ALMA MATER STUDIORUM - UNIVERSITA DI BOLOGNA
CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA

Corso di Laurea Triennale in Ingegneria Elettronica per l'Energia e l'Informazione

Interfacciamento di GnuRadio con dispositivi hardware

Tesi in
Comunicazioni Digitali e Internet

Relatore:

Chiar.mo Prof. Ing.
DAVIDE DARDARI

Candidato:

Bohdan Bohdanovich Kezyk

ANNO ACCADEMICO 2022-2023

Indice

Capitolo 1

Introduzione

L'argomento principale di questa tesi è costituito dalla spiegazione del funzionamento del Software GnuRadio erelativo interfacciamento di dispositivi hardware. GnuRadio è stato creato circa venti anni fa dall'ingegnere elettronico Erik Blossom. L'idea principale di questo software è di convertire tutti i problemi hardware relativi all'implementazione di un dispositivo radio in problemi software. Per lavorare con il software è disponibile un'interfaccia GnuRadio Companion (GRC) che è un sistema grafico semplice e molto intuitivo da usare per la progettazione e la costruzione di componenti Software Defined Radio (SDR). Nel primo approccio verso GRC, andremo a spiegare passo a passo la sua interfaccia, come vengono utilizzati i suoi blocchi, i tipi di dato che abbiamo e il segnale in uscita che otterremo. Il principio di funzionamento di GRC è basato sul Diagramma di Flusso, combinato da vari blocchi di elaborazione dei segnali che vengono forniti da una libreria dentro l'interfaccia GRC. Vedremo anche come viene creato e modificato un blocco dall'inizio alla fine, che può essere utilizzato dentro il nostro diagramma di flusso. Infine, verrà illustrato l'interfacciamento di GnuRadio con dispositivi hardware mediante alcuni esempi pratici.

Capitolo 2

GnuRadio

GnuRadio è un progetto di Software Defined Radio (SDR) lanciato circa venti anni fa da **Eric Blossom**, un ingegnere elettronico. L'idea principale che sta alla base di questo progetto, come afferma il suo stesso fondatore, è quella di convertire tutti i problemi hardware in problemi software, cioè spostare la complessità relativa alla progettazione di un apparato radio dal livello hardware a quello software, e portare il software più vicino possibile all'antenna.
Eric Blossom cominciò a lavorare a questo progetto perché era rimasto deluso dalle altre applicazioni di tipo SDR disponibili all'epoca: queste, infatti, avevano tutte la caratteristica di avere una natura proprietaria, mentre Eric desiderava portare la filosofia del free-software anche nel mondo SDR. Richard Stallman, il fondatore del Progetto Gnu, apprezzò l'idea di Eric Blossom e accettò di portare il nuovo progetto sotto l'egidia Gnu.
Fino ad ora, il progetto GnuRadio non ha deluso i suoi estimatori e i numerosi appassionati del mondo radio. Eric Blossom, insieme al suo collega e collaboratore Matt Ettus, hanno realizzato un progetto in grado di convertire un qualunque PC in un ricevitore o trasmettitore radio di buona qualità; l'unico hardware aggiuntivo necessario è rappresentato da un ricevitore RF "low-cost" e da un convertitore analogico-digitale in grado di trasformare il segnale ricevuto in campioni digitali. GnuRadio è uno strumento di sviluppo software gratuito che permette di sviluppare un ricevitore o trasmettitore radio personalizzato (anche molto diverso dai modelli normalmente in commercio) semplicemente combinando ed interconnettendo dei moduli software appropriati, proprio come se essi fossero dei blocchi funzionali (il package attualmente comprende circa 400 moduli, ma altri possono essere aggiunti alla libreria originale). Ciascun modulo è in grado di eseguire una specifica funzionalità di elaborazione del segnale (per esempio un mixer, un phase lock loop, un filtro), con un comportamento in tempo reale e con una elevata portata. Per questo motivo, è consigliabile utilizzare un PC con un'adeguata capacità di elaborazione e una buona disponibilità di memoria RAM.
Con l'approccio previsto da GnuRadio, il progettista diventa uno sviluppatore software che costruisce la radio creando prima un grafo (con una modalità molto simile a quanto avviene nella teoria dei grafi) in cui i vertici sono i blocchi di elaborazione del segnale ed i rami rappresentano i flussi di dati tra essi. I blocchi che implementano funzioni di elaborazione del segnale sono in genere codificati in linguaggio C++, mentre la struttura del grafo viene definita in Python. GnuRadio è molto conosciuta ed ampiamente usata specialmente in ambito accademico tra gli hobbysti e tra gli amanti della radio. Essa viene utilizzata sia per realizzare degli apparati radio concreti e funzionanti, sia semplicemente come un progetto di ricerca nell'area della comunicazione e trasmissione di tipo wireless. I moduli software di GnuRadio supportano vari tipi di modulazioni (GMSK, PSK, QAM, OFDM), vari codici di correzione degli errori (Reed-

Solomon, Viterbi, Turbo Codes) e diverse funzionalità di elaborazione del segnale (filtri, trasformate di Fourier veloci, equalizzatori, timing recovery).
Le applicazioni GnuRadio sono principalmente scritte in Python; tuttavia, gli algoritmi critici e di basso livello, come pure i moduli di elaborazione del segnale, sono scritti usando il linguaggio di programmazione C++, con un utilizzo esteso di istruzioni in virgola mobile. Python viene essenzialmente usato per descrivere il grafo di flusso, dopodiché la maggior parte del lavoro viene fatto in C++. Il mediatore fra Python e C++ è *The Swig* (Simplified Wrapper and Interface Generator). I blocchi sono sincroni e si distingue fra blocchi *source,* che non presentano ingressi, e blocchi *sink* che non producono uscite. GnuRadio è molto semplice da utilizzare, basti pensare che un ricevitore radio può essere realizzato in un modo estremamente veloce ed immediato in pochissimi passi. Inoltre, lo sviluppo di un algoritmo di elaborazione del segnale può essere compiuto usando un set di dati pre-registrati oppure generati ad-hoc (simulatore), consentendo così di effettuare lo sviluppo senza necessariamente disporre di un vero e proprio hardware RF. Un esempio di hardware minimale richiesto per cominciare a lavorare con GnuRadio è offerto dal dispositivo SDR ADALM-PLUTO.

2.1 Vantaggi di GnuRadio

Prima che GnuRadio fosse creato, durante lo sviluppo di dispositivi di comunicazione radio, un ingegnere doveva sviluppare un circuito specifico per il rilevamento di una particolare classe di segnale, progettare un circuito integrato in grado di codificare o decodificare quella particolare trasmissione, ed eseguire il debug di questi utilizzando apparecchiature costose. Per completare il tutto doveva scrivere un programma di comunicazione, che richiede uno studio approfondito della materia.
Ai giorni nostri il software GnuRadio, che è uno strumento di sviluppo gratuito e open source, racchiude le funzionalità in blocchi riutilizzabili facilmente, offrendo un'eccellente modularità, un'ampia libreria di algoritmi standard ed è fortemente ottimizzato per un'ampia varietà di piattaforme comuni. Inizialmente viene fornito con un estesa serie di esempi. Utilizzando questi blocchi, è possibile eseguire molte attività standard, come la normalizzazione dei segnali, la sincronizzazione, le misurazioni e la visualizzazione, semplicemente collegando il blocco appropriato al grafico del flusso di elaborazione del segnale. Inoltre, c'è la possibilità di creare nuovi blocchi che possono essere utilizzati nel diagramma di flusso con i blocchi già esistenti.
Pertanto, GnuRadio è principalmente una struttura per lo sviluppo di blocchi di elaborazione del segnale e la loro interazione. Tuttavia, GnuRadio in sé non è un software pronto per fare qualcosa di specifico. È compito dell'utente costruire qualcosa di utile, sebbene sia già dotato di molti utili esempi funzionanti. Bisogna pensarlo come un insieme di elementi costitutivi.
GnuRadio Companion (GRC) è un sistema grafico semplice da usare per la progettazione e la costruzione di componenti *Software Defined Radio (SDR)*. GnuRadio Companion fornisce funzioni comuni, come sorgenti di segnale, elaborazione del segnale e sink di segnale, blocchi che possono essere selezionati e posizionati sullo schermo. Una volta

posizionati, i blocchi possono essere cablati, proprio come in LabView, e il flusso di dati può essere controllato in questo modo. GnuRadio Companion include anche blocchi che consentono di costruire un'interfaccia GUI, che può essere utilizzata per visualizzare i dati e controllare l'applicazione. Se un blocco non esiste, può essere creato. Poiché GnuRadio utilizza Python, gli utenti possono utilizzare questo linguaggio potente e flessibile per creare nuovi blocchi che possono essere importati in GRC.
L'installazione di GnuRadio Companion è consigliata su **Linux**. Ma può essere installato anche su altri sistemi operativi, come Windows e Mac OS. Il processo di installazione su diversi sistemi viene spiegato in Appendice.

Capitolo 3

Fondamenti di GnuRadio Companion (GRC)

Vediamo in questo capitolo il primo approccio verso il software GnuRadio Companion. Spiegheremo la sua interfaccia, i suoi blocchi come vengono utilizzati, il tipo di dato che abbiamo e il segnale in uscita che otterremo.

3.1 Interfaccia GnuRadio Companion

Una volta avviato GRC, ci troveremo davanti all'interfaccia. L'interfaccia è abbastanza intuitiva e comprende cinque sezioni [1]. Nella parte superiore, abbiamo il classico menù, dal quale potremo selezionare le opzioni per la creazione e lettura di un nuovo progetto, oltre a quelle relative all'esecuzione ed al debug. Sulla destra avremo un elenco che mostrerà i blocchi disponibili suddivisi per la funzione dedicata. Al centro avremo l'area di lavoro, nella quale possiamo già notare due blocchi (*Options* e *Variable)*, su cui ritorneremo a breve. In basso a sinistra abbiamo la sezione di log. Infine, nella parte centrale un'altra sezione che ci mostra le variabili attualmente nell'area di lavoro, come in Figura 3.1.1.

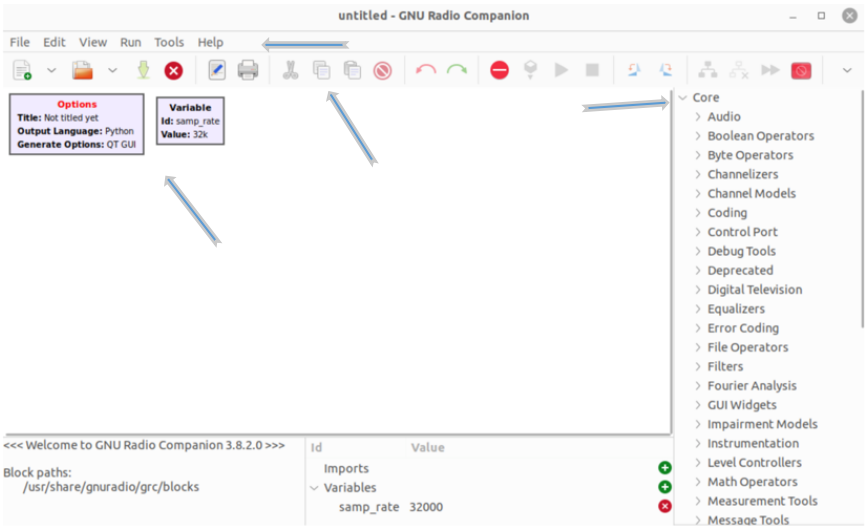


Figura 3.1.1. Interfaccia GRC.

Prima di fare qualsiasi operazione occorre salvare il file progetto. (File-Save), come in Figura 3.1.2.

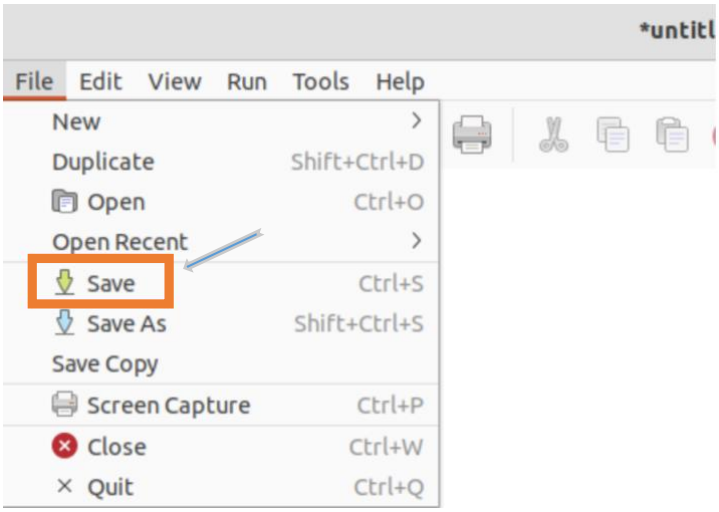


Figura 3.1.2. Salvataggio progetto.

Per accedere ai blocchi, per esempio blocco *Option*, fare doppio click, appare la schermata (Properties: Options), come in Figura 3.1.3. A questo punto siamo in grado di cambiare le proprietà del blocco in base a quello che dobbiamo fare. Id è il nome del file del diagramma di flusso Python e Il Title è una descrizione del diagramma di flusso. Per salvare le modifiche fare clic su OK.

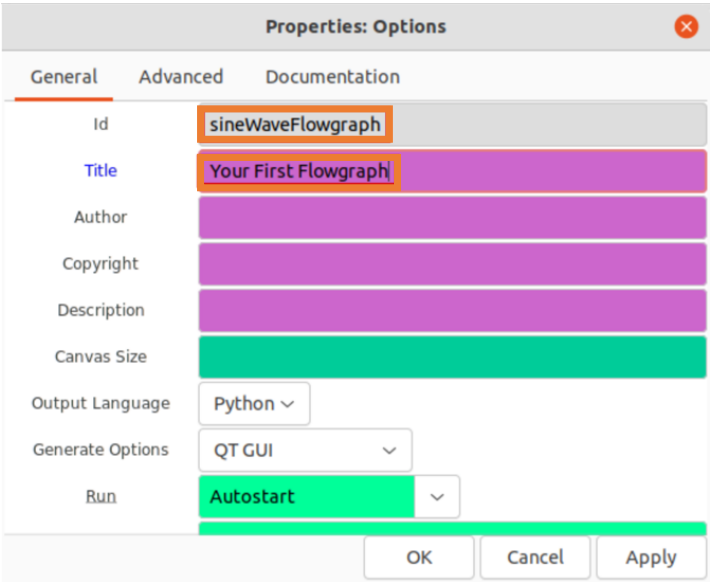


Figura 3.1.3. Proprietà del blocco Options.

Per creare il diagramma di flusso dobbiamo aggiungere dei blocchi. GnuRadio viene fornito con una libreria di blocchi di elaborazione del segnale posizionati sulla destra della schermata. I blocchi possono essere sfogliati utilizzando le frecce sulla destra (>), oppure usando la sequenza CTRL + F. Infine anche selezionando la lente di ingrandimento evidenziata in arancione.
Esempio: proviamo a cercare il blocco *Signal Source*. Una volta trovato possiamo trascinare il blocco nell'area di lavoro, come in Figura 3.1.4

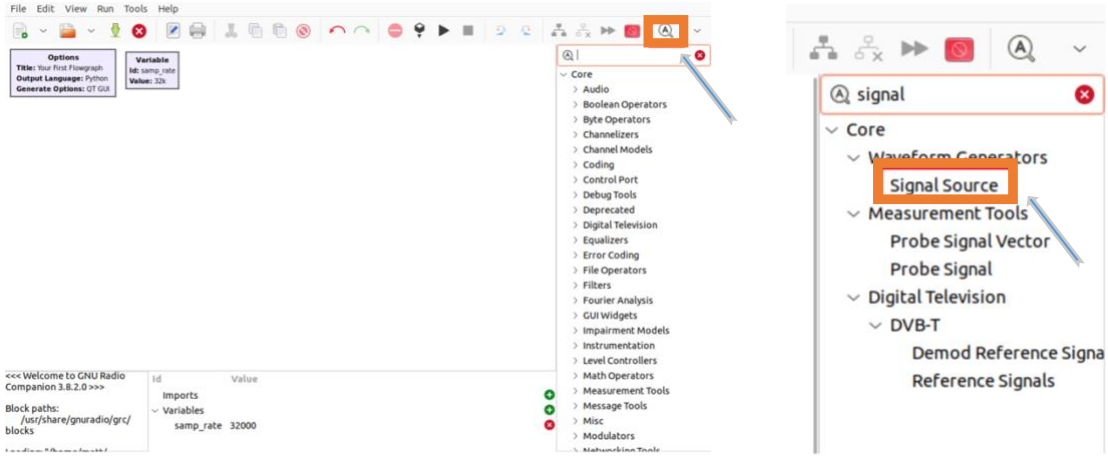


Figura 3.1.4. Ricerca blocco.

Ora per capire meglio vediamo un piccolo esempio dell'applicazione. Proviamo a individuare e inserire nell'area di lavoro i seguenti blocchi, come in Figura 3.1.5.

- *Signal Source* -> creerà una sinusoide complessa.
- *QT GUI Frequency Sink* -> visualizza l'ampiezza dello spettro di frequenza.
- *QT GUI Time Sink* -> visualizza il dominio del tempo.
- *Throttle* -> Un blocco dell'acceleratore dovrebbe essere utilizzato se e solo se il tuo diagramma di flusso non include alcun blocco di limitazione della velocità, che in genere è hardware (ad es. SDR, altoparlante, microfono). Il *Throttle Block* è in genere collegato direttamente all'uscita di un blocco sorgente non hardware (ad es. Signal Source), al fine di limitare la velocità con cui quel blocco sorgente crea campioni.

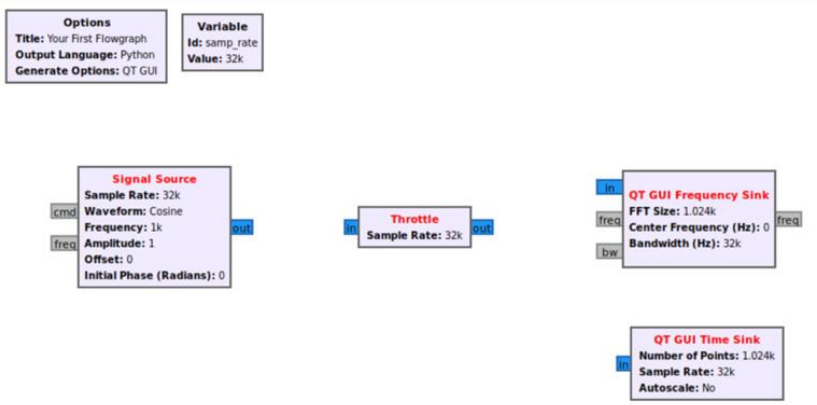


Figura 3.1.5. Diagramma di flusso.

La fase successiva è collegare i blocchi. Eseguire un clic sull'uscita di *Signal Source* (evidenziata in rosso) per poi proseguire con un clic sull'input del *Throttle* (evidenziato in arancione), come in Figura 3.1.6. Notiamo che Il testo del blocco *Signal Source* è cambiato da rosso a nero. Il testo rosso indica che un blocco ha ancora un input o un output che deve essere connesso prima che il diagramma di flusso possa essere eseguito.

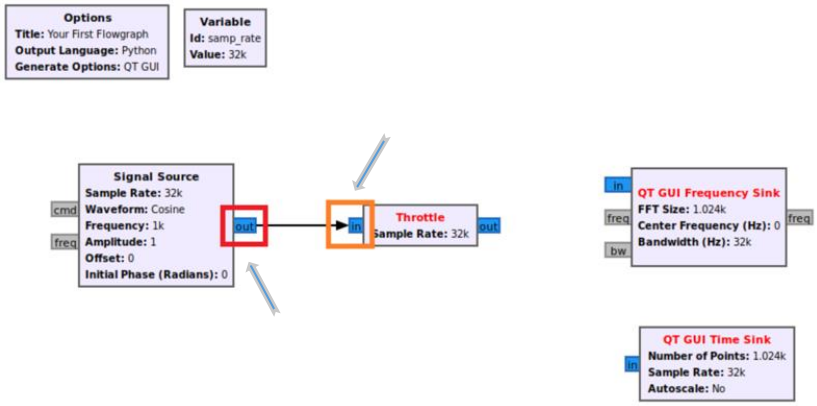


Figura 3.1.6. Collegamento dei blocchi.

Per concludere colleghiamo l'uscita di *Throttle* all'ingresso del *QT GUI Frequency Sink* e *QT GUI Time Sink*, come in Figura 3.1.7.

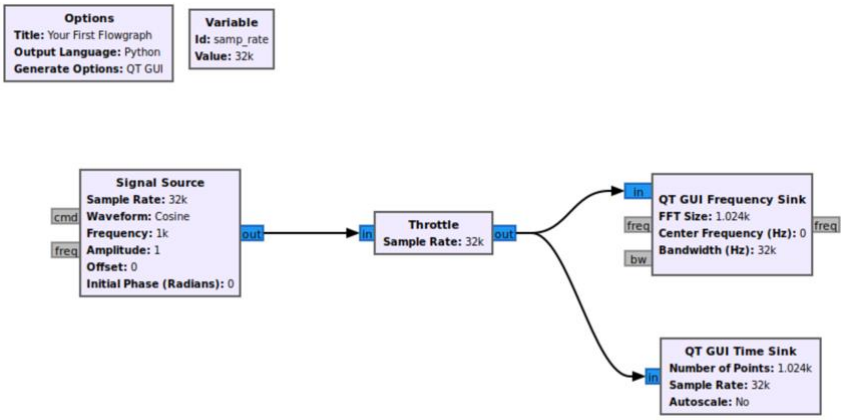


Figura 3.7. Collegamenti completati.

Dopo aver eseguito il collegamento corretto, premiamo il pulsante riproduci sulla interfaccia in alto, come in Figura 3.1.8.

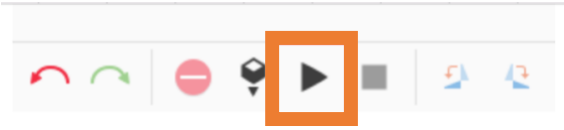


Figura 3.1.8. Riproduci.

Dopo qualche secondo, ci appare una nuova finestra che visualizza il segnale nel dominio del tempo e nel dominio della frequenza, come mostrato in Figura 3.1.9.

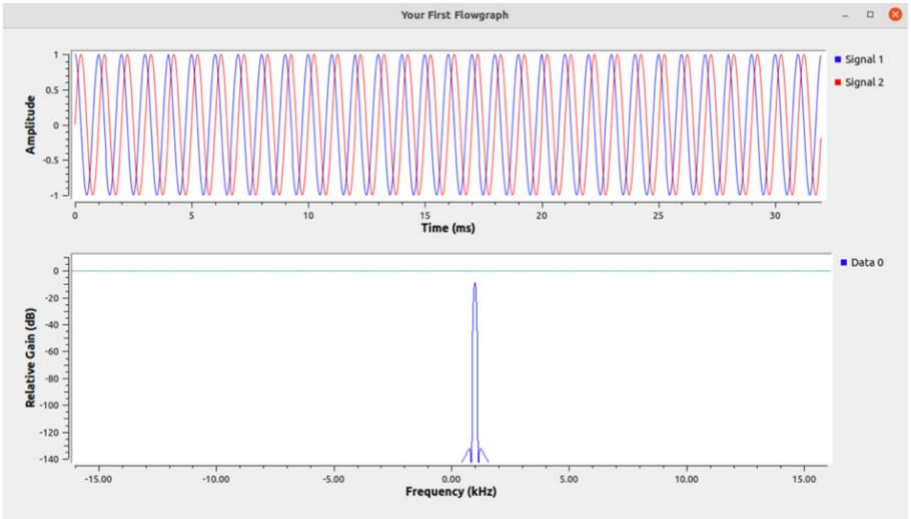


Figura 3.9. Segnale nel dominio del tempo e delle frequenze.

3.2 Variabili Python in GRC

Descriviamo come vengono utilizzati i tipi di dati Python in GRC e come utilizzare e modificare le variabili in un diagramma di flusso più sofisticato [2].

3.2.1 Float e numeri interi in GRC

GnuRadio Companion (GRC) utilizza i tipi di dati Python per rappresentare le variabili. I tipi di dati più semplici descrivono i numeri. I numeri in Python possono essere in virgola mobile o interi.
I numeri interi possono essere convertiti in virgola mobile usando float(..) e i numeri in virgola mobile possono essere convertiti in numeri interi usando int(..).

Ad esempio usiamo:

- floatNumber = 3.14
- intNumber = 2

Per fare la conversione scriviamo.

- floatNumber = float(2)
- intNumber = int(3.14)

GRC visualizza i numeri in modo diverso rispetto a Python. Ad esempio, il valore di samp_rate viene aggiunto a ogni nuovo diagramma di flusso, come in Figura 3.2.1.

Figura 3.2.1. Variabile in GRC.

Il valore di samp_rate è 32000Hz ma GRC visualizza il valore 32kHz. GRC converte tutti i numeri in unità SI. Si noti che GRC può visualizzare un numero in un formato diverso da quello rappresentato in Python. Se noi vogliamo fare la modifica delle proprietà al blocco *Variable,* facciamo semplicemente il doppio click su di essi cambiando il valore di samp_rate.

3.2.2 Stringhe in GRC

Python usa sia le virgolette singole (') che le virgolette doppie (") per contenere le stringhe.

singleQuoteString = ‘string1’
doubleQuoteString = “string2”

Le stringhe possono essere utilizzate come variabili in GRC. Proviamo a cambiare le proprietà del blocco *Variable* in stringhe, come in Figura 3.2.2.

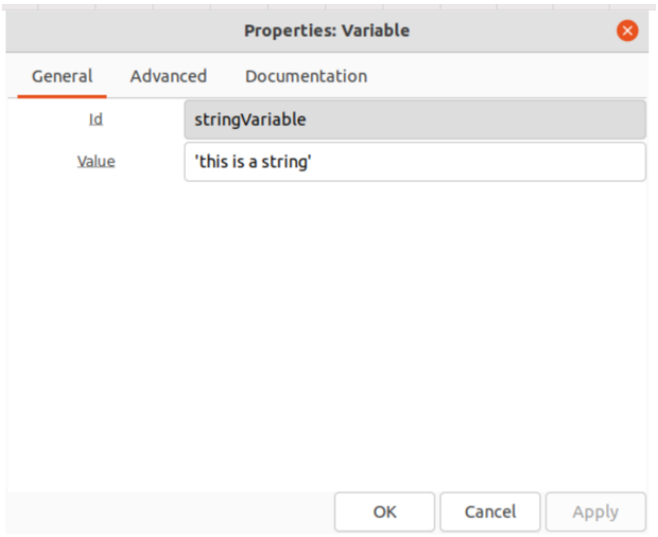


Figura 3.2.2. Scrittura delle stringhe.

3.2.3 Liste e tuple in GRC

Le variabili in GRC possono utilizzare **liste** Python, come in Figura 3.2.3. Per capire meglio, prendiamo sempre il blocco *Variable* e facciamo la modifica delle proprietà e nella voce Value scriviamo dentro le parentesi quadre [].

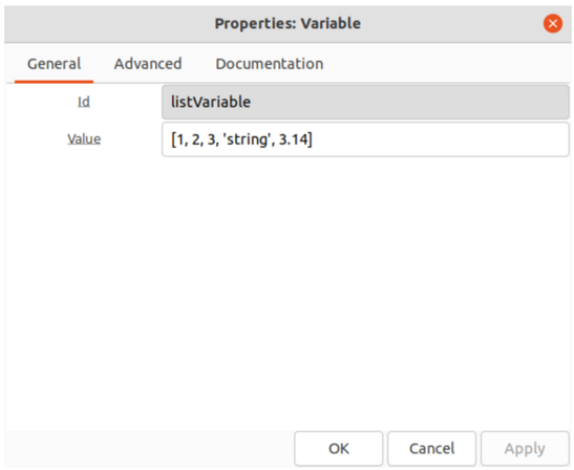


Figura 3.2.3. Nomenclatura delle **liste**.

Lista viene visualizzata in GRC, come in Figura 3.2.4.

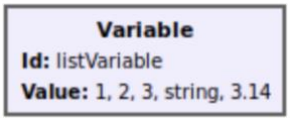


Figura 3.2.4. Visualizzazione **lista** in blocco *Variable*.

Le variabili in GRC possono utilizzare **tuple** Python. L'unica differenza dalle liste risulta che dentro il blocco *Variabile* la voce Value viene scritta dentro parentesi tonde (), come in Figura 3.2.5.

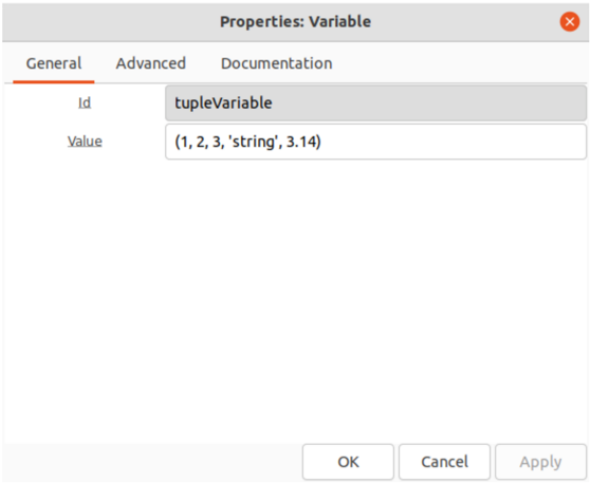


Figura 3.2.5. Nomenclatura delle **tuple**.

3.2.4 Comprensione della lista

Ogni variabile è una singola riga in Python, esempio *id=value.* La comprensione della lista può essere utilizzata per scrivere funzioni in una variabile.

Ad esempio: la comprensione della lista viene utilizzata per scorrere la lista stessa, aggiungendo +1 a tutte le voci e quindi moltiplicare ciascuna voce per 2.

```
listVariable = [0, 1, 2, 3]
listComprehensionExample = [( i+ 1)  *  2 for i in listVariable]
```

Questo esempio di comprensione della lista viene utilizzato in GnuRadio utilizzando due blocchi *Variable*, listVariable e listComprehensionExample, e inserendo i valori associati, come in Figura 3.2.6.

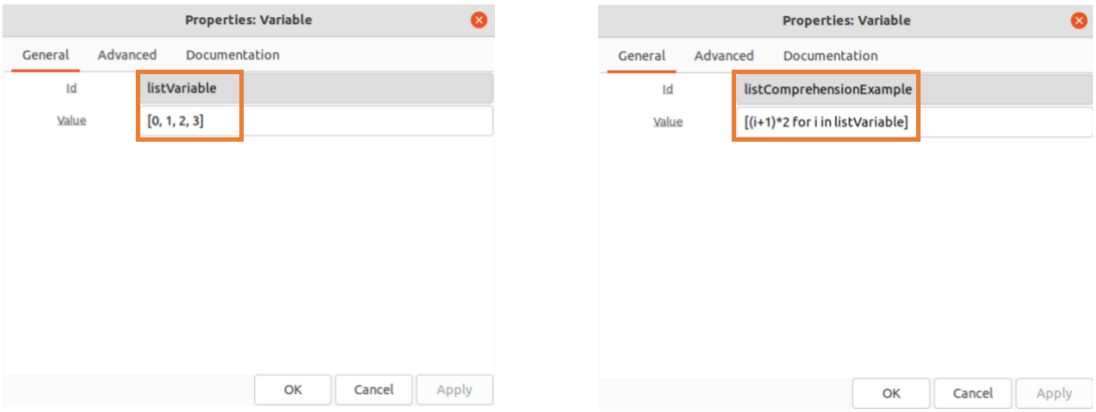


Figura 3.2.6. Comprensione delle variabili.

Le liste vengono visualizzati in GRC come in Figura 3.2.7.

Figura 3.2.7. Comprensione in GRC.

3.2.5 Proprietà dei colori in GnuRadio Companion

GRC utilizza uno schema di colori per rappresentare i tipi di dati durante la modifica delle proprietà del blocco. Vediamo un esempio delle proprietà per il blocco *QT GUI Frequency Sink*, come in Figura 3.2.8.

Figura 3.2.8. Proprietà *QT GUI Frequency Sink.*

Sono disponibili diversi colori per le proprietà *QT GUI Frequency Sink*: arancione, verde e viola. Ogni colore corrisponde a un diverso tipo di dati:

- Virgola mobile: arancione
- Intero: verde
- String: viola

Ad esempio: Larghezza di banda di colore arancione corrispondente a qualsiasi numero in virgola mobile. La dimensione FFT di colore verde corrisponde a un numero intero. L'etichetta Y di colore viola è una stringa che contiene parole utilizzate per descrivere l'asse verticale del grafico. I blocchi *Variable* non hanno un colore perché possono essere usati per rappresentare qualsiasi tipo di dato o oggetto.

3.3 Variabili nel Diagramma di Flusso

Vediamo come le variabili vengono aggiornate mentre un diagramma di flusso è in esecuzione. GRC consente a un utente di interagire con i flussigrafi GnuRadio, sia quelli creati da zero in modo interattivo, sia quelli letti da un file .grc. Quando l'utente GRC utilizza il pulsante di riproduzione per eseguire un diagramma di flusso, GRC crea un file Python (.py) che contiene il codice del diagramma di flusso. Il codice Python può avere variabili e un diagramma di flusso GnuRadio può avere variabili create dal blocco *Variable*. Ogni nuovo diagramma di flusso inizia con la variabile samp_rate [3].

I blocchi GnuRadio sono implementati come funzioni. I blocchi GnuRadio prendono parametri che ne modificano il comportamento. Tutti i blocchi nel diagramma di flusso come in Figura 3.3.1 utilizzano samp_rate come parametro. Creiamo un nuovo blocco *Variable* trascinando dalla libreria di blocchi a destra (mostrato in rosso).

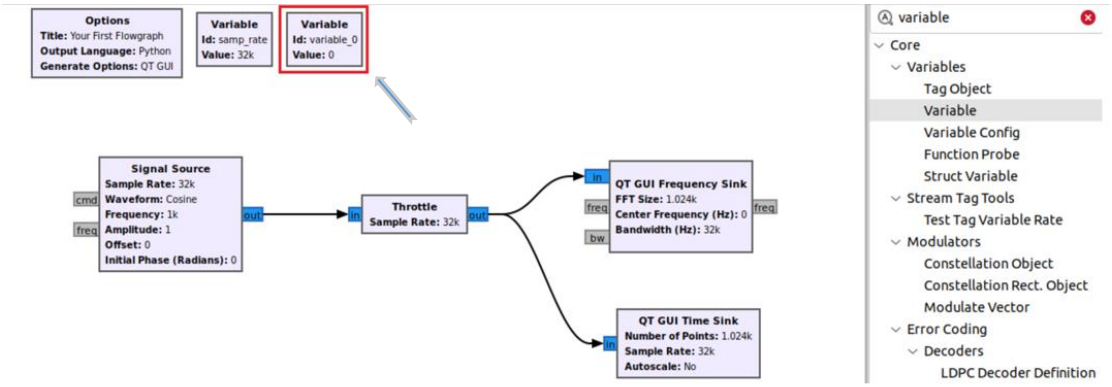


Figura 3.3.1. Nuovo blocco *Variable*.

Modifichiamo adesso i parametri del nuovo blocco *Variable* appena aggiunto in diagramma di flusso, modifichiamo il campo *id* che sarebbe il nome della variabile e

chiamiamolo *frequency*, modifichiamo anche la voce Value che è il valore del blocco *Variable* e mettiamo 4000Hz, come in Figura 3.3.2.

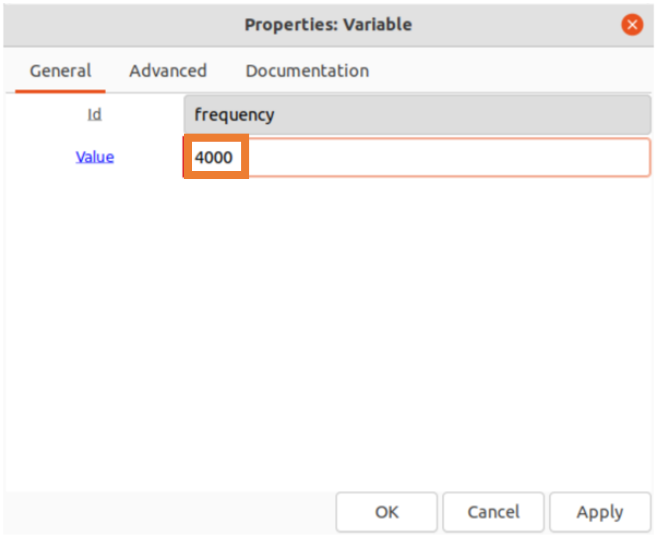


Figura 3.3.2. Proprietà nuovo blocco *Variable.*

Modifichiamo anche parametri del blocco *Signal Source* e vediamo che nella voce *Frequency* c'è scritto il valore 1000Hz, modifichiamo con il nome (*frequency*) della variabile che abbiamo appena modificato, come in Figura 3.3.3.

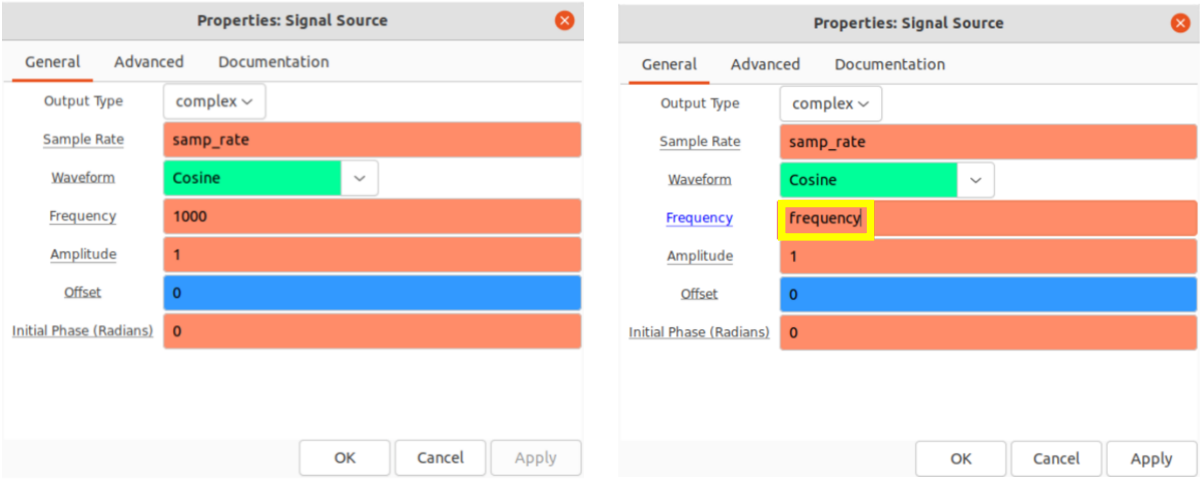


Figura 3.3.3. Cambio parametro blocco *Signal Source.*

Fare clic su OK per salvare le proprietà. La variabile di frequenza e il valore all'interno del blocco *Signal Source* vengono aggiornati, come in Figura 3.3.4.

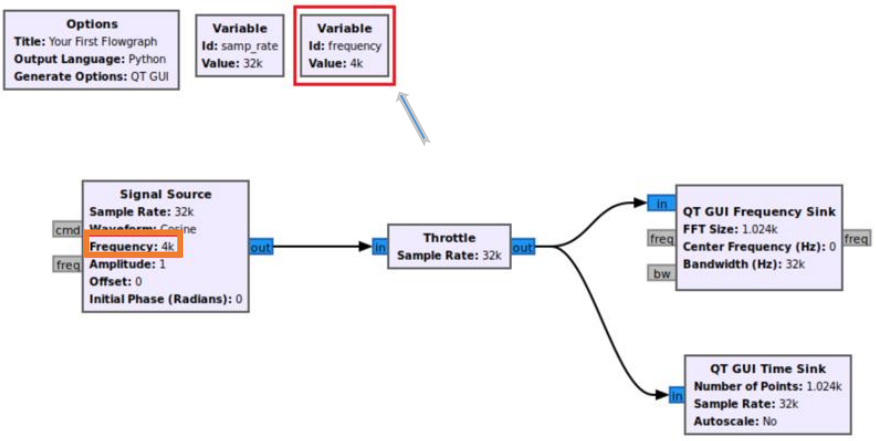


Figura 3.3.4. Diagramma di flusso aggiornato.

Ora se eseguiamo il programma otterremo il grafico del nostro Diagramma di Flusso, come in Figura 3.3.5.

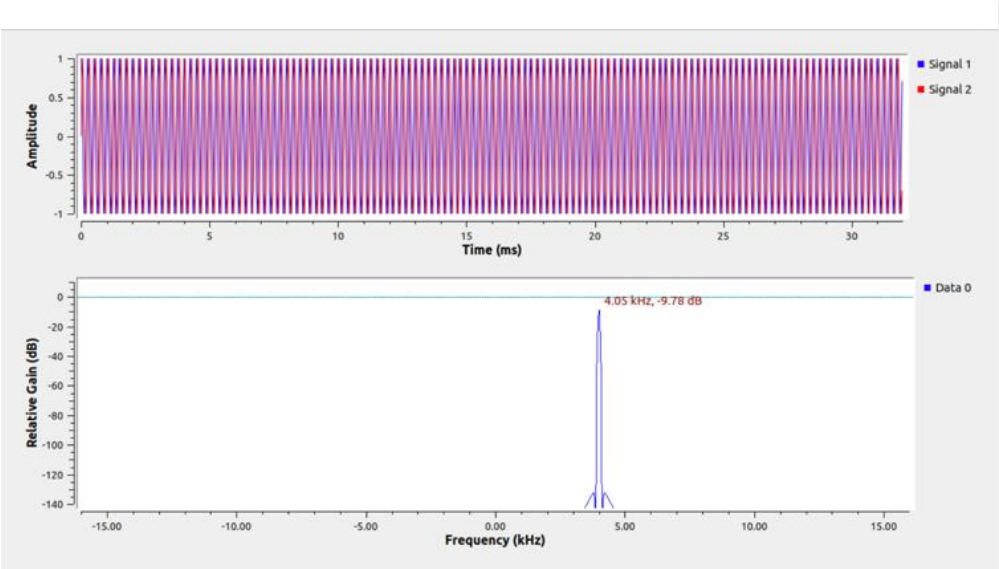


Figura 3.3.5. Grafico nel dominio del tempo e delle frequenze.

3.3.1 Variabili dipendenti

Le variabili possono essere dipendenti l'una dall'altra. I campi Id e Value vengono convertiti in una riga di Python nel modo seguente.

- Id=Value

La variabile frequency è stata modificata per accettare il valore 4000Hz, che è lo stesso di una riga di codice Python.

- frequency=4000Hz

La variabile di frequenza può anche dipendere da un'altra variabile. Modifichiamo la *frequency* per inserire il valore *samp_rate/3*, che per *samp_rate* = 32000Hz sarà una frequenza di 10.667Hz, come in Figura 3.3.6.

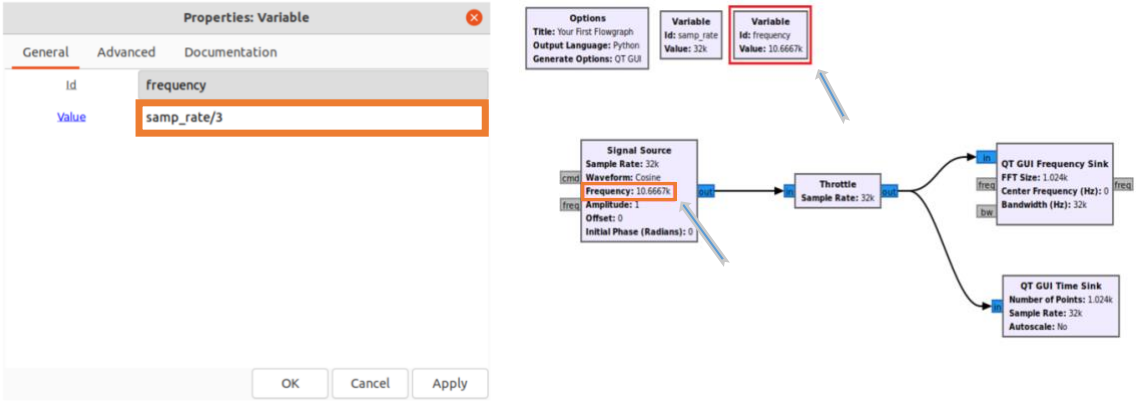


Figura 3.3.6. Variabili dipendenti.

3.4 Variabili di aggiornamento in fase di esecuzione

Vediamo come aggiornare le variabili mentre un diagramma di flusso è in esecuzione utilizzando i Widget della GUI QT [4].

3.4.1 Blocco QT GUI Range.

La libreria di blocchi GnuRadio viene fornita con widget QT GUI. I widget consentono l'interazione e la modifica di un diagramma di flusso durante l'esecuzione. Il widget *QT GUI Range* crea una barra di scorrimento che può essere utilizzata per aggiornare una variabile. Cerchiamo nella libreria il blocco *QT GUI Range* e lo trasciniamo nell'area di lavoro, come in Figura 3.4.1.

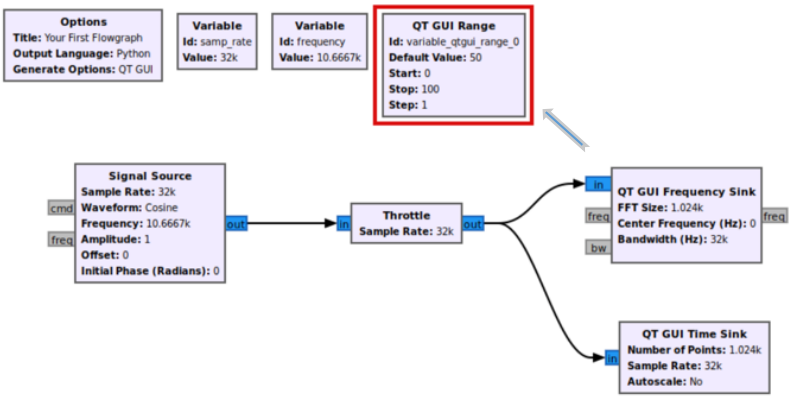


Figura 3.4.1. Nuovo blocco *QT GUI Range.*

La gamma QT GUI funziona come un blocco variabile. È necessario impostare i parametri predefiniti per *QT GUI Range*. Il blocco *QT GUI Range* sostituisce la variabile di frequenza, cambiando il campo Id in *frequency*. Il *Default Value* è il valore all'avvio del diagramma di flusso, impostiamo il valore predefinito su 0Hz. *Start* e *Stop* sono i valori di avvio e arresto del dispositivo di scorrimento, immpostiamo *-samp_rate/2* come valore iniziale e *samp_rate/2* come valore finale, sarà il range dentro il grafico dello spettro di frequenza. Il valore *Step* è la risoluzione del dispositivo di scorrimento, in questo esempio lo *Step* è impostato su 100Hz.

Viene visualizzato un messaggio di errore.

- ID “frequency is not unique”

Il messaggio di errore viene visualizzato perché un blocco *Variable* e *GUI QT Range* utilizzano entrambi la frequency come nome. Questo problema verrà risolto successivamente. Facciamo clic su OK per salvare le proprietà, come in Figura 3.4.2.

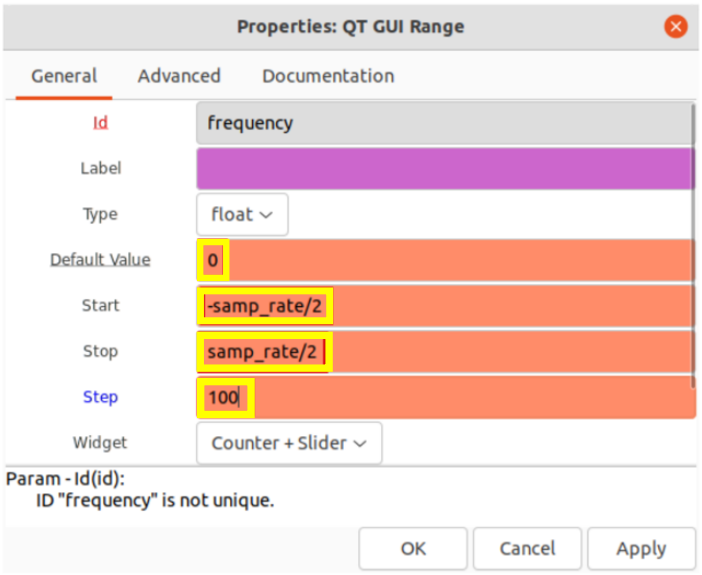


Figura 3.4.2. Proprietà del blocco *QT GUI Range.*

Proseguiamo cliccando sul blocco *Variable* con il tasto destro e selezioniamo Disable, oppure premere D sulla tastiera. Il blocco è ora ignorato e l'errore è stato risolto, come in Figura 3.4.3.

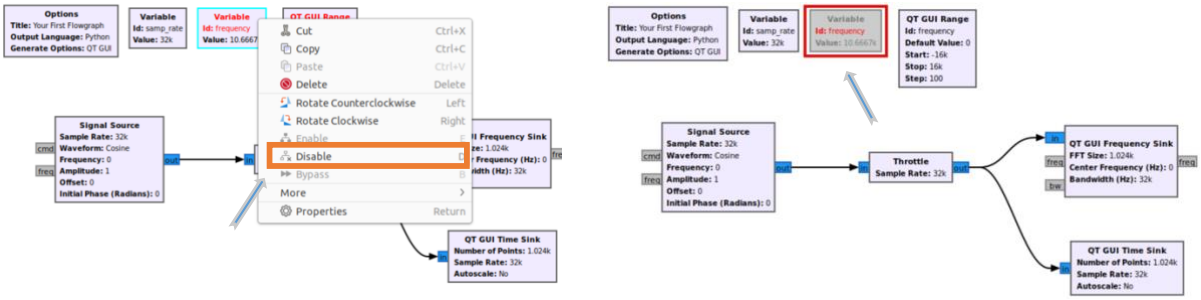


Figura 3.4.3. Correzione dell'errore precedente.

Eseguiamo il diagramma di flusso facendo clic sulla freccia o sul pulsante Riproduci. Il diagramma di flusso inizia con una frequenza pari a 0Hz, il valore predefinito immesso nel blocco *QT GUI Range*.

Il parametro di frequenza può quindi essere aggiornato, come in Figura 3.4.4 numerato in rosso con quadrature verdi.

1.Trasciniamo la barra di scorrimento
2.Inserimento di un valore
3.Fare clic sulle frecce su o giù

La frequenza è stata aggiornata a -5000Hz che si riflette in colore arancione nel grafico dello spettro di frequenza.

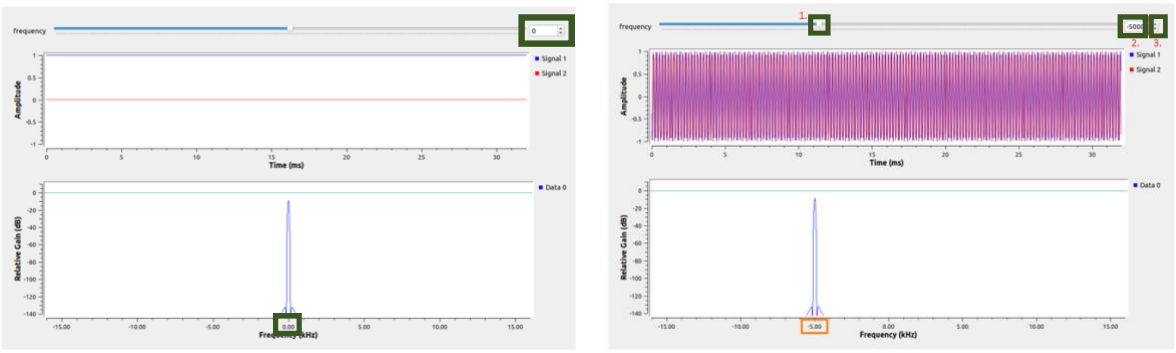


Figura 3.4.4. Modifica dello spettro di frequenza.

3.4.2 Blocco GUI QT Chooser

QT GUI Chooser crea un menù a discesa di opzioni per una variabile. Prendiamo il blocco e lo trasciniamo in area di lavoro, poi cambiamo i parametri predefiniti per il blocco *Chooser*. Il selettore crea un elenco di opzioni da selezionare mentre il diagramma di flusso è in esecuzione. In questo esempio vengono utilizzate tre frequenze: 0Hz; 1000Hz; -2000Hz.
Aggiorniamo le seguenti proprietà nel blocco *Chooser*.

Il campo Option è il valore della variabile, Label è una descrizione di testo che viene visualizzata nel menù a tendina. Figura 3.4.5 mostra un esempio dell'opzione evidenziata in arancione e dell'etichetta evidenziata in giallo.

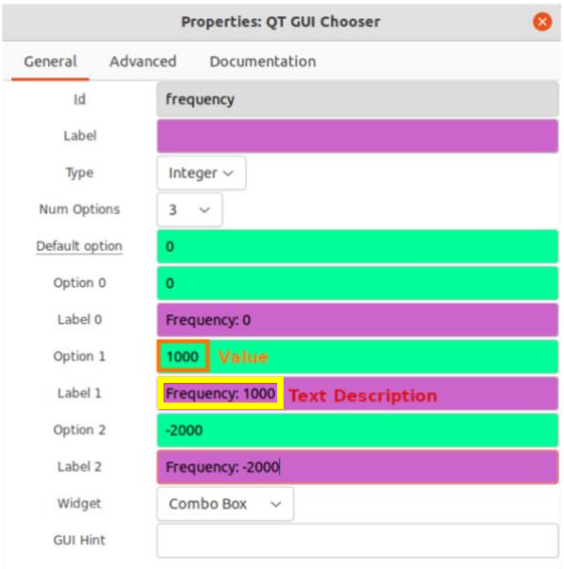


Figura 3.4.5. Proprietà del blocco *QT GUI Chooser.*

L'esecuzione del diagramma di flusso utilizzerà il valore predefinito Frequency: 0Hz all'avvio. La casella a discesa nell'angolo in alto a sinistra mostra che è stata selezionata la Frequency: 0Hz. Il dominio del tempo e il dominio della frequenza mostrano entrambi un segnale con frequenza 0Hz. I valori vengono selezionati facendo clic sul menù a discesa, facciamo clic su Frequenza: 1000Hz, e vedremo l'aggiornamento nello spettro di frequenza in Figura 3.4.6.

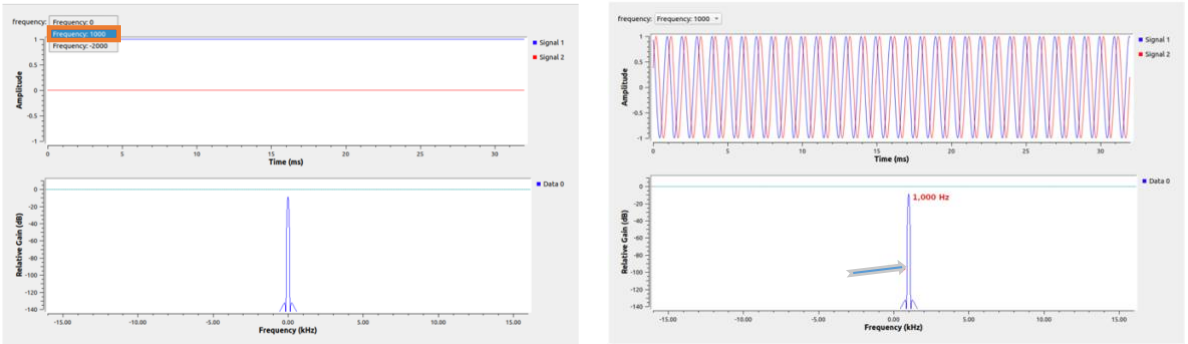


Figura 3.4.6. Cambiamento della frequenza con menù a discesa.

3.5 Tipi di dati del segnale

Vediamo i tipi di dati che possono essere utilizzati per rappresentare i segnali. Ogni porta di input e output su un blocco avrà un tipo di dati associato. Il tipo di dati è identificato dal colore della porta di input e output. I tipi di dati di GnuRadio possono essere trovati aprendo GnuRadio Companion (GRC) e facendo clic su (Help, Types). Una finestra visualizza i tipi di dati e i colori associati, come in Figura 3.5.1 [5].

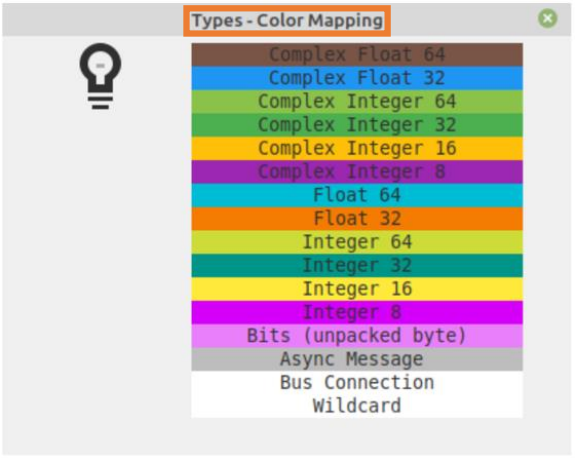


Figura 3.5.1. Tipi di dati.

Questi colori corrispondono alle porte di input e output per i blocchi in GRC, come in Figura 3.5.2. I tipi di dati più comuni nei blocchi GnuRadio sono Complex Float 32 in blu e Float 32 in arancione. Altri colori includono il tipo di dati Integer 16 (o short) in giallo e il tipo di dati Integer 8 (o char) in viola.

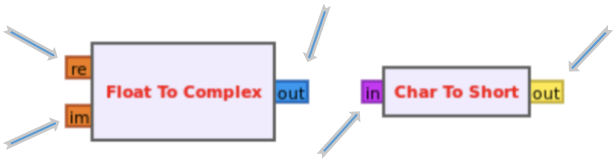


Figura 3.5.2. Colori delle porte input e output.

3.5.1 Tipo di dati complesso

Il seguente diagramma di flusso utilizza il tipo di dati Complex Float 32, che utilizza una coppia di float a 32 bit per rappresentare le parti reali e immaginarie di un campione complesso. Prendiamo il nostro diagramma di flusso che abbiamo sempre utilizzato, come in Figura 3.5.3.

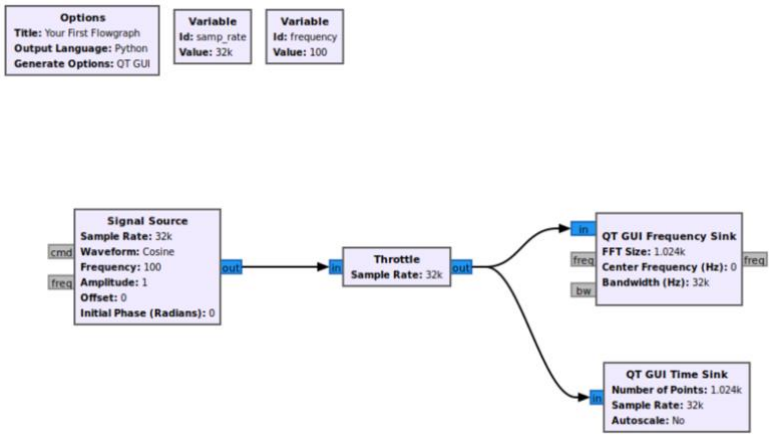


Figura 3.5.3. Diagramma di flusso.

L'esecuzione del diagramma di flusso mostra il segnale complesso tracciato nel dominio del tempo. Il segnale 1 è il componente reale e il segnale 2 è il componente immaginario del segnale complesso. Ogni campione complesso è quindi di 64 bit: un float di 32 bit per il componente reale e un float di 32 bit per il componente immaginario, come in Figura 3.5.4.

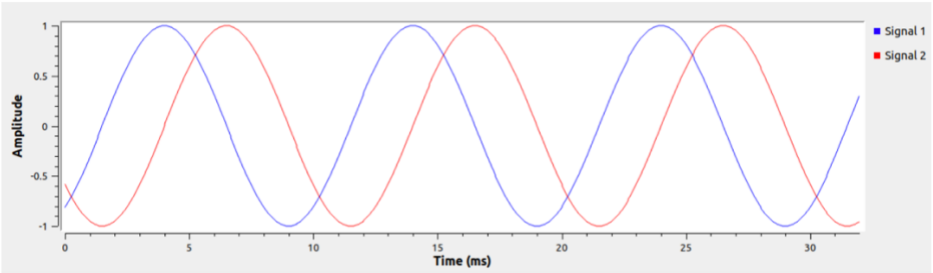


Figura 3.5.3. Grafico del segnale complesso nel dominio del tempo.

3.5.1 Tipo di dati Float

Molti blocchi GnuRadio supportano più tipi di dati. Il tipo di dati del blocco *Signal Source* può essere modificato facendo doppio clic su di esso. Nella voce Output Type selezionare dal menù il tipo di dato float e confermarlo. Il blocco *Signal Source* creerà una vera sinusoide, rappresentata dalla porta di uscita in arancione. Notare che la freccia che collega *Signal Source* a *Throttle* è rossa, a indicare un errore di mancata corrispondenza del tipo di dati, come in Figura 3.5.3.

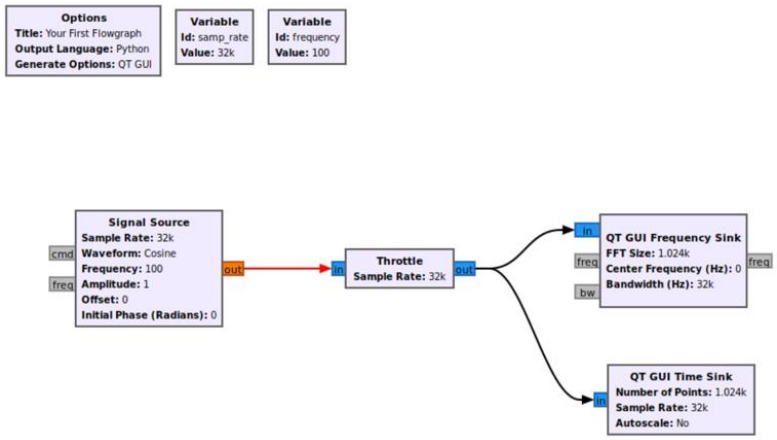


Figura 3.5.3. Cambiamento tipo di dato.

L'errore viene risolto convertendo tutti gli altri blocchi nel tipo di dati Float. Cliccando sul blocco una volta lo si seleziona, evidenziandolo in azzurro. I tipi di dati possono essere modificati premendo UP o DOWN sulla tastiera. Il diagramma di flusso è completo dopo che tutti i tipi di dati sono stati convertiti in Float, come in Figura 3.5.4.

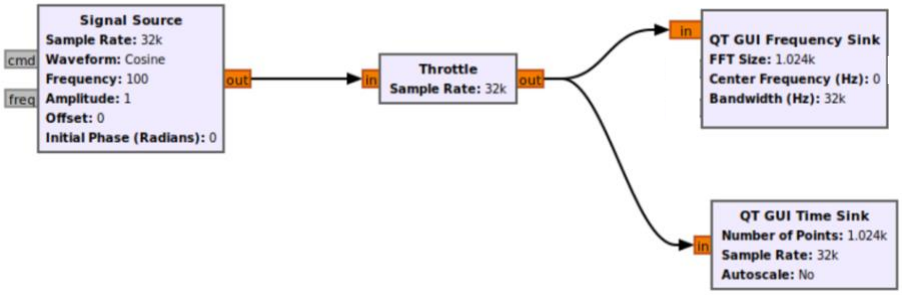


Figura 3.5.4. Conversione tipo di dato in float.

Il blocco *Signal Source* crea un output reale, che viene visualizzato come unico segnale nel dominio del tempo, come in Figura 3.5.5.

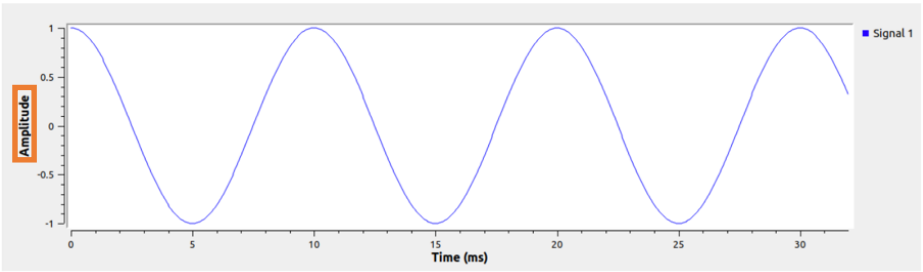


Figura 3.5.5. Grafico nel dominio del tempo reale.

3.6 Conversione di tipi di dati

Vediamo di illustrare come eseguire la conversione tra tipi di dati [6].

3.6.1 Tipo di dati char o byte

Il tipo di dati Char o Byte è un altro tipo di dati utile per rappresentare i dati binari. Il tipo di dati Byte è rappresentato dal colore viola in GRC, etichettato Integer 8. Prendiamo il blocco *Random Source* e trasciniamolo nell'area di lavoro. Per impostazione predefinita, il blocco utilizza il tipo di dati verde Integer 32. Facciamo doppio clic sul blocco per aprire le proprietà e modificare il tipo di dati in byte, come in Figura 3.6.1.

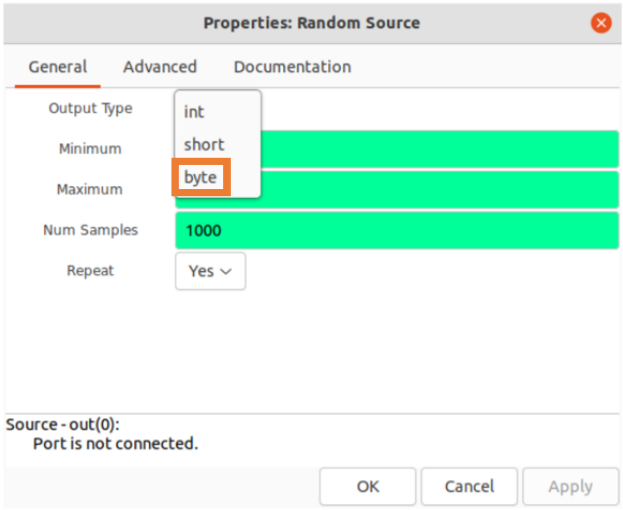


Figura 3.6.1. Cambiamento del tipo di dato.

Il *Random Source* è ora convertito nel tipo di dati viola Char o Byte, come in Figura 3.6.2.

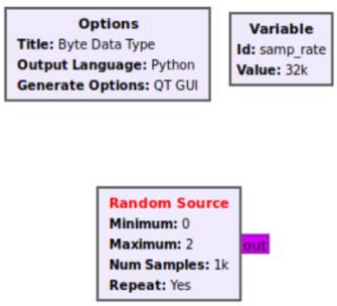


Figura 3.6.2. Conversione del tipo di dato visto in GRC.

3.6.2 Conversione di byte a Float 32

I parametri predefiniti di *Random Source* genereranno in modo casuale i valori 0 e 1. Aggiungiamo il blocco *QT GUI Time Sink* e il blocco *Throttle* nell'area di lavoro e colleghiamo i blocchi, come in Figura 3.6.3.

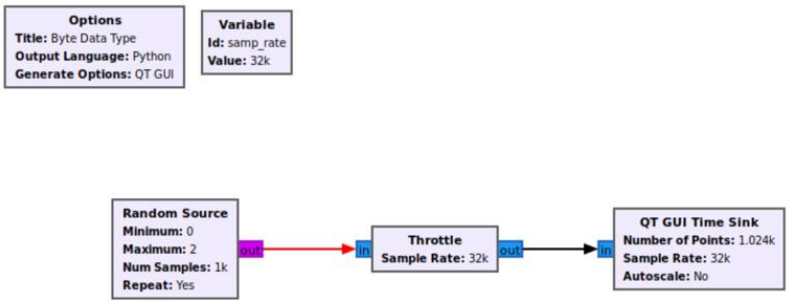


Figura 3.6.3. Diagramma di flusso.

La freccia rossa tra i blocchi *Random Source* e *Throttle* mostra un errore del tipo di dati che deve essere corretto. Doppio clic sul blocco *Throttle* e modifichiamo il tipo di dato in byte. Modificando il blocco *Throttle* apparirà una nuova freccia rossa, che mostra una connessione di tipo di dati tra *Throttle* e *QT GUI Time Sink* sbagliata. Il blocco *QT GUI Time Sink* non ha un tipo di dati char e quindi serve un nuovo blocco che ci permette di connetterli. La libreria di blocchi GnuRadio viene fornita con una varietà di convertitori di tipi di dati come elencati in *Type Converters*. Prendiamo il blocco *Char to Float* e trasciniamo in area di lavoro. Colleghiamolo al diagramma di flusso e vedremo che tutti gli errori in rosso scompariranno, come in Figura 3.6.4.

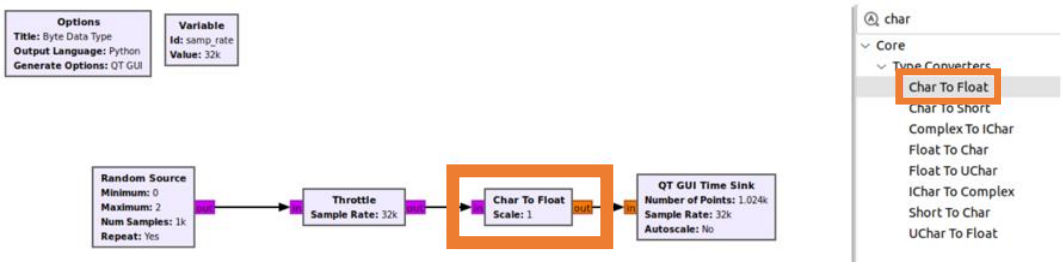


Figura 3.6.4. Convertitori tipi di dato.

Premiamo il pulsante Riproduci per avviare il diagramma di flusso. La *QT GUI Time Sink* visualizzerà ora i dati dal blocco *Random Source* che è randomizzato 0 e 1, come in Figura 3.6.5.

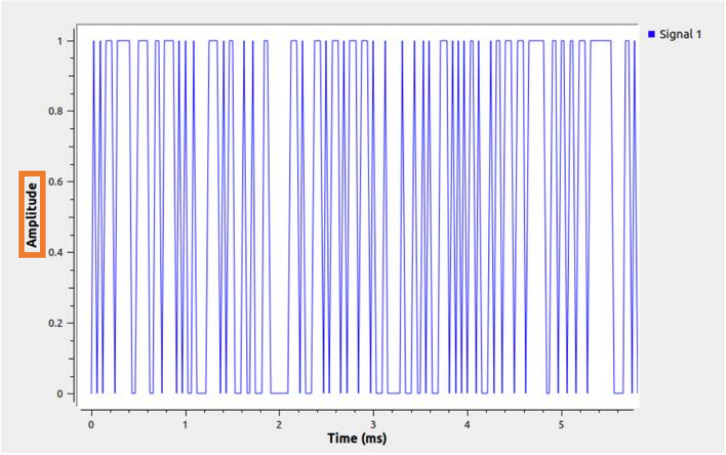


Figura 3.6.5. Grafico del segnale random 0 e 1.

3.7 Compressione/Decompressione dei Bit

Vediamo come comprimere i bit in un byte utilizzando il blocco *Pack K Bits* e come decomprimere un byte in bit utilizzando il blocco *Unpack K Bits* [7].

3.7.1 Avvio del diagramma di flusso dei bit di compressione

L'impacchettamento dei bit in un byte è utile per rappresentare i dati binari (al contrario dei campioni RF digitalizzati) e quando si utilizzano i blocchi del modulatore: Constellation Modulator, GFSK Mod e OFDM Transmitter.
Creiamo un nuovo diagramma di flusso, prendiamo il blocco *Random Source* all'area di lavoro, facciamo clic su *Random Source* per selezionarlo. Premendo i tasti SU o GIÙ per selezionare il tipo di dati byte, indicato dal colore della porta di uscita viola, come in Figura 3.7.1.

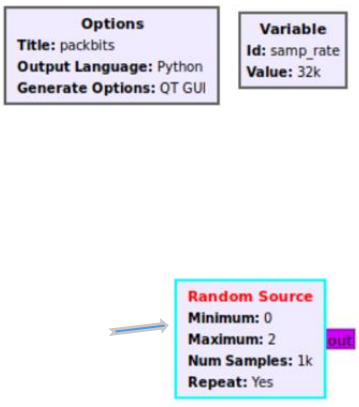


Figura 3.7.1. Cambiamento del tipo di dato in byte.

Random Source genera byte con un valore minimo (Minimum) fino a un valore massimo (Maximum-1). In questo caso, Minimum = 0 e Maximum = 2, quindi creerà 0 e 1 binari. Un blocco *Pack K Bits* viene utilizzato per parallelizzare, o impacchettare, più bit in un singolo byte per rappresentare valori binari più grandi.
Aggiungiamo i blocchi *Throttle, Pack K Bits, Char to Float e QT GUI Histogram Sink* al diagramma di flusso e colleghiamoli, come in Figura 3.7.2.

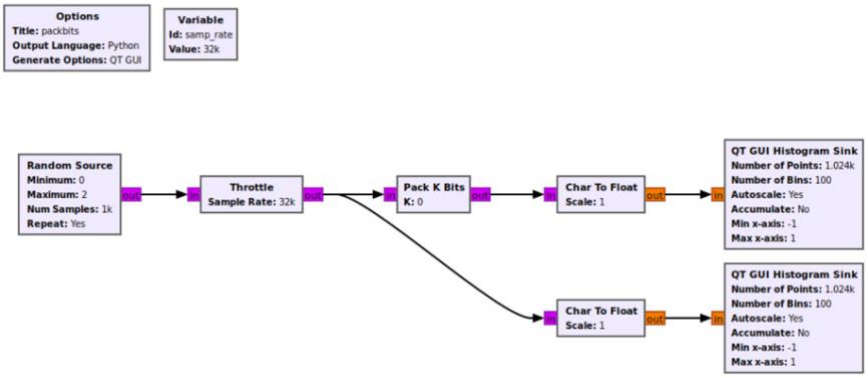


Figura 3.7.2. Diagramma di flusso con blocchi elencati.

3.7.2 Il blocco Packing K Bits

Il blocco *Pack K Bits* prende K bit e li inserisce in un byte riempiendo prima il bit meno significativo (LSB). Vediamo in un esempio dove K=4. La *Random Source* genererà prima il bit B0. Questo verrà ricevuto da Pack K Bits e quindi memorizzato nell'LSB, [0 0 0 0 0 0 0 B0].
Il secondo bit generato da Pack K Bits è B1, che viene quindi memorizzato da *Pack K Bits*, [0 0 0 0 0 0 B1 B0].
Seguendo questo andamento, i successivi bit B2 e B3 verranno quindi memorizzati come, [0 0 0 0 B3 B2 B1 B0]. Vediamo esempio in Figura 3.7.3.

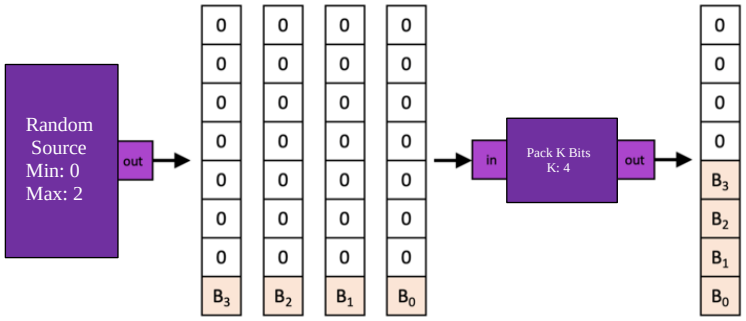


Figura 3.7.3. Inserzione bit in byte.

I Bit sono stati impacchettati, poiché K=4, il byte 0000B3B2B1B0 verrà prodotto come output e verrà avviato un nuovo byte. Il valore di output del byte in decimale (base - 10) è.

= (SI3*23) + (SI2*22) + (SI1*21) + (SI0*20)
= (SI3*8) + (SI2*4) + (SI1*2) + (SI0)

Ad esempio, se:
B0=0
B1=1
B2=0
B3=1

Il byte sarebbe rappresentato da 00001010 e il valore decimale è:

8+0+2+0=10.

3.7.3 Completiamo e impostiamo il diagramma di flusso

Modifichiamo le proprietà del blocco *Pack K Bits*, mettiamo K=4. Quattro bit producono numeri da $2^0=1$ a $2^4-1=15$.

Modifichiamo anche le proprietà del blocco *QT GUI Histogram Sink*.

- Title: 4 bit
- Number of Bins: 1024
- Max x-axis: 16

Modifichiamo anche le proprietà del secondo blocco *QT GUI Histogram Sink.*

- Title: 1 bit
- Number of Bins: 1024
- Max x-axis: 16

Il diagramma di flusso dovrebbe risultare, come in Figura 3.7.4.

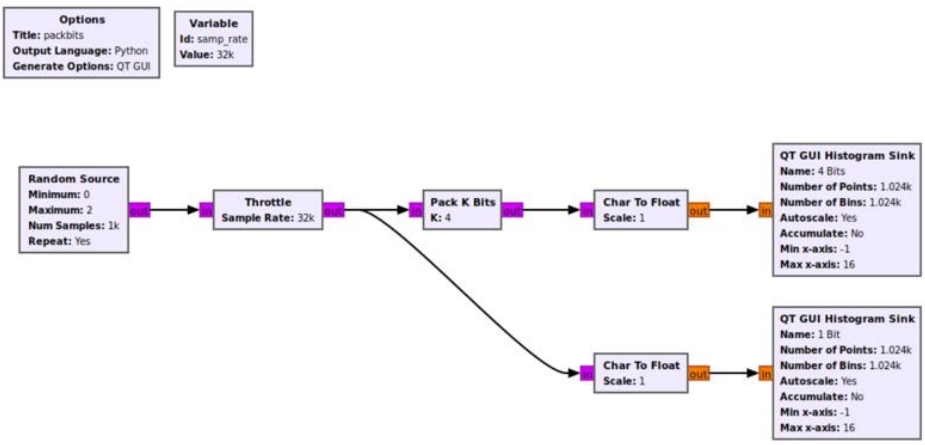


Figura 3.7.4. Diagramma di flusso con le modifiche.

Eseguiamo il diagramma di flusso. L'istogramma a 1 bit mostra i valori 0 e 1, mentre l'istogramma a 4 bit mostra i valori da 0 a 15, come in Figura 3.7.5.

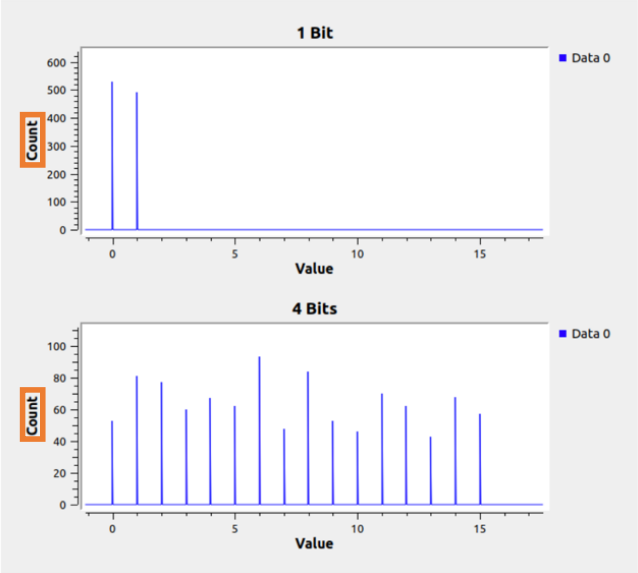


Figura 3.7.5. Istogramma a 1 e a 4 bit.

3.7.4 Decompressione dei Bit

La decompressione serializza un byte in una stringa di bit. Aggiungiamo il blocco *Unpack K Bits* all'area di lavoro e colleghiamo tra il blocco *Pack K Bits* e il blocco *Char to Float*, come in Figura 3.7.6. Modifichiamo le proprietà del blocco *Unpack K Bits* e scriviamo K=4.

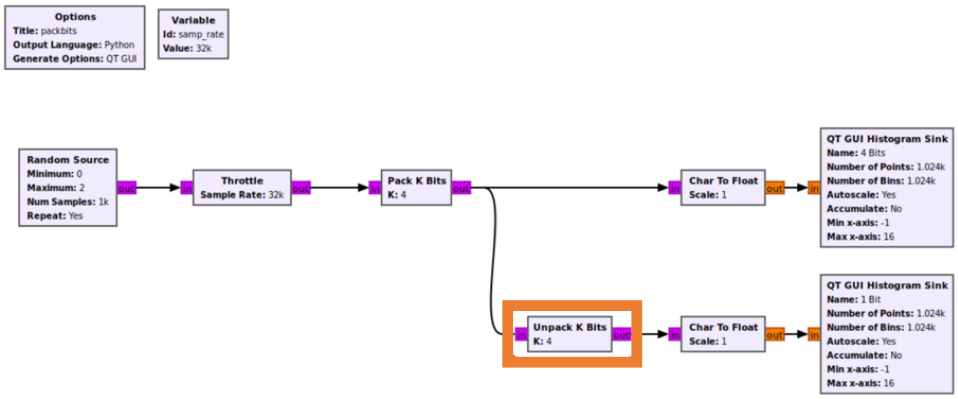


Figura 3.7.6. Aggiunta del blocco *Unpack K Bits.*

Eseguiamo il diagramma di flusso. L'istogramma a 1 bit mostra che i 4 bit impacchettati vengono decompressi (serializzati) nei valori 0 e 1, come in Figura 3.7.7.

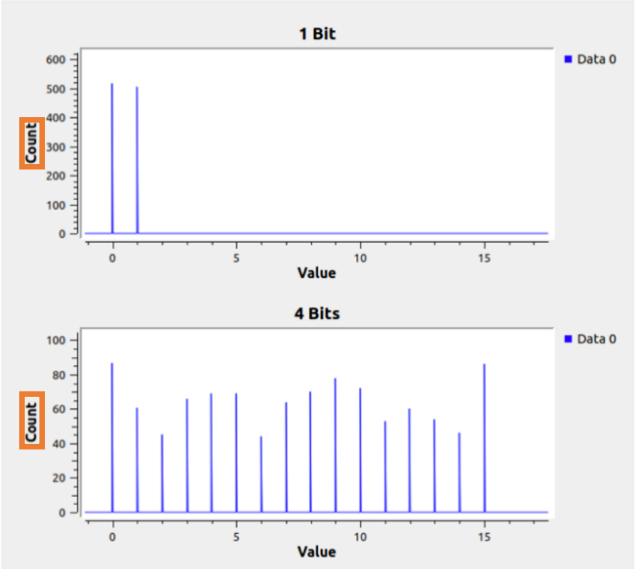


Figura 3.7.7. Decompressione dei bit.

La decompressione inizia prima con (LSB) e procede al bit più significativo (MSB). Dall'esempio precedente, i *Pack K Bits* hanno prodotto un byte con i bit 0000B3B2B1B0. Il blocco *Unpack K Bits* produce prima un'uscita con il bit B0, poi B1, B2 e B3 e i 4 zeri rimanenti nel byte vengono ignorati. Il blocco *Unpack K Bits* inverte perfettamente l'operazione del flusso di input *Pack K Bits*. Ciò può essere verificato aggiungendo un blocco *QT GUI Time Sink* con due input. Modifichiamo la voce (*Type = Float e Number of Inputs=2*) e colleghiamo diagramma di flusso, come in Figura 3.7.8.

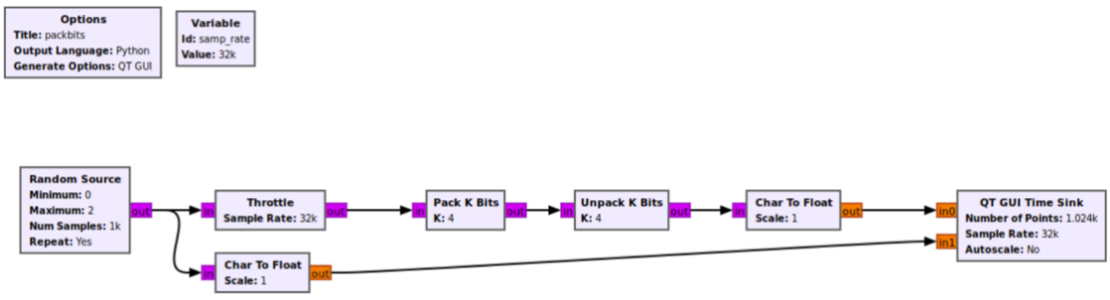


Figura 3.7.8. Diagramma di flusso con le modifiche elencate sopra.

Eseguiamo diagramma di flusso precedente e otterremo un grafico come in Figura 3.7.9. Per visualizzare il grafico nel dettaglio, cliccare con il pulsante sinistro del mouse e trascinare su una porzione più piccola per ingrandire.

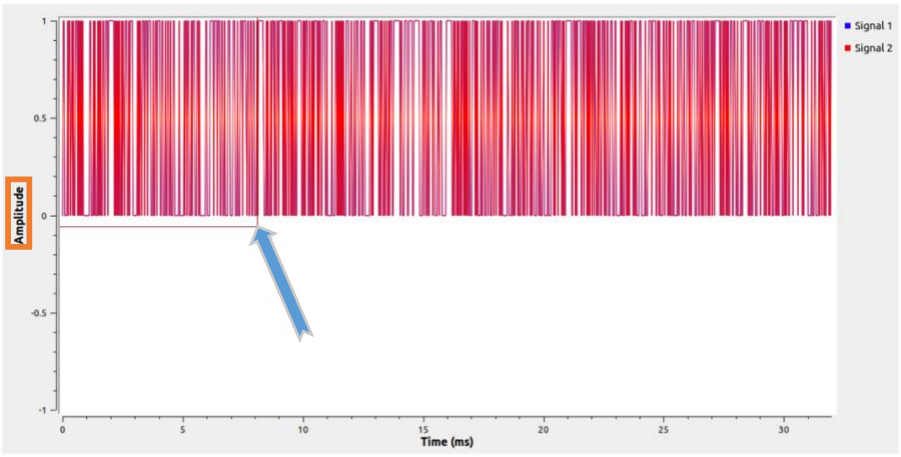


Figura 3.7.9. Zoom del Grafico.

Le due trame sono perfettamente sovrapposte. L'input del blocco Pack K Bits è esattamente lo stesso dell'output del blocco Unpack K Bits. Ciò dimostra come Pack K Bits e Unpack Bits eseguono operazioni inverse perfette, come in Figura 3.7.10.

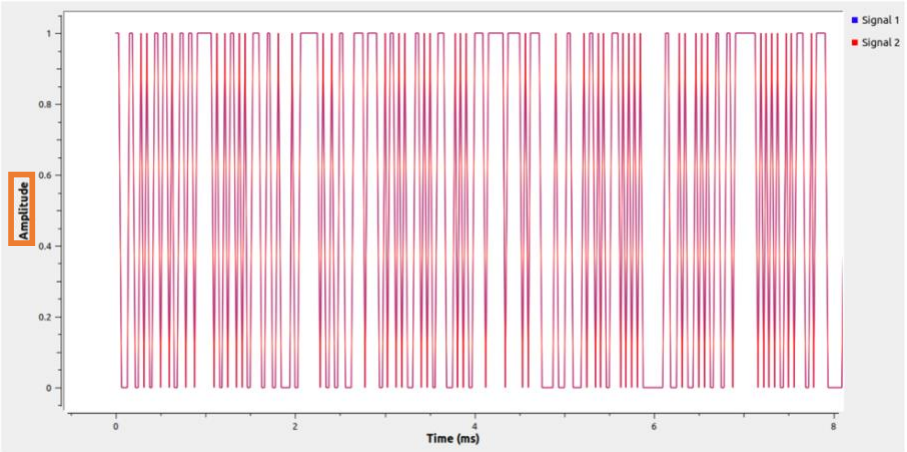


Figura 3.7.10. Diagramma dati perfettamente sovrapposte.

3.8 Flussi e vettori

Descriviamo le differenze tra un flusso e un vettore [8].

3.8.1 Flussi

I blocchi in GnuRadio possono essere collegati utilizzando flussi o vettori. Un flusso trasporta 1 campione per ogni istanza temporale e produce dati serializzati. Un flusso deve avere un tipo di dati, ad esempio Float 32 o Byte. Prendiamo un blocco *Signal Source* che produce un flusso Complex Float 32, come in Figura 3.8.1.

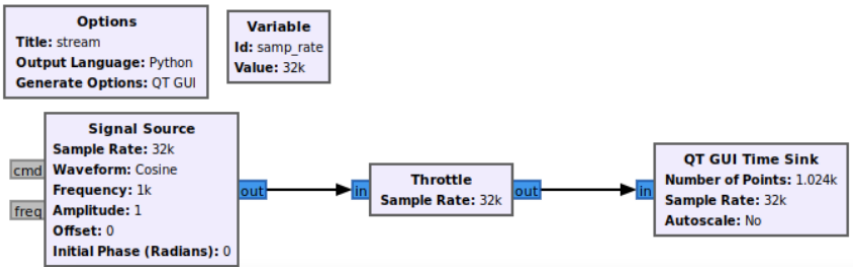


Figura 3.8.1. Diagramma di flusso con dato Complex Float 32.

Proviamo a eseguire il nostro diagramma di flusso e vedremo che l'output del blocco in ogni istanza di tempo contiene 1 campione complesso, come in Figura 3.8.2.

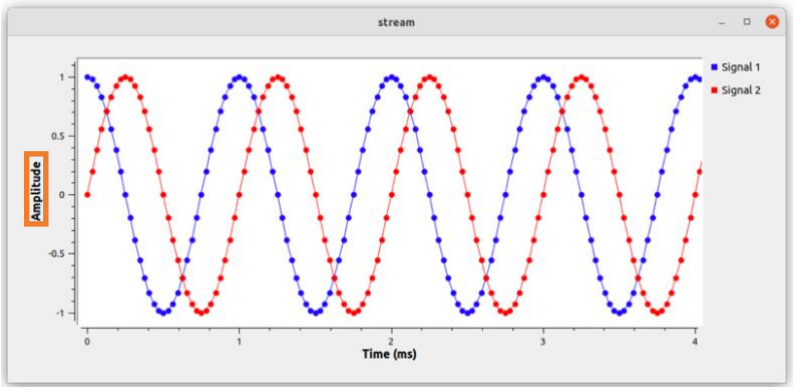


Figura 3.8.2. Grafico con campioni complessi.

3.8.2 Vettori

I vettori trasportano più campioni per istanza temporale e rappresentano dati in parallelo. Un flusso rappresenta un'istanza scalare in ogni momento. Un vettore rappresenta un array in ogni istanza di tempo. GRC utilizza colori più chiari per rappresentare i flussi e colori più scuri per rappresentare gli output vettoriali, come in Figura 3.8.3.

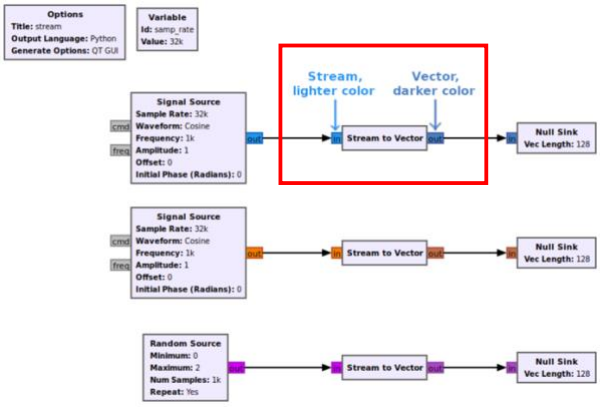


Figura 3.8.3. Rappresentazione colori dei flussi e vettori.

3.8.3 Esempio di diagramma di flusso da flussi a vettore

L'esempio seguente descrive come convertire un flusso in un vettore e viceversa. Due flussi sinusoidali complessi vengono convertiti in un vettore a due elementi, visualizzati e quindi riconvertiti nei loro due flussi indipendenti. Aggiungiamo due blocchi *Signal Source* all'area di lavoro. Modifichiamo i parametri della seconda sorgente di segnale in modo che abbia una frequenza di 100Hz e un'ampiezza di 0,1 per distinguerla visivamente dalla prima sorgente di segnale. Prendiamo altro blocco *Streams to Vector* e mettiamolo nell'area di lavoro collegandolo ai blocchi *Signal Source*.
Il blocco *Streams to Vector* funge da collegamento. Il blocco preleva un campione dalla porta in0 e lo inserisce nel primo elemento nel vettore di output. Il blocco *Streams to Vector* preleva un campione dalla porta in1 e lo inserisce nel secondo elemento nel vettore di output. Il blocco combina due input di flussi in un output vettoriale bidimensionale, come in Figura 3.8.4.

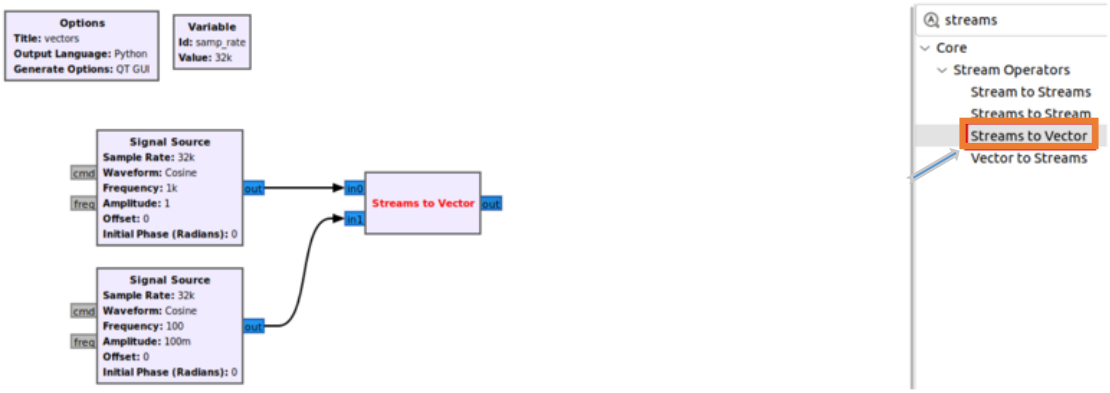


Figura 3.8.4. Convertitore da flussi a vettori.

Prendiamo il blocco *Vector to Stream* e aggiungiamolo al diagramma di flusso, prendiamo anche tre blocchi *QT GUI Time Sink* e aggiungiamoli al diagramma di flusso, come in Figura 3.8.5. Il blocco *Vector to Stream* serializzerà il vettore in un flusso. I campioni all'uscita di *Vector to Stream* saranno interlacciati.

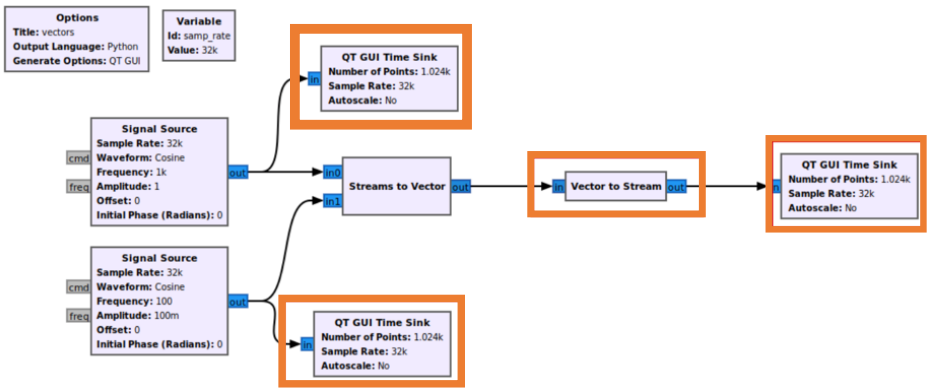


Figura 3.8.5. Diagramma di flusso con blocchi elencati.

Modifichiamo i titoli per i tre blocchi *QT GUI Time Sink* in modo che possano essere distinti l'uno dall'altro. Per prima cosa modifichiamo il titolo (Signal Source A) del blocco *QT GUI Time Sink* connesso al blocco *Signal Source* che ha frequenza 1000Hz e ampiezza 1. Modifichiamo il secondo blocco *QT GUI Time Sink* con il titolo (Signal Source B) che è connesso alla seconda Signal Source con frequenza 100Hz e ampiezza 0.1. Nell'ultimo blocco *QT GUI Time Sink* mettiamo il titolo (Interleaved Signal Sources) connesso al blocco *Vector to Stream*, come in Figura 3.8.6.

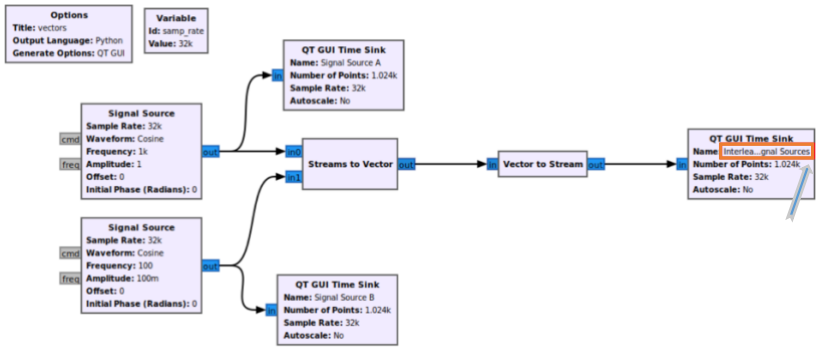


Figura 3.8.6. Blocchi *QT GUI Time Sink* con proprietà modificati.

L'esecuzione del diagramma di flusso visualizza tre sinusoidi in funzione del tempo, Signal Source A, Signal Source B e Interleaved Signal Sources. Il grafico di Interleaved Signal Sources mostra campioni interlacciati da Signal Source A e Signal Source B, come in Figura 3.8.7.

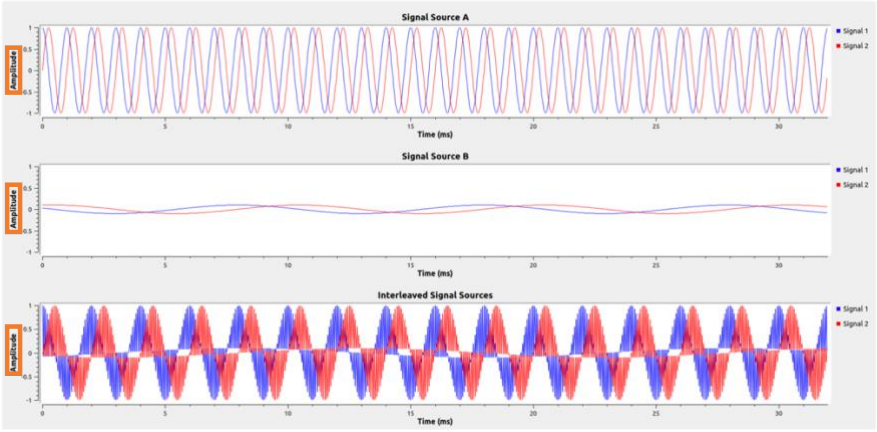


Figura 3.8.7. Output in funzione del tempo da flusso a vettori.

3.8.4 Esempio di diagramma di flusso da vettore a flussi

L'esempio seguente serializza i dati vettorizzati, riconvertendoli in due flussi. Prendiamo il nostro diagramma di flusso e facciamo qualche modifica. Aggiungiamo il blocco *Vector to Streams* e colleghiamolo al blocco *Streams to Vector*. Il blocco *Vector to Streams* deserializza i campioni vettoriali e li converte in flussi, eseguendo l'operazione inversa. Successivamente prendiamo i nostri due blocchi *QT GUI Time Sink,* li spostiamo e li

riconnettiamo alle uscite del blocco *Vector to Streams,* ottenendo diagramma di flusso seguente, come in Figura 3.8.8.

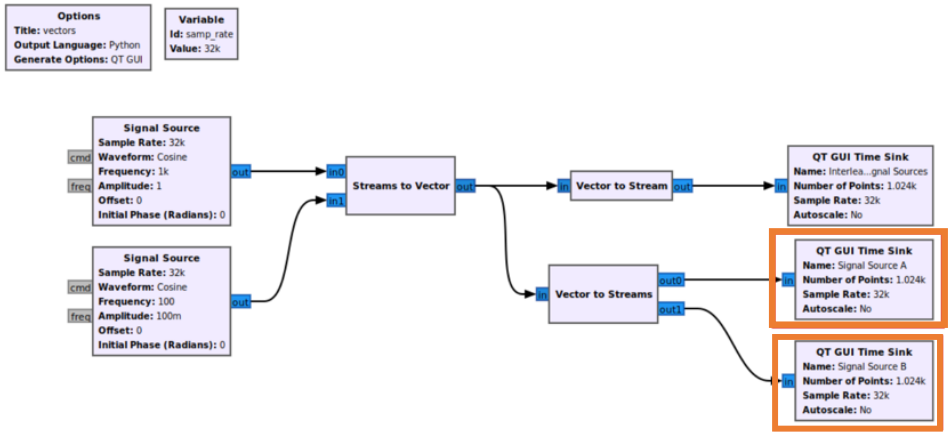


Figura 3.8.8. Diagramma di flusso da vettori a flussi.

Eseguiamo il diagramma di flusso. Notiamo che i campioni vettoriali sono stati nuovamente separati in due flussi, come in Figura 3.8.9.

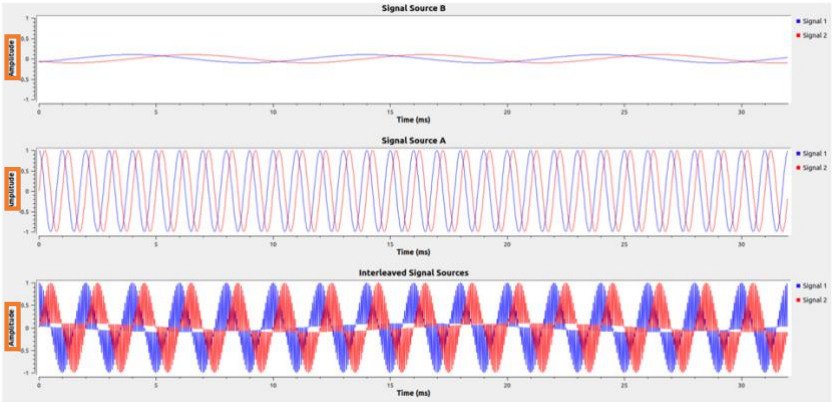


Figura 3.8.9. Output in funzione del tempo da vettori a flussi.

3.9 Blocchi Gerarchici e Parametri

Descriviamo come creare un blocco gerarchico, o blocco Hier, in GRC [9].

3.9.1 Creazione del diagramma di flusso iniziale

Un blocco gerarchico viene utilizzato come scatola per semplificare più blocchi GnuRadio in un singolo blocco. Il blocco gerarchico di esempio sarà un blocco del cambio di frequenza che moltiplica una sorgente di segnale rispetto a un segnale di ingresso, come in Figura 3.9.1.

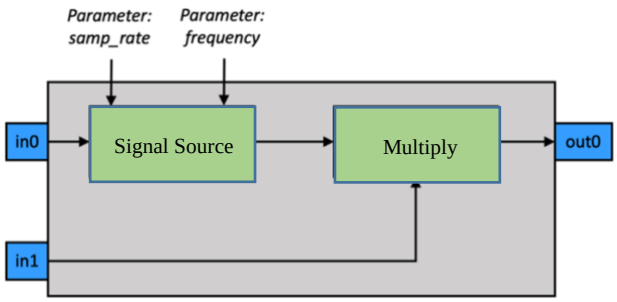


Figura 3.9.1. Blocco gerarchico FrequencyShifter.

Il primo passo è creare il diagramma di flusso. Prendiamo i seguenti blocchi nell'area di lavoro, *Signal Source, Multiply, Noise Source, Low Pass Filter, Throttle, QT GUI Frequency Sink, QT GUI Range* e li colleghiamo, come in Figura 3.9.2.

Aggiorniamo le proprietà del blocco *QT GUI Range*:

- Id: frequency
- Default Value: 0Hz
- Start: -samp_rate/2
- Stop: samp_rate/2

Aggiorniamo anche le proprietà del blocco *Low Pass Filter:*

- Cutoff Freq (Hz): samp_rate/4
- Transition Width (Hz): samp_rate/8

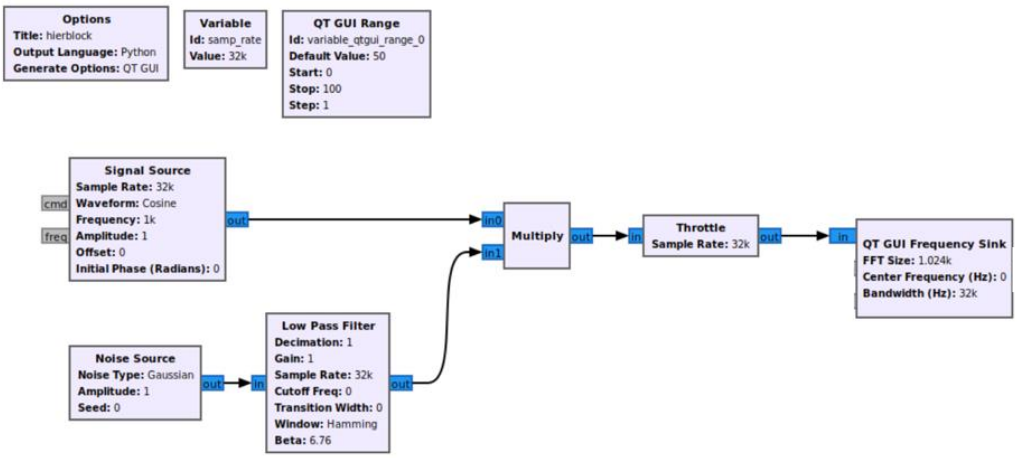


Figura 3.9.2. Diagramma di flusso di esempio.

3.9.2 Creazione del blocco gerarchico

Nell'area di lavoro selezioniamo i blocchi *Signal Source* e *Multiply* inclusa la loro connessione. I blocchi selezionati saranno visibili di colore azzurro. Facciamo clic con il tasto destro sui blocchi evidenziati e selezioniamo (More - Create Hier), come in Figura 3.9.3.

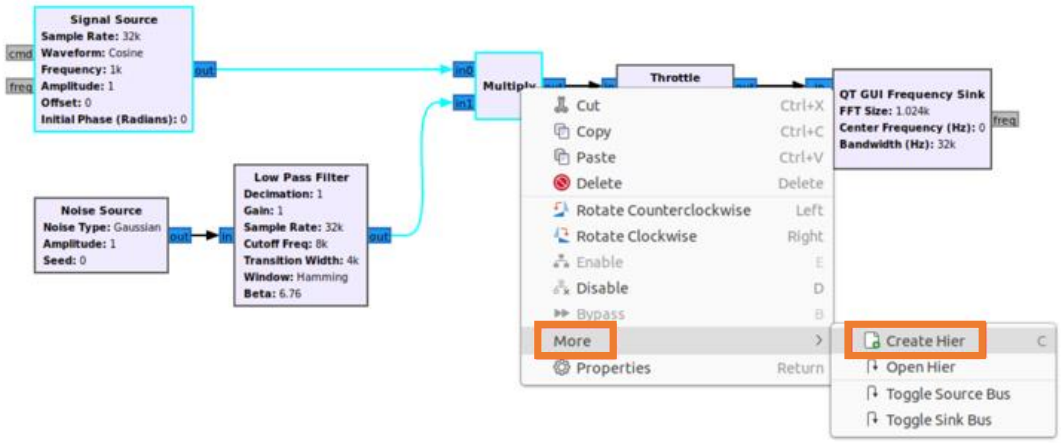


Figura 3.9.3 Passi per creare blocco gerarchico.

Un nuovo diagramma di flusso viene creato in una nuova scheda GRC, come in Figura 3.9.4.

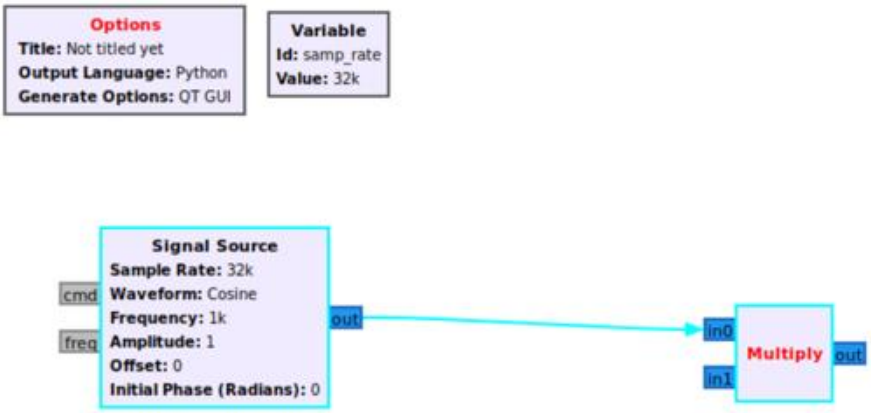


Figura 3.9.4. Nuovo digramma di flusso dopo le operazioni svolte.

Cambiamo le proprietà del blocco *Option*.

- Id: FrequencyShifter
- Title: Frequency Shifter Block
- Generate Options: Hier Block

Le restanti proprietà cambieranno mostrando la Category. La Category si troverà nella libreria dei blocchi sulla destra di GRC. Il blocco gerarchici si troverà sotto la categoria

GRC Hier Blocks, invece di Core dove si trovano gli altri blocchi GnuRadio, come in Figura 3.9.5.

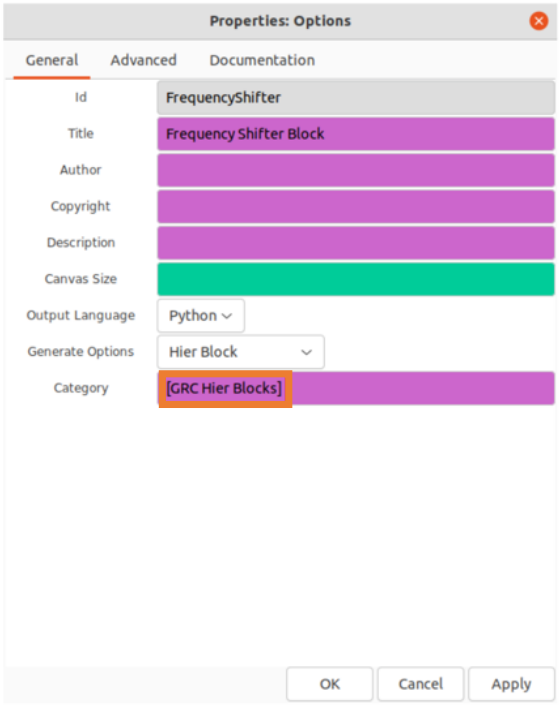


Figura 3.9.5. Modifica delle proprietà del blocco *Option.*

3.9.3 Variabili e Parametri

Una variabile è diversa da un parametro in GnuRadio. Un parametro crea un'interfaccia per il blocco gerarchico per accettare un valore da una fonte esterna, dove variabile esiste solo internamente al blocco gerarchico, come in Figura 3.9.6.

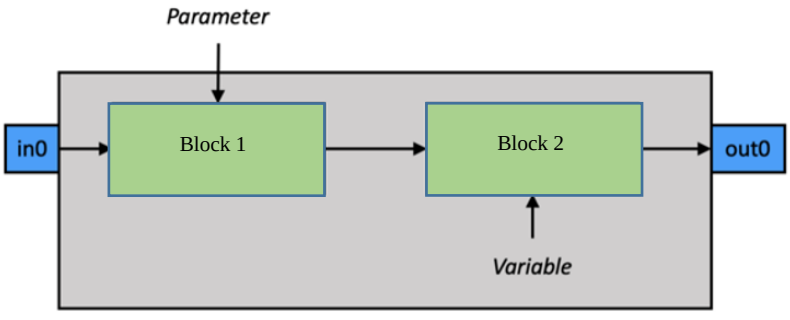


Figura 3.9.6. Blocco gerarchico.

Ad esempio, è possibile accedere alla variabile samp_rate solo dall'interno del blocco gerarchico. Il samp_rate deve essere convertito in un parametro in modo che possa essere aggiornato da un altro blocco nel diagramma di flusso più grande. Eliminiamo la

variabile samp_rate e aggiungiamo un blocco *Parameter* nell'area di lavoro GRC, come in Figura 3.9.7.

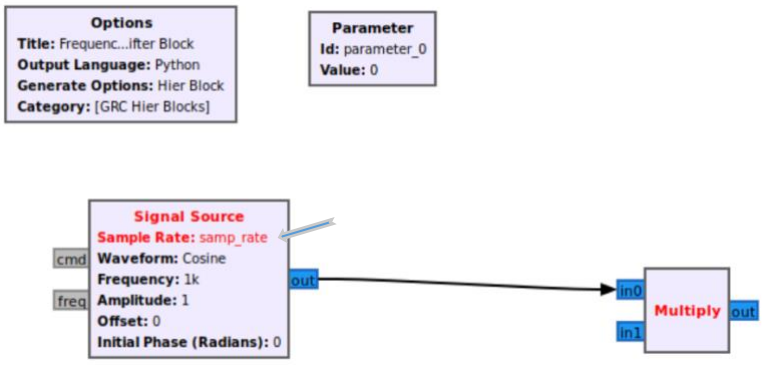


Figura 3.9.7. Diagramma di flusso con blocco *Parameter.*

Modifichiamo le proprietà del blocco *Parameter*:

- Id: samp_rate
- Label: Sample Rate
- Type: Float

Aggiungiamo un altro blocco *Parameter* e modifichiamo le sue proprietà:

- Id: frequency
- Label: Frequency
- Type: Float

Aggiungiamo e modifichiamo le proprietà del blocco *Signa Source*. Sostituiamo la voce *Frequency* in *frequency*, come in Figura 3.9.8.

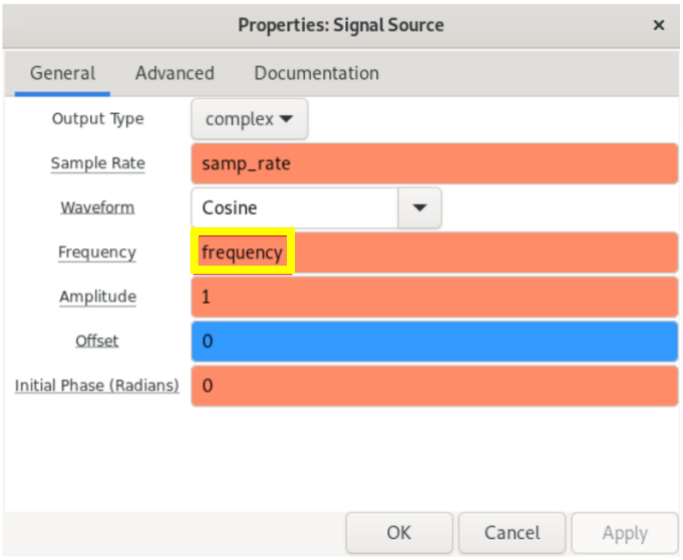


Figura 3.9.8. Modifica proprietà blocco *Signal Source.*

Otterremo il diagramma di flusso seguente, come in Figura 3.9.9.

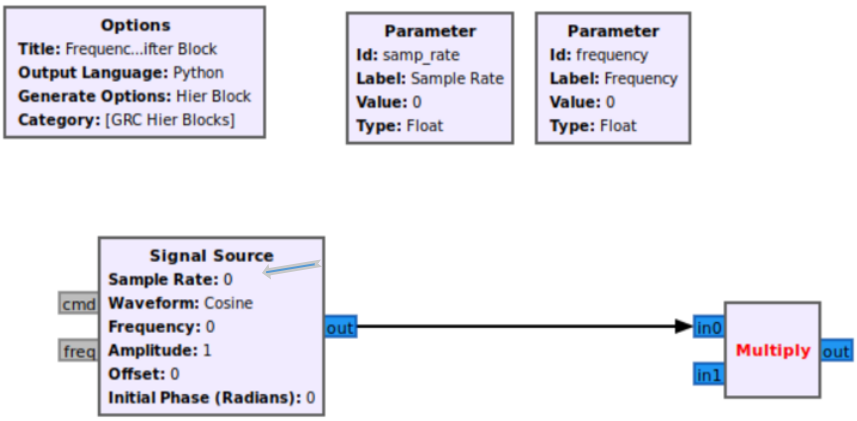


Figura 3.9.9. Diagramma di flusso con blocchi modificati.

3.9.4 Porte di ingresso e uscita

Un *Pad* viene utilizzato per specificare le porte di input e output su un blocco gerarchico. Aggiungi un *Pad Source* e un *Pad Sink* al diagramma di flusso per fungere da porte di ingresso e uscita, come in Figura 3.9.10.

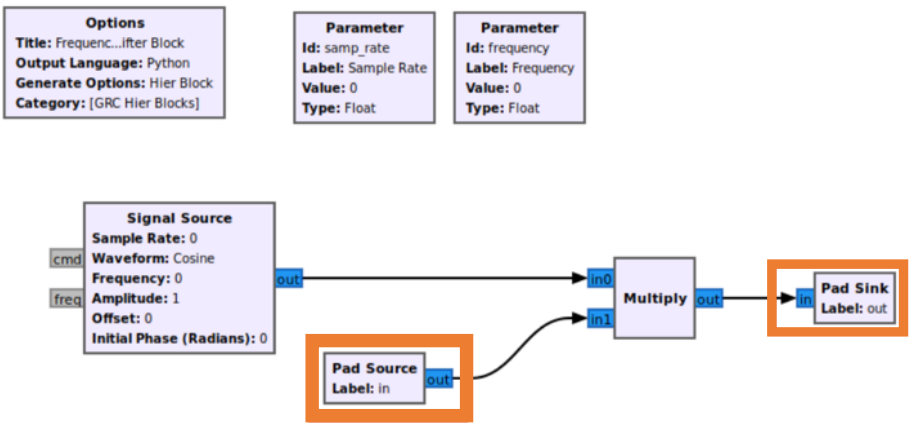


Figura 3.9.10. Diagramma di flusso con blocchi *Pad*

3.9.5 Generiamo il codice Hier Block

Facciamo clic su (Generate the flow graph) per creare il codice sorgente del blocco gerarchico, come in Figura 3.9.11.

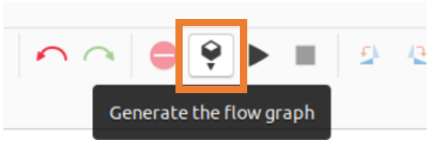


Figura 3.9.11. Comando per generare il codice del blocco gerarchico.

Verranno creati un file Python .py e un file YAML .yml. Per GNURadio v3.8 i file verranno creati nella home directory.

```
/home/$USER/.grc_gnuradio/
```

```
matt@ubuntu:~/.grc_gnuradio$ ls
FrequencyShifter.py  FrequencyShifter.py.block.yml  __pycache__
matt@ubuntu:~/.grc_gnuradio$
```

Per GNU Radio v3.10, i file verranno creati nella directory in cui è salvato il file .grc. Crea la directory .grc_gnuradio e copia i file .py e .yml lì:

```
$ mkdir /home/$USER/.grc_gnuradio
$ cp FrequencyShifter.block.yml /home/$USER/.grc_gnuradio/
$ cp FrequencyShifter.py /home/$USER/.grc_gnuradio/
```

GRC deve aggiornare l'elenco interno dei blocchi prima che il blocco *Frequency Shifter* possa essere utilizzato in un diagramma di flusso. Fare clic sul pulsante (Reload Blocks), come in Figura 3.9.12.

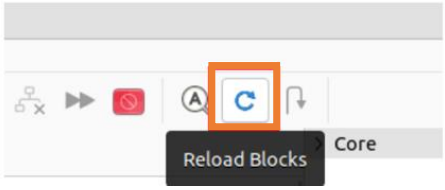


Figura 3.9.12. Comando per aggiornare elenco dei blocchi.

Vedremo che c'è una nuova categoria GRC Hier Blocks nella libreria di blocchi sotto Core e il *Frequency Shifter Block* può essere utilizzato nei diagrammi di flusso, come in Figura 3.9.13.

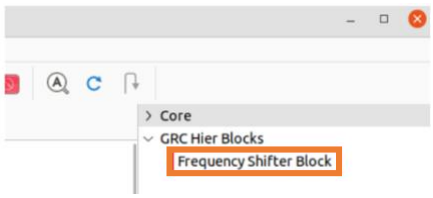


Figura 3.9.13. Nuova categoria nella libreria dei blocchi.

3.9.6 Utilizzo del blocco gerarchico

Il blocco gerarchico ora può essere utilizzato in un diagramma di flusso. Nel diagramma di flusso iniziale vengono eliminati i blocchi *Signal Source* e *Multiply*. Aggiungiamo il blocco appena creato *Frequency Shifter Block* all'area di lavoro e colleghiamolo al resto del diagramma di flusso, come in Figura 3.9.14.

Modifichiamo anche le proprietà del *Frequency Shifter Block* aggiungendo le variabili:

- Frequency: frequency
- Sample Rate: samp_rate

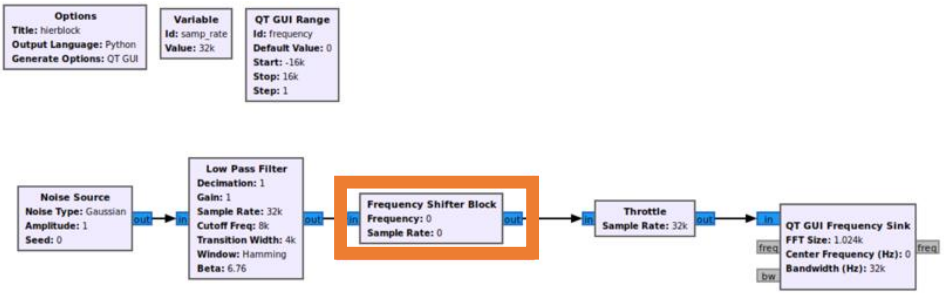


Figura 3.9.14. Diagramma di flusso con un blocco appena creato.

L'esecuzione del diagramma di flusso farà apparire la finestra *QT GUI Frequency Sink* con il cursore *QT QUI Range*. Trascinando il cursore della frequenza, il valore passerà attraverso il parametro *Frequency Shifter Block*, causando la modifica della frequenza centrale del segnale, come in Figura 3.9.15.

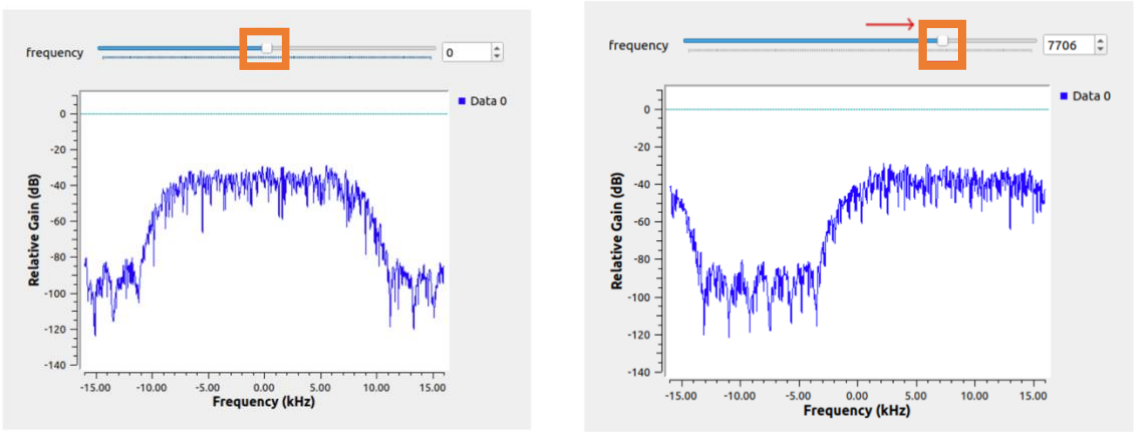


Figura 3.9.15. Output in funzione delle frequenze.

3.9.7 Eliminazione di un blocco gerarchico

Un blocco gerarchico può essere cancellato dalla memoria GRC rimuovendo i file da /home/$USER/.grc_gnuradio. Entriamo in un prompt dei comandi e spostiamoci nella directory .grc_gnuradio.

Per GNU Radio v3.8 il comando per eliminare è.

- rm FrequencyShifter.py FrequencyShifter.py.block.yml

Per GNU Radio v3.10 il comando per eliminare è.

- rm FrequencyShifter.py FrequencyShifter.block.yml

Facciamo ora clic sul pulsante Ricarica blocchi per aggiornare la memoria dei blocchi di GRC. Vedremo che la categoria GRC Hier Blocks viene eliminata e rimangono solo i blocchi Core, come Figura 3.9.16.

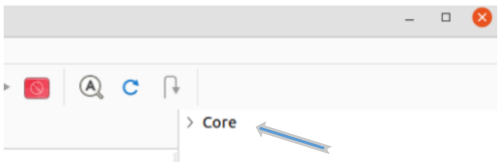


Figura 3.9.16. Libreria dei blocchi aggiornata dopo eliminazione.

Capitolo 4

Creazione e modifica di blocchi Python

In questo capitolo vedremo come verranno creati e modificati dei blocchi Python. Usiamo Embedded Python Block' che può essere utilizzato solo nel diagramma di flusso in cui è stato creato. Per installare e utilizzare un blocco Python in qualsiasi diagramma di flusso leggere sul sito [10].

4.1 Creazione del primo blocco

Vediamo come creare un blocco di elaborazione del segnale con *Embedded Python Block*. Il blocco di esempio somma o moltiplica i due input in base a un parametro. *l'Embedded Python Block'* può essere utilizzato solo nel diagramma di flusso in cui è stato creato [11].

4.1.1 Apertura dell'editor di codice

Embedded Python Block è uno strumento per creare rapidamente un blocco all'interno di un diagramma di flusso. Cerchiamo il *Python Block* e aggiungiamolo all'area di lavoro, come in Figura 4.1.1.

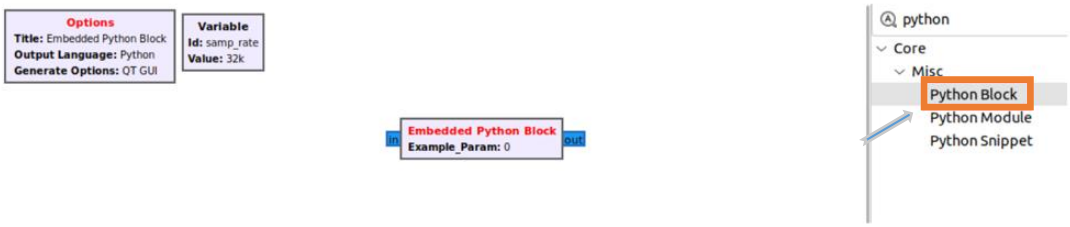


Figura 4.1.1. Diagramma di flusso con *Embedded Python Block.*

Modifichiamo le proprietà del blocco *Embedded Python Block*. Il *Embedded Python Block* ha due proprietà.

1. Code: una casella di selezione che contiene un collegamento al codice Python per il blocco.
2. Example_Param: un parametro di input per il blocco.

Facciamo clic su (Open in Editor) per modificare il codice Python. Viene visualizzato un prompt con la scelta dell'editor di testo da utilizzare per scrivere il codice Python. Fare clic su (Use Default), come in Figura 4.1.2.

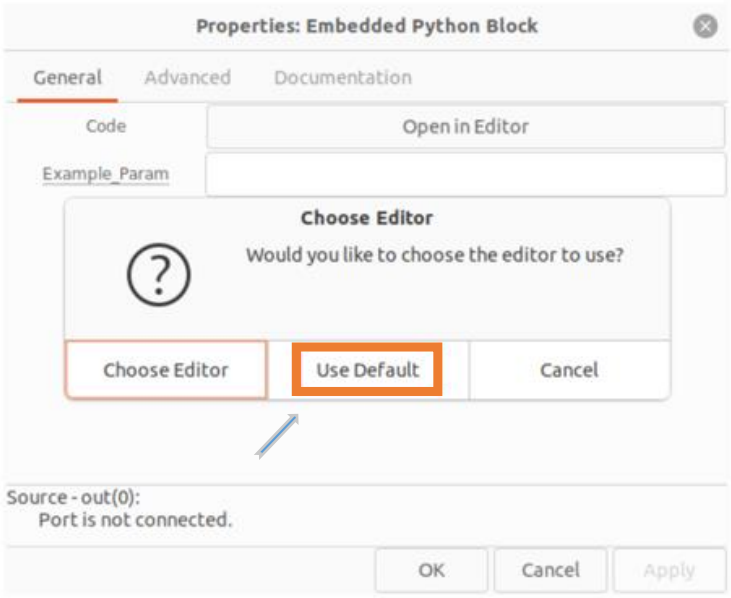
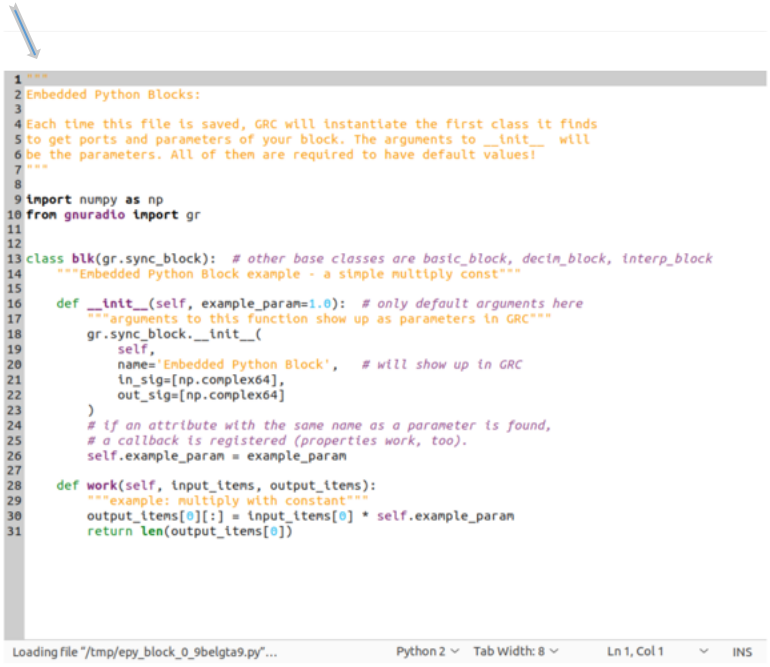


Figura 4.1.2. Proprietà *Embedded Python Block.*

Una finestra dell'editor mostra il codice Python per il *Embedded Python Block*, come in Figura 4.1.3.



```
"""
Embedded Python Blocks:

Each time this file is saved, GRC will instantiate the first class it finds
to get ports and parameters of your block. The arguments to __init__  will
be the parameters. All of them are required to have default values!
"""

import numpy as np
from gnuradio import gr


class blk(gr.sync_block):  # other base classes are basic_block, decim_block, interp_block
    """Embedded Python Block example - a simple multiply const"""

    def __init__(self, example_param=1.0):  # only default arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='Embedded Python Block',   # will show up in GRC
            in_sig=[np.complex64],
            out_sig=[np.complex64]
        )
        # if an attribute with the same name as a parameter is found,
        # a callback is registered (properties work, too).
        self.example_param = example_param

    def work(self, input_items, output_items):
        """example: multiply with constant"""
        output_items[0][:] = input_items[0] * self.example_param
        return len(output_items[0])
```


Figura 4.1.3. Finestra per la modifica di codice Python.

4.1.2 Componenti di un blocco Python

Ci sono tre sezioni importanti nel codice a blocchi di Python, come in Figura 4.1.4.

1. Dichiarazioni di importazione in verde
2. Funzione __init__ (*__init__* function) in blu
3. Funzione lavoro (*work()*) in rosso

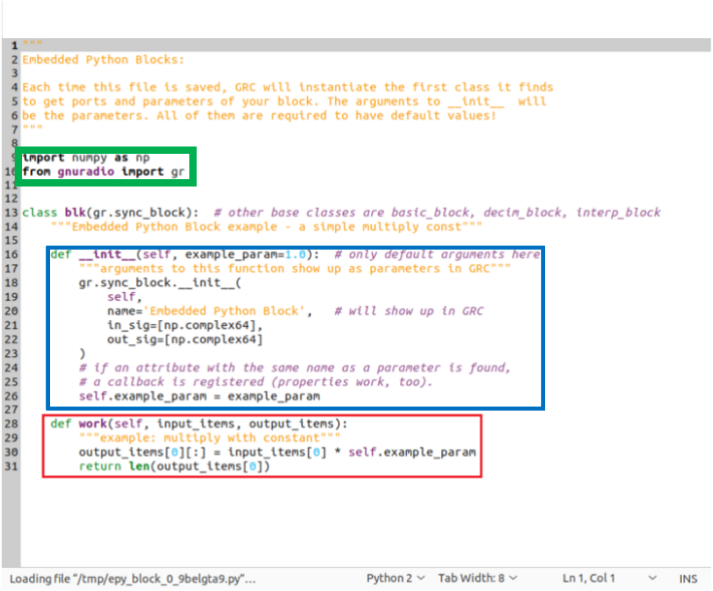


Figura 4.1.4. Sezioni codice Python.

1. La dichiarazione di importazione include le librerie NumPy and GNU Radio.
2. Funzione _init_.
    - Acetta il parametro example_param con un argomento predefinito di 1.0
    - Dichiara che il blocco ha un input e un output np.complex64, che è il tipo di dato GNU Radio Complex Float 32
    - Memorizza la veriabile self.example_param del parametro di input
3. La funzione *work()*.
    - Ha i parametri input_items e output output_items
    - Applica un'operazione matematica a input_items e archivia il risultato in output_items
    - Restituisce il numero di campioni prodotto

4.1.3 Modifica del nome del parametro

Il codice viene modificato per aggiungere il comportamento personalizzato. Il primo passo è rinominare example_param in additionalFlag per essere più descrittivo. Dal menu dell'editor selezionare Find and Replace, come in Figura 4.1.5.

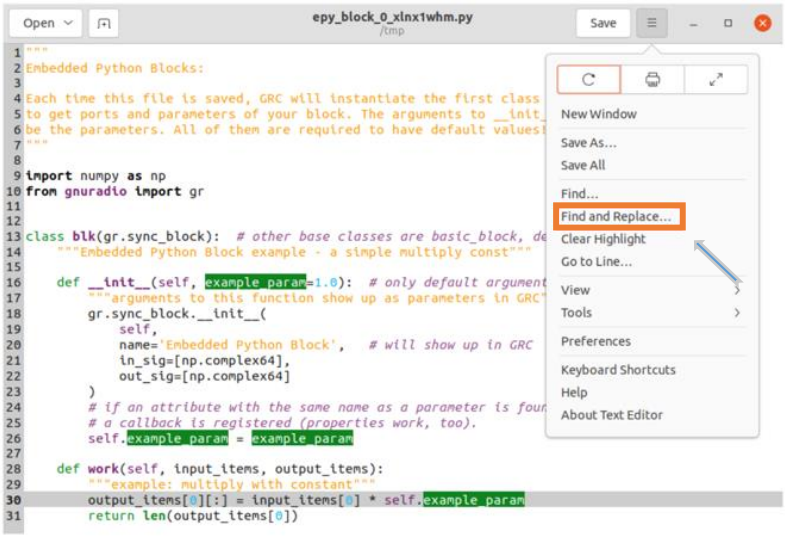


Figura 4.1.5. Modifica del nome del parametro.

Accediamo e cambiamo i seguenti parametri: Find > example_param, Replace with > additionFlag, e facciamo click su Replace All, come in Figura 4.1.6.

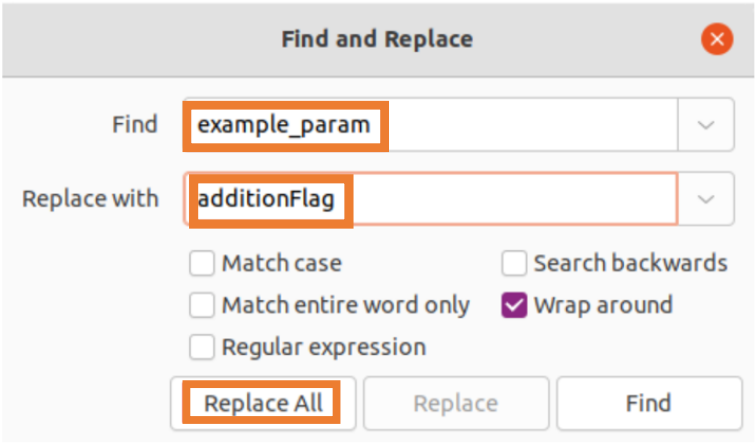


Figura 4.1.6. Cambiamento dei parametri.

Il parametro viene modificato e il codice Python viene aggiornato, come in Figura 4.1.7.

Figura 4.1.7. Aggiornamento codice Python.

Proviamo adesso a cambiare il valore predefinito in True invece di 1.0. Per fare ciò devo entrare in Find and Replace e cambiare parametri in modo seguente.

Find > additionFlag=1.0
Replace with > adittionFlag=True

Dopo di che clicchiamo su Replace All. Infine salviamo tutto cliccando su Salva e ritorniamo alla finestra di GRC. Notiamo che il blocco Python incorporato visualizza il parametro Additionflag: 1, invece di Example_Param: 0, come in Figura 4.1.8.

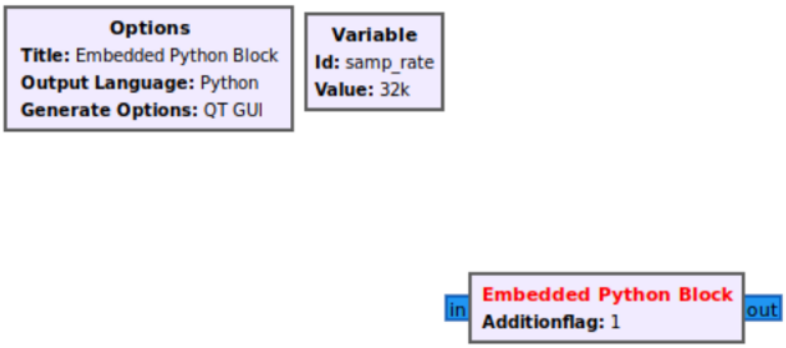


Figura 4.1.8. Blocco *Embedded Python Block* con le modifiche.

4.1.4 Modifica degli ingressi del blocco

Il blocco predefinito ha un singolo input e un singolo output; tuttavia, abbiamo bisogno di due input per il blocco. Per aggiungere un input, aggiungiamo un secondo np.complex64 all'elenco in_sig nella stessa maniera come abbiamo fatto sopra con Find and Replace. Cambiamo anche il nome del blocco con Add or Multiply Block, come in Figura 4.1.9.

```
"""
Embedded Python Blocks:

Each time this file is saved, GRC will instantiate the first class it finds
to get ports and parameters of your block. The arguments to __init__  will
be the parameters. All of them are required to have default values!
"""

import numpy as np
from gnuradio import gr


class blk(gr.sync_block):  # other base classes are basic_block, decim_block, interp_block
    """Embedded Python Block example - a simple multiply const"""

    def __init__(self, additionFlag=True):  # only default arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='Add or Multiply Block',   # will show up in GRC
            in_sig=[np.complex64,np.complex64],
            out_sig=[np.complex64]
        )
        # if an attribute with the same name as a parameter is found,
        # a callback is registered (properties work, too).
        self.additionFlag = additionFlag

    def work(self, input_items, output_items):
        """example: multiply with constant"""
        output_items[0][:] = input_items[0] * self.additionFlag
        return len(output_items[0])
```

Figura 4.1.9. Aggiunta di secondo ingresso.

Otteniamo il seguente diagramma di flusso, come in Figura 4.1.10.

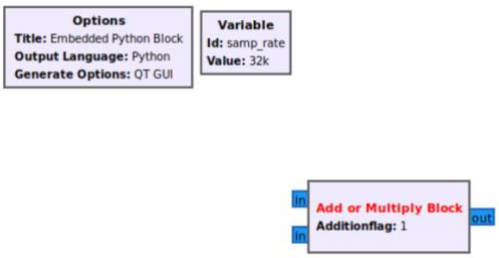


Figura 4.1.10. Diagramma di flusso con blocco modificato.

4.1.5 Modifica della Funzione Lavoro (*work()*)

La funzione *work()* deve essere modificata con il seguente pseudo-codice per il blocco Python.

if (additionFlag is True)
  then add the two inputs
else
  then multiply the two inputs

Ricordare di rientrare con multipli di 4 spazi (4, 8, 12, ecc.) quando si iniziano nuove righe in Python. Dopo la modifica del codice ricordare di salvare, come in Figura 4.1.11.

```python
"""
Embedded Python Blocks:

Each time this file is saved, GRC will instantiate the first class it finds
to get ports and parameters of your block. The arguments to __init__  will
be the parameters. All of them are required to have default values!
"""

import numpy as np
from gnuradio import gr


class blk(gr.sync_block):  # other base classes are basic_block, decim_block, interp_block
    """Embedded Python Block example - a simple multiply const"""

    def __init__(self, additionFlag=True):  # only default arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='Add or Multiply Block',   # will show up in GRC
            in_sig=[np.complex64,np.complex64],
            out_sig=[np.complex64]
        )
        # if an attribute with the same name as a parameter is found,
        # a callback is registered (properties work, too).
        self.additionFlag = additionFlag

    def work(self, input_items, output_items):
        """example: add or multiply based on flag"""
        if (self.additionFlag == True):
            output_items[0][:] = input_items[0][:] + input_items[1][:]
        else:
            output_items[0][:] = input_items[0][:] * input_items[1][:]
        return len(output_items[0])
```

Figura 4.1.11. Modifica della finzione *(work()).*

4.1.6 Collegamento del diagramma di flusso

Ritorniamo al GRC. Facciamo doppio clic sul blocco *Add or Multiply Block*, modifichiamo la proprietà della Additionflag con Tru e salviamo.
Prendiamo i due blocchi *Signal Source*, un blocco *Throttle*, un blocco *QT GUI Time Sink* e un blocco *QT GUI Frequency Sink* nell'area di lavoro GRC e colleghiamoli secondo il seguente diagramma di flusso, come in Figura 4.1.12. Impostiamo la frequenza del secondo blocco *Signal Source* su 3000Hz.

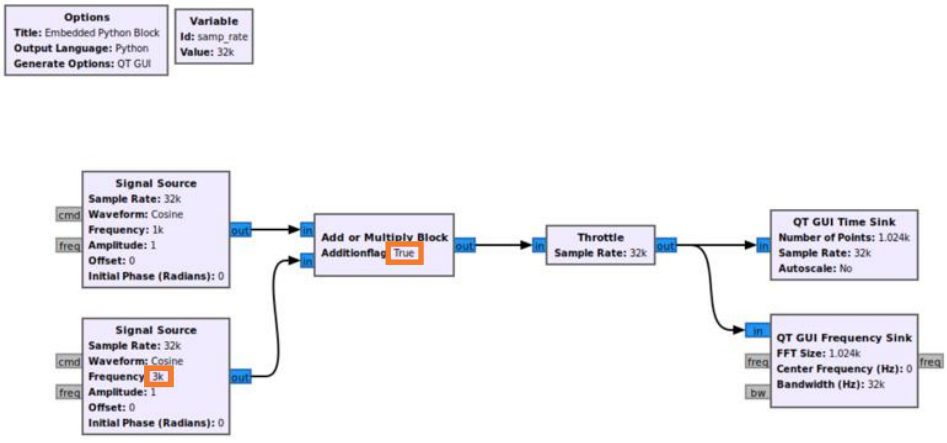


Figura 4.1.12. Diagramma di flusso con blocchi aggiunti e modificati.

Eseguiamo il diagramma di flusso. Selezionando True nel blocco *Add or Multiply Block* si esegue la somma dei segnali delle due sorgenti. I grafici mostrano la somma delle due sinusoidi, una con frequenza di 1000Hz e l'altra a 3000Hz. L'asse y nel grafico *QT GUI Time Sink* taglia parzialmente l'ampiezza delle sinusoidi. Facciamo clic sul pulsante della rotellina per visualizzare il menu di visualizzazione e selezionare Auto Scale, come in Figura 4.1.13.

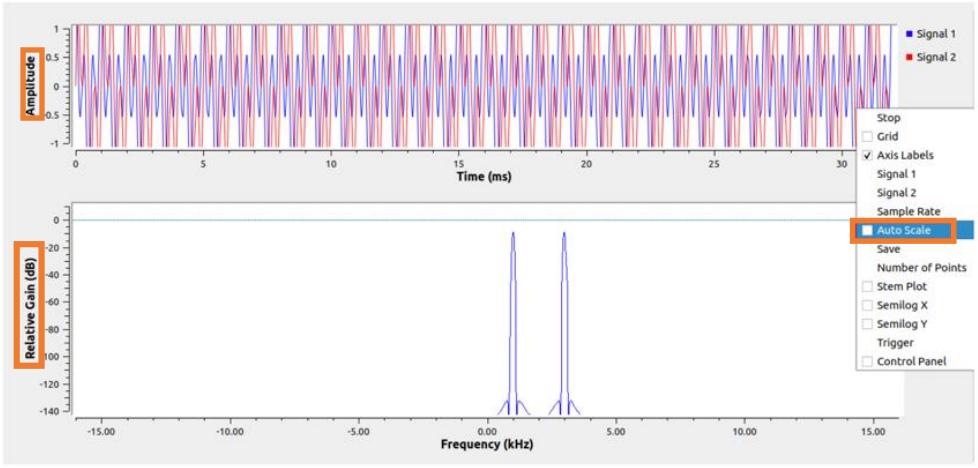


Figura 4.1.13. Output diagramma di flusso eseguita con l'ampiezza tagliata.

Si può quindi vedere l'ampiezza completa delle due sinusoidi, come in Figura 4.1.14.

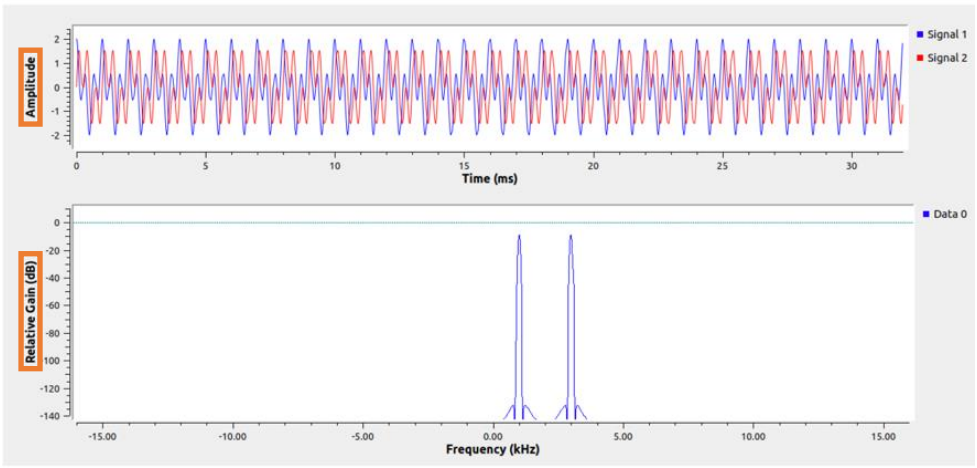


Figura 4.1.14. Output diagramma di flusso eseguita con ampiezza completa.

Fermiamo il diagramma di flusso chiudendo la *QT GUI Time Sink* o premendo il pulsante quadrato in GRC, come in Figura 4.1.15.

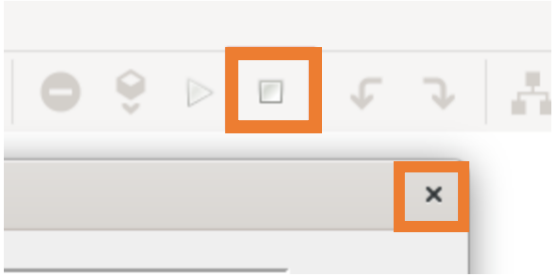


Figura 4.1.15. Arresto diagramma di flusso.

Cambiamo le proprietà nel blocco *Add o Multiply Block,* nella voce Additionflag cambiamo con False*,* come in Figura 4.1.16.

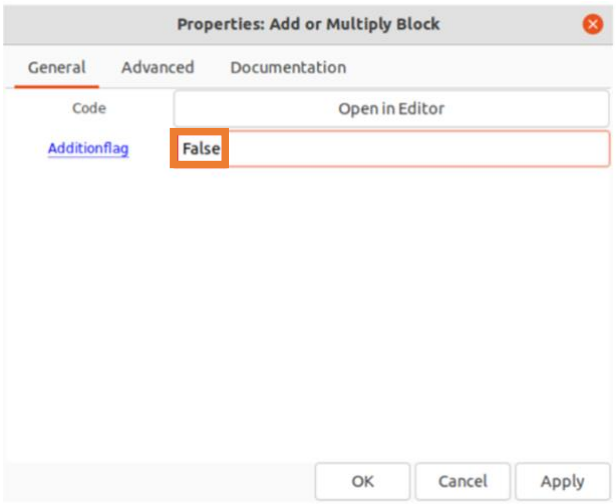


Figura 4.1.16. Modifica proprietà blocco *Add o Multiply Block.*

Per definizione, la moltiplicazione di due sinusoidi complessi produce una sinusoide con la somma delle due frequenze. Pertanto, la moltiplicazione della sorgente del segnale di frequenza 1000Hz e frequenza 3000Hz è una sinusoide complessa di frequenza 4000Hz. Questa sinusoide complessa viene visualizzata durante l'esecuzione del diagramma di flusso, come in Figura 4.1.17.

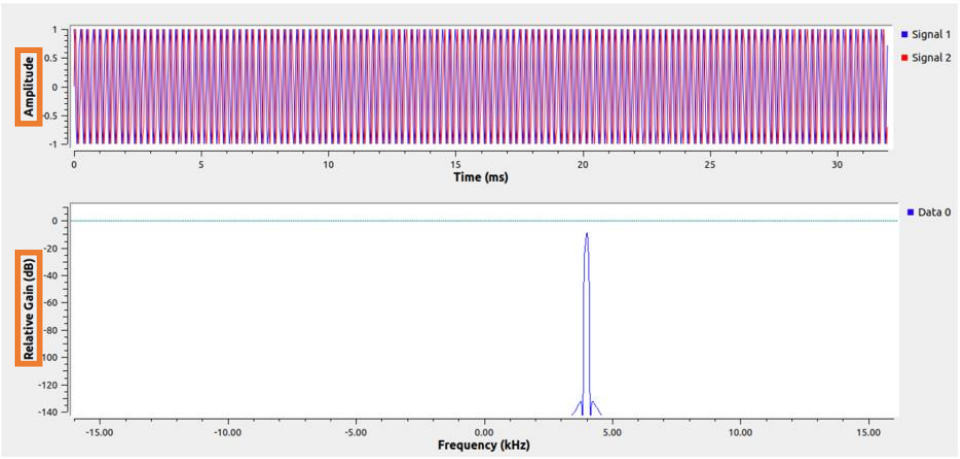


Figura 4.1.17. Sinusoide complessa.

4.2 Blocco Python con vettori

Blocco Python con vettori descrive come il Python Embedded Block può essere modificato per accettare input e output vettoriali e come l'indicizzazione del vettore input_items è diversa tra vettori e flussi [12].

Creiamo prima il diagramma di flusso aggiungendo i seguenti blocchi. *Signal Source, Throttle, Stream to Vector, Embedded Python Block, Vector to Stream, QT GUI Time Sink, Virtual Sink, Virtual Source, Variable.*

Modifichiamo le proprietà del blocco *Signal Source.*

- Output Type: float
- Frequency: 100Hz

Modifichiamo le proprietà del blocco *Variable*.

- Id: vectorLength
- Value: 16Hz

Modifichiamo le proprietà del blocco *Stream to Vector*.

- Num Items: vectorLength

Modifichiamo le proprietà del blocco *Vector to Stream*.

- Num Items: vectorLength

Modifichiamo le proprietà del blocco *Virtual Sink*.

- Stream Id: sinusoid

Modifichiamo le proprietà del blocco *Virtual Source*.

- Stream Id: sinusoid

Modifichiamo le proprietà del blocco *QT GUI Time Sink.*

- Autoscale: Yes
- Number of Inputs: 2

Colleghiamo i blocchi secondo il seguente diagramma di flusso, come in Figura 4.2.1.

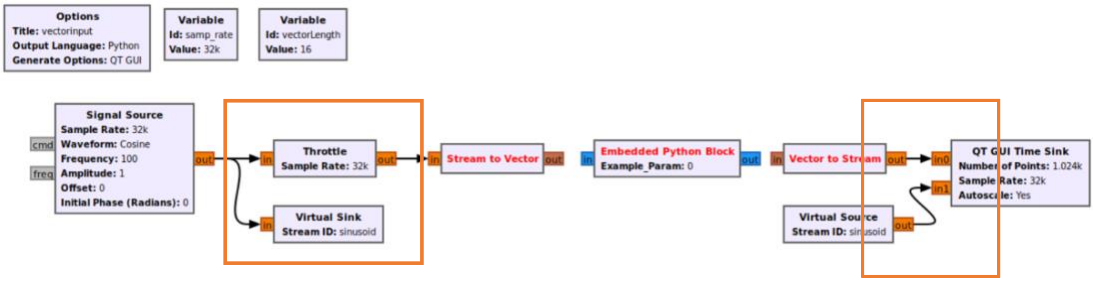


Figura 4.2.1 Diagramma di flusso con blocchi elencati.

4.2.1 Accettazione di input e output vettoriali

Il blocco Python incorporato deve essere modificato nella maniera seguente.

1. accettare input vettoriali
2. produrre output vettoriali
3. modificare i tipi di dati in float

Per quanto riguarda l'esercitazione sui flussi e sui vettori, da un punto di vista di livello superiore, i flussi sono in genere solo un caso speciale di vettori, quelli che hanno un solo elemento di dati in parallelo. Questo è il motivo per cui possiamo semplicemente modificare i parametri in_sig e out_sig per utilizzare vettori anziché flussi.
Poiché il caso dei flussi è così comune essi hanno una sintassi semplice, mentre la sintassi per i vettori è solo un po' più complessa. In particolare, quando si utilizzano i vettori, è necessario specificare la lunghezza del vettore assieme al tipo di dati degli elementi nei vettori. Lo facciamo usando una tupla in Python.
Facciamo doppio clic sul blocco per modificare il codice sorgente. Cambiamo i seguenti parametri, come in Figura 4.2.2.

def _init_(self, vectorSize=16):
name='Max Hold Block',
in_sig=[(np.float32,vectorSize)],
out_sig=[(np.float32,vectorSize)]
Rimuoviamo la riga self.example_param = example_param.
Rimuoviamo la moltiplicazione per self.example_param:

```
import numpy as np
from gnuradio import gr


class blk(gr.sync_block):  # other base classes are basic_block, decim_block, interp_block
    """Embedded Python Block example - a simple multiply const"""

    def __init__(self, vectorSize=16)  # only default arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='Max Hold Block',   # will show up in GRC
            in_sig=[(np.float32,vectorSize)],
            out_sig=[(np.float32,vectorSize)]
        )
        # if an attribute with the same name as a parameter is found,
        # a callback is registered (properties work, too).


    def work(self, input_items, output_items):
        """example: multiply with constant"""
        output_items[0][:] = input_items[0]
        return len(output_items[0])
```

Figura 4.2.2 Codice Python con le modifiche.

Salviamo il codice e colleghiamo il *Max Hold Block* al resto del diagramma di flusso, come in Figura 4.2.3.

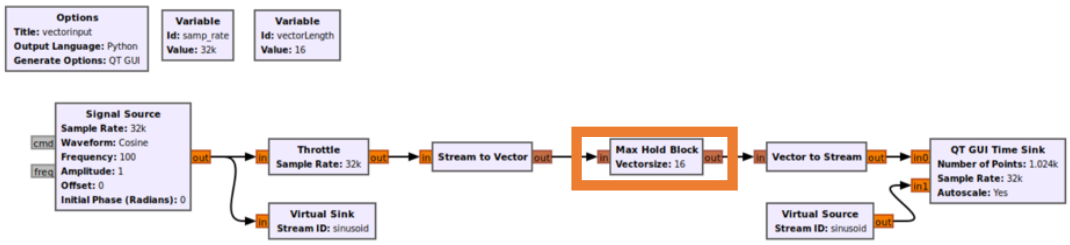


Figura 4.2.3. Diagramma di flusso con la modifica del blocco *Max Hold Block.*

4.2.2 Segnalazione per le mancate corrispondenze di lunghezza del vettore

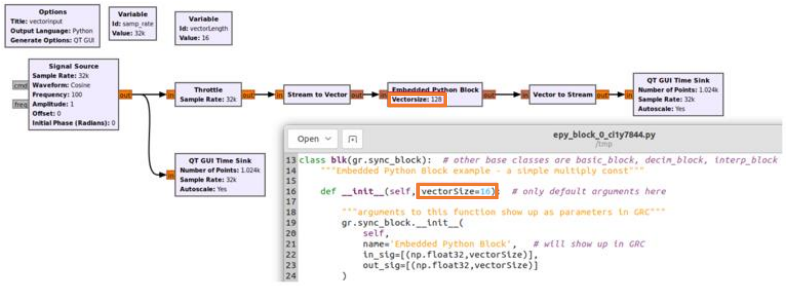
L'Embedded Python Block ha una differenza che altri moduli Out of Tree non hanno. Prima che un diagramma di flusso possa essere eseguito controlliamo GRC per garantire che tutti i tipi di dati collegati e le dimensioni dei vettori corrispondano. Durante questo processo il valore predefinito di vectorSize nella __init()__ function, diventa

def _init_(self, vectorSize=16):

Esso viene utilizzato per definire la dimensione dei vettori per l'input e l'output.

in_sig=[(np.float32,vectorSize)],
out_sig=[(np.float32,vectorSize)]

Verifica della correttezza del diagramma di flusso. In questo caso vectorSize=16. GRC presuppone che le porte di input e output siano vettori con lunghezza 16, anche se viene passato un parametro diverso attraverso le proprietà del blocco. La Figura 4.2.4 seguente mostra come una dimensione vettoriale di 128 nel blocco *Embedded Python Block* viene passata come parametro che GRC non rileva come un errore, ma il diagramma di flusso andrà in crash una volta eseguito.




```
Traceback (most recent call last):
  File "/home/username/vectorinput.py", line 250, in <module>
    main()
  File "/home/username/vectorinput.py", line 226, in main
    tb = top_block_cls()
  File "/home/username/vectorinput.py", line 188, in __init__
    self.connect((self.blocks_stream_to_vector_0, 0), (self.epy_block_0, 0))
  File "/usr/lib/python3/dist-packages/gnuradio/gr/hier_block2.py", line 48, in wrapped
    func(self, src, src_port, dst, dst_port)
  File "/usr/lib/python3/dist-packages/gnuradio/gr/hier_block2.py", line 111, in connect
    self.primitive_connect(*args)
  File "/usr/lib/python3/dist-packages/gnuradio/gr/runtime_swig.py", line 4531, in primitive_connect
    return _runtime_swig.top_block_sptr_primitive_connect(self, *args)
RuntimeError: itemsize mismatch: stream_to_vector0:0 using 64, Embedded Python Block0:0 using 512
```

Figura 4.2.4. Errore dopo esecuzione del diagramma di flusso.

In alternativa, GRC mostrerà un errore se il parametro predefinito non corrisponde alla dimensione del vettore degli altri blocchi. In questo caso, la lunghezza del vettore predefinita nel codice è 128 ma il parametro passato è 16, come in Figura 4.2.5.

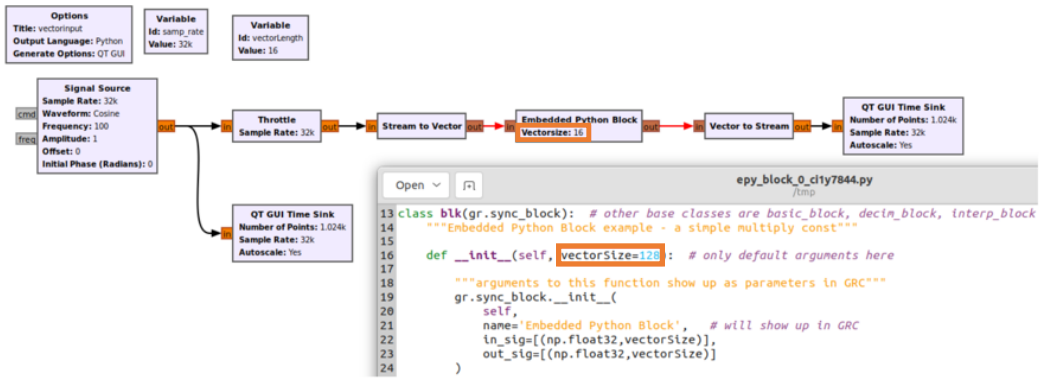


Figura 4.2.5. Non corrispondenza tra parametro e la dimensione del vettore.

4.2.3 Indicizzazione dei flussi

Per un flusso, gli input e gli output possono essere indicizzati utilizzando sia il numero di porta che l'indice del campione.
L'indicizzazione basata sul numero di porta restituisce tutti i campioni di input per una porta specifica. Per esempio:

- input_items[0]

Restituisce tutti i campioni di input sulla porta 0.
La riga seguente restituisce il quarto esempio di input sulla porta 0:

- input_items[0][3]

Invece l'indicizzazione per i flussi è generalizzata a:

- input_items[portIndex][sampleIndex]
- output_items[portIndex][sampleIndex]

Figura 4.2.6 mostra come visualizzare i flussi di indicizzazione.

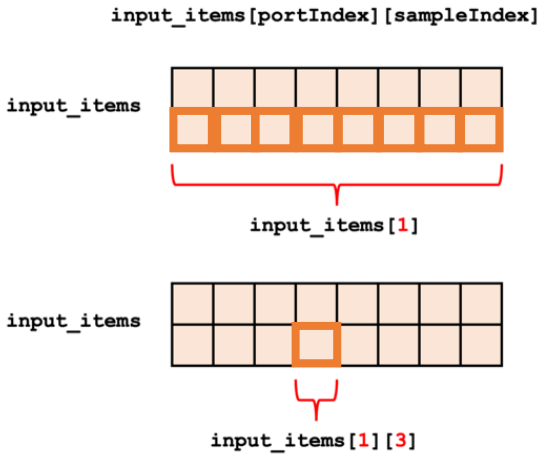


Figura 4.2.6. Flussi di indicizzazione.

4.2.4 Indicizzazione dei vettori

L'input input_items e l'output output_items includono una dimensione aggiuntiva quando si usano i vettori. I vettori aggiungono una dimensione aggiuntiva, rappresentata come vectorIndex. Input_items e output_items diventano matrici tridimensionali.

- input_items[portIndex][vectorIndex][sampleIndex]
- output_items[portIndex][vectorIndex][sampleIndex]

L'indicizzazione basata su portIndex restituisce un array bidimensionale di tutti i vettori e campioni. Esempio:

- input_items[portIndex]
- output_items[portIndex]

L'indicizzazione basata su portIndex e vectorIndex restituisce un array unidimensionale di campioni. Esempio:

- input_items[portIndex][vectorIndex]
- output_items[portIndex][vectorIndex]

L'indicizzazione basata su portIndex, vectorIndex e sampleIndex restituisce un singolo campione. Esempio:

- input_items[portIndex][vectorIndex][sampleIndex]
- output_items[portIndex][vectorIndex][sampleIndex]

Di seguito è riportato un esempio visivo di indicizzazione vettoriale, come in Figura 4.2.7.

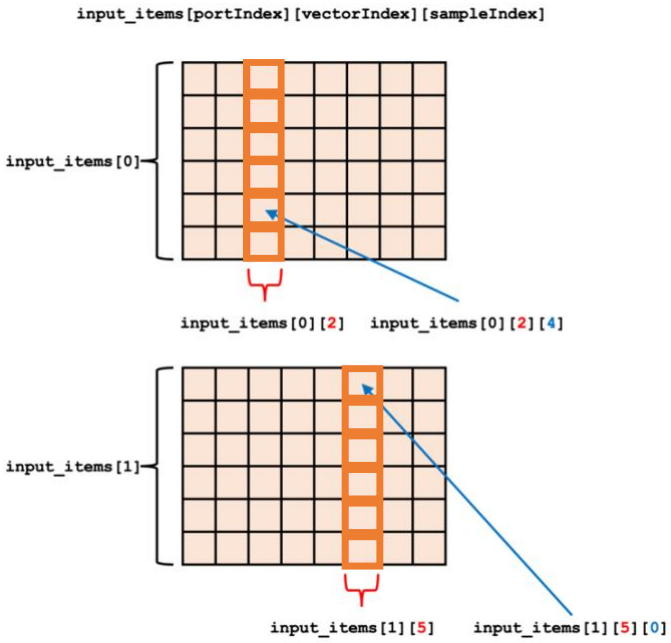


Figura 4.2.7. Indicizzazione vettoriale.

4.2.5 Creazione della funzione Max Hold

La funzione *work()* viene modificata per includere la funzione Max Hold che fa vedere il massimo livello ogni certo numero di campioni.

Aggiungiamo un loop su tutti i vettori in input_items[0].

- *for vectorIndex in range (len(input_items[0])):*

Calcoliamo il valore massimo del vettore.

- *maxValue = np.max(input_items[0][vectorIndex])*

Loop su ciascuno dei campioni di input.

- *for sampleIndex in range(len(input_items[0][vectorIndex])):*

Assegniamo ogni campione di output maxValue.

- *output_items[0][vectorIndex][sampleIndex] = maxValue*

Il codice dovrebbe essere simile alla seguente editor, come in Figura 4.2.8.

```python
class blk(gr.sync_block):  # other base classes are basic_block, decim_block, interp_block
    """Embedded Python Block example - a simple multiply const"""

    def __init__(self, vectorSize=16):  # only default arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='Max Hold Block',   # will show up in GRC
            in_sig=[(np.float32,vectorSize)],
            out_sig=[(np.float32,vectorSize)]
        )
        # if an attribute with the same name as a parameter is found,
        # a callback is registered (properties work, too).


    def work(self, input_items, output_items):
        """example: multiply with constant"""

        for vectorIndex in range(len(input_items[0])):
            maxValue = np.max(input_items[0][vectorIndex])
            for sampleIndex in range(len(input_items[0][vectorIndex])):
                output_items[0][vectorIndex][sampleIndex] = maxValue

        return len(output_items[0])
```

Figura 4.2.8. Codice Python con le modifiche.

Salviamo il codice. Eseguiamo il diagramma di flusso. L'output mostrerà una sinusoide rossa e una sinusoide blu con un max-hold applicato ogni 16 campioni, come in Figura 4.2.9.

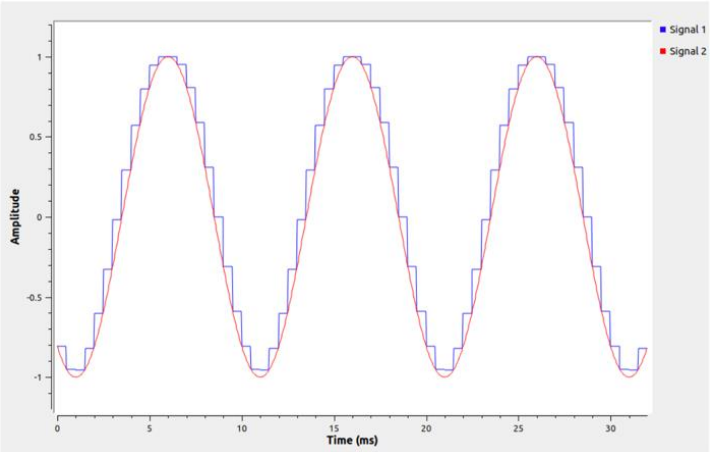


Figura 4.2.9. Output della sinusoide con Max-Hold.

4.2.6 Più porte vettoriali

Il Max Hold Block è stato modificato per aggiungere una seconda porta di input e output del vettore. Aggiungiamo i seguenti blocchi all'area di lavoro: *Noise Source, Stream to Vector, Vector to Stream, Virtual Sink, Virtual Source, QT GUI Time Sink.*

Modifichiamo le seguenti proprietà dei blocchi.

- *Noise Source* Output Type: float
- *Stream to Vector* Num Items: vectorLength

- *Vector to Stream* Num Items: vectorLength
- *Virtual Sink* Stream Id: noise
- *Virtual Source* Stream Id: noise
- *QT GUI Time Sink* Autoscale: Yes
  Number of Inputs: 2

Colleghiamo tutti blocchi e vediamo il diagramma di flusso, come in Figura 4.2.10.

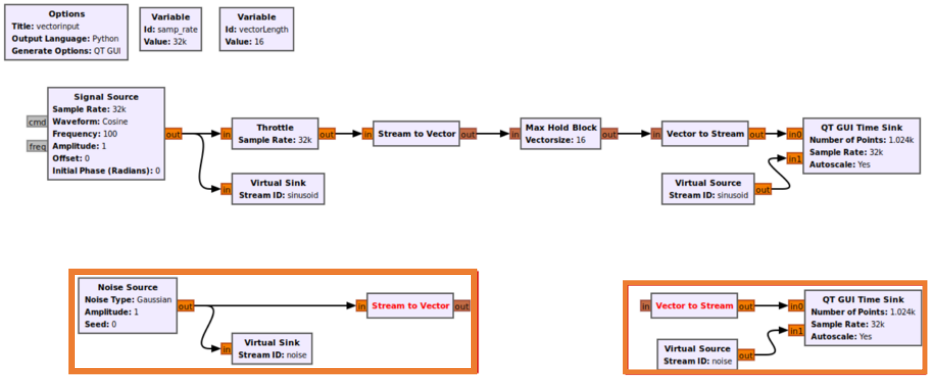


Figura 4.2.10. Diagramma di flusso con blocchi elencati e modificcati.

Modifichiamo il codice per il *Max Hold Block*. Aggiungiamo un secondo input e output vettoriale, come in Figura 4.2.11.

in_sig=[(np.float32,vectorSize), (np.float32,vectorSize)],
out_sig=[(np.float32,vectorSize), (np.float32,vectorSize)]

```
13 class blk(gr.sync_block):  # other base classes are basic_block, decim_block, interp_block
14     """Embedded Python Block example - a simple multiply const"""
15
16     def __init__(self, vectorSize=16):  # only default arguments here
17         """arguments to this function show up as parameters in GRC"""
18         gr.sync_block.__init__(
19             self,
20             name='Max Hold Block'   # will show up in GRC
21             in_sig=[(np.float32,vectorSize),(np.float32,vectorSize)],
22             out_sig=[(np.float32,vectorSize),(np.float32,vectorSize)]
23         )
24         # if an attribute with the same name as a parameter is found,
25         # a callback is registered (properties work, too).
26
27
28     def work(self, input_items, output_items):
29         """example: multiply with constant"""
30
31         for vectorIndex in range(len(input_items[0])):
32             maxValue = np.max(input_items[0][vectorIndex])
33             for sampleIndex in range(len(input_items[0][vectorIndex])):
34                 output_items[0][vectorIndex][sampleIndex] = maxValue
35
36         return len(output_items[0])
```

Figura 4.2.11. Modifica codice Python del *Max Hold Block.*

La funzione work() viene modificata per eseguire la funzione max hold su entrambe le porte di ingresso.

Includiamo un loop esterno su tutte le porte di input.

- *for portIndex in range(len(input_itemo)):*

Cambiamo tutta l'indicizzazione [0] in [portIndex].

- for portIndex in range (len(input_items)):
  for vectorIndex in range(len(input_items[portIndex])):
  maxValue = np.max(input_items[portIndex][vectorIndex])
  for sampleIndex in range(len(input_items[portIndex][vectorIndex])):
  output_items[portIndex][vectorIndex][sampleIndex] = maxValue

Il codice ora dovrebbe essere simile alla seguente editor, come in Figura 4.2.12.

```
class blk(gr.sync_block):  # other base classes are basic_block, decim_block, interp_block
    """Embedded Python Block example - a simple multiply const"""

    def __init__(self, vectorSize=16):  # only default arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='Max Hold Block',   # will show up in GRC
            in_sig=[(np.float32,vectorSize)],
            out_sig=[(np.float32,vectorSize)]
        )
        # if an attribute with the same name as a parameter is found,
        # a callback is registered (properties work, too).


    def work(self, input_items, output_items):
        """example: multiply with constant"""

        for portIndex in range(len(input_items)):
            for vectorIndex in range(len(input_items[portIndex])):
                maxValue = np.max(input_items[portIndex][vectorIndex])
                for sampleIndex in range(len(input_items[portIndex][vectorIndex])):
                    output_items[portIndex][vectorIndex][sampleIndex] = maxValue

        return len(output_items[0])
```

Figura 4.2.12. Modifica codice Python del *Max Hold Block.*

Salviamo il codice e colleghiamo i blocchi, come in Figura 4.2.13.

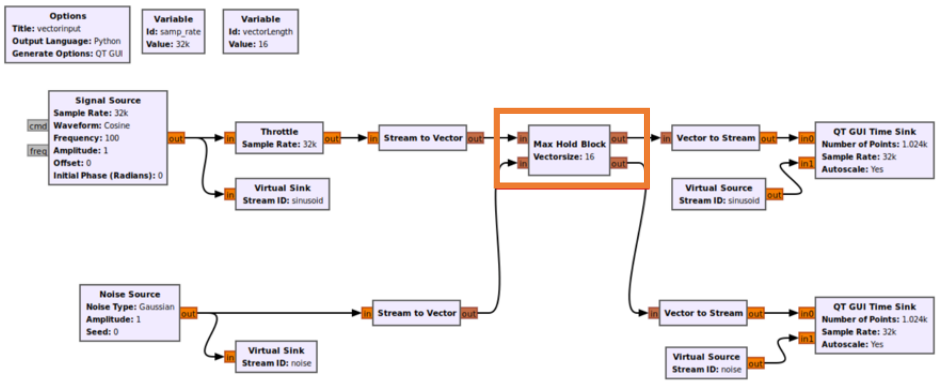


Figura 4.2.13. Diagramma di flusso con le modifiche e collegamenti eseguiti.

Eseguendo il diagramma di flusso verranno ora generate due uscite di massima tenuta, una per la sorgente di rumore e una per la sinusoide, come in Figura 4.2.14.

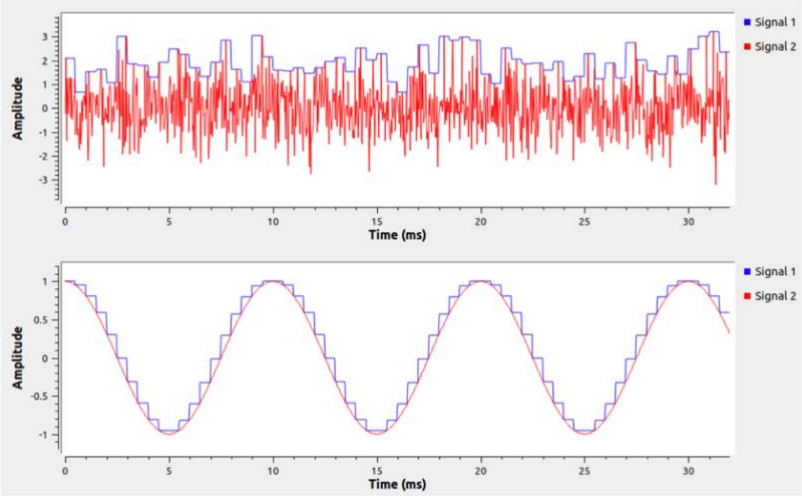


Figura 4.2.14. Output diagramma di flusso.

4.3 Passaggio di messaggi Python Block

Vediamo come leggere e scrivere messaggi usando *Embedded Python Block*. I messaggi sono un modo asincrono per inviare informazioni tra i blocchi. I messaggi sono utili per trasmettere i dati di controllo, mantenere uno stato coerente tra i blocchi e fornire alcune forme di feedback non dati ai blocchi in un diagramma di flusso [13].

I messaggi hanno diverse proprietà:

- Non esiste alcuna garanzia basata sull'orologio campione quando i messaggi arriveranno.
- I messaggi non sono associati a un campione specifico come un tag.
- Le porte di input e output dei messaggi non devono essere collegate in GRC.
- Le porte dei messaggi utilizzano il tipo polimorfico (PMT).

Le porte dei messaggi sono indicate da un colore grigio e le loro connessioni sono contraddistinte da linee tratteggiate, come in Figure 4.3.1.

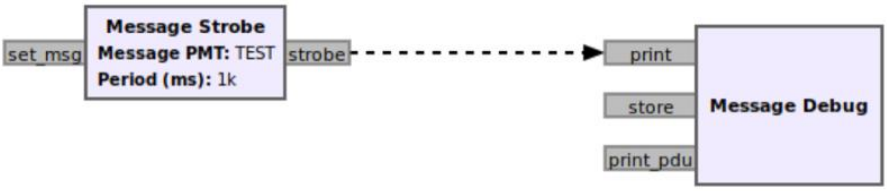


Figura 4.3.1. Porte dei messaggi.

Vediamo un diagramma di flusso che mostra come:

- Aggiungere porte di invio e ricezione di messaggi ai blocchi Python.
- Mandare messaggi.
- Ricevere e gestire i messaggi.
- Adatta il comportamento del blocco nella funzione work() in base ai messaggi ricevuti.

Vengono creati due blocchi Python incorporati e personalizzati per:

- Selezionare uno dei due segnali di ingresso in base a un messaggio di ricezione.
- Contare il numero di campioni e inviare un messaggio al blocco multiplexer per cambiare gli ingressi.

Aggiungiamo i seguenti blocchi al diagramma di flusso e colleghiamo *Noise Source, Signal Source, Python Block, Throttle, QT GUI Time Sink.* Avremo il diagramma di flusso seguente, come in Figura 4.3.2.

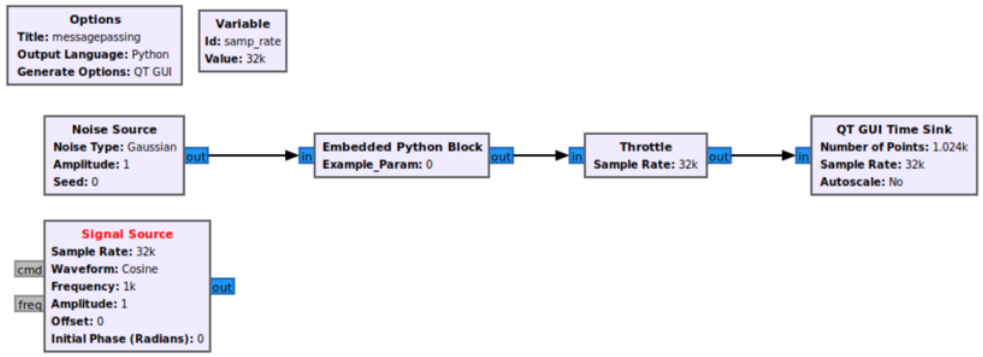


Figura 4.3.2. Diagramma di flusso con blocchi elencati.

4.3.1 Multiplexer: definiamo il blocco

Facciamo doppio clic su *Embedded Python Block* e apriamo il codice sorgente nell'editor.

Eseguiamo le seguenti modifiche:

- example_param non è necessario, quindi rimuoviamo la variabile example_param dalla __*init()*__function.

- def __init__(self): # solo argomenti predefiniti qui

Eliminiamo anche la riga:

- self.example_param = example_param

Cambiamo il nome del blocco in Multiplexer:

- name='Multiplexer',

Aggiungiamo un secondo input al blocco:

- in_sig=[np.complex64, np.complex64],

Eliminiamo la moltiplicazione per example_param:

- output_items[0][:] = input_items[0]

Si otterrà il seguente editor, come in Figura 4.3.3.

```
"""
Embedded Python Blocks:

Each time this file is saved, GRC will instantiate the first class it finds
to get ports and parameters of your block. The arguments to __init__  will
be the parameters. All of them are required to have default values!
"""

import numpy as np
from gnuradio import gr


class blk(gr.sync_block):  # other base classes are basic_block, decim_block, interp_block
    """Embedded Python Block example - a simple multiply const"""

    def __init__(self):  # only default arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='Multiplexer',   # will show up in GRC
            in_sig=[np.complex64, np.complex64],
            out_sig=[np.complex64]
        )
        # if an attribute with the same name as a parameter is found,
        # a callback is registered (properties work, too).


    def work(self, input_items, output_items):
        """example: multiply with constant"""
        output_items[0][:] = input_items[0]
        return len(output_items[0])
```

Figura 4.3.3. Codice Python con le modifiche.

Ricordiamo che Python richiede l'indentazione corretta. Per l'impostazione predefinita, *l'Embedded Python Block* utilizza il rientro in multipli di quattro spazi. La combinazione di tabulazioni e spazi può presentare un errore di sintassi, come in Figura 4.3.4.

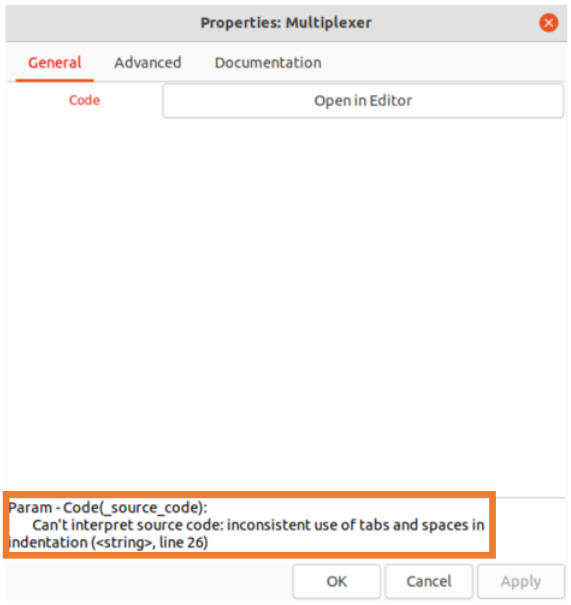


Figura 4.3.4. Errore di sintassi.

Salviamo il codice e torniamo a GRC. Il nome del blocco viene modificato e il blocco ha due ingressi. Colleghiamo la sorgente di rumore e la sorgente di segnale ai due ingressi, come in Figura 4.3.5.

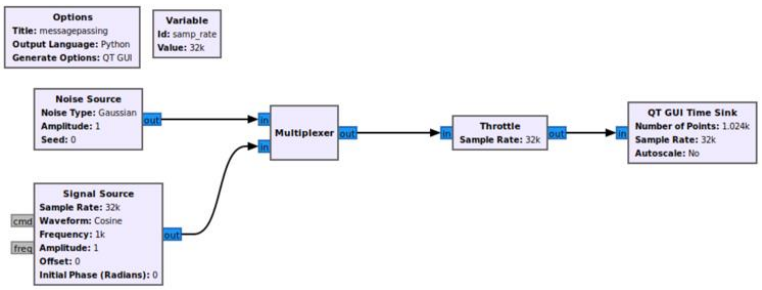


Figura 4.3.5. Diagramma di flusso con blocco *Multiplexer.*

4.3.2 Multiplexer: definizione della porta di input del messaggio

Ritorniamo all'editor di codice. È necessario aggiungere una porta per i messaggi di input.

Creiamo una variabile per memorizzare il nome della porta del messaggio:

- self.selectPortName = 'selectPort'

Aggiungiamo una riga per creare o registrare la porta di input del messaggio:

- self.message_port_register_in(pmt.intern(self.selectPortName))

Aggiungiamo una linea per connettere la porta di input con un gestore di messaggi:

- self.set_msg_handler(pmt.intern(self.selectPortName), self.handle_msg)

Otterremo il seguente editor, come in Figura 4.3.6.

```
class blk(gr.sync_block):  # other base classes are basic_block, decim_block, interp_block
    """Embedded Python Block example - a simple multiply const"""

    def __init__(self):  # only default arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='Multiplexer',   # will show up in GRC
            in_sig=[np.complex64, np.complex64],
            out_sig=[np.complex64]
        )
        # if an attribute with the same name as a parameter is found,
        # a callback is registered (properties work, too).
        self.selectPortName = 'selectPort'
        self.message_port_register_in(pmt.intern(self.selectPortName))
        self.set_msg_handler(pmt.intern(self.selectPortName), self.handle_msg)
        self.selector = True

    def handle_msg(self, msg):
        self.selector = pmt.to_bool(msg)
```

Figura 4.3.6. Codice Python con le modifiche.

Salvataggio del codice. Si noti che gli errori di sintassi sono elencati nelle proprietà *dell'Embedded Python Block*, come in Figura 4.3.7.

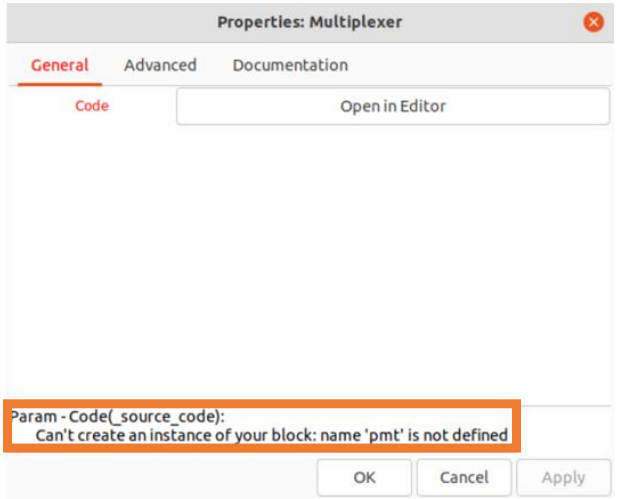


Figura 4.3.7. Errori di sintassi.

Questo errore indica che la libreria pmt deve essere importata. Ritorniamo all'editor di codice e aggiungiamo l'istruzione corretta, come in Figura 4.3.8.

```
"""
Embedded Python Blocks:

Each time this file is saved, GRC will instantiate the first class it finds
to get ports and parameters of your block. The arguments to __init__  will
be the parameters. All of them are required to have default values!
"""

import numpy as np
from gnuradio import gr
import pmt
```

Figura 4.3.8. Aggiungere la libreria pmt.

4.3.3 Multiplexer: creazione di un gestore di messaggi

Un gestore di messaggi è la funzione che interviene quando viene ricevuto un messaggio. La funzione del gestore dei messaggi deve essere definita. Questo gestore di messaggi passa tra le due porte di input in base al messaggio ricevuto. Il messaggio ricevuto è un valore booleano vero o falso.

Definiamo una nuova variabile sotto __init()__ che è il selettore di input:

- self.selector = True

Definiamo la funzione handle_msg():

- def handle_msg(self, msg):
  self.selector = pmt.to_bool(msg)

La funzione pmt.to_bool() accetta il messaggio PMT e quindi converte il tipo di dati precedenti nel tipo di dati booleano di Python. I PMT vengono utilizzati nel passaggio di messaggi a tipi di dati astratti. Ad esempio, i messaggi possono essere utilizzati per inviare e ricevere stringhe, float, numeri interi e liste. Dopo le modifiche otterremo il seguente editor, come in Figura 4.3.9.

```
class blk(gr.sync_block):  # other base classes are basic_block, decim_block, interp_block
    """Embedded Python Block example - a simple multiply const"""

    def __init__(self):  # only default arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='Multiplexer',   # will show up in GRC
            in_sig=[np.complex64, np.complex64],
            out_sig=[np.complex64]
        )
        # if an attribute with the same name as a parameter is found,
        # a callback is registered (properties work, too).
        self.selectPortName = 'selectPort'
        self.message_port_register_in(pmt.intern(self.selectPortName))
        self.set_msg_handler(pmt.intern(self.selectPortName), self.handle_msg)
        self.selector = True

    def handle_msg(self, msg):
        self.selector = pmt.to_bool(msg)

```

Figura 4.3.9. Codice Python con le modifiche.

4.3.4 Multiplexer: utilizzo di un messaggio in work()

L'interfaccia esterna del multiplexer è completa. La funzione work() del blocco viene modificata per aggiungere l'operazione di multiplexing, come in Figura 4.3.10.

Aggiungiamo il seguente codice alla funzione work():

- if (self.selector):
  output_items[0][:] = input_items[0]
  else:
  output_items[0][:] = input_items[1]

Il blocco multiplexer seleziona la porta 0 se self.selector = True e la porta 1 se self.selector = False. Il valore predefinito di self.selector è definito nella funzione __init()__.

```
13 class blk(gr.sync_block):  # other base classes are basic_block, decim_block, interp_block
14     """Embedded Python Block example - a simple multiply const"""
15
16     def __init__(self):  # only default arguments here
17         """arguments to this function show up as parameters in GRC"""
18         gr.sync_block.__init__(
19             self,
20             name='Multiplexer',   # will show up in GRC
21             in_sig=[np.complex64, np.complex64],
22             out_sig=[np.complex64]
23         )
24         # if an attribute with the same name as a parameter is found,
25         # a callback is registered (properties work, too).
26         self.selectPortName = 'selectPort'
27         self.message_port_register_in(pmt.intern(self.selectPortName))
28         self.set_msg_handler(pmt.intern(self.selectPortName), self.handle_msg)
29         self.selector = True
30
31     def handle_msg(self, msg):
32         self.selector = pmt.to_bool(msg)
33
34     def work(self, input_items, output_items):
35         """example: multiply with constant"""
36         if (self.selector):
37             output_items[0][:] = input_items[0]
38         else:
39             output_items[0][:] = input_items[1]
40         return len(output_items[0])
```

Figura 4.3.10. Codice Python con le modifiche.

Salviamo il codice e torniamo a GRC. Il blocco Multiplexer ha una porta di messaggio selectPort, come in Figura 4.3.11.

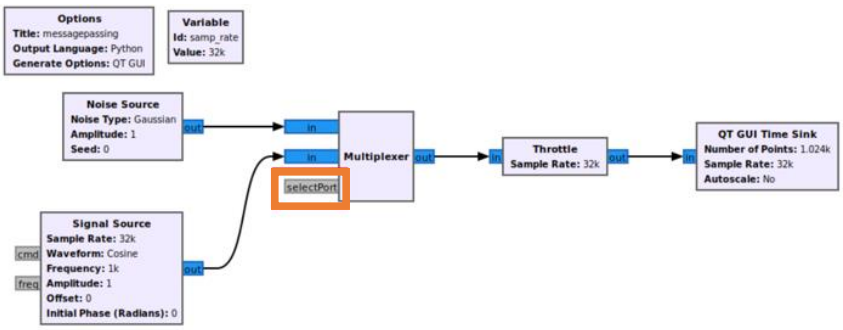


Figura 4.3.11. Diagramma di flusso con blocco *Multiplexer* modificato.

Eseguiamo il diagramma di flusso per assicurarci che tutto sia corretto prima di procedere. Come accennato nell'introduzione, non è necessario che una porta di messaggio sia collegata per eseguire un diagramma di flusso.
Il valore predefinito di self.selector è True. La funzione work() del multiplexer seleziona la porta 0 e la invia all'output.
La *QT GUI Time Sink* visualizza il rumore, come in Figura 4.3.12.

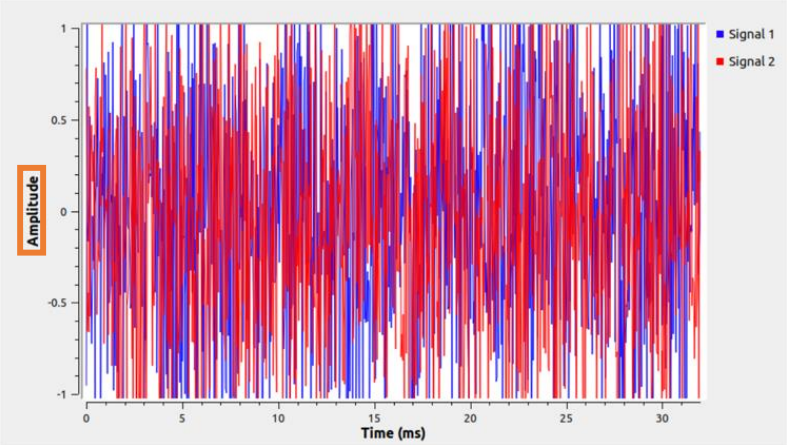


Figura 4.3.12. Output del diagramma di flusso.

4.3.5 Selector Control: definizione del blocco

Un altro *Embedded Python Block* viene utilizzato per contare il numero di campioni che ha ricevuto e quindi inviare un messaggio di controllo al blocco multiplexer per attivare o disattivare il selettore. Aggiungiamo un nuovo blocco Python al diagramma di flusso, *tra Multiplexer e Throttle*. Trascina e rilascia un NUOVO Blocco Python dalla libreria dei blocchi, come in Figura 4.3.13.
IMPORTANTE: NON COPIARE E INCOLLARE I BLOCCHI ESISTENTI.

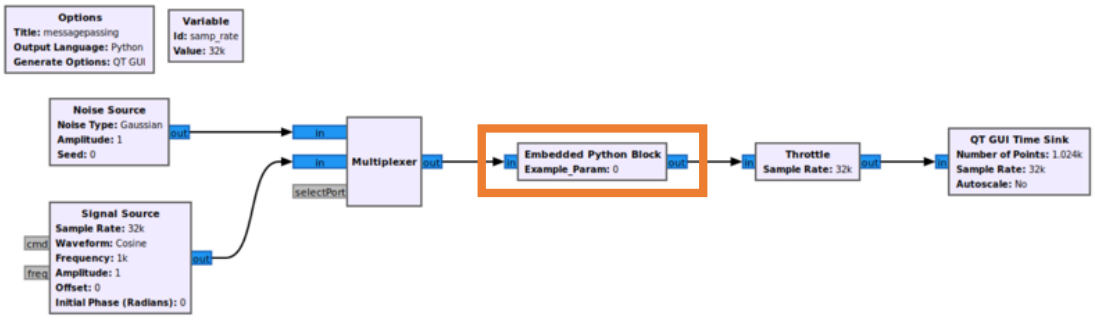


Figura 4.3.13. Diagramma di flusso con *Embedded Python Block* agiunto.

Modifichiamo il codice del *Embedded Python Block*.

Modificare il parametro example_param nella funzione __init()__:

- def __init__(self, Num_Samples_To_Count=128):

Cambiamo il nome del blocco:

- name='Selector Control',

Memorizziamo Num_Samples_To_Count come variabile privata:

- self.Num_Samples_To_Count = Num_Samples_To_Count

Rimuoviamo la moltiplicazione nella funzione work():

- example_param

Otterremo seguente editor, come in Figura 3.3.14.

```
import numpy as np
from gnuradio import gr

class blk(gr.sync_block):  # other base classes are basic_block, decim_block, interp_block
    """Embedded Python Block example - a simple multiply const"""

    def __init__(self, Num_Samples_To_Count=128):  # only default arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='Selector Control',   # will show up in GRC
            in_sig=[np.complex64],
            out_sig=[np.complex64]
        )
        # if an attribute with the same name as a parameter is found,
        # a callback is registered (properties work, too).
        self.Num_Samples_To_Count = Num_Samples_To_Count

    def work(self, input_items, output_items):
        """example: multiply with constant"""
        output_items[0][:] = input_items[0]
        return len(output_items[0])

```

Figura 4.3.14. Codice Python con le modifiche.

4.3.6 Selector Control: definizione della porta di output dei messaggi

Importiamo la libreria pmt, creiamo una variabile self.portName nella funzione __init()__ che è il nome della porta di output come stringa messageOutput:

- self.portName = 'messageOutput'

Una porta di messaggio viene creata, o registrata, aggiungendo la riga nella funzione __init()__:

- self.message_port_register_out(pmt.intern(self.portName))

Otteniamo il seguente editor, come in Figura 4.3.15.

```
import numpy as np
from gnuradio import gr
import pmt

class blk(gr.sync_block):  # other base classes are basic_block, decim_block, interp_block
    """Embedded Python Block example - a simple multiply const"""

    def __init__(self, Num_Samples_To_Count=128):  # only default arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='Selector Control',   # will show up in GRC
            in_sig=[np.complex64],
            out_sig=[np.complex64]
        )
        # if an attribute with the same name as a parameter is found,
        # a callback is registered (properties work, too).
        self.Num_Samples_To_Count = Num_Samples_To_Count
        self.portName = 'messageOutput'
        self.message_port_register_out(pmt.intern(self.portName))


    def work(self, input_items, output_items):
        """example: multiply with constant"""
        output_items[0][:] = input_items[0]
        return len(output_items[0])
```

Figura 4.3.15. Codice Python con le modifiche.

Salviamo il codice e torniamo a GRC. Il blocco *Selector Control* ha una porta di output dei messaggi, come in Figura 4.3.16.

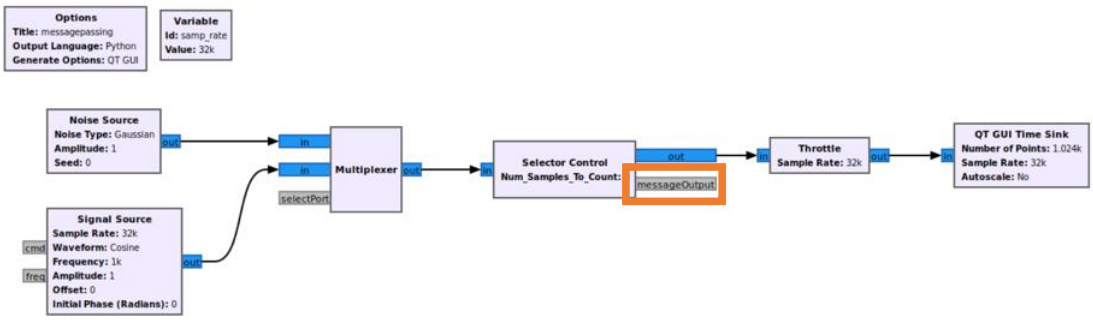


Figura 4.3.16. Diagramma di flusso con blocco *Selector Control* modificata.

4.3.7 Selector Control: invio di un messaggio in work()

Non è necessario definire un gestore di messaggi per una porta di output. Tuttavia, la funzione work() deve essere modificata per creare la logica per l'invio dei messaggi.

Creiamo due variabili in __init()__:

- self.state = True
- self.counter = 0

Aggiungiamo la riga per aumentare il numero di campioni contati per ogni chiamata a work():

- self.counter = self.counter + len(output_items[0])

Otterremo il seguente editor, come in Figura 4.3.17.

```
class blk(gr.sync_block):  # other base classes are basic_block, decim_block, interp_block
    """Embedded Python Block example - a simple multiply const"""

    def __init__(self, Num_Samples_To_Count=128):  # only default arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='Selector Control',   # will show up in GRC
            in_sig=[np.complex64],
            out_sig=[np.complex64]
        )
        # if an attribute with the same name as a parameter is found,
        # a callback is registered (properties work, too).
        self.Num_Samples_To_Count = Num_Samples_To_Count
        self.portName = 'messageOutput'
        self.message_port_register_out(pmt.intern(self.portName))

        self.state = True
        self.counter = 0


    def work(self, input_items, output_items):
        """example: multiply with constant"""

        self.counter = self.counter + len(output_items[0])

        output_items[0][:] = input_items[0]
        return len(output_items[0])
```

Figura 4.3.17. Codice Python con le modifiche.

Aggiungiamo la logica per inviare un messaggio una volta superato il contatore:

- if (self.counter > self.Num_Samples_To_Count):
  PMT_msg = pmt.from_bool(self.state)
  self.message_port_pub(pmt.intern(self.portName), PMT_msg)
  self.state = not(self.state)
  self.counter = 0

La logica traduce il tipo di dati booleano di Python self.state in un PMT utilizzando la chiamata alla funzione pmt.from_bool() e quindi invia, o pubblica, il messaggio sulla porta del messaggio di output. La variabile self.state viene commutata sul valore opposto e il contatore viene azzerato, come in Figura 4.3.18.

```
    def work(self, input_items, output_items):
        """example: multiply with constant"""

        self.counter = self.counter + len(output_items[0])

        if (self.counter > self.Num_Samples_To_Count):
            PMT_msg = pmt.from_bool(self.state)
            self.message_port_pub(pmt.intern(self.portName), PMT_msg)
            self.state = not(self.state)
            self.counter = 0

        output_items[0][:] = input_items[0]
        return len(output_items[0])
```

Figura 4.3.18. Codice Python con le modifiche.

Salviamo il codice e torniamo a GRC. Modifichiamo la voce Num_Samples_To_Count con il valore 32000Hz nelle proprietà del blocco *Selector Control*, come in Figura 4.3.19.

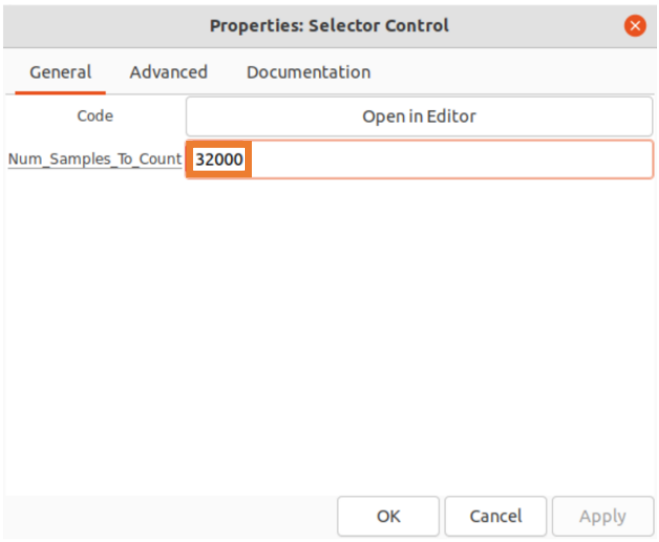


Figura 4.3.19 Modifica delle proprietà del blocco *Selector Control*.

Aggiungiamo il blocco *Message Debug* e connettiamo a esso la porta messageOutput. L'esecuzione del diagramma di flusso mostra che i messaggi vengono inviati a una velocità di uno al secondo, alternando (vero) e (falso). Tuttavia, la porta del messaggio di output dal controllo del selettore non è ancora collegata alla porta del messaggio di input del multiplexer, quindi il time sink della GUI QT mostra solo rumore, come in Figura 4.3.20.

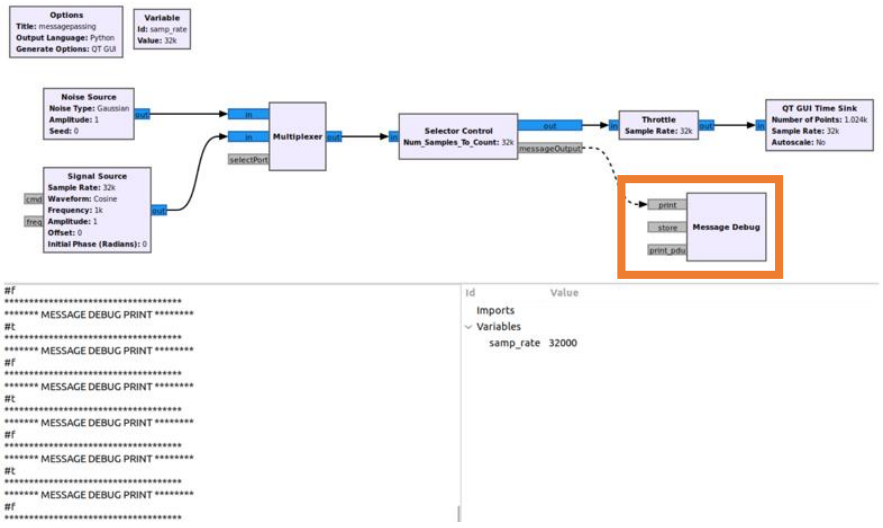


Figura 4.3.20. Diagramma di flusso con blocco *Message Debug*.

Collegare la porta del messaggio di uscita del *Selector Control* alla porta del messaggio di ingresso del *Multiplexer*. Si noti che la linea tratteggiata viaggia dietro i due blocchi e può essere difficile da vedere, come in Figura 4.3.21.

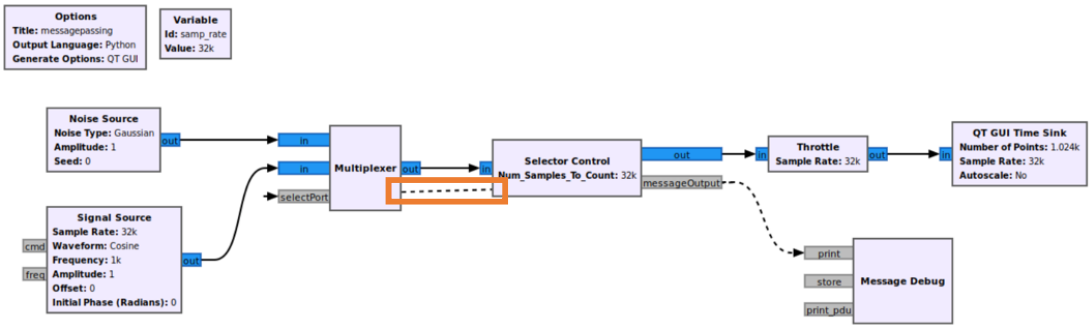


Figura 4.3.21. Collegamento delle porte di messaggi.

I blocchi *Virtual Sinks e Virtual Sources* possono essere utilizzati per ripulire alcune delle connessioni e rendere il diagramma di flusso più facile da comprendere. Facciamo clic sulla linea tratteggiata ed eliminiamola. Trasciniamo e rilasciamo i due blocchi *Virtual Sink e Virtual Source* nell'area di lavoro. Modifichiamo l'ID flusso in *message* sia per il *Virtual Sinks* che per *Virtual Sources*, quindi colleghiamoli nel diagramma di flusso, come in Figura 4.3.22.

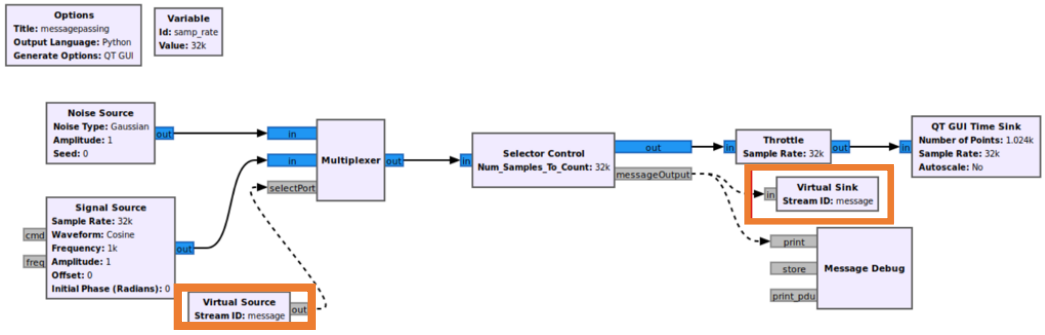


Figura 4.3.22. Diagramma di flusso con blocchi aggiunti e collegati.

Eseguiamo il diagramma di flusso. Il *QT GUI Time Sink* mostra un'uscita alternata tra la sorgente del rumore e la sorgente del segnale, come in Figura 4.3.23.

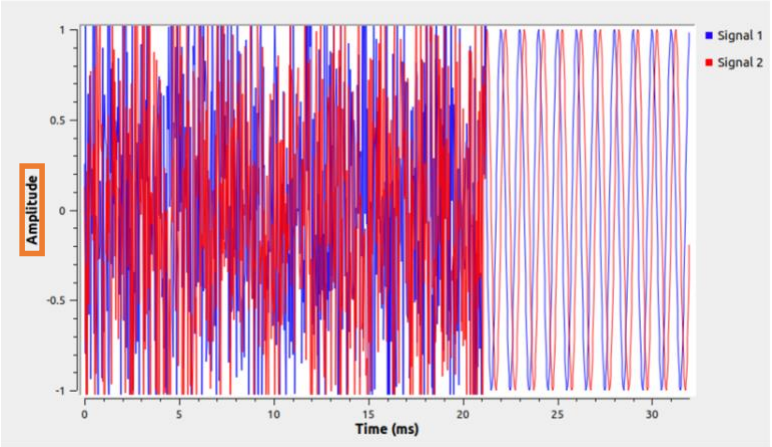


Figura 4.3.23. Output in uscita del diagramma di flusso.

4.4 Tag di Python Block

Mostriamo come creare due blocchi Python incorporati per rilevare quando il segnale di ingresso attraversa la soglia, scrivere un tag, leggere il tag nel blocco separato e aggiornare l'output con il tempo trascorso dall'ultimo rilevamento [14].

4.4.1 Panoramica dei tag

I tag sono un modo per trasmettere informazioni insieme a campioni RF (Radiofrequenza) digitalizzati in modo sincrono nel tempo. I tag sono particolarmente utili quando i blocchi a valle devono sapere su quale campione il ricevitore è stato sintonizzato su una nuova frequenza o per includere una sequenza temporale con campioni specifici.
I messaggi trasmettono informazioni in modo asincrono senza alcuna garanzia temporale basata sull'orologio e i tag sono informazioni associate a campioni RF specifici. I tag viaggiano insieme a campioni RF digitalizzati in flussi di dati e vettori, inclusi Complex Float 32, Float 32, Byte e tutti gli altri formati.

I tag vengono aggiunti utilizzando la riga:

- self.add_item_tag(outputPortNumber, absoluteIndex, key, value)

outputPortNumber - determina a quale flusso di output viene aggiunto il tag.
absoluteIndex - è l'indice di esempio a cui viene aggiunto il tag.
Il diagramma di flusso conta ogni campione e il primo campione prodotto è l'indice campione assoluto 0. La chiave è un tipo PMT contenente il nome della variabile da memorizzare e il valore è un altro tipo PMT che contiene le informazioni da memorizzare, come in Figura 4.4.1.

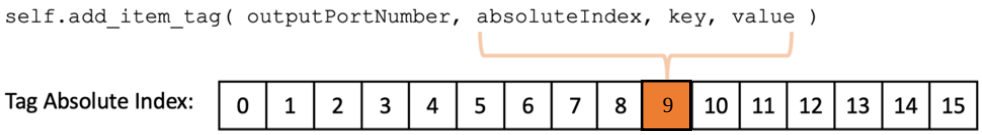


Figura 4.4.1. Tag.

La lettura dei tag può essere eseguita con la funzione:

- tagTuple=self.get_tags_in_window(inputPortNumber,relativeIndexStart,relativeIndexStop)

La lettura dei tag in una finestra vengono letti in base all'indice relativo all'interno del vettore input_items corrente, come in Figura 4.4.2.

Il modo più semplice per ottenere tutti i tag corrispondenti agli esempi input_items correnti è con la chiamata di funzione:

tagTuple=self.get_tags_in_window(inputPortNumber,0,len(input_items[inputPortNumber]))

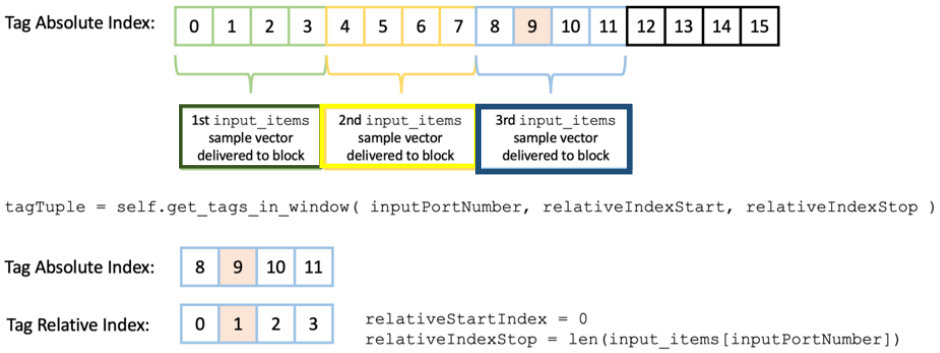


Figura 4.4.2. Lettura dei tag.

4.4.2 Creiamo un segnale di prova

Prendiamo seguenti blocchi: *GLFSR Source, Repeat, Multiply Const, Add Const, Single Pole IIR Filter, Throttle, QT GUI Time Sink.*

Modifichiamo seguenti parametri:

- GLFSR Source — Degree: 32
- Repeat — Interpolation: 128
- Multiply Const — Constant: 0.5
- Add Const — Constant: 0.5
- Single Pole IIR Filter — Alpha: 0.05
- samp_rate Variable — Value: 3200
- QT GUI Time Sink — Number of Points: 2048; Autoscale: Yes

Cambiamo tutti i blocchi in input e output in Float. Colleghiamoli tutti secondo il seguente diagramma di flusso, come in Figura 4.4.3

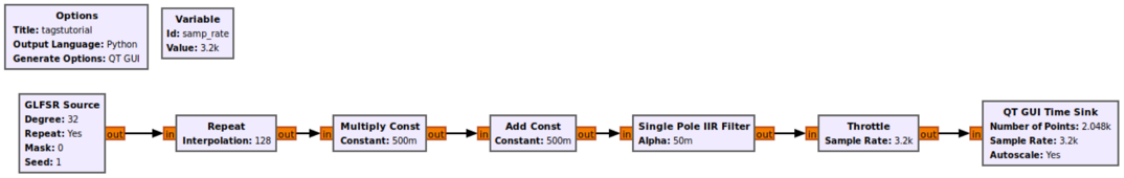


Figura 4.4.3 Diagramma di flusso con blocchi elencati.

Eseguendo il diagramma di flusso viene generata una sequenza pseudo-casuale di 0 e 1 filtrati, come in Figura 4.4.4.

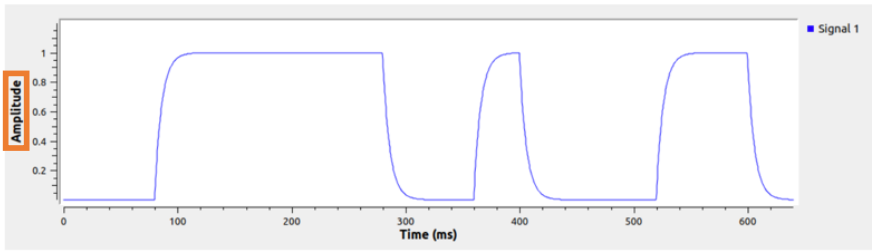


Figura 4.4.4. Output di 0 e 1 casuali.

4.4.3 Threshold Detector: definizione del blocco

Trasciniamo un blocco Python e facciamo doppio clic su di esso per modificare il codice sorgente. Ricordiamo che gli *Embedded Python Blocks* usano il rientro in multipli di quattro spazi.

Cambiamo il nome della variabile example_param e aggiungiamo un nuovo parametro report_period:

- def __init__(self, threshold=1.0, report_period=128):

Aggiorniamo il nome del blocco:

- name='Threshold Detector',

Cambiamo i tipi di input e output in Float:

- in_sig=[np.float32],
- out_sig=[np.float32]

Cambiamo il nome della variabile da self.example_param:

- self.threshold = threshold
- self.report_period = report_period

Rimuoviamo la moltiplicazione per self.example_param:

- output_items[0][:] = input_items[0]

Il codice è simile al seguente editor, come in Figura 4.4.5.

```
import numpy as np
from gnuradio import gr


class blk(gr.sync_block):  # other base classes are basic_block, decim_block, interp_block
    """Embedded Python Block example - a simple multiply const"""

    def __init__(self, threshold=1.0, report_period=128):  # only default arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='Threshold Detector',   # will show up in GRC
            in_sig=[np.float32],
            out_sig=[np.float32]
        )
        # if an attribute with the same name as a parameter is found,
        # a callback is registered (properties work, too).
        self.threshold = threshold
        self.report_period = report_period

    def work(self, input_items, output_items):
        """example: multiply with constant"""
        output_items[0][:] = input_items[0]
        return len(output_items[0])
```

Figura 4.4.5. Codice Python con le modifiche.

Salviamo il codice e torniamo a GRC. Il blocco è simile, come in Figura 4.4.6.

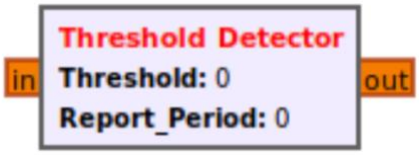


Figura 4.4.6. Blocco *Threshold Detector* in GRC con le modifiche.

Se il blocco non si aggiornasse correttamente, potrebbe esserci un problema con la sintassi di Python. Facciamo doppio clic sul *Embedded Python Block* per visualizzare eventuali errori di sintassi, come in Figura 4.4.7.

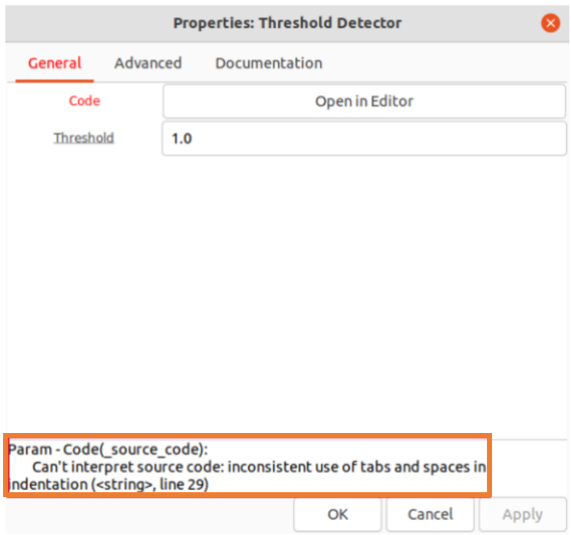


Figura 4.4.7. Errori di sintassi.

Aggiungiamo un blocco *Virtual Sink* e *Virtual Source* al diagramma di flusso.

Modifichiamo le seguenti proprietà del blocco:

- Virtual Sink — Stream ID: signal
- Virtual Source — Stream ID: signal
- QT GUI Time Sink — name: "Threshold Detector"
- Threshold Detector — Threshold: 0.75; Report Period: 128

Colleghiamo i blocchi secondo il diagramma di flusso, come in Figura 4.4.8.

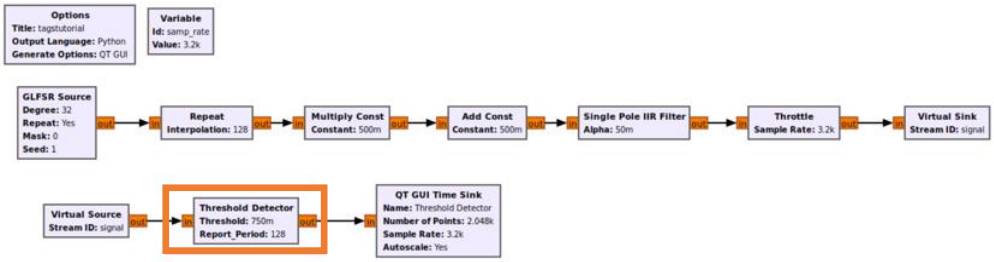


Figura 4.4.8. Diagramma di flusso modificato.

4.4.4 **Threshold Detector** (Rilevatore di soglia)**: scrittura di tag**

Modifichiamo il codice del blocco Threshold Detector.

Importiamo la libreria pmt:

- import pmt

Aggiungiamo due nuove variabili, self.timer e self.readyForTag sotto la funzione __init()__:

- self.timer = 0
- self.readyForTag = True

Otterremo il seguente editor, come in Figura 4.4.9.

```
import numpy as np
from gnuradio import gr
import pmt

class blk(gr.sync_block):  # other base classes are basic_block, decim_block, interp_block
    """Embedded Python Block example - a simple multiply const"""

    def __init__(self, threshold=1.0, report_period=128):  # only default arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='Threshold Detector',   # will show up in GRC
            in_sig=[np.float32],
            out_sig=[np.float32]
        )
        # if an attribute with the same name as a parameter is found,
        # a callback is registered (properties work, too).
        self.threshold = threshold
        self.report_period = report_period

        self.timer = 0
        self.readyForTag = True
```

Figura 4.4.9. Codice Python con le modifiche.

La funzione work() deve essere modificata. Creiamo un ciclo for per scorrere tutti i campioni di input:

- for index in range(len(input_items[0])):

Devono essere scritte tre sezioni di codice. Il primo blocco scrive il livello di ampiezza in un tag denominato una volta raggiunta o superata la soglia. Il tag viene scritto solo se la variabile di stato self.readyForTag è True. Una volta scritto un tag, la variabile di stato self.readyForTag viene impostata su False.

```
# scrivi il tag
if (input_items[0][index] >= self.threshold and self.readyForTag == True):
   # definisce la chiave come 'detect'
   key = pmt.intern('detect')
   # ottiene il valore di rilevamento
   value = pmt.from_float(np.round(float(input_items[0][index]),2))
   # tag indice da scrivere
   writeIndex = self.nitems_written(0) + index
   # aggiungi l'oggetto tag (key, value pair)
   self.add_item_tag(0, writeIndex, key, value )
   # tag è stato scritto, imposta lo stato
   self.readyForTag = False
```

Il prossimo blocco di codice viene utilizzato per eseguire il timer. Il timer aumenta di 1 per ogni campione di input fintanto che self.readyForTag è False:

```
# aumenta il timer di 1
if (self.readyForTag == False):
   self.timer = self.timer + 1
```

Il terzo blocco di codice controlla la variabile di stato self.readyForTag. Una volta che self.timer raggiunge il valore massimo, il timer viene azzerato e la variabile di stato self.readyForTag viene impostata su True:

```
# imposta il flag per scrivere
if (self.timer >= self.report_period):
    # reimposta il timer
    self.timer = 0
    # ripristina lo stato una volta che il timer raggiunge il valore massimo
    self.readyForTag = True
```

Otterremo il seguente editor, come in Figura 4.4.10.

```
32  def work(self, input_items, output_items):
33      """example: multiply with constant"""
34
35      # loop over every input sample
36      for index in range(len(input_items[0])):
37
38          # write the tag
39          if (input_items[0][index] >= self.threshold and self.readyForTag == True):
40              # define the key as 'detect'
41              key = pmt.intern('detect')
42              # get the detection value
43              value = pmt.from_float(np.round(float(input_items[0][index]),2))
44              # tag index to be written
45              writeIndex = self.nitems_written(0) + index
46              # add the tag object (key, value pair)
47              self.add_item_tag(0, writeIndex, key, value )
48              # tag has been written, set state
49              self.readyForTag = False
50
51          # increase  the timer by 1
52          if (self.readyForTag == False):
53              self.timer = self.timer + 1
54
55          # set flag to write
56          if (self.timer >= self.report_period):
57              # reset timer
58              self.timer = 0
59              # reset state once timer hits max value
60              self.readyForTag = True
61
62      output_items[0][:] = input_items[0]
63      return len(output_items[0])
```

Figura 4.4.10. Codice Python con le modifiche.

Eseguendo il diagramma di flusso i tag vengono visualizzati nella *QT GUI Time Sink*, come in Figura 4.4.11.

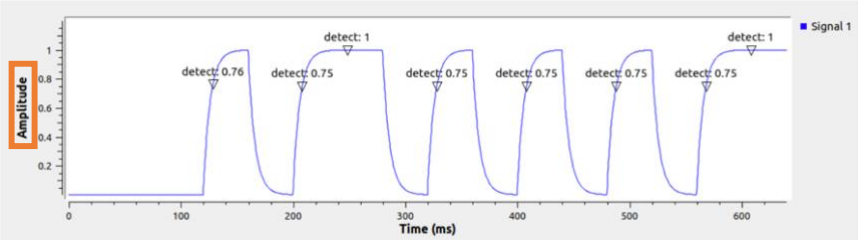


Figura 4.4.11. Output con le modifiche eseguite.

4.4.5 **Detection Counter** (Contatore di rilevamento**): definizione del blocco**

Viene creato un nuovo *Embedded Python Block* per leggere i tag, contare il numero di campioni dall'ultimo tag e produrre tale numero come output. Trasciniamo e rilasciamo un nuovo blocco Python nell'area di lavoro GRC. Non copiare e incollare il blocco Python

esistente, crea solo una seconda copia di *Threshold Detector*. Facciamo doppio clic sul *Embedded Python* e modifichiamo il codice.

Rimuoviamo il parametro self.example_param:

- def __init__(self):

Cambiamo il nome:

- name='Detection Counter',

Rendere mobili le porte di input e output:

- in_sig=[np.float32],
- out_sig=[np.float32]

Rimuoviamo la riga self.example_param = example_param e la moltiplicazione per self.example_param:

- output_items[0][:] = input_items[0]

Otterremo il seguente codice in editor, come in Figura 4.4.12.

```
import numpy as np
from gnuradio import gr


class blk(gr.sync_block):  # other base classes are basic_block, decim_block, interp_block
    """Embedded Python Block example - a simple multiply const"""

    def __init__(self):  # only default arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='Detection Counter',   # will show up in GRC
            in_sig=[np.float32],
            out_sig=[np.float32]
        )
        # if an attribute with the same name as a parameter is found,
        # a callback is registered (properties work, too).


    def work(self, input_items, output_items):
        """example: multiply with constant"""
        output_items[0][:] = input_items[0]
        return len(output_items[0])
```

Figura 4.4.12. Codice Python con le modifiche.

Salviamo il codice. Aggiungiamo un'altra *QT GUI Time Sink* e modifichiamo le proprietà:

- Name: "Detection Counter"
- Number of Points: 2048
- Autoscale: Yes

Colleghiamo il blocco *Detection Counter* dopo il rilevamento della soglia, come in Figura 4.4.13.

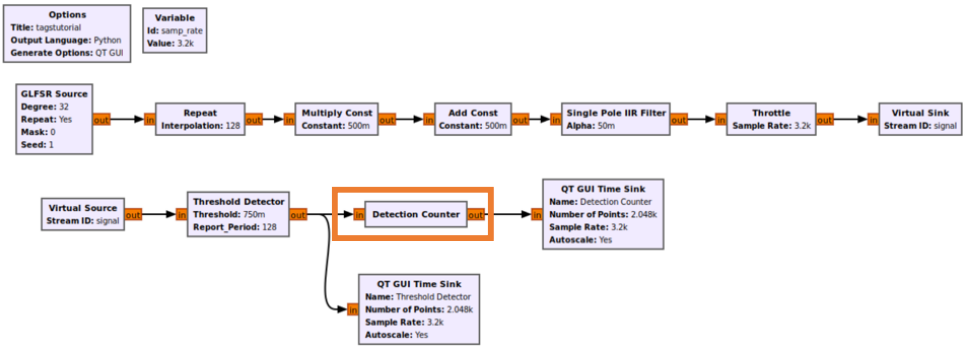


Figura 4.4.13. Diagramma di flusso modificato.

4.4.6 Detection Counter: lettura di tag

Il blocco *Detection Counter* deve essere modificato per leggere i tag.

Importiamo la libreria pmt:

- importa pmt

Aggiungi una nuova variabile sotto __init()__:

- self.samplesSinceDetection = 0

Otterremo il seguente codice in editor, come in Figura 4.4.14.

```
import numpy as np
from gnuradio import gr
import pmt

class blk(gr.sync_block):  # other base classes are basic_block, decim_block, interp_block
    """Embedded Python Block example - a simple multiply const"""

    def __init__(self):  # only default arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='Detection Counter',   # will show up in GRC
            in_sig=[np.float32],
            out_sig=[np.float32]
        )
        # if an attribute with the same name as a parameter is found,
        # a callback is registered (properties work, too).
        self.samplesSinceDetection = 0
```

Figura 4.4.14. Codice Python con le modifiche.

Modifichiamo la funzione work() per leggere i tag:

# recupera tutti i tag associati a input_items[0]
tagTuple = self.get_tags_in_window(0, 0, len(input_items[0]))

Passiamo in rassegna tutti i tag con il rilevamento della chiave, calcoliamo il relativo offset e memorizziamolo in un elenco:

```
# dichiara una lista
relativeOffsetList = []
# scorre tutti i tag 'detect' e memorizza il relativo offset
per tag in tagTuple:for tag in tagTuple:
   if (pmt.to_python(tag.key) == 'detect'):
      relativeOffsetList.append( tag.offset - self.nitems_read(0) )
```

Eseguiamo una sorta di offset in modo che siano ordinati dal più piccolo al più grande:

```
# ordina l'elenco degli offset relative
relativeOffsetList.sort()
```

Passiamo attraverso tutti i campioni di output:

```
# scorre tutti i campioni di output
for index in range(len(output_items[0])):
```

Per ogni campione di output, produci un output che sia il numero di campioni dall'ultimo tag di rilevamento:

```
# l'output campiona dal contatore di rilevamento
output_items[0][index] = self.samplesSinceDetection
```

Se l'indice del campione di output corrente è maggiore o uguale all'indice del tag di rilevamento corrente, rimuovere il valore di offset dall'elenco e reimpostare il contatore di campioni self.samplesSinceDetection. Altrimenti, aumentare il contatore di campioni di 1.

```
# assicurati che l'elenco non sia vuoto e se il campione di input corrente
# è maggiore o uguale al successivo
if (len(relativeOffsetList) > 0 and index >= relativeOffsetList[0]):
   # cancella l'offset
   relativeOffsetList.pop(0)
   # resetta il contatore delle uscite
   self.samplesSinceDetection = 0
else:
   # non è stato rilevato un tag di rilevamento, quindi continua ad aumentare
   # il contatore di output
   self.samplesSinceDetection = self.samplesSinceDetection + 1
```

Rimuoviamo l'assegnazione dell'output:

```
output_items[0][:] = input_items[0]
```

La funzione lavoro è simile a seguente editor, come in Figura 4.4.15.

```python
def work(self, input_items, output_items):
    """example: multiply with constant"""

    # get all tags associated with input_items[0]
    tagTuple = self.get_tags_in_window(0, 0, len(input_items[0]))

    # declare a list
    relativeOffsetList = []

    # loop through all 'detect' tags and store their relative offset
    for tag in tagTuple:
        if (pmt.to_python(tag.key) == 'detect'):
            relativeOffsetList.append( tag.offset - self.nitems_read(0) )

    # sort list of relative offsets
    relativeOffsetList.sort()

    # loop through all output samples
    for index in range(len(output_items[0])):

        # output is now samples since detection counter
        output_items[0][index] = self.samplesSinceDetection

        # make sure the list is not-empty, and if the current input sample
        # is greater than or equal to the next
        if (len(relativeOffsetList) > 0 and index >= relativeOffsetList[0]):
            # clear the offset
            relativeOffsetList.pop(0)
            # reset the output counter
            self.samplesSinceDetection = 0
        else:
            # a detect tag has not been seen, so continue to increase
            # the output counter
            self.samplesSinceDetection = self.samplesSinceDetection + 1

    return len(output_items[0])
```

Figura 4.4.15. Codice Python con le modifiche.

Salviamo il codice. Eseguiamo il diagramma di flusso. L'output è simile alla seguente Figura 4.4.16. Si noti che tutti i tag dall'input a *Detection Counter* vengono automaticamente trasmessi all'output.

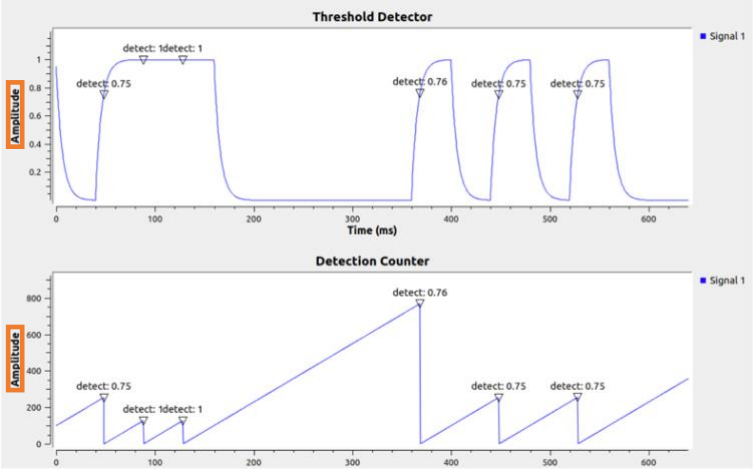


Figura 4.4.16. Output con tag.

4.4.7 Propagazione dei Tag

Per impostazione predefinita, tutti i tag di input vengono propagati a tutti i tag di output. Può essere utile ridurre o rimuovere completamente i tag da determinati flussi. Il blocco *Tag Gate* può essere utilizzato per rimuovere i tag da un flusso di output. Colleghiamo il *Tag Gate* dopo il blocco del contatore di rilevamento, come in Figura 4.4.17.

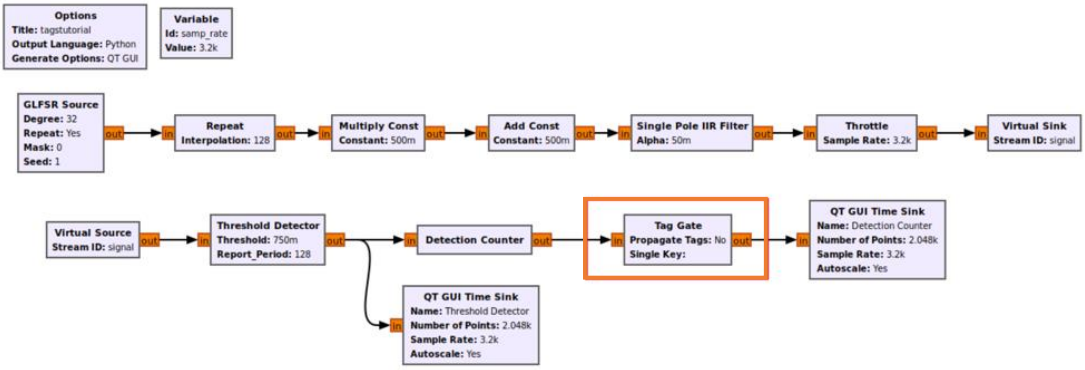


Figura 4.4.17 Diagramma di flusso con blocco *Tag Gate* aggiunto.

Eseguendo il diagramma di flusso i tag vengono rimossi dalla *QT GUI Time Sink*, come in Figura 4.4.18.

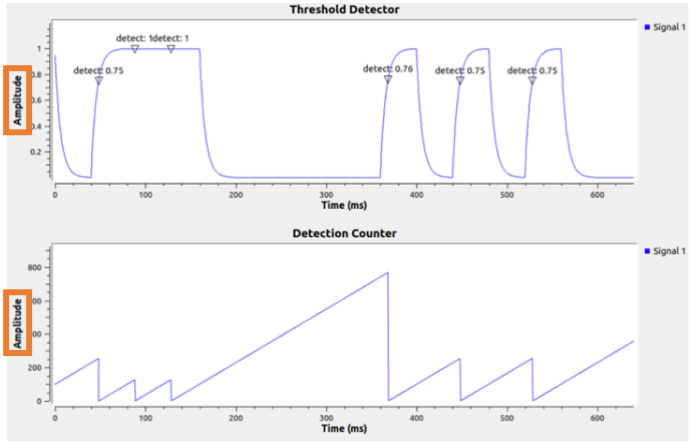


Figura 4.4.18. Output con tag rimossi.

Capitolo 5

Realizzazione di una Software Defined Radio mediante la board ADALM-PLUTO

5.1 Software Defined Radio (SDR)

In un sistema Software Defined Radio (SDR) gli strumenti di elaborazione dei segnali sono implementati via software. Questo permette di ottenere setup estremamente flessibili, in quanto nel caso in cui sia necessario modificare la configurazione è sufficiente cambiare il codice relativo ai componenti hardware d'interesse. Questo consente un risparmio in termini di costo non essendo necessario acquistare ulteriore materiale. Ciò significa che lo stesso hardware può essere utilizzato per ottenere setup differenti.
Esempi di componenti hardware sostituiti da blocchi software sono filtri, modulatori, demodulatori, amplificatori e similari. Particolare rilievo è attribuito ai convertitori digitale-analogico (DAC) e analogico-digitale (ADC), che consentono la comunicazione del sistema con blocchi RF che ricevono e trasmettono segnali. Inoltre, è richiesta la presenza di un computer per salvare ed elaborare i dati. In Figura 5.1.1 è visibile la schematizzazione di una struttura SDR. Si nota che sono presenti tre componenti distinte: hardware, software e mezzi di collegamento. Ogni componente viene valutato secondo alcuni parametri. L'hardware e il software vengono valutati in base alla banda di funzionamento e alla sample rate in cui operano. Inoltre, i componenti hardware valutano la sensibilità al rumore, il consumo di potenza e l'ingombro. Una caratteristica fondamentale del software è invece la semplicità del codice e la sua flessibilità. Il mezzo che unisce l'hardware del setup all'host, dove risiede il codice, deve avere una banda passante adeguata. Tipicamente si tratta di cavi Ethernet o USB. Per realizzare un sistema SDR esistono diversi software dedicati e GnuRadio è quello più utilizzato.

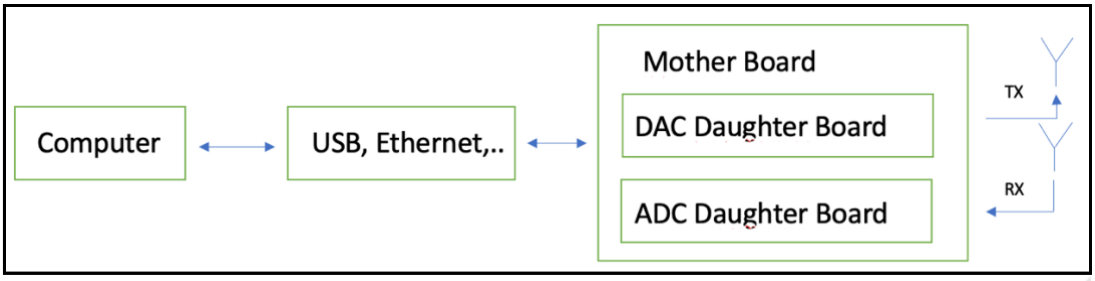


Figura 5.1.1. Struttura di un sistema SDR.

5.2 ADALM-PLUTO

ADALM-PLUTO Active Learning Module (PlutoSDR) è un dispositivo a basso costo facile da usare commercializzato da Analog Devices Inc. che può essere utilizzato per avvicinarsi alla tecnologia Software Defined Radio (SDR) oppure per apprendere i concetti e gli argomenti base della ingegneria elettronica, oppure ancora per approfondire argomenti avanzati nel campo delle comunicazioni in radio frequenza, sia in modalità "istruttore" che in modalità "auto-apprendimento". PlutoSDR consente agli studenti di comprendere meglio il mondo che li circonda ed è applicabile a tutti i livelli e in tutti gli ambiti. Il PlutoSDR è uno strumento che "amplifica" la relazione tra teoria e attività pratica nel campo della radiofrequenza. Basato sull'AD9363 di Analog Devices offre un canale di ricezione e un canale di trasmissione che possono essere gestiti in full duplex e viene fornito con due antenne che coprono 824-894 MHz e 1710-2170 MHz. PlutoSDR è completamente autonomo è interamente alimentato tramite USB con il firmware predefinito. È supportato da sistemi operativi come OS X, Windows, Linux e consente l'esplorazione e la comprensione dei sistemi RF. Inoltre, è supportato da MATLAB, Simulink e utilizza i blocchi drain e source GNU Radio, libiio, una libreria con C, C++, C# e binding Python. In Figura 5.2.1 possiamo vedere come è rappresentato il dispositivo ADALM-PLUTO, con il su cavetto USB di collegamento e alimentazione, completo di due antenne GSM.

Figura 5.2.1. ADALM-PLUTO.

5.2.1 Dati tecnici ADALM-PLUTO

- Modulo autonomo di apprendimento RF portatile
- Piattaforma di sperimentazione economica
- Copertura RF da 325 MHz a 3,8 GHz (anche da 70 MHz fino a 6 GHz i trasmissione e recezione "per avere questa copertura bisogna fare una modifica software")
- ADC e DAC a 12 bit a velocità flessibile

- Un trasmettitore e un ricevitore (SMA femmina, 50 Ω)
- Half o full-duplex
- Supporto MATLAB, Simulink
- Blocchi drain e source GNU Radio
- API Libiio, C, C++, C# e Python
- Interfaccia USB 2.0
- Involucro in plastica
- Alimentato tramite USB
- Fino a 20 MHz di larghezza di banda istantanea (I/Q complesso) (anche 56 MHz di larghezza di banda)

5.3 Interfacciamento ADALM-PLUTO con GnuRadio.

5.3.1 Interfacciamento ADALM-PLUTO con il Host

Dopo aver collegato ADALM-PLTU la prima cosa da verificare è la comunicazione con il Pc. Successivamente in base al sistema operativo digitiamo il seguente comando nel prompt dei comandi.

- Sistema operativo: Mac Os, Linux – ifconfig
- Sistema operativo: Windows – ipconfig

Dopo che abbiamo scritto il comando e premuto invio appare la seguente Figura 5.3.1, dove si evidenzia l'indirizzo ip del dispositivo e la sua maschera. Per controllare i dati del dispositivo quando lo colleghiamo al nostro Pc, deve apparire l'icona con il nome ADALM-PLUTO ed entrando avremo tutti i dati del dispositivo. Se non otterremo il risultato prefissato come in Figura 5.3.1, potrebbe dipendere dai driver di collegamento con ADALM-PLUTO non installati. Per risolvere il problema possiamo consultare [15].

Figura 5.3.1. Collegamento ADALM-PLUTO al Host.

Dopo che abbiamo controllato che il nostro dispositivo dialoga con il host, dobbiamo controllare se ADALM-PLUTO trasmette i dati. In base al sistema operativo digitiamo il seguente comando dentro il prompt dei comandi.

- Mac Os, Linux – ping 192.168.2.10
- Windows – ping 192.168.2.10

Dopo l'invio del comando vedremo il LED1 del dispositivo lampeggiare, ciò significa che sta trasmettendo i dati; vedremo anche in schermata quanti pacchetti abbiamo trasmessi, ricevuti e persi, come in Figura 5.3.2.

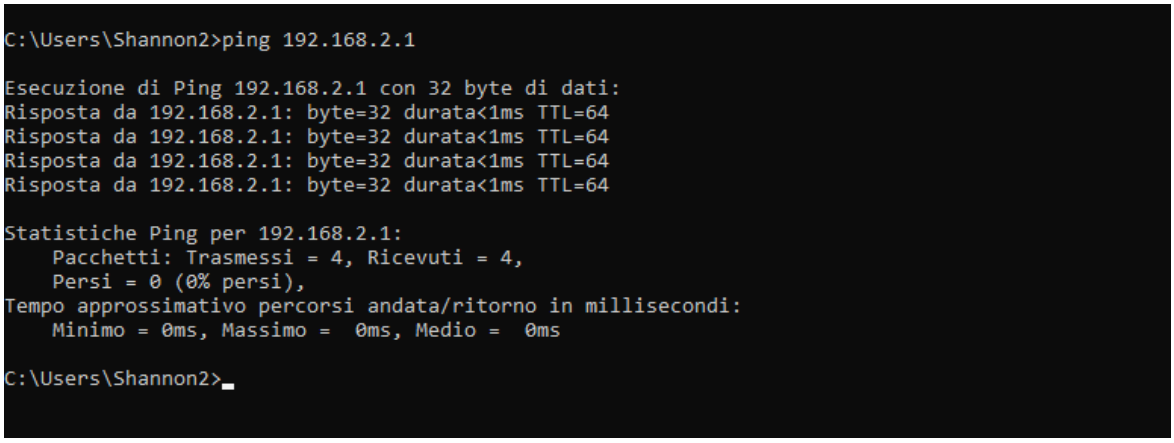


Figura 5.3.2. Trasmissione dei dati.

5.3.2 Trasferimento dati ADALM-PLUTO con software GnuRadio

Quando installiamo il software GnuRadio, devono essere già presenti i blocchi per interfacciarsi con ADALM-PLUTO. Andiamo nella libreria, cerchiamo i blocchi *PlutoSDR Source e PlutoSDR Sink* e trasciniamoli in area di lavoro, come in Figura 5.3.3.

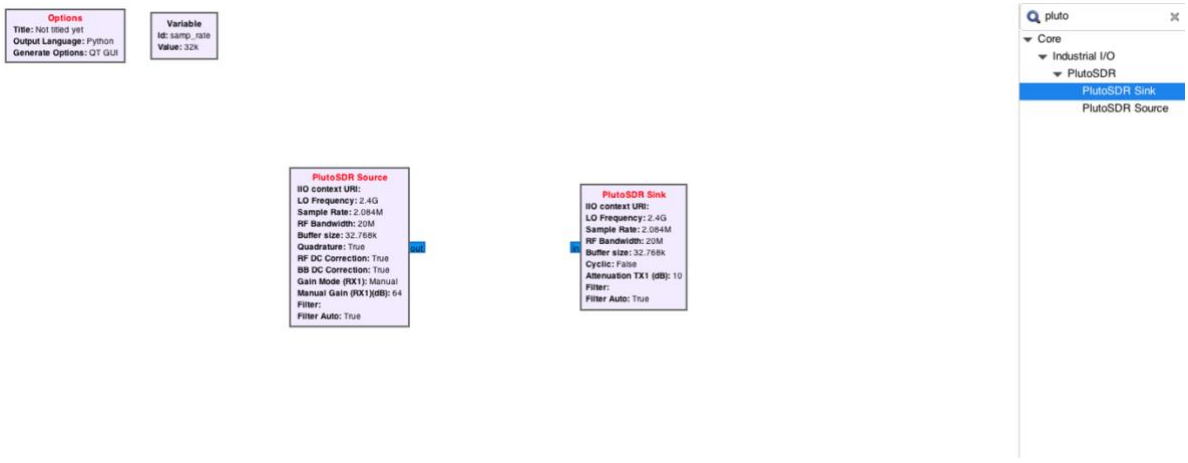


Figura 5.3.3. Blocchi PlutoSDR.

Descriviamo le proprietà del blocco *PlutoSDR Source e PlutoSDR Sink,* che possono essere modificati in base a quello che vogliamo ottener.

PlutoSDR Source:

- LO Frequency - Seleziona la frequenza dell'oscillatore locale RX.
- IIO context URI - Indirizzo IP dell'unità, ad es. "ip:192.168.2.1" (senza virgolette).
- Sample Rate - Frequenza di campionamento in campioni al secondo, questo definirà quanta larghezza di banda riceve il tuo SDR (il parametro della larghezza di banda RF di seguito definisce solo il filtro). limiti: >= 520833 e <= 61440000.
- RF Bandwidth - Larghezza di banda RF configura i filtri analogici RX: RX TIA LPF e RX BB LPF. limiti: >= 200000 e <= 52000000.
- Buffer size - Dimensione del buffer interno in campioni. I blocchi IIO inseriranno/emetteranno solo un buffer di campioni alla volta. Per ottenere la frequenza di campionamento continua più elevata, prova a utilizzare un numero in milioni.
- Quadrature - True/False.
- RF DC Correction - True/False.
- BB DC Correction - True/False
- Gain Mode (Rx1) - Seleziona una delle modalità disponibili: manual, slow_attack, hybrid e fast_attack. Per la maggior parte delle applicazioni di rilevamento dello spettro, utilizzare un guadagno manuale, in modo da sapere effettivamente quando un segnale è presente o meno e la sua potenza relativa.
- Manual Gain (Rx1)(dB) - Valore di guadagno, massimo di 71 dB o 62 dB quando la frequenza centrale è superiore a 4 GHz.
- Filter - Consente di caricare una configurazione del filtro FIR da un file.
- Filter Auto - Se abilitato, carica un filtro predefinito e quindi abilita velocità di campionamento/banda base inferiori.

PlutoSDR Sink:

- IIO context URI - Indirizzo IP dell'unità, ad es. "ip:192.168.2.1" (senza virgolette).
- LO Frequency - Seleziona la frequenza dell'oscillatore locale TX.
- Sample Rate - Frequenza di campionamento in campioni al secondo, questo definirà quanta larghezza di banda trasmette il tuo SDR (il parametro della larghezza di banda RF sotto definisce solo il filtro). limiti: >= 520833 e <= 61440000. Un filtro FIR deve essere caricato o impostato su auto per valori inferiori a 2,083 MSPS.
- RF Bandwidth - Larghezza di banda RF configura i filtri analogici TX: TX BB LPF e TX Secondary LPF. limiti: >= 200000 e <= 52000000.
- Buffer size - Dimensione del buffer interno in campioni. I blocchi IIO inseriranno/emetteranno solo un buffer di campioni alla volta. Per ottenere la frequenza di campionamento continua più elevata, prova a utilizzare un numero in milioni.

- Cyclic - Impostare su "true" se si desidera la modalità "ciclica". In questo caso, il primo buffer di campioni verrà ripetuto su PlutoSDR fino all'arresto del programma. Il blocco PlutoSDR IIO riporterà la sua elaborazione come completa: i blocchi collegati al blocco PlutoSDR IIO non verranno più eseguiti, ma il resto del grafico di flusso sì.
- Attenuation TX1 (dB) - Controlla l'attenuazione per TX1. L'intervallo va da 0 a 89,75 dB in passi di 0,25 dB. Nota: l'uscita massima si verifica con un'attenuazione pari a 0.
- Filter - Consente di caricare una configurazione del filtro FIR da un file.
- Filter Auto - Se abilitato, carica un filtro predefinito e quindi abilita velocità di campionamento/banda base inferiori.

5.3.3 Esempio 1: come interfacciare ADALM-PLUTO con il nostro Pc

Una volta fatto il test che ADALM-PLUTO comunica con il host, apriamo il nostro programma GnuRadio Companion, trasciniamo in area di lavoro i blocchi *PlutoSDR Source*, *PlutoSDR Sink, Signal Source e QT GUI Sink* e colleghiamoli come in Figura 5.3.4.

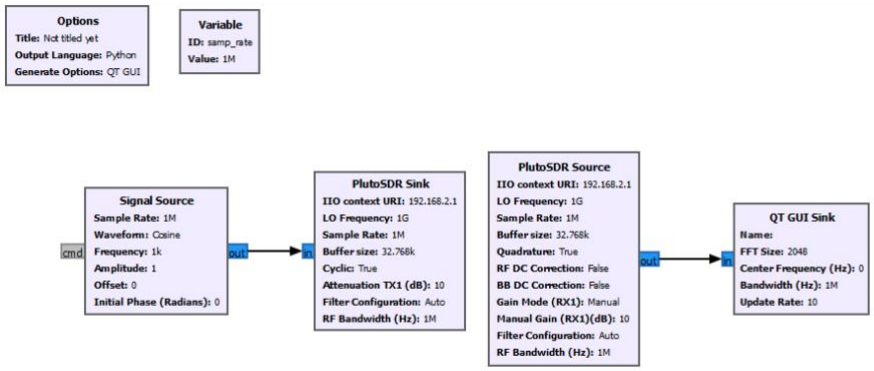


Figura 5.3.4. Diagramma di flusso con i blocchi PlutoSDR.

Verifichiamo la sincronizzazione. Con i blocchi collegati in questa maniera noi trasmettiamo dei dati e riceviamo quello che abbiamo trasmesso. La sorgente del segnale verrà immessa nella sincronizzazione e trasmetteremo dei segnali con *PlutoSDR Sink.* La *PlutoSDR Source* riceverà dei segnali che saranno visualizzati da *QT GUI Sink.* Per ottimizzare il risultato possiamo collegare il trasmettitore e ricevitore del dispositivo con un loop, come in Figura 5.3.5.
IMPORTANTE: MAI INIZIARE A TRASMETTERE SENZA ALCUNA CONNESSIONE AL TRASMETTITORE, SAREBBE COME CREARE UN CORTOCIRCUITO AL DISPOSITIVO, RISCHIANDO DI BRUCIARLO.

Figura 5.3.5. ADALM-PLUTO collegamento antenna a loop.

Modifichiamo le proprietà dei blocchi, otterremo seguente Figura 5.3.6.

Variable:

- Id – samp_rate
- Value – 1000000 Hz

PlutoSDR Sink:

- IIO context URI - 192.168.2.1
- LO Frequency – 1000000000 HZ
- Sample Rate – samp_rate
- RF Bandwidth - 1000000
- Buffer size – 0x8000
- Cyclic - True
- Attenuation TX1 (dB) – 10 dB
- Filter Auto – True

PlutoSDR Source:

- IIO context URI - 192.168.2.1
- LO Frequency – 1000000000 Hz
- Sample Rate – samp_rate
- RF Bandwidth - 1000000
- Buffer size – 0x8000
- Quadrature - True

- RF DC Correction - False.
- BB DC Correction - False
- Gain Mode (Rx1) - Manual
- Manual Gain (Rx1)(dB) – 10 dB
- Filter Auto – True

Signal Source:

- Sample Rate – samp_rate
- Waveform – Cosine
- Frequency – 10000 Hz
- Amplitude – 1
- Ofset – 0

QT GUI Sink:

- FFT Size – 2048 Hz
- Bandwidth(Hz) – samp_rate

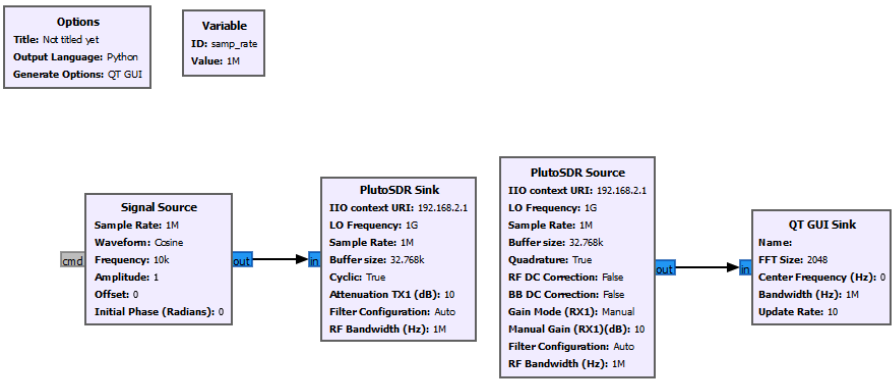


Figura 5.3.6. Diagramma di flusso con proprietà modificate.

Eseguendo il nostro diagramma di flusso, otterremo un grafico dove viene mostrato il nostro coseno complesso con frequenza positiva di 10KHz, essendo un coseno complesso con frequenza positiva avremo uno spostamento positivo, come in Figura 5.3.7.

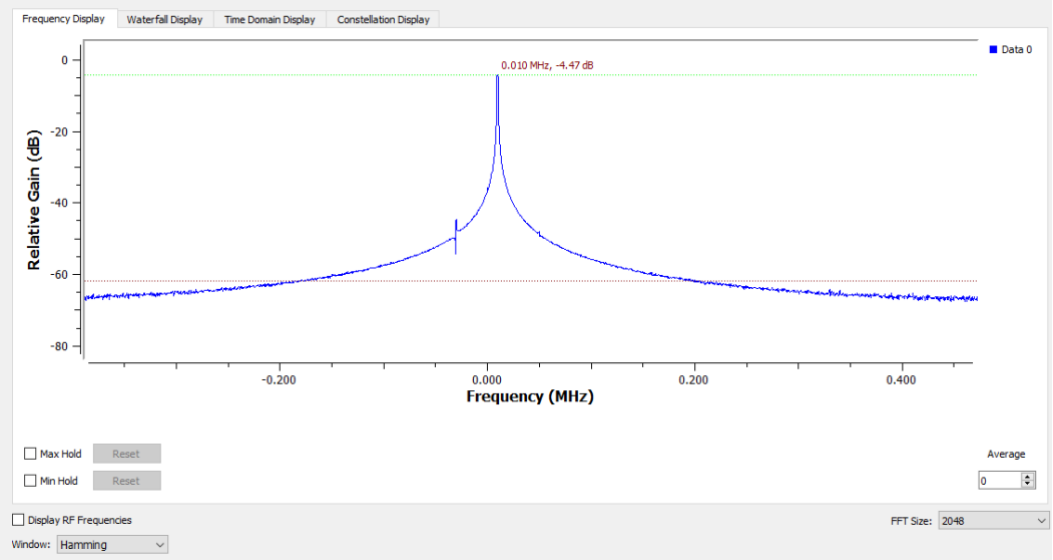


Figura 5.3.7. Spettro in dominio delle frequenze.

Vediamo anche il grafico del coseno complesso nel dominio del tempo, come in Figura 5.3.8.

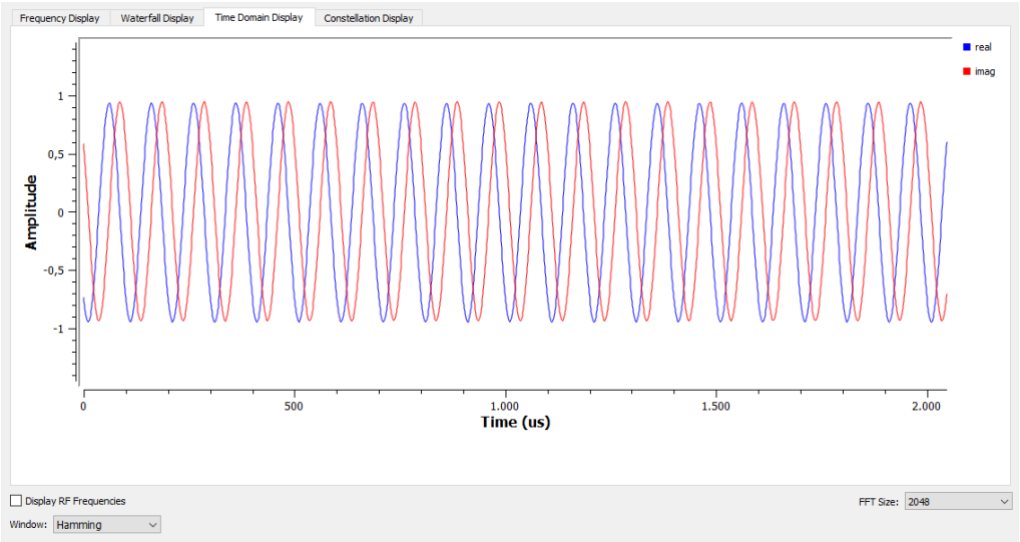


Figura 5.3.8. Coseno complesso nel dominio del tempo.

Possiamo vedere anche lo spettro di frequenze nel dominio del tempo, come in figura 5.3.9.

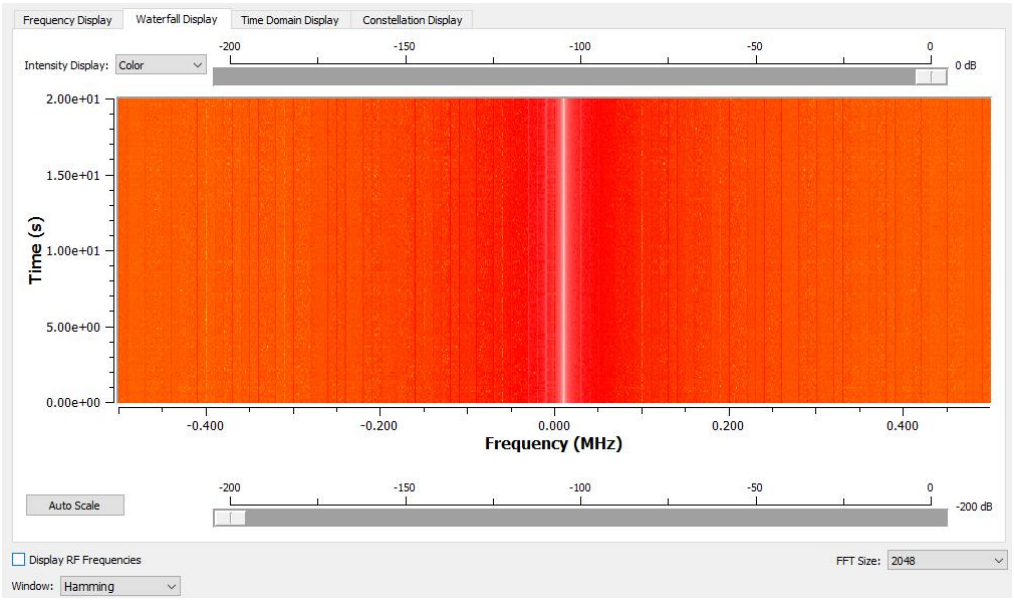


Figura 5.3.9. Spettro delle frequenze nel dominio del tempo.

Ora se facciamo la modifica della frequenza nel blocco *Signal Source,* mettendo Frequency=100kHz, otterremo il seguente spettro nel dominio delle frequenze con alcune armoniche dato che la trasmissione non è perfetta, come in Figura 5.3.10.

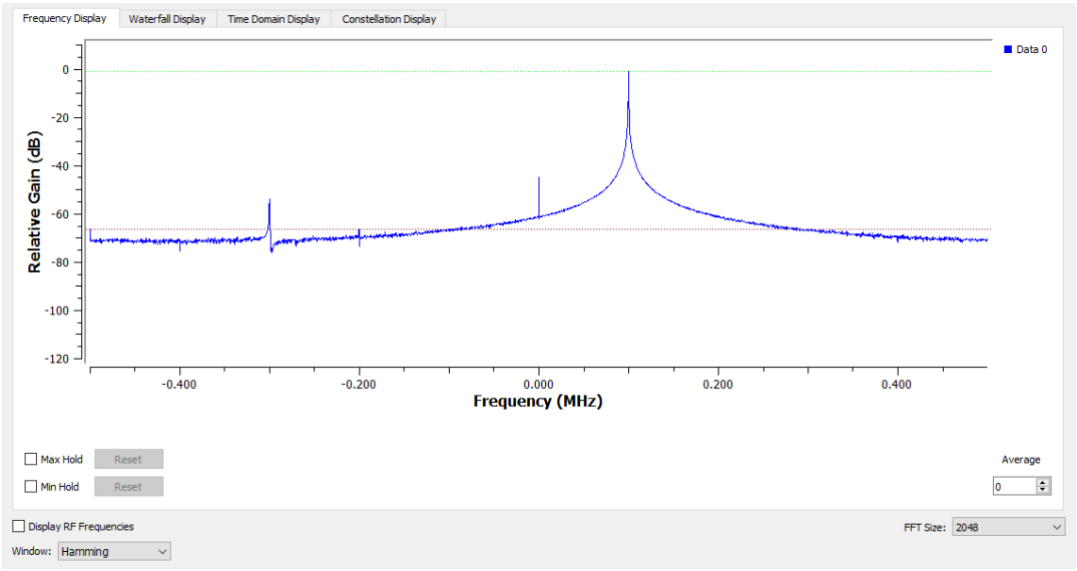


Figura 5.3.10. Spettro nel dominio delle frequenze.

Nel dominio del tempo vedremo come il coseno complesso e diventato più veloce, come in Figura 5.3.11.

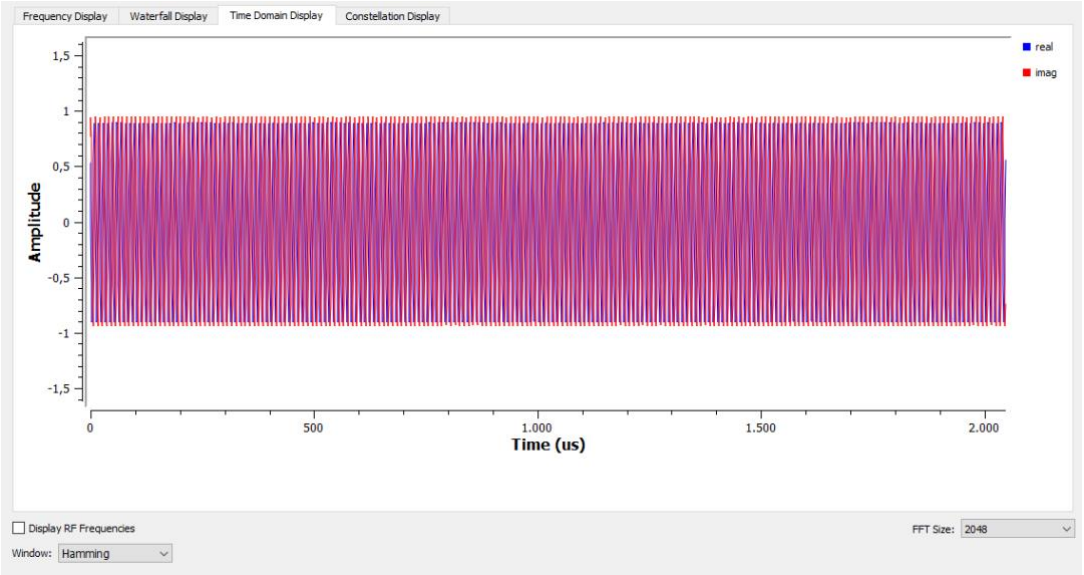


Figura 5.3.11. Coseno complesso nel dominio del tempo.

5.3.4 Esempio 2: Percezione del segnale in movimento.

Costruiamo l'applicazione con la capacità di trasmettere e ricevere un segnale. Proviamo a costruire qualcosa che riesca a percepire i riflessi di una persona o di qualsiasi oggetto che si muove nella stanza. Dato che le due antenne sono molto vicine, la maggior parte del segnale viene captato direttamente dall'antenna ricevente; quindi, dovremmo cercare un picco che è stato riflesso da qualcosa in movimento. In questo esempio useremo una frequenza che arriva molto vicino al massimo della banda di ADALM-PLUTO cioè 3,5GHz.

Andiamo nella libreria di GnuRadio Companion, cerchiamo i seguenti blocchi e trasciniamoli in area di lavoro.
Per la trasmissione, *PlutoSDR Sink, 2 blocchi QT GUI Range, QT GUI Time Sink.*
Per la ricezione, *PlutoSDR Source, QT GUI Range, QT GUI Frequency Sink, QT GUI Time Sink.*

Modifichando le proprietà dei blocchi, otterremo seguente Figura 5.3.12.

Variable:

- Id – samp_rate
- Value – 2400000 Hz

Blocchi Trasmissione.

PlutoSDR Sink:

- IIO context URI - 192.168.2.1
- LO Frequency – 3500000000 HZ
- Sample Rate – int(samp_rate)
- RF Bandwidth - 1000000
- Buffer size – 32768
- Cyclic - False
- Attenuation TX1 (dB) – tx_attenuation
- RF Bandwidth – 20000000
- Filter Auto – True

Signal Source:

- Sample Rate – samp_rate
- Waveform – Cosine
- Frequency – tone _freq
- Amplitude – 1
- Ofset – 0

QT GUI Time Sink:

- Sample Rate – samp_rate
- Line 1 Label – TX re
- Line 2 Label – TX im

QT GUI Range:

- ID – tone_freq
- Type – float
- Default Value – 500
- Start – (-2e3)
- Stop – 2e3
- Step – 1

QT GUI Range:

- ID – tx_attenuation
- Type – float
- Default Value – 10
- Start – 0
- Stop – 100
- Step – 1

Blocchi ricezione.

PlutoSDR Source:

- IIO context URI - 192.168.2.1
- LO Frequency – 3500000000 Hz
- Sample Rate – int(samp_rate)
- Buffer size – 32768
- Quadrature - True
- RF DC Correction - True
- BB DC Correction - True
- Gain Mode (Rx1) - Manual
- Manual Gain (Rx1)(dB) – rx_gain
- RF Bandwidth – 20000000
- Filter Auto – True

QT GUI Range:

- ID – rx_gain
- Type – float
- Default Value – 64
- Start – 0
- Stop – 70
- Step – 1

QT GUI Time Sink:

- Sample Rate – samp_rate
- Line 1 Label – RX re
- Line 2 Label – RX im

QT GUI Frequency Sink:

- Sample Rate – samp_rate
- Line 1 Label – RX ref

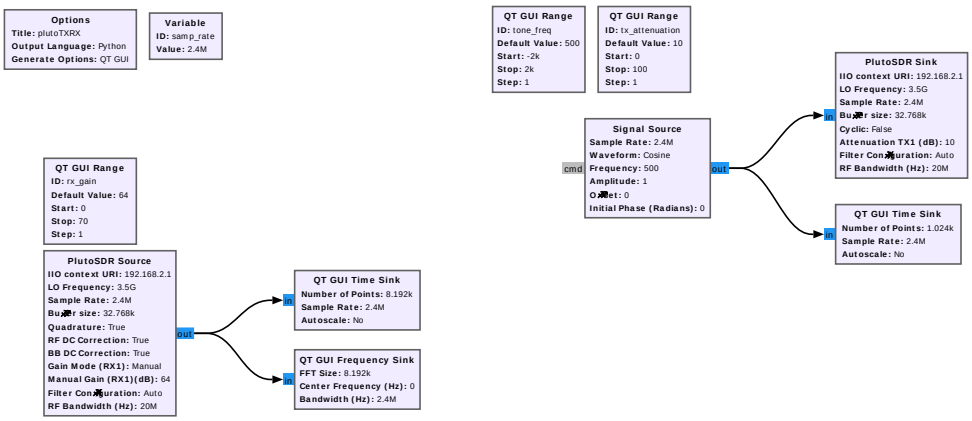


Figura 5.3.12. Diagramma di flusso con proprietà modificate.

Eseguendo il nostro diagramma di flusso mostrato in Figura 5.3.12, otterremo un grafico dove si noterà il picco a 0Hz come in Figura 5.3.13.

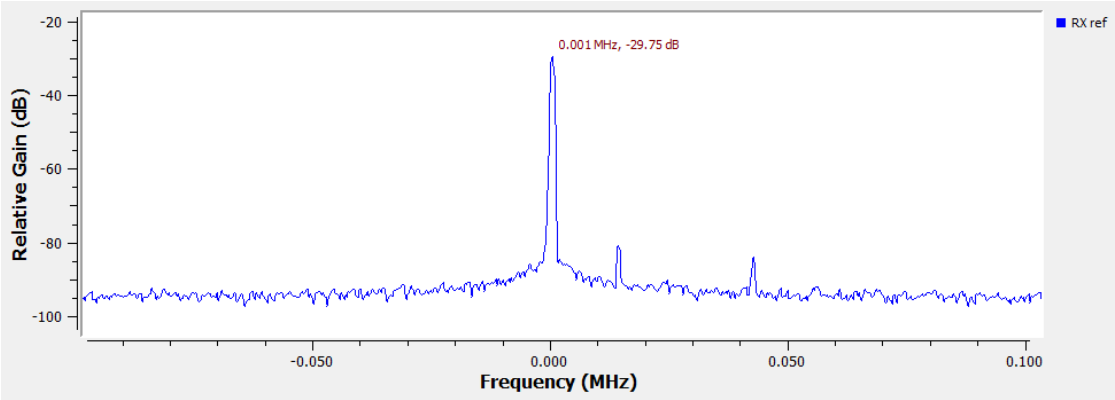


Figura 5.3.13. Spetro di frequenza.

Prendendo un oggetto, nel mio caso la scatola stessa del dispositivo ADALM-PLUTO come in Figura 5.3.14 e movendolo davanti alle antenne del dispositivo fino a una distanza 30-40cm, vedremo piccole variazioni del nostro picco mostrato in Figura 5.3.13.

Figura 5.3.14. Movimento oggetto.

Per vedere meglio la variazione del segnale, lo facciamo passare dentro un filtro "*Low Pass Filter*".

Aggiungiamo i blocchi, *Low Pass Filter, QT GUI Time Sink, QT GUI Frequency Sink e* modifichiamo le proprietà. Si otterrà il seguente risultato come in Figura 5.3.15.

Low Pass Filter:

- Decimation - 50
- Sample Rate – sample rate
- Cutoff Freq (frequenza di taglio) – tone_freq*2
- Transition Width (Hz) - 250

QT GUI Time Sink:

- Sample Rate – samp_rate/50
- Line 1 Label – LPF re
- Line 2 Label – LPF im

QT GUI Frequency Sink:

- Sample Rate – samp_rate/50
- Line 1 Label – LPF

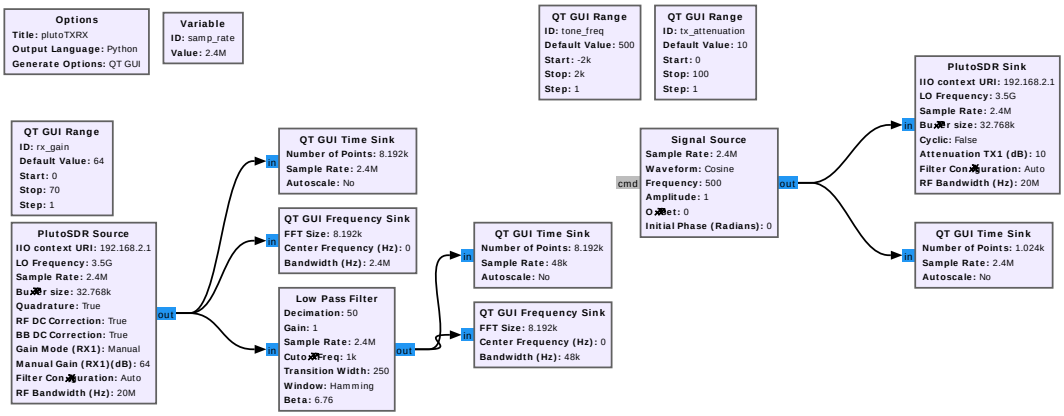


Figura 5.3.15. Diagramma di flusso con proprietà modificate.

Eseguendo il nostro diagramma di flusso mostrato in Figura 5.3.15, otterremo il nostro segnale filtrato passa-basso. Vedremo un picco abbastanza stretto a 500Hz, come in Figura 5.3.16.

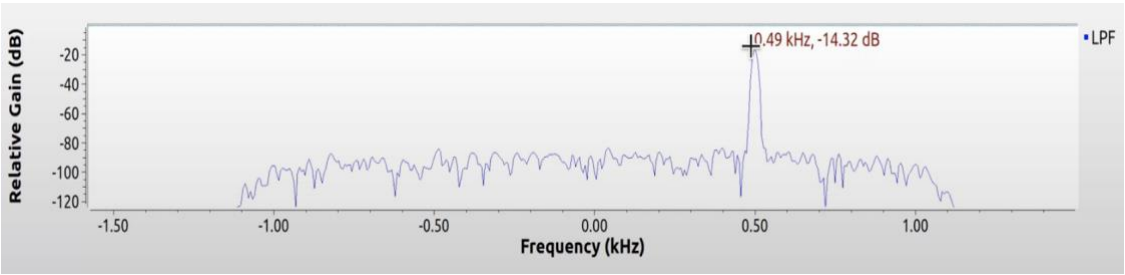


Figura 5.3.16. Spettro di frequenza di un segnale filtrato.

Ora se facciamo muovere un oggetto o semplicemente una mano davanti all'antenne del dispositivo ADALM-PLUTO, vedremo delle variazioni laterali del nostro picco, come in Figura 5.3.17.

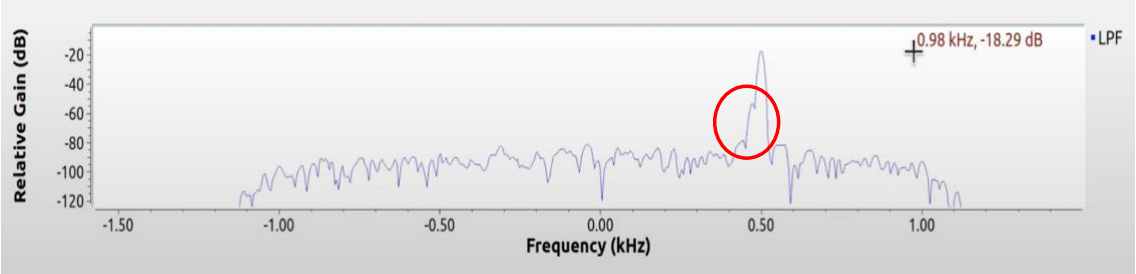


Figura 5.3.17. Spettro di frequenza con movimento.

Nel prossimo passaggio filtreremo il picco che abbiamo ottenuto precedentemente nel grafico dello spettro di frequenze. Per ottenere questo, avremo bisogno di altro filtro "*Band Reject Filter*" che moltiplicheremo per una costante, facendo passare dal blocco "*Multiply Const*".
Aggiungiamo i blocchi, *Band Reject Filter, Multiply Const, QT GUI Range, QT GUI Time Sink, QT GUI Frequency Sink e* modifichiamo le proprietà. Otterremo così il nuovo diagramma di flusso come in Figura 5.3.18.

Band Reject Filter:

- Sample Rate – sample rate/50
- Low Cutoff Freq (frequenza di taglio) – 495
- High Cutoff Freq – 505
- Transition Width - 5

Multiply Const:

- Constant – nom_gain

QT GUI Range:

- ID – nom_gain
- Default Value - 1
- Start – 0
- Stop – 100
- Step - 1

QT GUI Time Sink:

- Sample Rate – samp_rate/50
- Line 1 Label – band reject re
- Line 2 Label – band reject im

QT GUI Frequency Sink:

- Sample Rate – samp_rate/50
- Line 1 Label – band reject

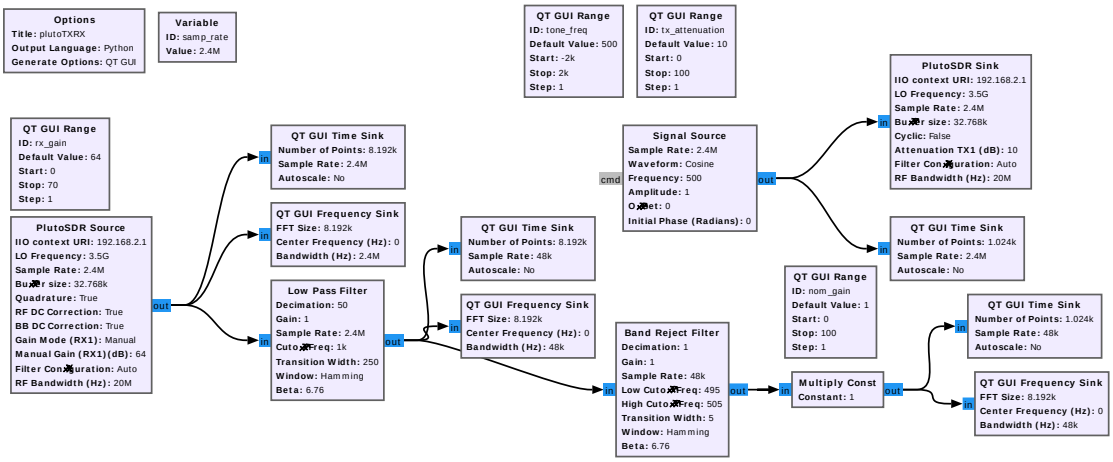


Figura 5.3.18. Diagramma di flusso con proprietà modificate.

Eseguendo il diagramma di flusso in Figura 5.3.18, noteremo che non rimante molto del precedente picco di 500Hz dopo il passaggio dal filtro "*Band Reject Filter*". Se muoviamo un oggetto davanti alle antenne del dispositivo ADALM-PLUTO, si noterà una variazione del picco a 500Hz. In Figura 5.3.19 è mostrato il confronto del segnale dopo e prima del filtro "*Band Reject Filter*".

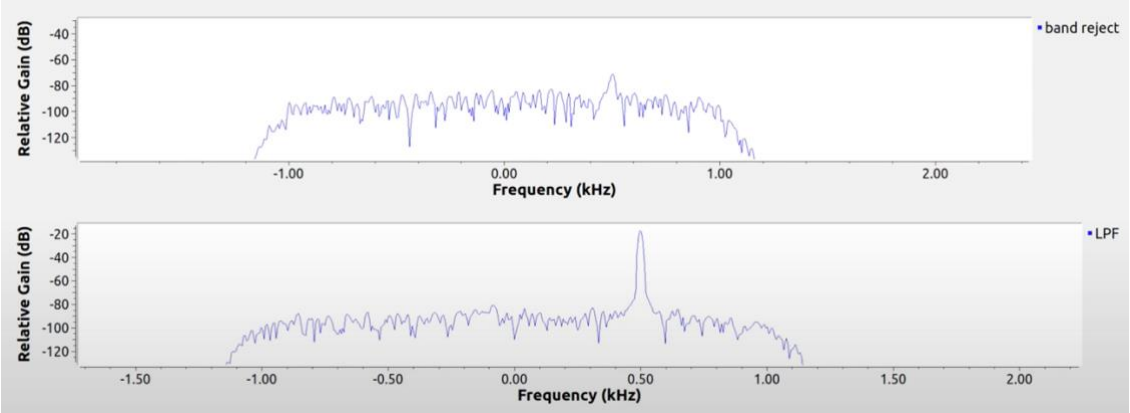


Figura 5.3.19. Spettro di frequenza dopo e prima del filtro "*Band Reject Filter*".

Questo segnale possiamo farlo passare anche dalla scheda audio per sentire/notare la differenza del suono quando muoviamo un oggetto davanti al dispositivo ADALM-PLUTO. Per fare questo dobbiamo trasformare il segnale complesso in quello reale usando il blocco *"Complex To Real"*. Per fare la sincronizzazione audio aggiungiamo il blocco *"Audio Sink"* e modificando i parametri, otterremo Figura 5.3.20.

Audio Sink:

- Sample Rate – 48000 Hz

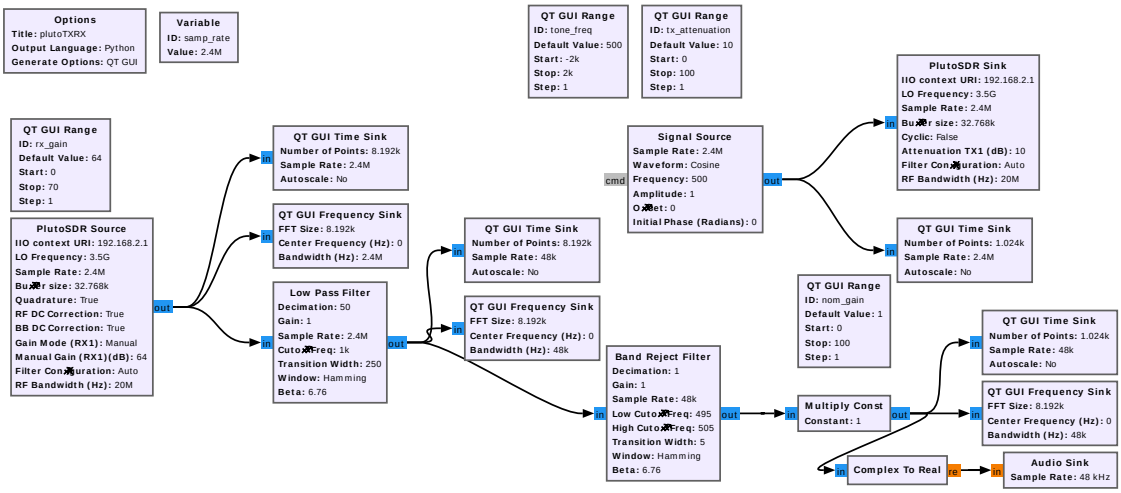


Figura 5.3.120. Diagramma di flusso con proprietà modificate.

Eseguendo il diagramma di flusso in Figura 5.3.20, sentiremo un certo suono. Ogni volta che passiamo un oggetto davanti alle antenne del dispositivo ADALM-PLUTO il suono cambierà, catturando il movimento dell'oggetto nella stanza.

Capitolo 6

Conclusioni

Lo studio svolto ha permesso di mostrare che il Software GnuRadio è un progetto Hardware/Software che fornisce una vasta libreria di strumenti con i quali è possibile costruire SDR. La caratteristica fondamentale di GnuRadio è quella di essere una libreria organizzata a blocchi, che realizzano specifiche funzioni. Esempi di blocchi sono filtri, mixers, automatic gain control (AGC), modulatori, demodulatori, ecc. Questo permette di sperimentare la SDR senza dover preventivamente conoscere i dettagli implementativi dei blocchi di processamento digitale. Uno strumento in particolare, il GnuRadio Companion, permette di manipolare i blocchi di GnuRadio in maniera visuale e di costruire manualmente la propria SDR, o qualsiasi altro sistema DSP, semplicemente interconnettendo i blocchi funzionali. Questo consente di sperimentare la SDR senza nemmeno dover conoscere la programmazione in Python. La SDR, se approcciata gradualmente e sfruttando la grande semplificazione offerta da GnuRadio, diventa davvero aperta ad un vasto pubblico di sperimentatori senza competenze molto specifiche. Inoltre, GnuRadio è un progetto Free Software, vuol dire che può essere usato e modificato liberamente da qualsiasi utente senza alcun costo.
Per svolgere interfacciamento con dispositivo Hardware esterno, GnuRadio fornisce dei blocchi per dispositivo Hardware. GnuRadio non si limita, però, a interfacciarsi ad un solo Hardware, ha possibilità di interfacciamento con altri dispositivi. In questa tesi è stato utilizzato il sistema chiamato SDR ADALM-PLUTO. Attraverso alcuni esempi è stato mostrata la versatilità e semplicità offerta da GnuRadio nel realizzare sistemi di elaborazione dei segnali con dispositivi hardware esterni.

Appendice

Installazione GnuRadio Companion

Sistema operativo Linux

Il modo consigliato per installare GnuRadio sulla maggior parte delle piattaforme è utilizzare i pacchetti binari già disponibili. Lo sviluppo di GnuRadio può essere veloce e i binari forniti dalla tua distribuzione potrebbero essere obsoleti. Controlla se la versione che stai installando è aggiornata. A volte le vecchie versioni non vengono aggiornate nei sistemi di packaging. Se trovi un bug in una versione precedente di GnuRadio, controlla se il bug esiste ancora nella versione più recente di GNU.
Iniziamo l'installazione; la prima cosa da fare è installare Ubuntu PPA, ma prima bisogna controllare se ci sono vecchie versioni installate di GnuRadio. Se esistono occorre assicurarsi di disinstallare prima tutte le versioni precedentemente installate di GnuRadio.

Aprendo il terminale del nostro sistema operativo ed eseguendo il comando, verrà scaricato l'elenco dei pacchetti dall'archivio aggiornati per ottenere le informazioni sui pacchetti più recenti.

```
sudo apt-get update
```

Attendiamo fino al completamento dell'aggiornamento per installare il GnuRadio Companion, eseguendo il comando successivo.

```
sudo atp-get install gnuradio
```

Attendiamo il completamento dell'installazione, successivamente verifichiamo sul nostro PC l'icona di GnuRadio Companion che sarà pronta all'uso [16].

Sistema operativo Windows

Ci sono alcuni programmi di installazione binari GnuRadio non ufficiali che sono mantenuti dalla comunità:
1) Installatore Radioconda [17]. Questo programma di installazione binario tende ad essere il più aggiornato, ma occore controllare l'elenco dei pacchetti per vedere se include ciò di cui si ha bisogno. Utilizza il gestore di pacchetto/ambiente conda che fornisce un facile accesso a migliaia di Python e altri pacchetti che non sono strettamente correlati a GnuRadio. Ciò ti consente anche di rimanere aggiornato senza doverlo reinstallare. Poiché questo fornisce gli stessi pacchetti conda-forge disponibili senza il programma di installazione, la guida all'installazione di conda potrebbe essere utile per istruzioni aggiuntive, incluso come creare moduli OOT aggiuntivi dalla sorgente.

Dopo aver installato Radioconda otterrai un "GnuRadio Companion" aggiunto al tuo menu di avvio. Avviamo GnuRadio Companion.
2) Il programma di installazione di Geof Nieboer ospitato su gcndevelopment.com (recentemente irraggiungibile dal 2022-07, attualmente accessibile dal 2022-10-23). Questo è un programma di installazione binario per Windows 7/8/10 a 64 bit che include tutte le dipendenze per Windows, una distribuzione Python personalizzata, driver SDR comunemente usati e diversi blocchi OOT.
3) Programma di installazione dell'ambiente di sviluppo Pothos SDR. Questo programma di installazione binario include GnuRadio, Pothos, CubicSDK e altri strumenti. Storicamente è stato aggiornato circa una o due volte all'anno.

Sistema operativo Mac OS

GnuRadio è stato compilato e installato su OSX 10.4 ("Tiger") fino a 10.15 ("Catalina") eseguendo qualsiasi versione compatibile di Xcode su tutti i Mac recenti e molti vecchi, siano essi Intel o PowerPC/PPC. C'è pochissimo supporto per ottenere le librerie e le applicazioni in background installate su OSX 10.5 o versioni precedenti, né Intel a 32 bit o qualsiasi PPC, sebbene tutte queste dovrebbero essere possibili. Il supporto principale è per i Mac basati su Intel a 64 bit con OSX 10.6 o versioni successive.
Prerequisiti: l'esecuzione di quasi tutte le interfacce grafiche (GUI) di GNU Radio richiederà prima il download e l'installazione di X11/XQuartz. Attraverso OSX 10.8, Apple ha fornito un mezzo per installare X11.app, ma XQuartz è sempre stato più aggiornato e quindi è consigliato per l'uso. A partire da OSX 10.9, Apple non ha più fornito una versione completa e funzionante di X11.app quindi, usa XQuartz fin dall'inizio. Per instalazione XQuartus [18].
Nel passo successivo installare il MacPorts per la tua versione del sistema operativo Mac [19]. Assicurati di seguire le modifiche all'ambiente shell di MacPorts necessarie in modo tale che gli eseguibili installati di MacPorts vengano trovati prima di tutti gli altri. Queste sono le uniche modifiche all'ambiente shell necessarie per eseguire qualsiasi MacPorts. Una volta installato MacPorts, GnuRadio e tutte le sue dipendenze possono essere installate dal terminale inserendo il seguente comando.

```
sudo port install gnuradio
```

Questo metodo di installazione di GnuRadio è costantemente aggiornato da Michael Dickens, e quindi è il metodo consigliato per installare GnuRadio su Mac OS X.

MacPorts richiede che alcune variabili di ambiente siano impostate nella shell. Quando MacPorts viene installato utilizzando il programma di installazione del pacchetto macOS, dopo l'installazione viene eseguito uno script "postflight" che aggiunge o modifica automaticamente un file di configurazione della shell nella tua home directory.
Una volta terminata la l'installazione digitare sul terminale il segunte codice.

```
export PATH=/opt/local/bin:/opt/local/sbin:$PATH
```

Ciò pone i percorsi MacPorts all'inizio di PATH in modo che i file MacPorts abbiano la precedenza sui file originali [20].

L’Installazione è conclusa. Scrivendo sul terminale seguente codice, noi apriamo interfaccia di GnuRadio Companion, sulla quale possiamo svolgere le nostre applicazioni ma vediamo anche la versione corrente della nostra GnuRadio Companion [21].

gnuradio-copmpanion

Ringraziamenti

Ringrazio il Professor Davide Dardari per avermi dato l'opportunità di lavorare ad un progetto di tesi così stimolante e costruttivo e per essere stato una guida sempre presente durante la realizzazione dello stesso.

Grazie alla mia famiglia per avermi sempre sostenuto e per aver reso possibile questo mio percorso. Un Grazie particolare alla mia ragazza che ha creduto in me fin dall'inizio e mi ha spinto a iniziare gli studi.

Infine, è doveroso ringraziare i miei compagni di corso, che mi hanno accompagnato in questi anni e che ricorderò sempre con immenso affetto.

Bibliografia

[1] https://wiki.gnuradio.org/index.php?title=Your_First_Flowgraph
[2] https://wiki.gnuradio.org/index.php?title=Python_Variables_in_GRC
[3] https://wiki.gnuradio.org/index.php?title=Variables_in_Flowgraphs
[4] https://wiki.gnuradio.org/index.php?title=Runtime_Updating_Variables
[5] https://wiki.gnuradio.org/index.php?title=Signal_Data_Types
[6] https://wiki.gnuradio.org/index.php?title=Converting_Data_Types
[7] https://wiki.gnuradio.org/index.php?title=Packing_Bits
[8] https://wiki.gnuradio.org/index.php?title=Streams_and_Vectors
[9] https://wiki.gnuradio.org/index.php?title=Hier_Blocks_and_Parameters
[10]https://wiki.gnuradio.org/index.php?title=Creating_Python_OOT_with_gr-modtool
[11] https://wiki.gnuradio.org/index.php?title=Creating_Your_First_Block
[12] https://wiki.gnuradio.org/index.php?title=Python_Block_with_Vectors
[13] https://wiki.gnuradio.org/index.php?title=Python_Block_Message_Passing
[14] https://wiki.gnuradio.org/index.php?title=Python_Block_Tags
[15] https://github.com/ryanvolz/radioconda
[16] https://wiki.gnuradio.org/index.php?title=LinuxInstall
[17] https://wiki.gnuradio.org/index.php?title=WindowsInstall
[18] https://www.xquartz.org
[19] https://www.macports.org/install.php
[20] https://guide.macports.org/chunked/installing.shell.html
[21] https://wiki.gnuradio.org/index.php?title=MacInstall