IPv6 and IoT

Martin Báchor

April 2025

STU Faculty of Electrical Engineering and Information Technology

Automatic Delay Control in GNU Radio

Status of this Memo

This memo provides information for the IPv6 and IoT. Distribution of this memo is unlimited.

Copyright Notice

Abstract

This document describes the design and implementation of an automatic delay control in GNURadio. The solution leverages a custom Python block to detect delays. This report includes development, full source code, integration guidelines, and performance analysis.

Table of Contents

Introduction

This project focuses on the automatic detection and control of stream delay in GNU Radio, a widely used open-source software development toolkit for building software-defined radios (SDR).

In many signal processing applications, precise synchronization between data streams is essential — especially when comparing or combining signals that originate from different paths or sources. However, due to buffering, transmission latency, or processing delays, signals may arrive with an unknown delay offset.

The goal of this project is to automatically measure and report the delay between two input streams in real-time using an embedded or OOT block in gnuradio.

Distribution and Environment

This project was developed and tested in a controlled virtual environment to ensure consistent behaviour, compatibility, and reproducibility when working with GNU Radio and its associated libraries.

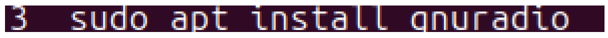
Operating System

Ubuntu Linux 24.04 LTS Running in a Virtual Machine (VM) using Oracle VM VirtualBox

The virtual environment allows easy isolation of dependencies and ensures that the system behaves predictably regardless of the host operating system.

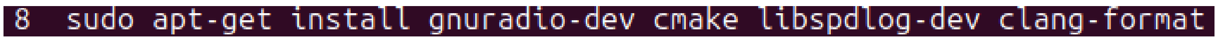
Software Stack

GNU Radio version 3.10.9



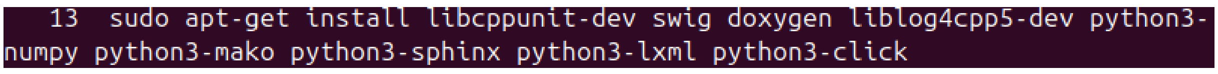
```
3  sudo apt install gnuradio
```

Modules For OOT



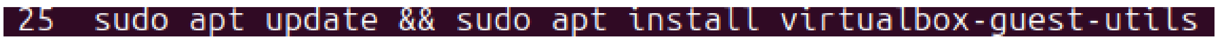
```
8  sudo apt-get install gnuradio-dev cmake libspdlog-dev clang-format
```

Python 3.12.3 and its libraries



```
   13  sudo apt-get install libcppunit-dev swig doxygen liblog4cpp5-dev python3-
numpy python3-mako python3-sphinx python3-lxml python3-click
```

Guest Additions: Installed for better integration (e.g., clipboard sharing, dynamic resolution)



```
25  sudo apt update && sudo apt install virtualbox-guest-utils
```

Libraries used:

NumPy For numerical operations such as cross-correlation

PMT (Polymorphic Types): GNU Radio messaging system used to send detected delay values through message ports. This is included by default with GNU Radio's Python bindings.

Counter: implementation of way to count buffer for delay values

Virtual Machine Configuration

- Base Memory (RAM): 4096 MB (4 GB)
- Processor Cores: 2
- Video Memory: 128 MB
- Storage: 20 GB dynamically allocated virtual hard disk

This configuration is sufficient for developing and running real-time signal processing applications in GNU Radio with low to medium complexity.

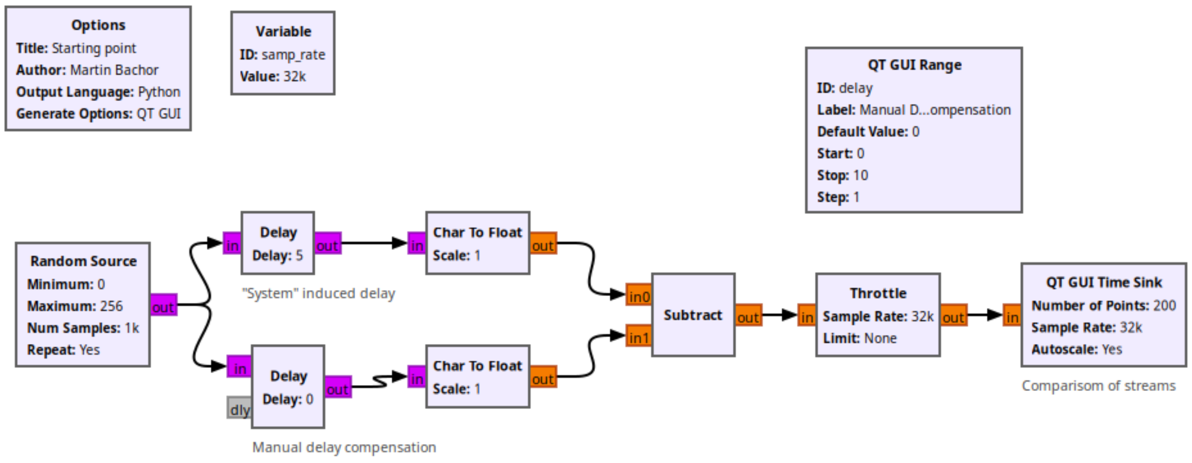
Development

This section outlines the development process of the project, describing the gradual design and implementation stages from the initial signal setup to delay detection and solution for noisy or unstable results in Embedded Python block.

Starting Point

The starting point of the project is a simple GNU Radio flowgraph designed to simulate a delay between two identical signals. The purpose of this setup is to test and validate delay detection logic using basic signal blocks. Below is a screenshot of the initial flowgraph:

Figure 1: Starting Point of GNU Radio



Blocks used:

- Random Source – Generator that produces a random sequence of numbers in our case it generates 1000 random numbers between 0 and 256 and once it generates, it will repeat the process.
- Delay 2x – This block takes signal that is coming to its input and delays it by fixed number of samples, specified in its parameters. In our case "System" inducted delay is set to 5 and Manual delay compensation is block that will simulate delay compensation.
- Subtract – This block takes two input signals and performs sample by sample subtraction.
- Throttle – Regulates flow of samples.
- QT GUY Time Sink – A graphical visualization of signal like an oscilloscope.
- QT GUY Range – provides a slider in GUY that allows user to dynamically change a value,

Embedded Python Block vs OOT (Out of Tree)

During development two approaches for custom functionality were considered: Embedded Python Block and OOT Modules.

OOT Module:

A more robust and scalable solution for creating custom blocks. Requires use of tools like gr_modtool to generate C++ or Python templates. Better suited for long-term or more complex projects [1]. Not used in this development because the functionality was contained enough for an embedded block approach.

Embedded Python Block

This block allows users to create a new custom block in Python directly inside GNU Radio Companion without needing to make and install an Out of Tree (OOT) Module. Fast to prototype and ideal for small custom logic, but less reusable and harder to manage as project grows [2]. Used in this implementation due to ist simplicity and fast experimentation.

Cross-Correlation Method

Cross-correlation measures the similarity between two sequences as a function of the displacement of one relative to the other. It reveals how one series (reference) is correlated with the other (target) when shifted by a specific amount.

Implementation of Cross-correlation Analysis in Python

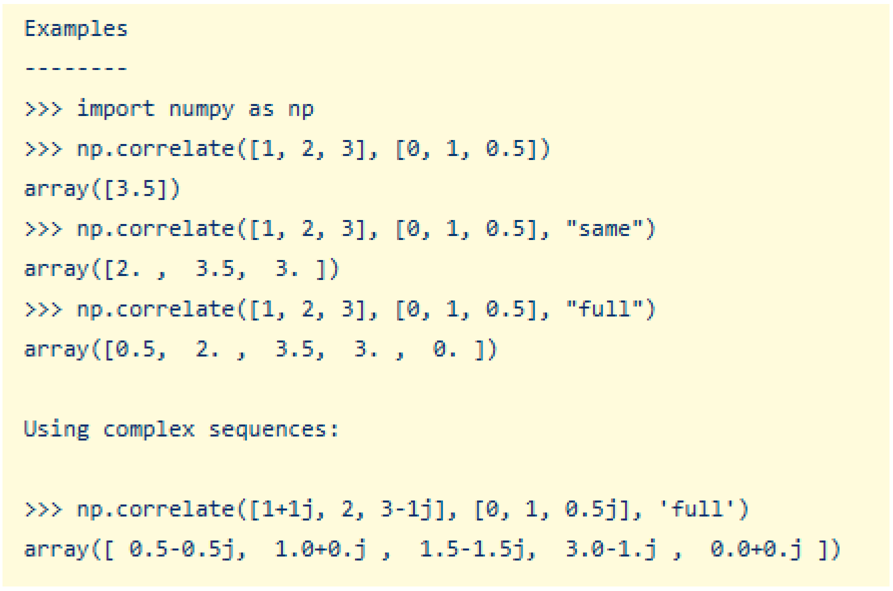
Function for Cross-Correlation in python uses numpy library that we installed at start. Utilizing NumPy's fast numerical operations for efficient cross-correlation computation. [3]

numpy.correlate(*a*, *v*, *mode='valid'*) - Cross-correlation of two 1-dimensional sequences.

Parameters: **-a, v *array_like*** - Input sequences.

-mode*{'valid', 'same', 'full'}, optional* **-** Refer to the **convolve** docstring. Note that the default is 'valid', unlike **convolve**, which uses 'full'. [4]

Figure 2: Examples of Cross-correlation in python [5]

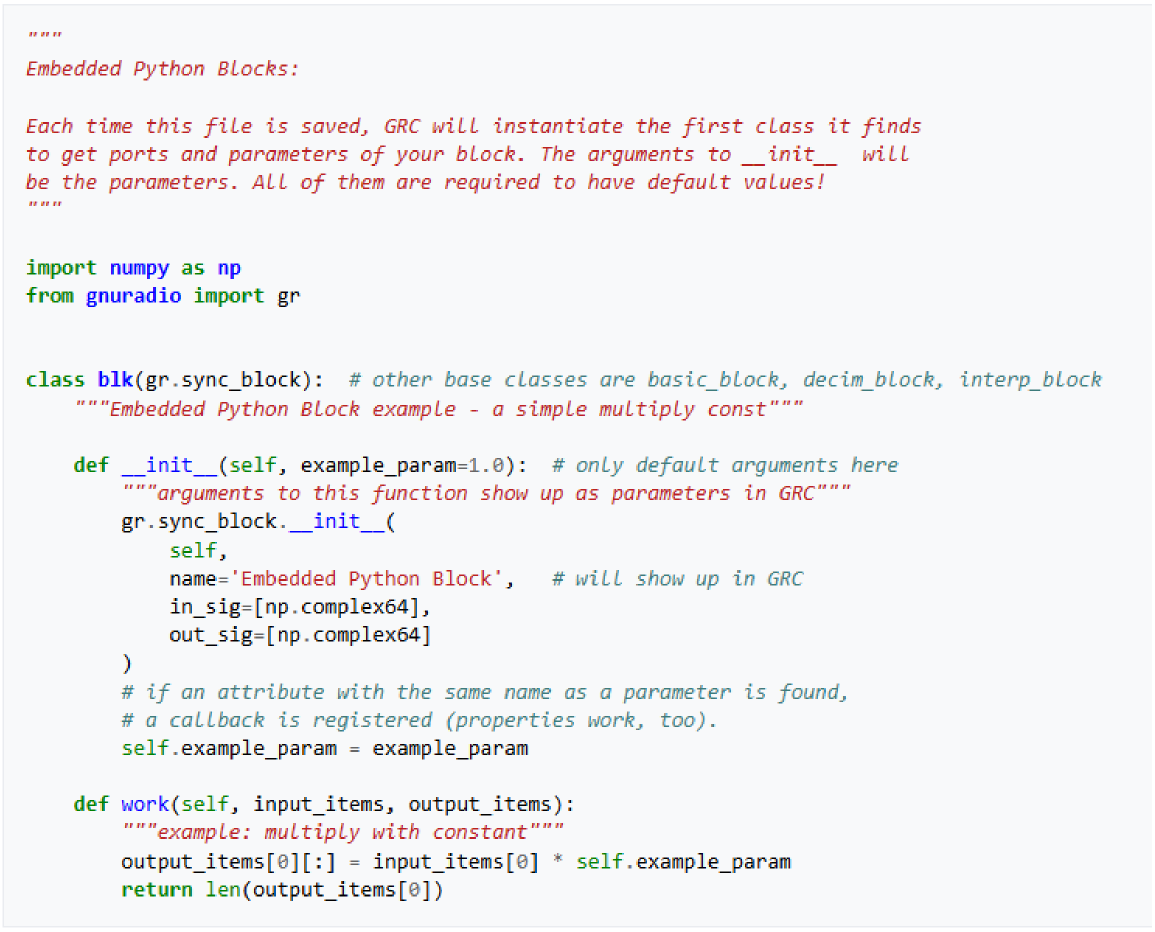


```
Examples
--------
>>> import numpy as np
>>> np.correlate([1, 2, 3], [0, 1, 0.5])
array([3.5])
>>> np.correlate([1, 2, 3], [0, 1, 0.5], "same")
array([2. ,  3.5,  3. ])
>>> np.correlate([1, 2, 3], [0, 1, 0.5], "full")
array([0.5,  2. ,  3.5,  3. ,  0. ])

Using complex sequences:

>>> np.correlate([1+1j, 2, 3-1j], [0, 1, 0.5j], 'full')
array([ 0.5-0.5j,  1.0+0.j ,  1.5-1.5j,  3.0-1.j ,  0.0+0.j ])
```

Implementation of Embedded Python Block

- Initial code in the Python block: pre-populated code that will be our starting point

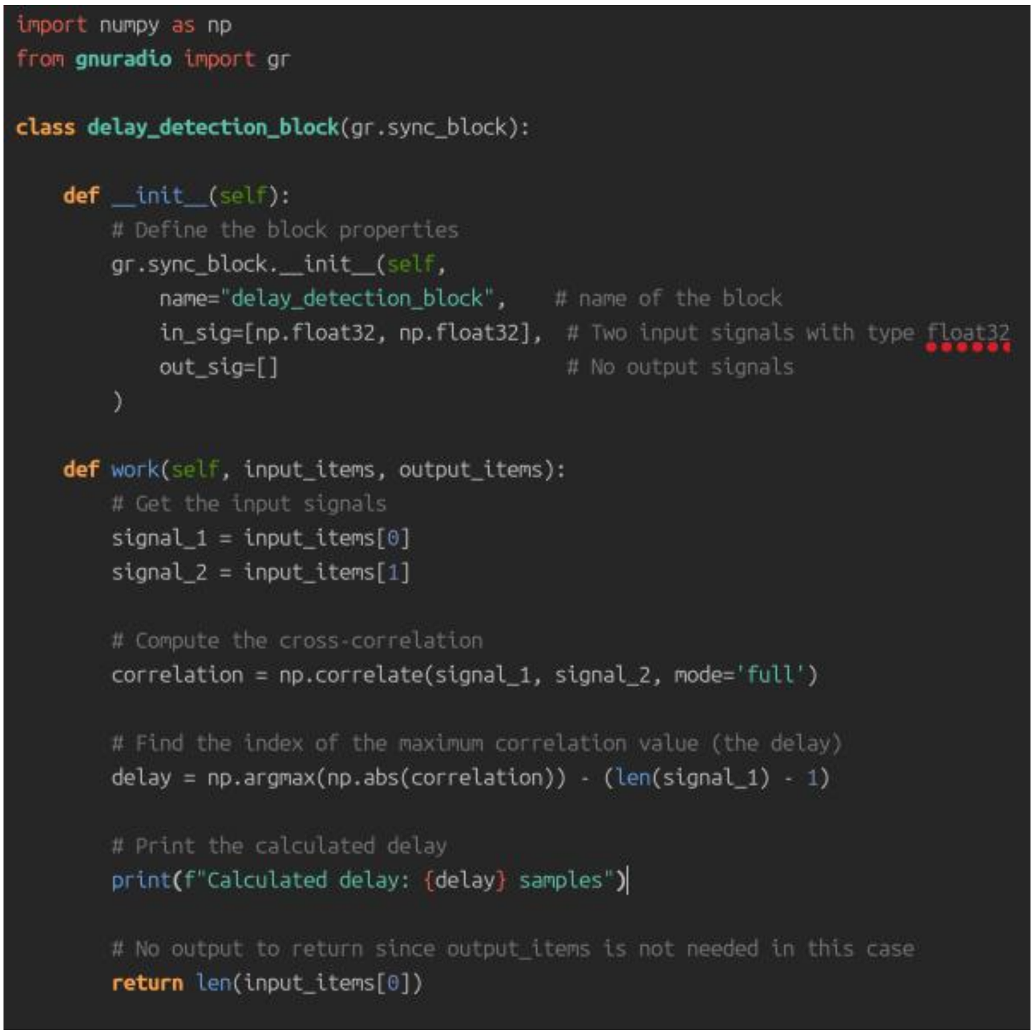
Figure 3: pre-populated code of python block [2]



```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, 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])
```

- **First implementation and testing:** This block receives two float input signals and calculates the delay between them using cross-correlation. Cross-correlation calculates similarity between two signals as one is shifted over other. Then by finding the position of maximum value in the correlation result, we determine by how many samples is signal shifted. This version of block was used primarily for proof of concept to verify that this method is working.

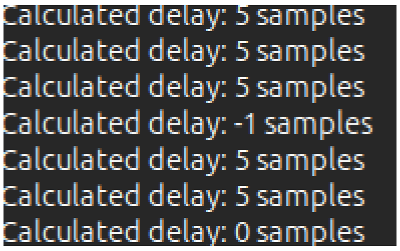
Figure 4: First version of block



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

class delay_detection_block(gr.sync_block):

    def __init__(self):
        # Define the block properties
        gr.sync_block.__init__(self,
            name="delay_detection_block",    # name of the block
            in_sig=[np.float32, np.float32],  # Two input signals with type float32
            out_sig=[]                        # No output signals
        )

    def work(self, input_items, output_items):
        # Get the input signals
        signal_1 = input_items[0]
        signal_2 = input_items[1]

        # Compute the cross-correlation
        correlation = np.correlate(signal_1, signal_2, mode='full')

        # Find the index of the maximum correlation value (the delay)
        delay = np.argmax(np.abs(correlation)) - (len(signal_1) - 1)

        # Print the calculated delay
        print(f"Calculated delay: {delay} samples")

        # No output to return since output_items is not needed in this case
        return len(input_items[0])
```

Output of this code looked like this, despite a few incorrect samples, system reliably detects the correct delay. Those inconsistencies are to be stabilized in next implementations.

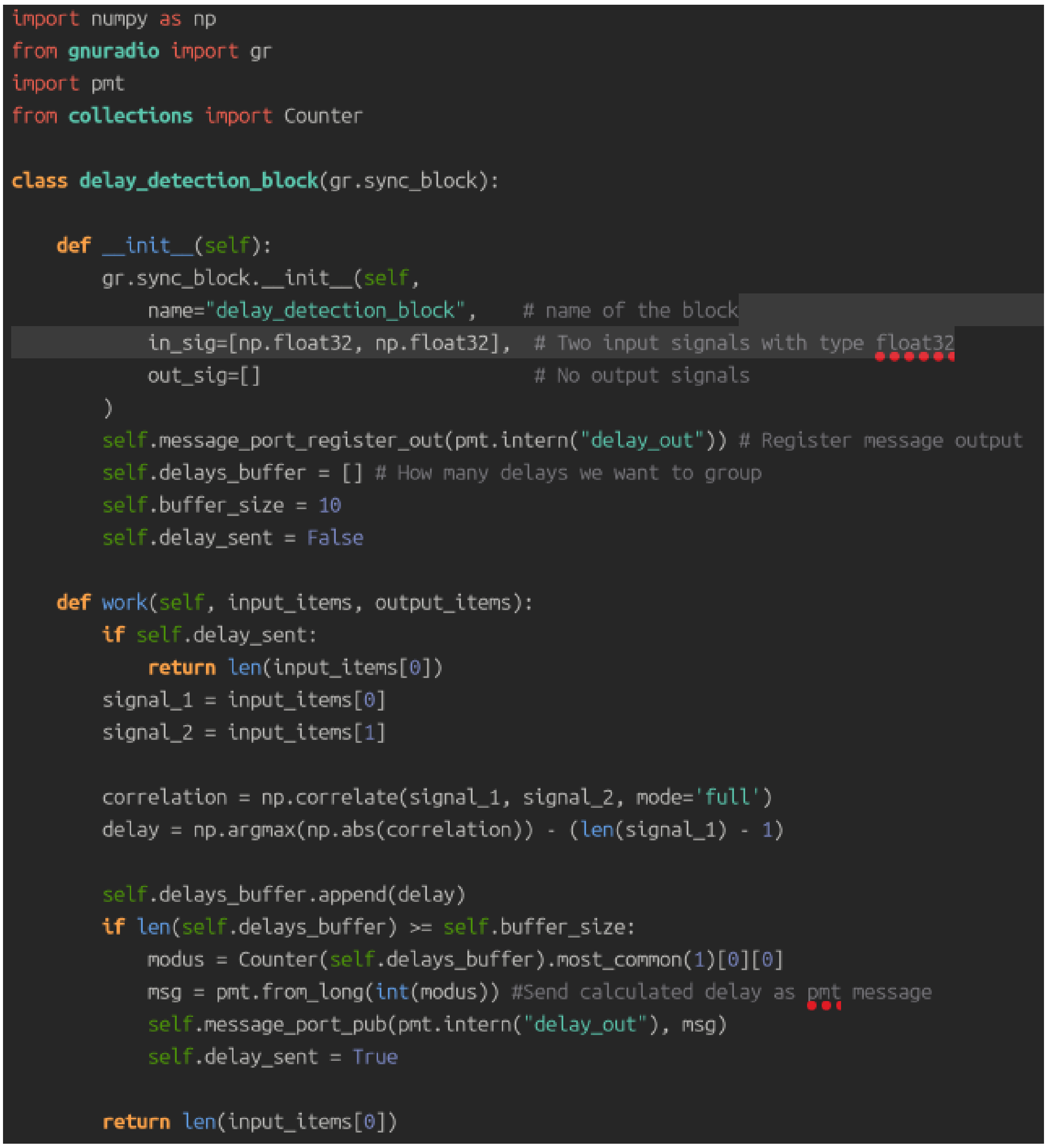
Figure 5: Output 1



```
Calculated delay: 5 samples
Calculated delay: 5 samples
Calculated delay: 5 samples
Calculated delay: -1 samples
Calculated delay: 5 samples
Calculated delay: 5 samples
Calculated delay: 0 samples
```

- Second implementation: Addition of output trough message port, uses pmt library, this allows block to send calculated delay to other blocks via message connection. Introduction of buffer for delay values to improve accuracy and stability, instead of using just one value this version takes multiple values (in this case 10) and stores then into buffer. After collecting 10 values, it computes modus or most common delay using Pythons Counter. This approach filters out noisy or unstable results. Second implementation uses a flag and only sends the first calculated delay. It prevents spamming other block with repeated messages and in our case, it stops the message of 0 delay value that would come next.

Figure 6: Second implementation

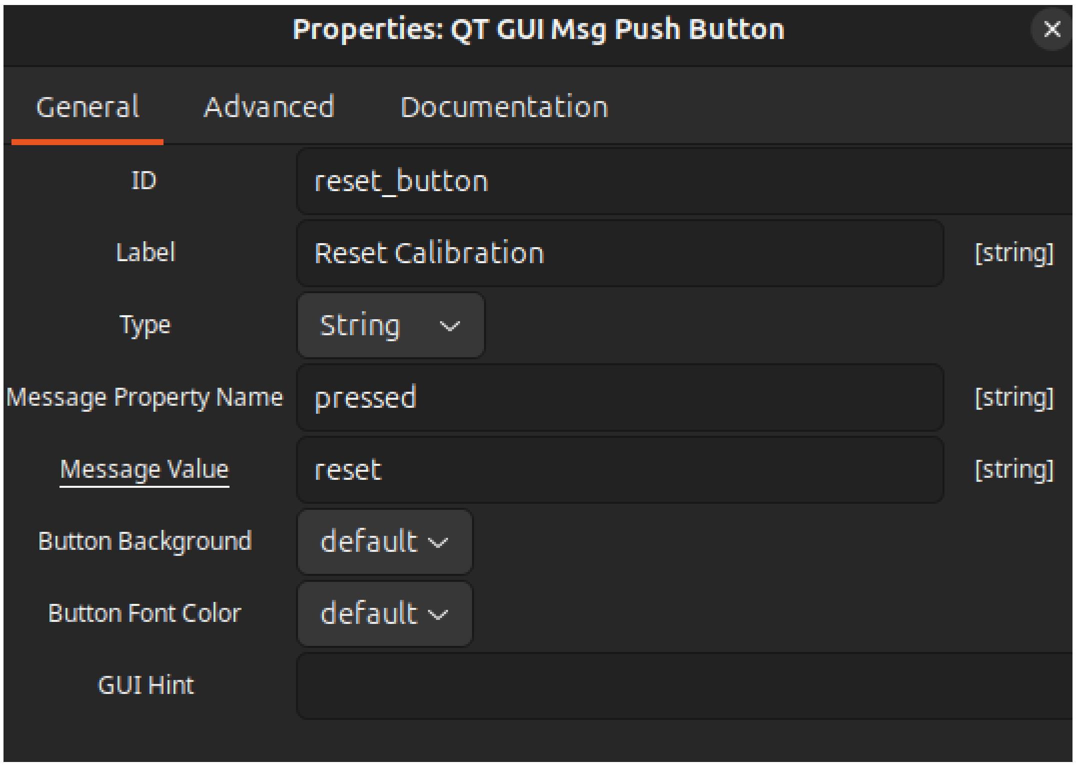


```python
import numpy as np
from gnuradio import gr
import pmt
from collections import Counter

class delay_detection_block(gr.sync_block):

    def __init__(self):
        gr.sync_block.__init__(self,
            name="delay_detection_block",    # name of the block
            in_sig=[np.float32, np.float32],  # Two input signals with type float32
            out_sig=[]                        # No output signals
        )
        self.message_port_register_out(pmt.intern("delay_out")) # Register message output
        self.delays_buffer = [] # How many delays we want to group
        self.buffer_size = 10
        self.delay_sent = False

    def work(self, input_items, output_items):
        if self.delay_sent:
            return len(input_items[0])
        signal_1 = input_items[0]
        signal_2 = input_items[1]

        correlation = np.correlate(signal_1, signal_2, mode='full')
        delay = np.argmax(np.abs(correlation)) - (len(signal_1) - 1)

        self.delays_buffer.append(delay)
        if len(self.delays_buffer) >= self.buffer_size:
            modus = Counter(self.delays_buffer).most_common(1)[0][0]
            msg = pmt.from_long(int(modus)) #Send calculated delay as pmt message
            self.message_port_pub(pmt.intern("delay_out"), msg)
            self.delay_sent = True

        return len(input_items[0])
```

- Final implementation: In final implementation of delay detection block, I introduced two key features to make block more interactive and efficient. I implemented control mechanism for starting calibration process and reset function for the block. This was achieved by using QT GUI Msg Push Button for external input, which sends “start” and “reset” messages to the block. Start button triggers delay detection process, while reset button clears any previous calibration data and sets the delay back to 0.

  Button setup and integration: add two QT GUI Msg Push button blocks from QT category. Connect them to message input of delay detection block and configure each button as shown. For start button its same but Message Value is “start”.

Figure 7: Reset button



Properties: QT GUI Msg Push Button

General | Advanced | Documentation

ID	reset_button	
Label	Reset Calibration	[string]
Type	String	
Message Property Name	pressed	[string]
Message Value	reset	[string]
Button Background	default	
Button Font Color	default	
GUI Hint		

Conclusion

This project successfully implements a custom GNU Radio block for detecting and calibrating delay between two input signals using cross-correlation. This block is enhanced with user interactivity through QT GUI Message Push Buttons, allowing for flexible control of the calibration process.

The final implementation is designed for robustness and ease of use, ensuring the calibration process is both reliable and user controlled. This makes the block suitable for signal alignment tasks in both simulation and real-world SDR applications.

References

[1] https://wiki.gnuradio.org/index.php?title=Creating_Python_OOT_with_gr-modtool

[2] https://wiki.gnuradio.org/index.php?title=Embedded_Python_Block

[3] https://www.geeksforgeeks.org/cross-correlation-analysis-in-python/

[4] https://numpy.org/doc/2.1/reference/generated/numpy.correlate.html

[5] https://github.com/numpy/numpy/blob/v2.1.0/numpy/_core/numeric.py#L712-L785