} \approx 65$$

De esta manera queda claro que  $\Delta f$  (detalle en el dominio de la frecuencia) y Frame Rate (velocidad de entrega de cada vector de N bins) se ajustan por separado.

La salida de Log Power FFT se conecta al Virtual Sink.

### 3.4 Virtual Sink

Actúa como “puerta de entrada” para cualquier flujo de muestras. Todo lo que llega a este bloque se redirige, a través del canal virtual, al bloque Virtual Source correspondiente (*Virtual Sink*, 2022).

### 3.5 Virtual Source

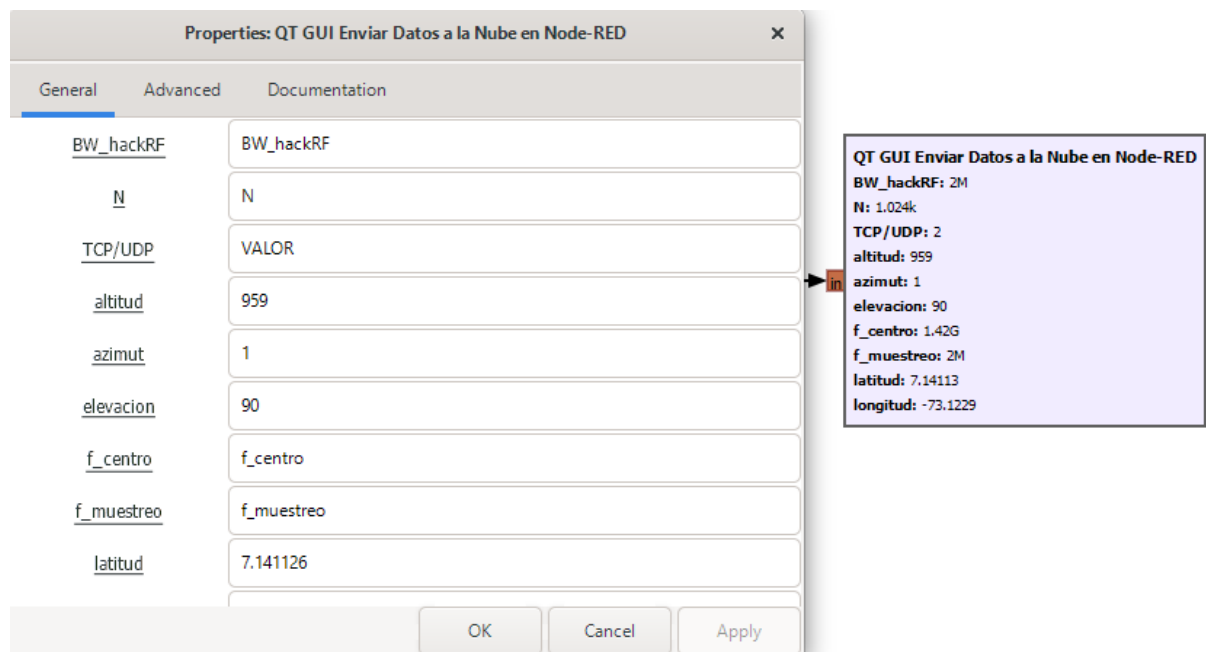
Están conectados inalámbricamente con virtual Sink. El Virtual Source se añadió, para verlo como cualquier señal  $x(t)$ , significa que puede ser cualquier señal de entrada y sirve para alimentar el *bloque QT GUI Enviar Datos a la Nube en Node-RED* (*Virtual Source*, 2022).

### 3.6 QT GUI Enviar Datos a la Nube en Node-RED

Es un bloque que se crea del conjunto de todos los bloques de un flujograma, para poder cambiar variables de una forma fácil, Los valores ajustados en este panel se vinculan a las variables globales de GRC y alimentan, en tiempo real, a los bloques TCPFlagger y UDPFlagger, que los envían posteriormente a los sockets correspondientes en Node-RED (*Pad Source*, 2022).

## Figura 16

*Panel de control QT GUI Enviar Datos a la Nube en Node-RED.*



*Nota.* Agrupa de forma gráfica las variables de transmisión (ancho de banda, frecuencia, coordenadas, protocolo, etc.) y las pone a disposición de Node-RED para envío de datos.

### 3.7 Complex to Float/Vector

Toma un stream de muestras complejas y, cada vez que acumula N muestras, calcula su módulo (valor absoluto) y las almacena en un vector de float32 de longitud N.

La salida de Complex to Float/Vector se conecta al bloque Virtual Sink (*Embedded Python Block* - GNU Radio, 2019).

#### Figura 17

*Configuración del bloque Complex to Float/Vector*

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#
# GNU Radio Python Block: complex_to_float_vector
#
# - 1 entrada: stream de complex64
# - 1 salida: vector float32 de longitud fija N
#

import numpy as np
from gnuradio import gr

class complex_to_float_vector(gr.basic_block):
    def __init__(self, N=1024):
        gr.basic_block.__init__(
            self,
            name="complex_to_float_vector",
            in_sig=[np.complex64],
            out_sig=[(np.float32, int(N))]
        )
        self.N = int(N)
        self.buffer = np.zeros(self.N, dtype=np.float32)

    def general_work(self, input_items, output_items):
        in0 = input_items[0]
        out0 = output_items[0]
        n_input = len(in0)

        if n_input < self.N:
            return 0

        slice_in = in0[:self.N]
        np.abs(slice_in, out=self.buffer)
        out0[0][:] = self.buffer

        self.consume(0, self.N)
        return 1
```

*Nota.* Convierte un flujo complejo (complex64) en ventanas de tipo float32 de longitud fija N, agrupando N muestras consecutivas en un vector.

3.8 Bloque Options

El bloque **Options** define los parámetros globales del flujo en GNU Radio Companion: solo se usa uno por diagrama y sirve para fijar metadatos (título, autor, descripción), tamaño de ventana y otras opciones que afectan cómo se genera y organiza el proyecto (*Options - GNU Radio*, 2025).

Figura 18

*Configuración de Options*

Properties: Options

General

Advanced

Documentation

ID	Nube	
Title	Enviar Datos a la Nube en Node-RED	[string]
Author	Wilder Carrillo Diego Campo	[string]
Copyright		[string]
Description		[string]
Output Language	Python	
Generate Options	Hier Block (QT GUI)	
Category	[bloques_proyecto]	[string]

OK

Cancel

Apply



*Nota.* La opción Generate Options: Hier Block (QT GUI) indica que este bloque se expandirá como un sub-flujo jerárquico capaz de contener elementos de interfaz Qt, mientras que Category: [bloques\_proyecto] lo sitúa dentro de la sección “bloques\_proyecto” en la paleta de GRC.

### 3.9 Bloques Para Una Señal Simulada

#### 3.9.1 Configuración del Bloque Wav File Source

El bloque Wav File Source se emplea como fuente de señal simulada, en reemplazo del hardware SDR durante las fases iniciales del prototipo. Esta solución permitió avanzar en el desarrollo mientras se resolvía la indisponibilidad del dispositivo HackRF One. A continuación, se detallan los parámetros configurados para este bloque en GNU Radio:

Nombre del bloque: Wav File Source Archivo cargado: voz\_grabada.wav Ruta del archivo: [Ver enlace de descarga en el anexo digital] Repetición (Repeat): Yes Formato de datos: Float Uso previsto: Proporcionar una señal continua durante toda la ejecución del flujo, permitiendo simular condiciones reales de adquisición (*Wav File Source*, 2022). Nota: El archivo voz\_grabada.wav se encuentra disponible como archivo adjunto y puede visualizarse en la base de datos de la biblioteca UIS o mediante el siguiente enlace: [https://drive.google.com/drive/u/1/folders/1yguh\\_6UKcVcPjRBp\\_0-Y\\_pJeQ0QAbCVv](https://drive.google.com/drive/u/1/folders/1yguh_6UKcVcPjRBp_0-Y_pJeQ0QAbCVv)

#### 3.9.2 Null Source y Float To Complex

El bloque Null Source se emplea para generar una señal nula que se combina con la salida real del archivo .wav usando el bloque Float To Complex. Esta combinación permite convertir las señales separadas en un flujo complejo de datos, con partes real e imaginaria, que es el formato esperado por los bloques de procesamiento siguientes (*Null Source*, 2022), (*Float To Complex*, 2025).

### 3.9.3 *Throttle*

Este bloque se encarga de limitar la velocidad de procesamiento de las muestras para evitar que el sistema consuma todos los recursos disponibles del CPU cuando se ejecuta sin hardware SDR real. En este caso, actúa sobre una señal compleja con frecuencia de muestreo de 2 MHz, definida por la variable `f_muestreo`. El largo del vector es 1, y la opción `Ignore rx_rate tag` está activada, lo cual es adecuado para flujos de datos sintéticos como los provenientes de un archivo `.wav` (*Throttle*, 2023).