Project 15

Developing Two-Transmit Channel SDR with Beamforming Capabilities

Peyton Adkins, Gabriel Faro, Alexandra Calvo, Joel Brigida, Jason Kurth

February 6th, 2024

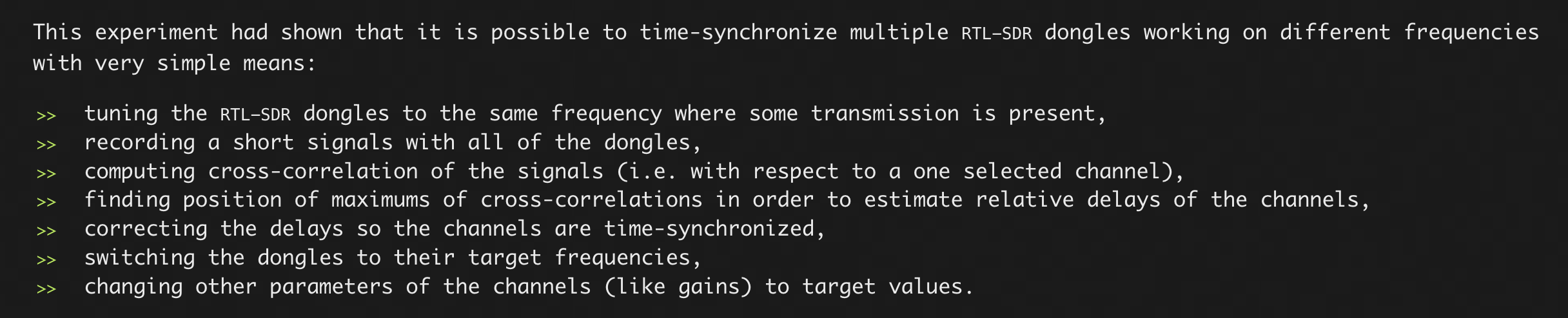
• • • Phase Coherence – Multi-RTL Research • • •

References

1. [[Multi-RTL GitHub Repo](https://github.com/ptrkrysik/multi-rtl)] {https://github.com/ptrkrysik/multi-rtl}
2. [[Multi-RTL's Author's Log Page](https://ptrkrysik.github.io/)] {https://ptrkrysik.github.io/}
3. [[RTL Coherence GitHub](https://github.com/tejeez/rtl_coherent?tab=readme-ov-file)] {https://github.com/tejeez/rtl\_coherent?tab=readme-ov-file}
4. [[Experimental RTL-SDR GitHub](https://github.com/keenerd/rtl-sdr)] {https://github.com/keenerd/rtl-sdr}
5. [[Delay Blocks in GNU Radio](https://www.gnuradio.org/doc/doxygen-3.7.5/classgr_1_1blocks_1_1delay.html)] {https://www.gnuradio.org/doc/doxygen-3.7.5/classgr\_1\_1blocks\_1\_1delay.html}
6. [[GNU Radio Delay Block Code](https://github.com/gnuradio/gnuradio/blob/main/gr-blocks/lib/delay_impl.cc)] {https://github.com/gnuradio/gnuradio/blob/main/gr-blocks/lib/delay\_impl.cc}

Summary of Methods for Achieving Time Sync

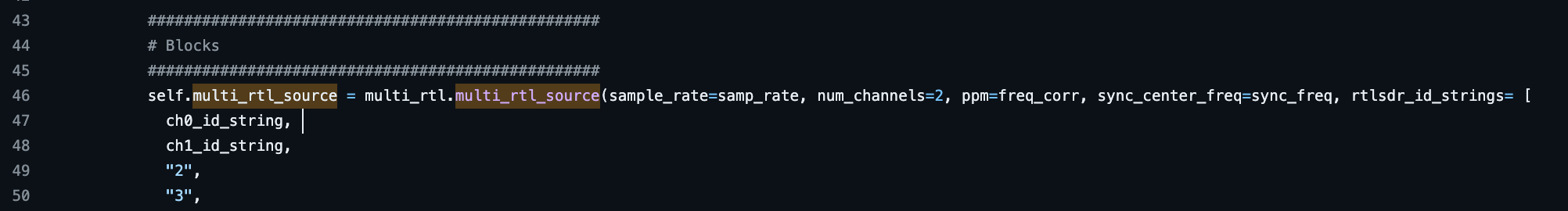
The basic principles of achieving time synchronization across multiple receive nodes with different frequencies according to the author of the multi-rtl project Piotr Krysik:

<https://ptrkrysik.github.io/>

In essence, there are two components to achieving multi-node coherence across multiple Pluto SDRs, phase sync and time sync. For phase sync, this can be achieved through an external clock dictating all Pluto SRDs to operate on its frequency. However, this does not guarantee time synchronization across multiple operations performed on the individual Pluto SDRs, let alone the collective whole of all possible antenna nodes.

In Krysik’s GNU Radio block, this is achieved within the context of building a GNU radio block itself. The main file for the initialization of the multi-rtl block is under the following link:  
<https://github.com/ptrkrysik/multi-rtl/blob/master/examples/mutlirtl_rx_to_cfile_2chan.py#L201>

The file primarily amounts to the configuration of the Multi-RTL GNU Radio block through GNU Radio’s various block configuration libraries. Options are defined and the block itself is initialized with a key class attribute called ‘multi\_rtl\_source’ in line 46:



https://github.com/ptrkrysik/multi-rtl

The file referencing the function that declares this attribute (of the same name) can be found within the file multi\_rtl\_source.py under the following link:  
<https://github.com/ptrkrysik/multi-rtl/blob/master/python/multi_rtl_source.py#L53>

It is here within this file that the bulk of time synchronization logic is nested.

Primary Logic for Time Sync

As previously stated, the basis for achieving time coherence and synchronization is through the following bullets of common interest:

1. Generate a reference signal that is used to compare node signals to
2. Computing a cross-correlation of each node’s signal with the reference signal
3. Using the resulting correlation to adjust each node’s time delay

The file found in the link [<https://github.com/ptrkrysik/multi-rtl/blob/master/python/multi_rtl_source.py#L53>] houses the multi-rtl code for achieving these steps. The final step, the implementation of an actual time delay for each node, is achieved within the context of having been constructed as a GNU Radio block.



https://github.com/ptrkrysik/multi-rtl

In short, the function compute\_and\_set\_delays() contains the main logic to achieve time sync but does so using GNU Radio’s library of block delay functionality. In the for loop starting at line 188, the delays stored for each channel (which in our case we could think of as Rx/Tx nodes) are applied to delay blocks assigned to each channel through the function .set\_dly(). More on this function and delay blocks can be found in GNU Radio’s documentation here:  
<https://www.gnuradio.org/doc/doxygen/classgr_1_1blocks_1_1delay.html>

The code for GNU Radio’s delay blocks can be sourced [here](https://github.com/gnuradio/gnuradio/blob/main/gr-blocks/lib/delay_impl.cc), showing the way in which inputs are either skipped or replaced with zeros (depending on the polarity of the input variable. For our own purposes, the ability to find a time delay value could work for an algorithm that asynchronously applies time delays to the input of each Rx node according to how off each one is to the reference signal.

Obtaining the Multi-RTL Block

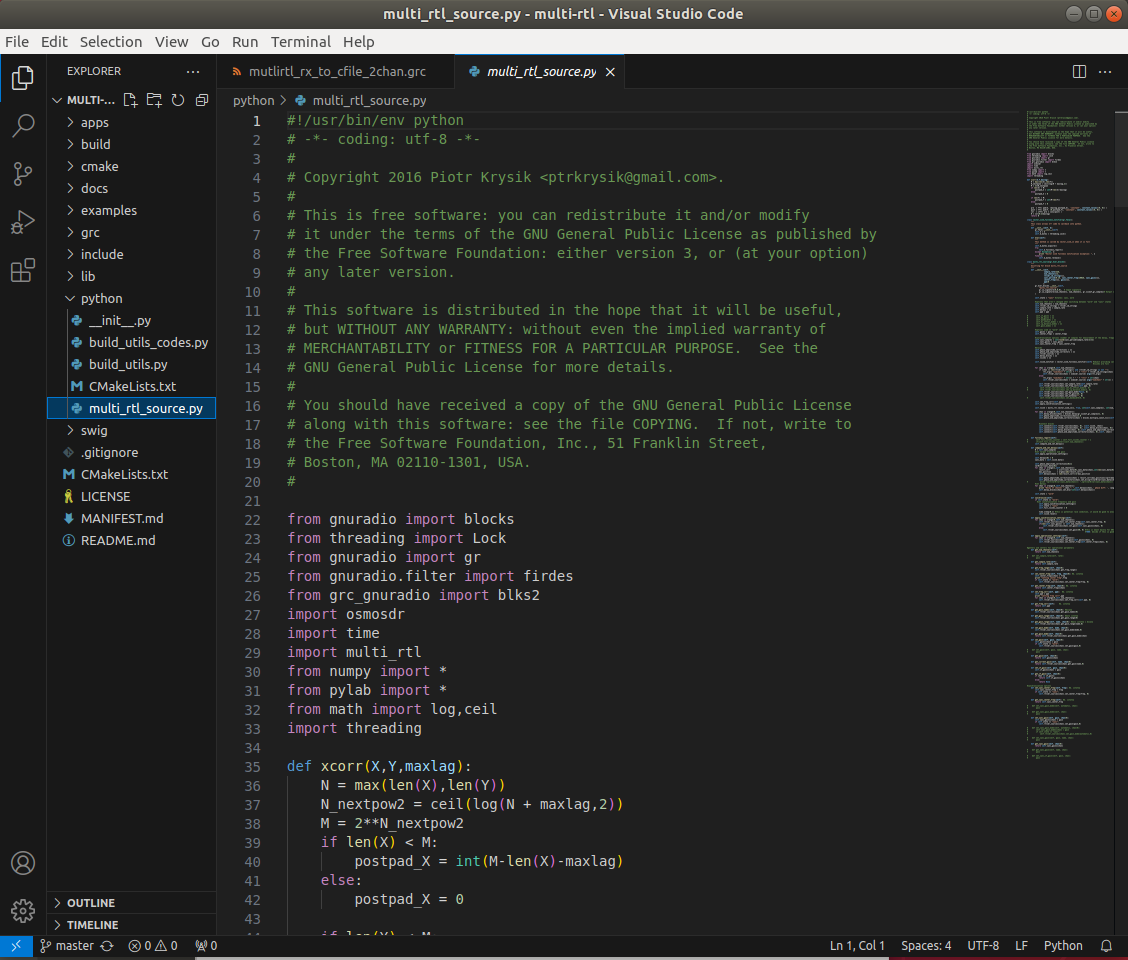
Downloading the Multi-RTL block is listed as a set of instructions at the bottom of the [GitHub page](https://github.com/ptrkrysik/multi-rtl). However, there are many unlisted specifications that inhibit the ability to download and test this block on any assumed system. Here are some of the specifications I have found in my research:

* The Multi-RTL block was built from GNU Radio version 3.7 and has historically not worked for any other version.
  + [[GitHub Issue Discussion](https://github.com/ptrkrysik/multi-rtl/issues/5)] {https://github.com/ptrkrysik/multi-rtl/issues/5}
* GNU Radio version 3.7 can only be accessed via Ubuntu 18 (Bionic Beaver)
  + [[Ubuntu VM ISO File](https://www.releases.ubuntu.com/bionic/)] {https://www.releases.ubuntu.com/bionic/}
  + [[PPA for GNU Radio 3.7](https://launchpad.net/~gnuradio/+archive/ubuntu/gnuradio-releases-3.7)] {https://launchpad.net/~gnuradio/+archive/ubuntu/gnuradio-releases-3.7}

To obtain this block, I followed the proceeding steps:

1. Download and install an Ubuntu 18.04.6 virtual machine [[Ubuntu VM ISO File](https://www.releases.ubuntu.com/bionic/)]
2. Set up the GNU Radio 3.7 repository [[PPA for GNU Radio 3.7](https://launchpad.net/~gnuradio/+archive/ubuntu/gnuradio-releases-3.7)]
3. Install GNU Radio using `sudo apt-get install gnuradio`, being sure to install all dependencies [[GNU Radio Linux install](https://wiki.gnuradio.org/index.php/InstallingGR)]
4. Follow the instructions in the Multi-RTL GitHub to install the block [[Multi-RTL GitHub Repo](https://github.com/ptrkrysik/multi-rtl)]





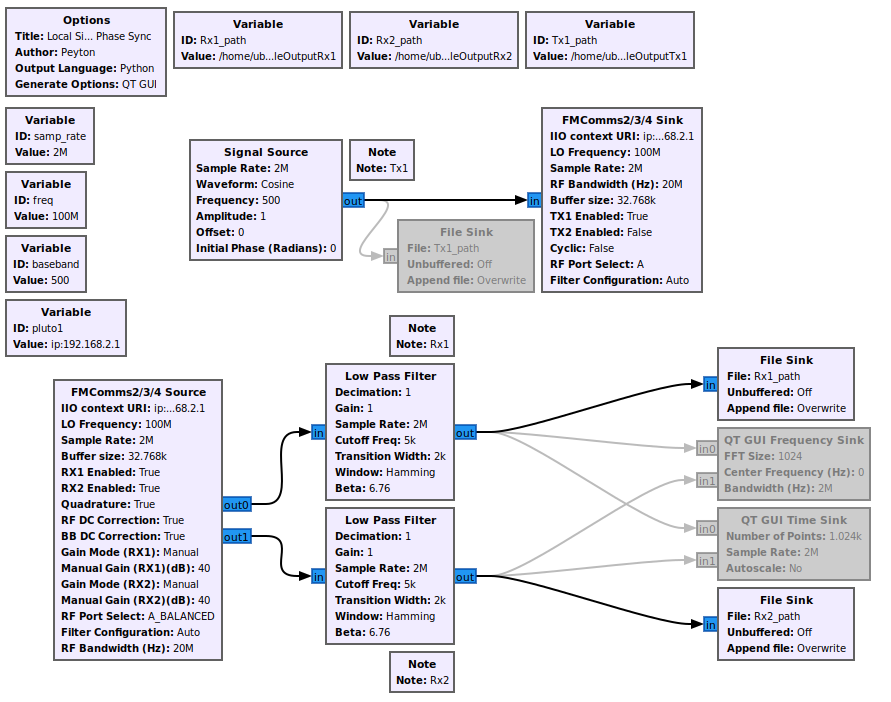
• • • Phase Coherence – Pythonic Approach • • •

Local Time Sync Testing

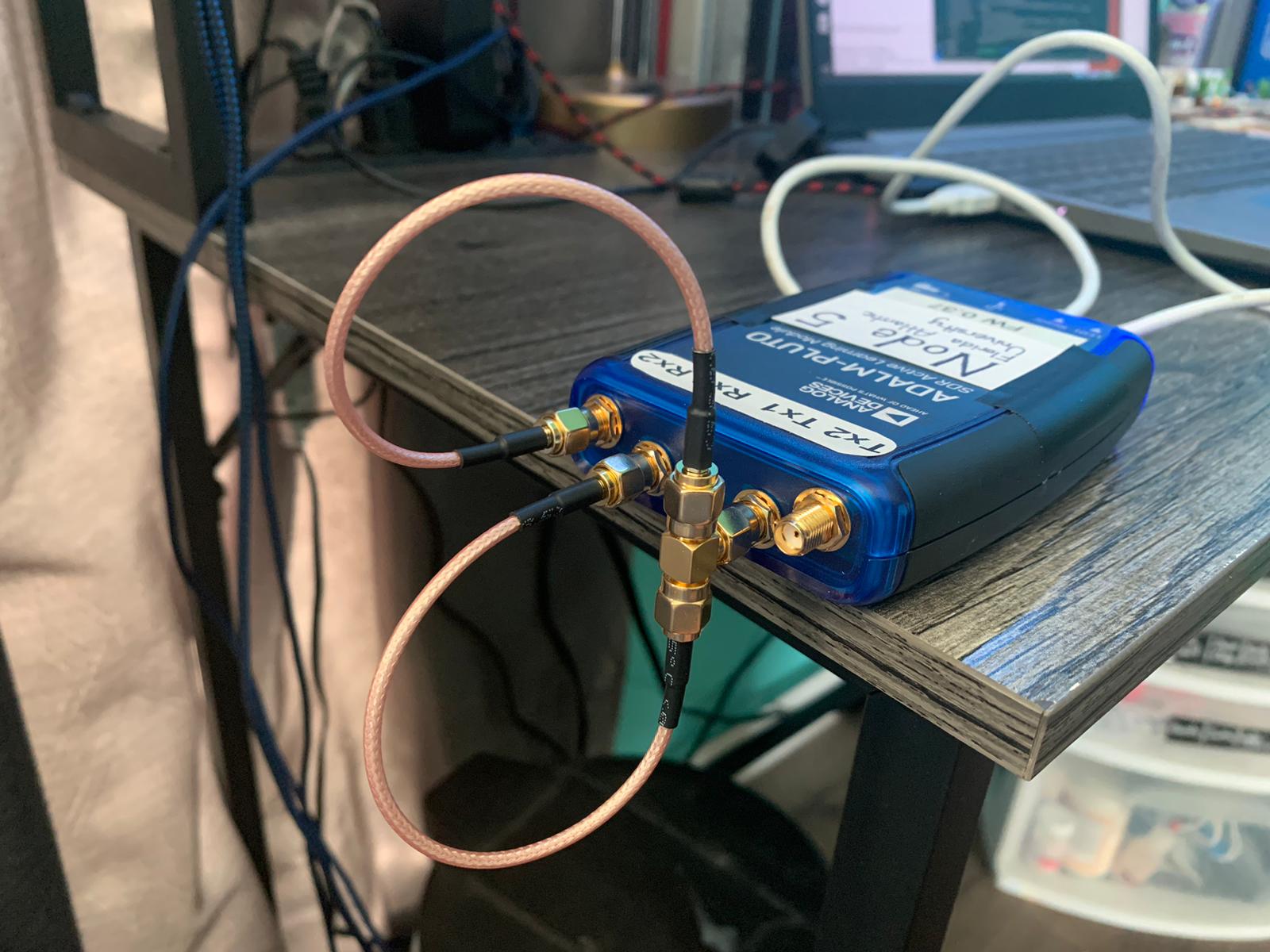
To fully realize the logic that goes into achieving phase sync across multiple Pluto SDRs, we must first conduct testing locally with a single Pluto with IQ streams captured to local files. To achieve this, we wrote a GNU Radio flowgraph that will configure an FMComms2/3/4 sink block to allow the Tx1 node to transmit a signal from a Signal Source block. Below are the specifications of this test:

* Sample Rate – Rx and Tx = 2 million per second
* LO Frequency - Rx and Tx = 100MHz
* Baseband – Tx Signal = 500Hz

I had conducted two variations of this test: one where the transmit signals are sourced over the air, and one test with direct wiring. In both cases, the flowgraph to generate and receive the samples is the same:



The above flowgraph captures the signal source transmitted from the first Tx node and the received signals of the Rx1 and Rx2 nodes. Below is an image of the setup with a single Pluto SDR in both test variations:

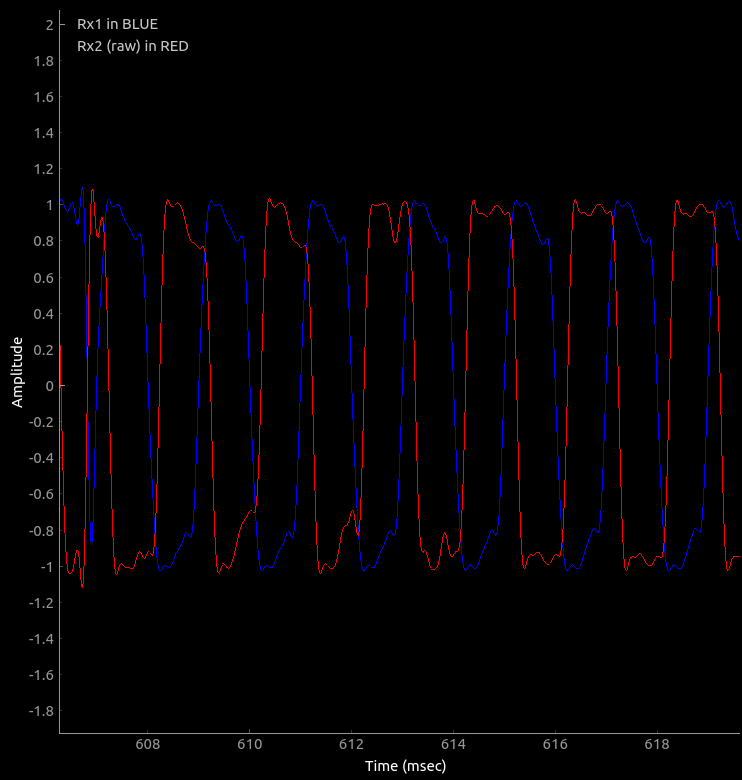
Over the air Direct Wire

For more close results, we look at the data sourced in files, one for each Rx node. A Python script takes this data and displays it in a graph to compare the captured samples more closely. The graph is configured to display the IQ samples relative to the time domain.

Over the air



Direct Wire

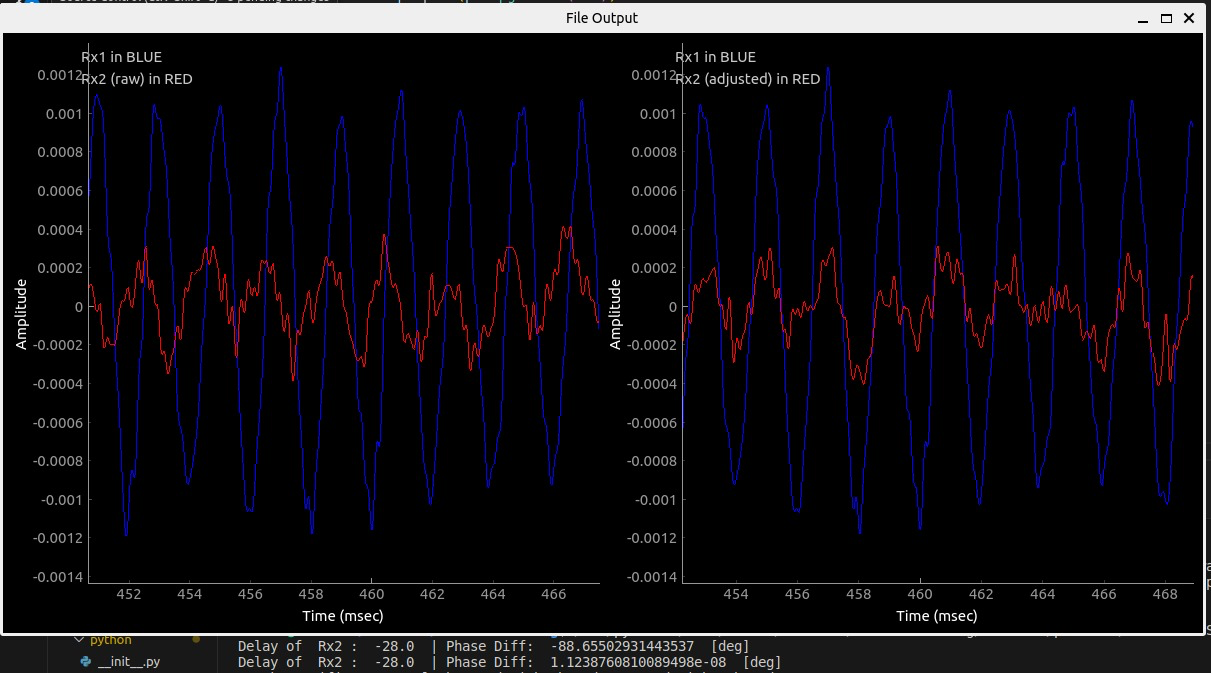


While not nearly as clear with data streamed over the air, both tests clearly show some level of misalignment between the two Rx nodes, even when sourced directly via wire. This misalignment of phase between the two Rx nodes of the same Pluto was striking. After all, Jon Kraft’s work involved the use of two Rx nodes of the same Pluto, and his logic performed quite functionality. However, it was this thought that led to a critical discovery; Jon Kraft inadvertently achieved phase sync between these two nodes.

Looking back on some of his work with the monopulse tracker algorithm (and other Pluto-centric algorithms like it), one key consistency stuck out: the phase calibration. In short, Kraft writes his code under the assumption that at the start of any algorithm for DOA or the like, the user must ‘calibrate’ the Pluto setup to reflect an actual situation. For example, if you have a transmit node directly facing the first Rx node (the reference node in his works), then you would change the phase calibration offset to the value of the phase returned from the calculation, essentially correcting the value to what it should be when the incoming signal is straight on to the linear array. [Here at 22:49](https://www.youtube.com/watch?v=2QXKuEYR4Bw) is where you can find Jon Kraft referencing this calibration value, but never fully explaining its significance.

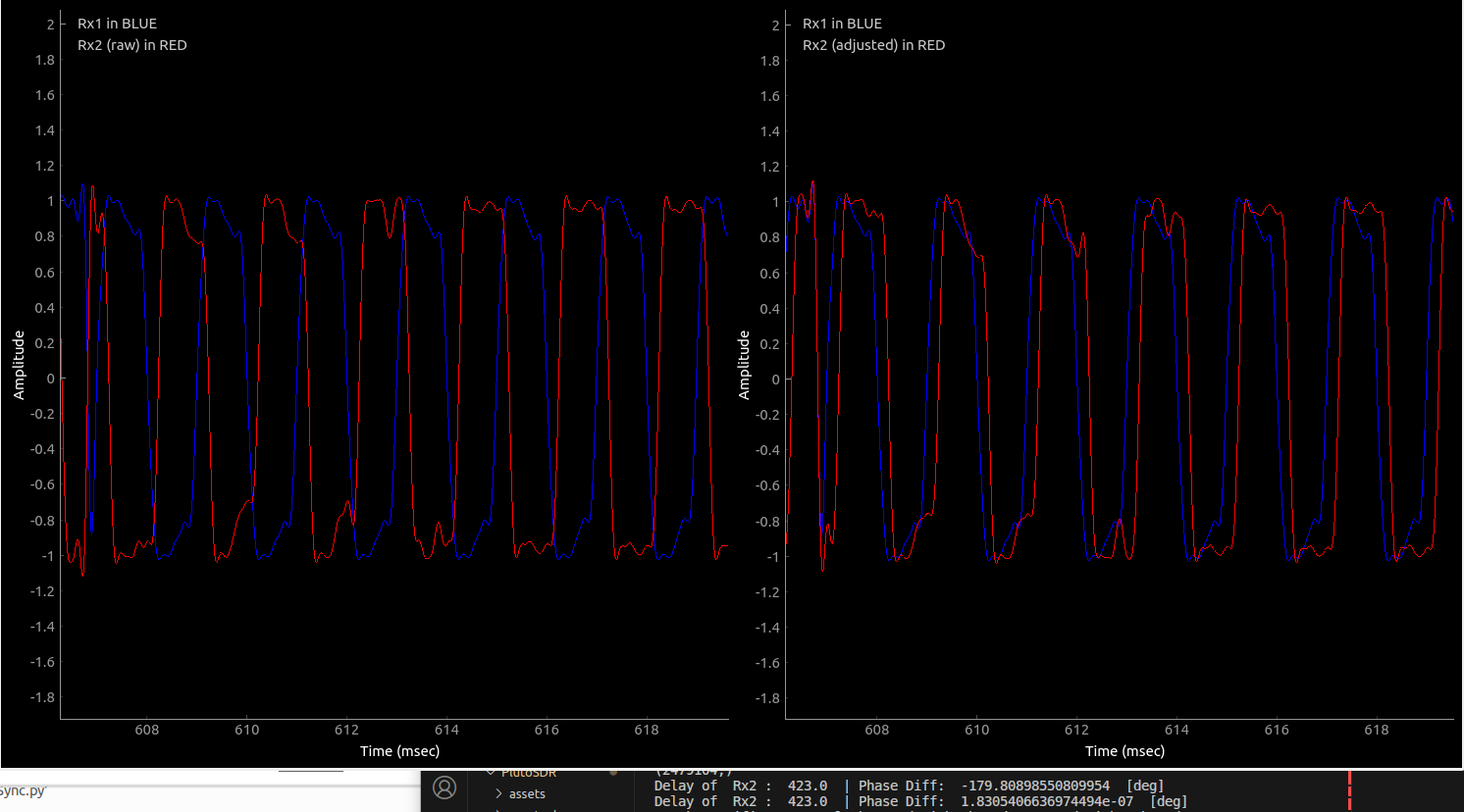
That’s when it hit me. Jon Kraft wasn’t simply calibrating the Plutos for an ambiguous purpose, **it was to account for the randomness of phases between the two Rx nodes**. Looking more into this, I found various small forums on the [Analog Devices Engineering Zone](https://ez.analog.com/) portal suggesting issues between the two Rx nodes on a single Pluto. Solutions for this problem were non-existent in a clean and clear format, let alone those for synchronizing multiple Pluto devices. This, however, was a critical breakthrough in research and allowed me to write a test script that achieved phase between the IQ samples of two file sinks in GNU Radio:

Over the air

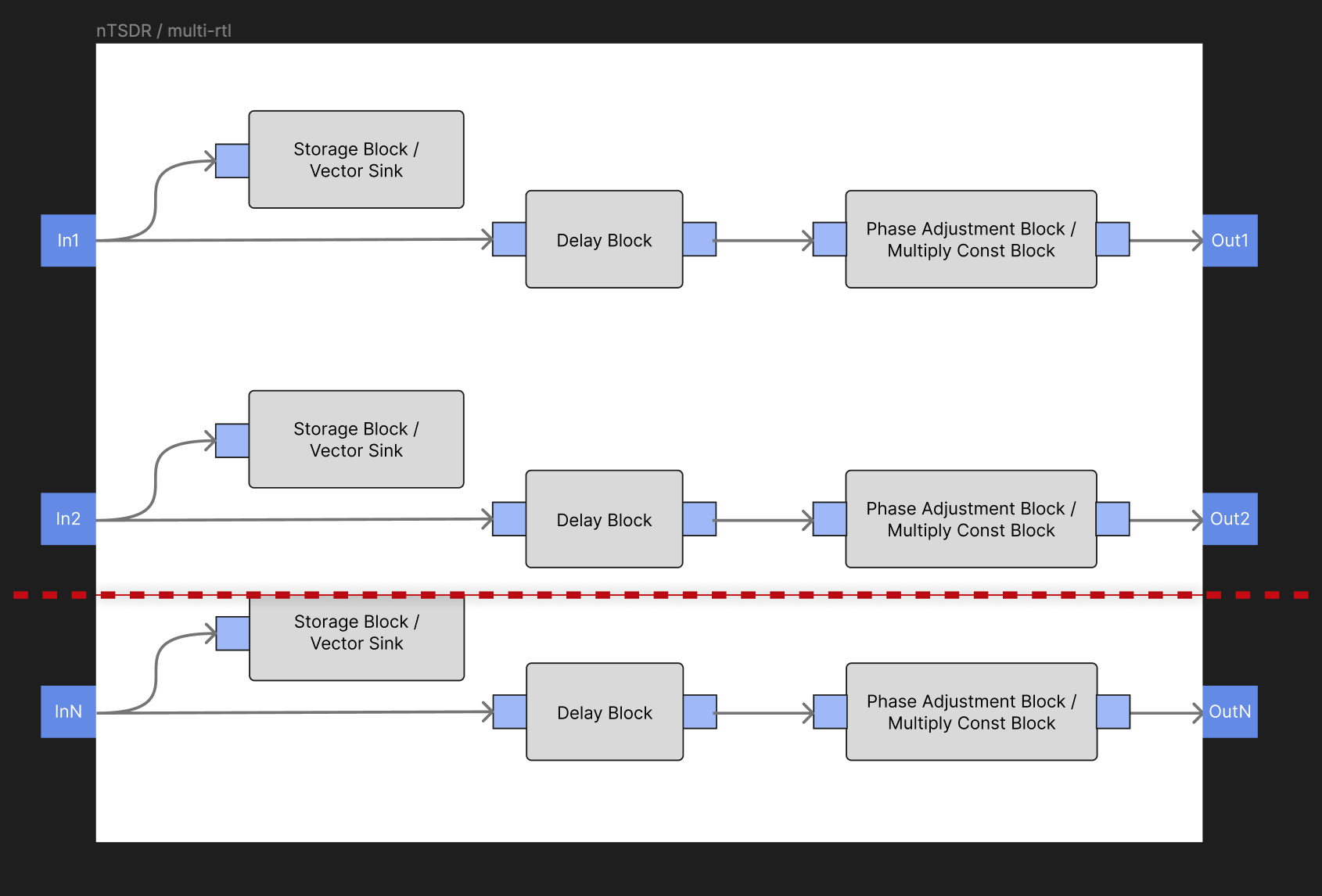


Graphically, it may be difficult to notice the difference, but as seen in the console below, a phase difference of ≈ -88.66° is initially found. Then, computations referenced from both Jon Kraft and Piotr Krysik’s works achieve a phase shift to then output a phase of ≈ 0°, proving phase alignment. Below is the same test with the same method, only with a wired setup:

Direct Wire



While an important benchmark for achieving phase sync, the utility of this script’s logic needs to be tested and considered. Some next steps that our group has taken include translating the code used to achieve phase sync locally to that of a GNU Radio block, but this process proved once again to be more complex. In direct translation, some aspects of the new environment must be considered. Firstly, let us look at the basic structure of the Multi-RTL block and exam the key features within it.



The original secret thought to achieve phase sync was the delay block from GNU Radio’s library. However, a closer look at the code for multi-RTL and its structure revealed a similar method implemented by Jon Kraft in the complex multiplication of a phase shift. The precise methods of this complex multiplication for each member are slightly different, thus leading to speculation as to the data structure of the IQ samples streamed within GNU Radio.

Kraft and Krysik’s Work in Tandem – Part I

While the previous tests for the validity of cross-correlation methods have appeared sound, a more direct approach would be to consider the implications of Kraft’s work and the cross-correlation methods proposed by Krysik’s solution. This is why we must recall the setup Kraft used in his work with the Pluto SDRs. Consider the procedure for calibrating the second Rx node in his single Pluto setup:

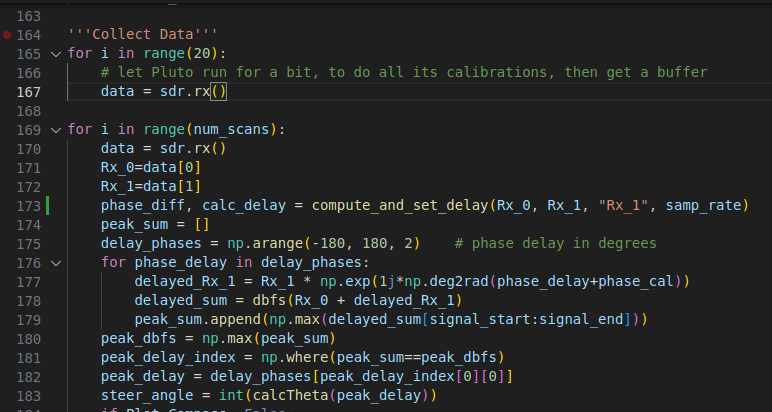
* A single Pluto SDR is configured to send and receive, having both receive antennae available to relay incoming IQ samples.
* The frequency of the transmitted node is 2.3GHz with an element spacing of 65mm.
* The transmitter is set to face the first Rx node perpendicularly to ensure a 0° angle of arrival and, thus, a 0° phase offset required for beamsteering.

Below is an image to illustrate the physical setup proposed by Kraft’s work:



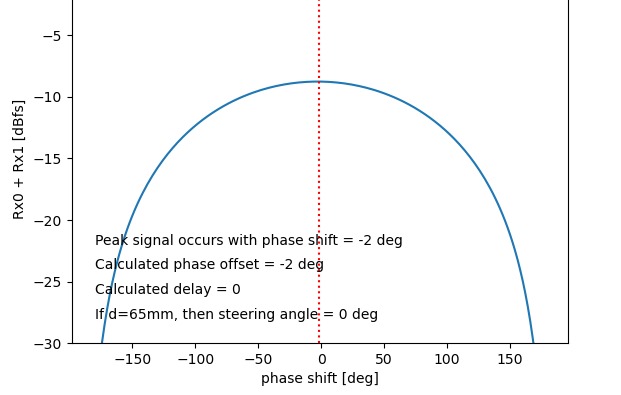
Kraft’s approach to dealing with the phase offsets that occur between the two Rx nodes is to mimic this setup above and find the peak signal through all possible phase degree shifts from -180° to 180°. In this manner, the estimated phase found for the proposed peak signal is the direct result of the internal phase difference of the Rx nodes, proven by the fact that when the transmitted signal arrives at the linear array at 0°, there should be no phase shift necessary (0° phase degree).

I had modified Kraft’s Pluto\_beamformer\_PlotPeaks\_youtube.py script to not only display the peak signal and the respective phase degree per usual, but to calculate the same phase degree via the cross-correlation found in Krysik’s Multi-RTL solution.



In addition to the function definitions necessary, I had called the cross-correlation calculation function in line 173 to return the phase degree and the delay value (which such a value proved to be confusing and generally unimportant for this test). In this manner, the phase degree would be calculated on the same data later used in the full scan to find the peak signal and the respective phase degree, allowing me to directly compare the results.

Among twelve different occurrences of the same test, I found the two values to be one in the same. Below is one of these many instances of success:



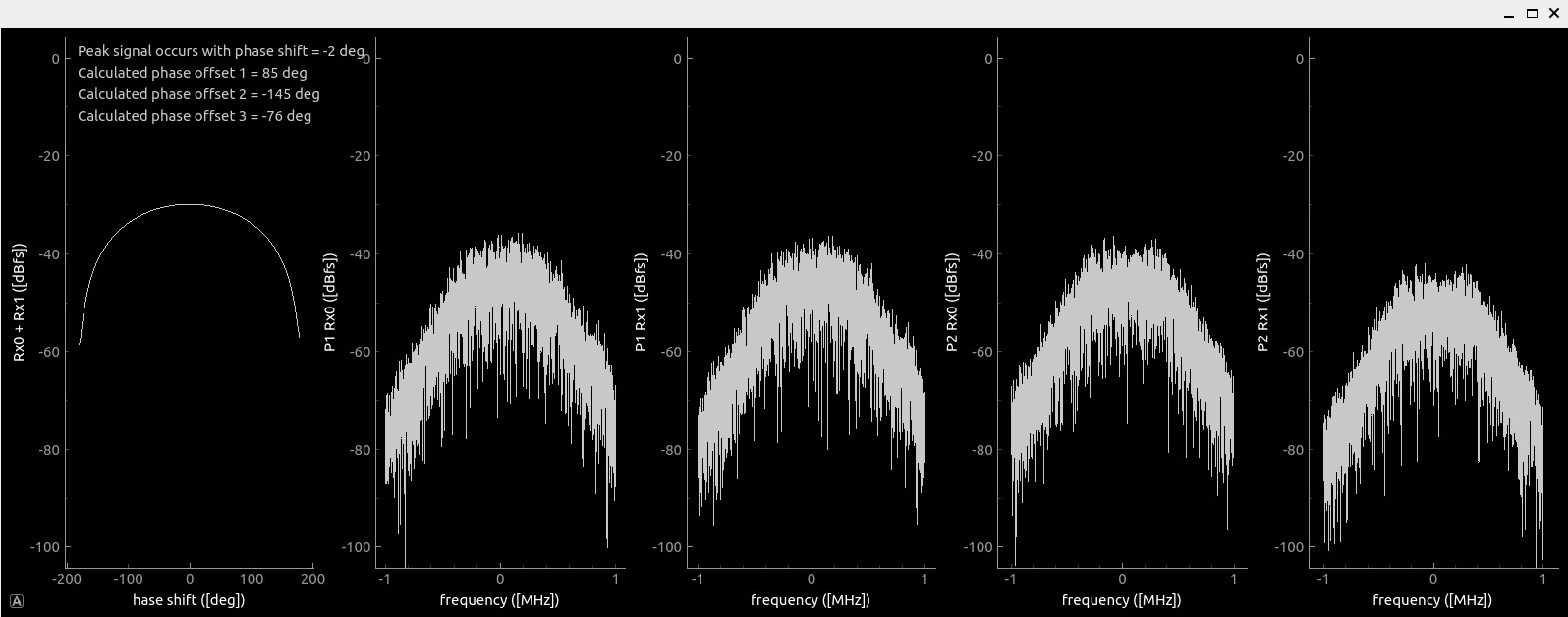
The upper-most number is Kraft’s original value sourced from finding the phase degree that yielded the strongest returned signal, but recall in this test, this should be 0° as the Tx node was placed directly perpendicular to the linear array. The value below this is the result of Krysik’s cross-correlation calculation, proving that such a method of finding this offset can be found with significantly less computational resources as well as open the possibility of using multiple Pluto SDRs.

Next steps include modifying the script once more to add functionality to two Pluto SDRs with the primary goal of achieving phase sync across the four Rx antennas.

Kraft and Krysik’s Work in Tandem – Part II

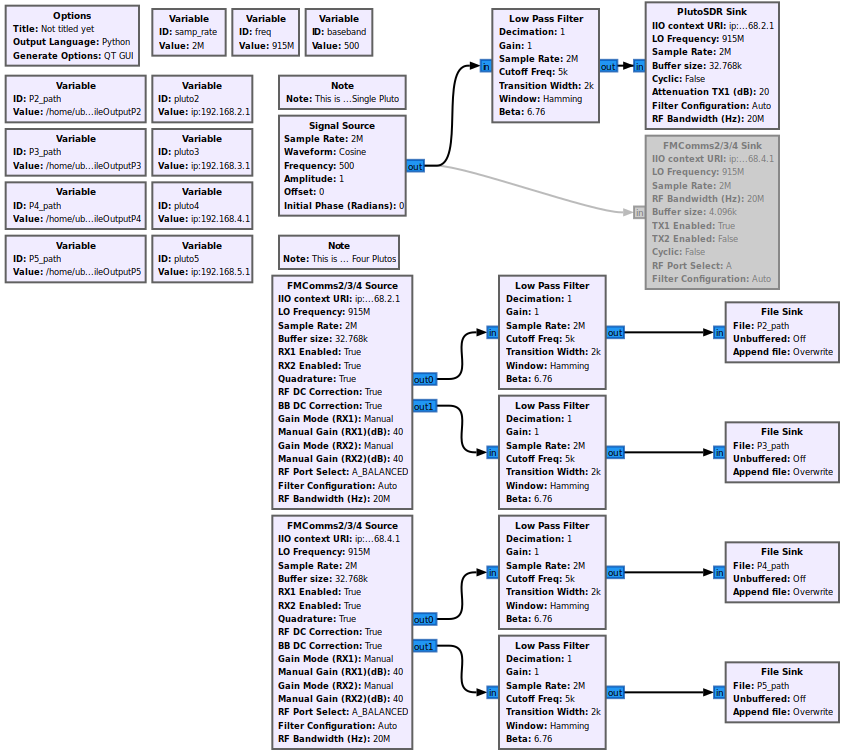
In the previous tests, the synchronization of two Pluto devices failed with a concerning result for how the second Pluto was receiving data, whether it be directly wired or over the air. The data from the Pluto device was wildly sporadic, causing the otherwise previously assumed cross-correlation between the nodes of the first Pluto to yield equally sporadic results. Below is a capture that shows the frequency displays of each Rx node. Note that the results in the top left corner for Rx3 and Rx4 were fluctuating wildly, which cannot be properly conveyed in this still image:

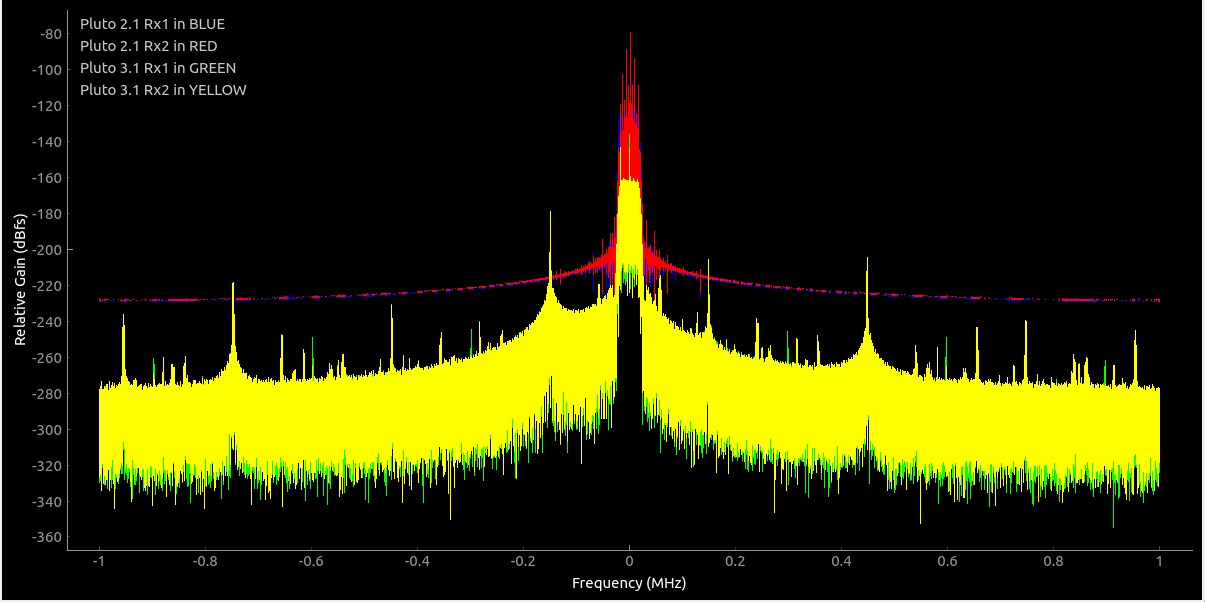
Direct Wire – Initial Test



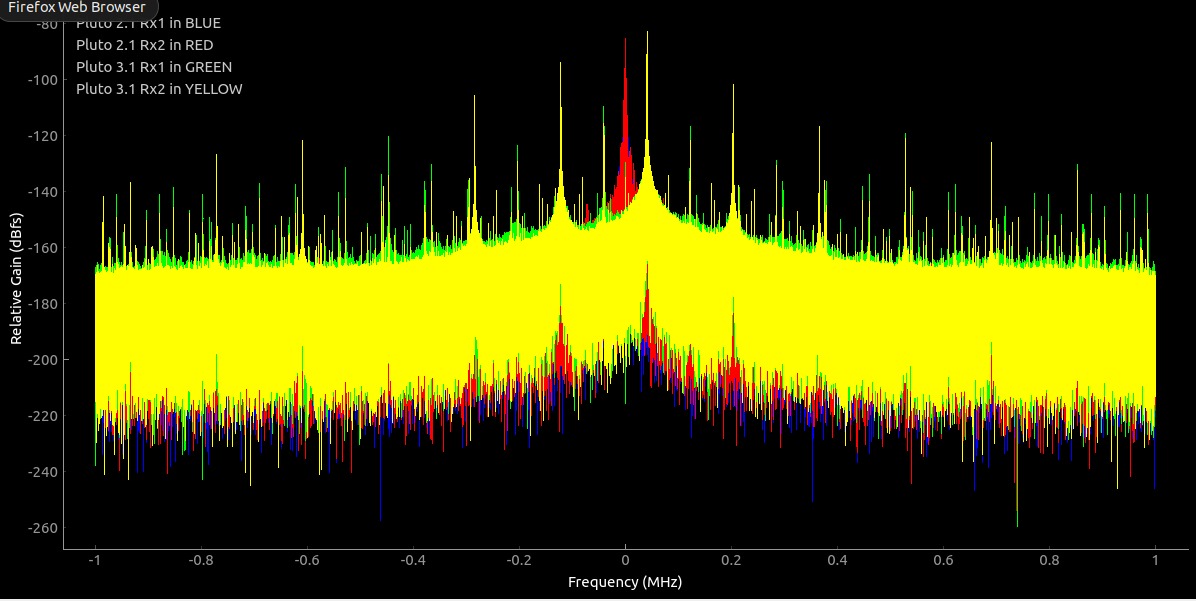
Note that only the second node (Rx2) had its phase shift adjusted and added to contribute to the shape of the beam. The other nodes’ offset values would cause a sporadic display that yielded no practical results.

A recent set of tests conducted by Joel B. and Peyton A. (myself) aimed to test the reason as to why the results of a second Pluto could be so inconsistent and seemingly random. The first idea related to an issue with USB ports. In the previous testing, each Pluto had been directly connected to a USB hub with one central port to connect into the local computer. In the proceeding test, the two Pluto SDRs were connected into two USB ports into the local computer. We begin using a simple GNU Radio flowgraph to store the data into a file sink to be displayed through a Python script. The flowgraph for this test can be seen below:

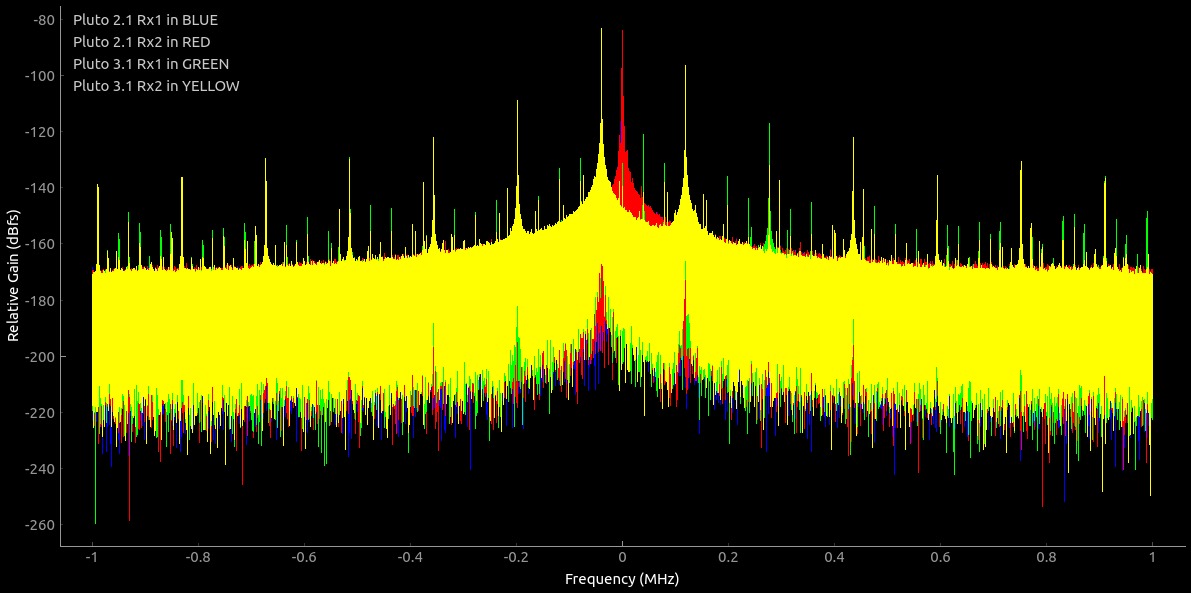
  
NOTE: The information in the top left is not dependable to the test; the green & yellow data correlated to one Pluto Rx pair and the red & yellow data to another. Below are the results:



Our flowgraph had contained lowpass filter blocks, so sensing that there may be an issue between the use of physical 915mHz filters, so we disabled them to try again. With no noticible results, we then tried setting also the Tx attenuation to 20dB:

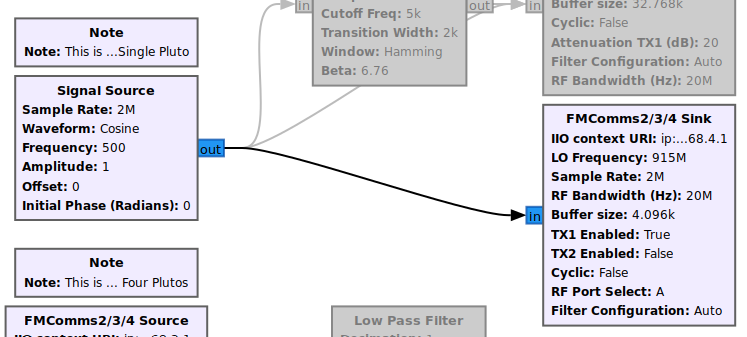


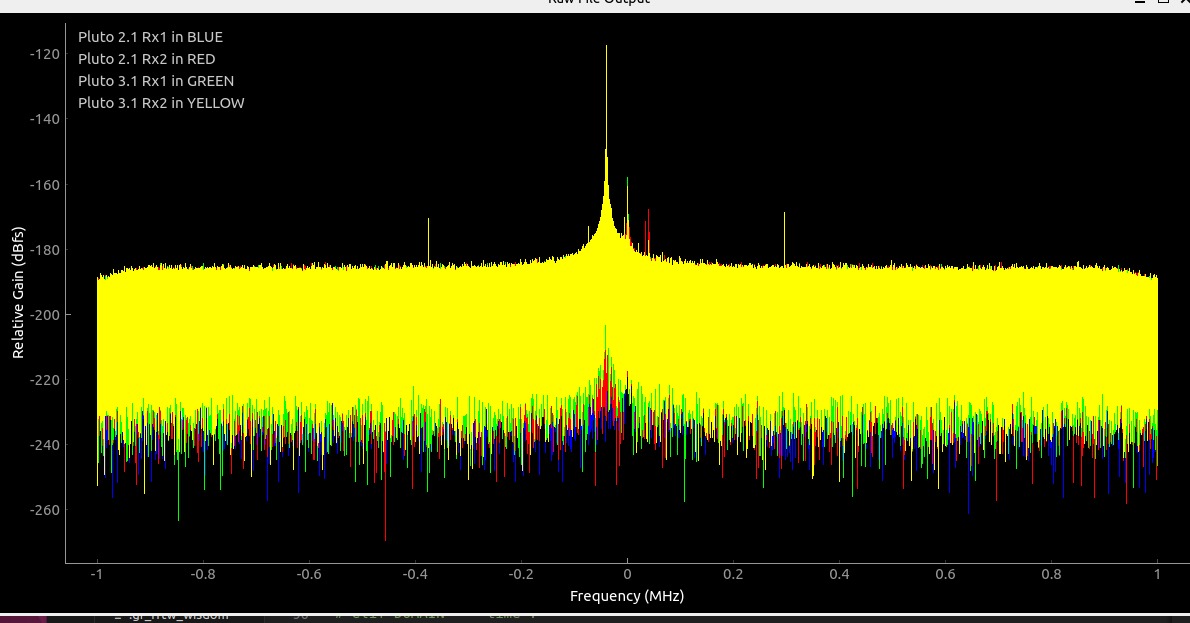
We observe noticeable variations in the frequency, plenty of noise spikes. We plan to observe the attenuation further, but a hypothesis came to mind regarding the Pluto SDRs themselves. In all previous testing, one Pluto had used its first Tx node for testing, which yielded promising results. To test to see if this was an issue between the specific devices, the Tx node was switched to the opposing Pluto, giving that responsibility over. Below are the results:



Surprisingly, the behavior of the received data did not change, apart from the graphical color differences due to the pairs being assigned to different roles in the code. This led us to believe that there was, indeed, some degree of issue regarding the introduction of another device as the first Pluto, the one given the Tx node responsibility and the one with the first Rx node set as the reference point, would yield consistent results.

Another aspect that we observed is our flowgraph used FMComms Source blocks for the Rx nodes, but a Pluto SDR Sink block for the Tx node. We switched this out with the respective FMComms Sink block with the necessary configurations. The buffer size we had ensured stayed consistent between the Pythonic script and the GNU Radio flowgraph:



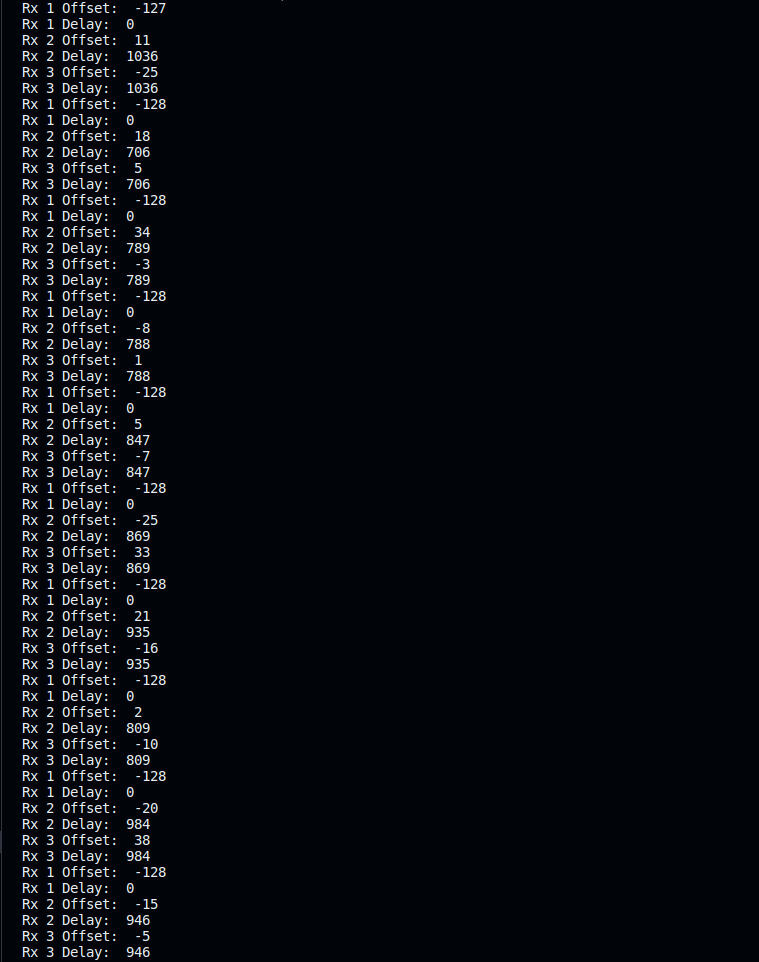


We had made further minor tests changing the Tx attenuation, with some differences between each. However, the takeaway from these tests was the need to further examine how the data is received and when for each Pluto. There may very well be an issue with the way in which processes are scheduled in the OS of a given system, let alone the physical differences in time of arrival concerning the Pluto SDR’s trigger periods.

Kraft and Krysik’s Work in Tandem – Part III

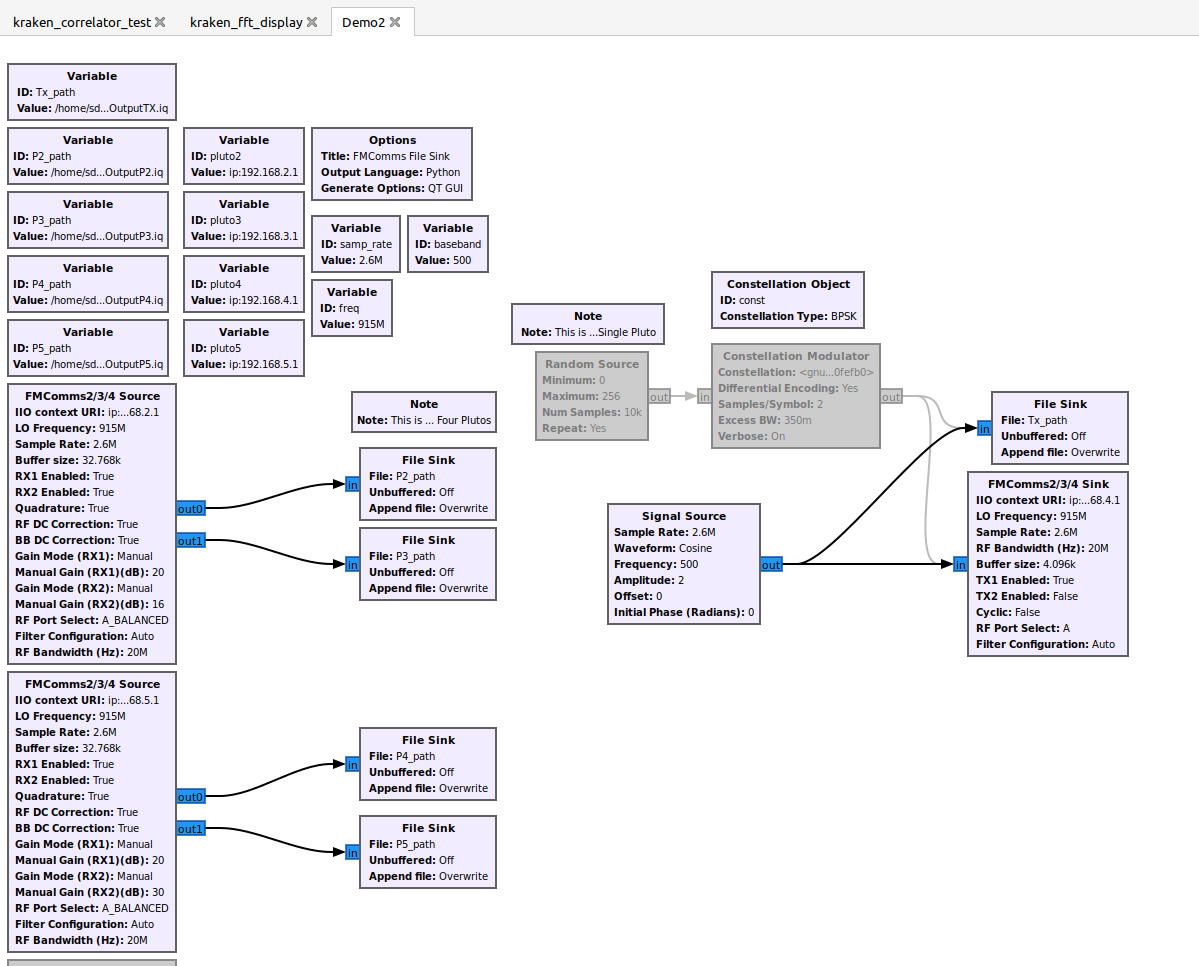
After much testing of prior knowledge about the way in which Pluto devices are registered in a CPU, our group took to examine Krysik’s values returned from his function that served as the basis for Multi-RTL. To recap, the Multi-RTL block utilized a phase offset between the received data on each dongle as well as a known ‘delay’ value. The precise usage of this delay value was an enigma at first, only known to be used as the parameter for GNU Radio’s Delay Block (more information about this block within this document).

A test was conducted to observe the returned ‘delay’ value and the returned phase offset from Krysik’s modified function used to find such values in Multi-RTL. Our test used three Pluto devices, one for the sole role of a single Tx node wired directly into each receiving node across the two other Pluto devices, making four Rx nodes total.



This is the terminal for which we calculated the ‘delay’ and the phase offset for each Rx node among two Pluto SDRs. As you can see, there is no ‘delay’ for the Pluto1 Rx1 when compared to the reference node (P1Rx0), which results in a consistent phase offset calculation each time data is pulled. However, a CPU scheduler/USB driven delay is found at complete random between the first and second Pluto (which we will now call the 'trigger delay'). Because this trigger delay is random each time new Rx data is pulled, this causes the phase offsets for the respective Pluto's Rx nodes to be equally random. This results in a tricky predicament in finding a consistent, reliable phase offset for the Rx nodes of any other Pluto besides the reference node's Pluto (Pluto 1).

To correct this trigger delay, we looked back at the found value from Krysik’s work to observe how it was implemented for correction. As a direct parameter for a GNU Radio exclusive block written in C++, a Python implementation was created a few weeks prior, but with no concrete proof of functionality. However, a new test was conducted using stored files from a GNU Radio flowgraph, which is displayed below:





Above is the file sink display for two Pluto SDRs (with 3rd at transmit) with a direct wire setup. The details for the transmit signal and baseband are provided in the GNU Flowgraph image. Here, we see the TX data in cyan, which may appear to be different in amplitude from the signals received, but this is a result of double splitting of the TX data with physical splitters. The amplitude of the TX data is 2amps whereas the received data is around 0.5amps (due to a double two-way split).

We observe in the terminal that the trigger delays for each RX node (not including P1Rx0, the reference node) have a discrete value which, upon processing for correction, return 0.0 after an additional check for this trigger delay. Additionally, the phase offsets are found, and a phase shift is implemented for each Rx node, which is most visible in the [CORRECTED] graph. However, the graph, we theorize, may not accurately show the trigger delay's correction as the graph places the data on the initial array index for each Rx data array.

To fully prove full phase sync (which is to say the trigger delay and the phase offsets are found and corrected), we utilize Kraft’s Plot Peaks algorithm once again to show how each Rx node can be adjusted to work in tandem. Below, we see the complete direct-wire setup for a sync test across 6 Rx nodes, which is a total of 4 Pluto devices (1 Pluto – 1 Tx, 3 Pluto – 2 Rx).



This Plot Peaks algorithm is modified to use PyQT graph, but the main differences involve the calculations for finding each Rx node’s trigger delays and phase offsets as well as correcting them each time new Rx data is pulled through a continuous loop. As shown, the expected phase delay resulting in the peak summated signal is, indeed, found at 0º, which to our genuine surprise, yielded phase sync across multiple Pluto devices.

However, the major flaw in this logic lies in the fact that the trigger delay is random each time new Rx data is pulled, which can be corrected, but this results in equally random phase offsets. This makes the logic used for a single Pluto not applicable as, having no trigger delay itself, it is unable to remain consistent in operation (“operation” referring to direction-finding, steering, etc). Each new pull of Rx data results in a new random trigger delay, and consequently random phase offsets, resulting in the above algorithm to work only when the Tx data is incoming at steering angle 0º (which is a phase delay of 0º).

• • • Phase Coherence – GNU Radio Blocks • • •

1. [[General GNU Radio Block Structure](https://wiki.gnuradio.org/index.php/BlocksCodingGuide)] {https://wiki.gnuradio.org/index.php/BlocksCodingGuide}
2. [[Creating Python OOT with gr-modtool](https://wiki.gnuradio.org/index.php?title=Creating_Python_OOT_with_gr-modtool)] {https://wiki.gnuradio.org/index.php?title=Creating\_Python\_OOT\_with\_gr-modtool}
3. [[Imbedded Python Block Reference](https://wiki.gnuradio.org/index.php?title=Creating_Your_First_Block)] {https://wiki.gnuradio.org/index.php?title=Creating\_Your\_First\_Block}
4. [[GNU Radio Delay Block API Reference](https://www.gnuradio.org/doc/doxygen-3.7.5/classgr_1_1blocks_1_1delay.html)] {https://www.gnuradio.org/doc/doxygen-3.7.5/classgr\_1\_1blocks\_1\_1delay.html}
5. [[GNU Radio Delay Block Implementation](https://github.com/gnuradio/gnuradio/blob/7d61746e27778c56eb8a805ad940c89ff265e313/gr-blocks/lib/delay_impl.cc#L27)] {https://github.com/gnuradio/gnuradio/blob/7d61746e27778c56eb8a805ad940c89ff265e313/gr-blocks/lib/delay\_impl.cc#L27}
6. [[GNU Radio PlutoSDR Source Reference](https://wiki.gnuradio.org/index.php/PlutoSDR_Source)] {https://wiki.gnuradio.org/index.php/PlutoSDR\_Source}
7. [[GNU Radio’s Block-Building Comprehensive Video Tutorial – Pythonic Method](https://youtu.be/CnJObODsx0I?si=VmzfgwWea_bgkDjW)] [1:28:12] {https://youtu.be/CnJObODsx0I?si=VmzfgwWea\_bgkDjW}
8. [[Simple GNU Radio Block Creation Video Tutorial – gr-modtool C++](https://youtu.be/0hJgCRnHTqM?si=CPZr820Cl2J1M_kw)] [15:58] {https://youtu.be/0hJgCRnHTqM?si=CPZr820Cl2J1M\_kw}

Because of the way the Multi-RTL project utilizes GNU Radio delay blocks to achieve time sync, there begins a question as to how to proceed moving forward. There are a few possible methods and tests that could help the development of achieving time sync moving forward. But to understand any possible method, we must first understand how GNU Radio blocks are created and run.

Delay Blocks in GNU Radio

Among the listed sources above, delay blocks are GNU Radio blocks that skip or rewrite a certain number of inputs to produce a ‘delay’ to those inputs. The way the Multi-RTL block functions is by creating and connecting delay blocks to each input (which in our case would be Rx/Tx nodes) and, once having found the appropriate delay quantities for each, sets those delay blocks with those delay values. This utilization of delay blocks is what makes Multi-RTL a valuable resource as the implementation of a delay is what we as a group have found to be difficult.

**METHOD 1: Abstract from the GNU Radio delay blocks in a Python script.**

If there is a way to reference and use GNU Radio’s delay blocks independently of a GNU Radio block context (i.e. using these delay blocks without the confines of developing our own GNU Radio block), we would be able to more easily create a solution for time sync. However, a major roadblock I had found in testing and research is the fact that delay blocks (or any other GNU Radio block) must be connected to other GNU Radio blocks to stream information in and out. This effectively means for our project is that we must create a block that holds in class inputs and class outputs that can be connected to delay blocks to allow for data to pass in and out. Otherwise, we can create a delay block reference and set its delay, but as a GNU Radio block, it is unable to input or output data without GNU Radio’s connect() methods.



https://github.com/ptrkrysik/multi-rtl

Above is the area of Multi-RTL's code that creates and connects delay blocks to itself, defined as a class (which is a GNU Radio block). To achieve this in the same way, we would also have to create a GNU Radio block to connect the delay to, at which point we would be developing a GNU Radio block entirely.

**METHOD 2: Create our own delay function in a Python script.**

Since the direct reference to GNU Radio’s delay blocks require more work than anticipated, another option would be to recreate the logic found within these delay blocks for use in our project. In other words, we can reference the [source code](https://github.com/gnuradio/gnuradio/blob/7d61746e27778c56eb8a805ad940c89ff265e313/gr-blocks/lib/delay_impl.cc#L27) for GNU Radio’s delay blocks and manufacture our own way of finding, setting, and implementing time delays.



https://github.com/gnuradio/gnuradio/blob/main/gr-blocks/lib/delay\_impl.cc

Recreating this logic would allow us to implement unique time delays for each input node, achieving time sync. As reiterated, the idea of the time delay is to take an array of values as an input and, in one of two methods, trim or pad the first d\_delta of inputs.

Also, we may be able to contain this snippet of code into an embedded python block within GNU Radio. In short, this allows for the insertion of Pythonic code as a block inside GNU Radio, which would greatly reduce the development time if we wished to test and run code within GNU Radio while still utilizing time sync.

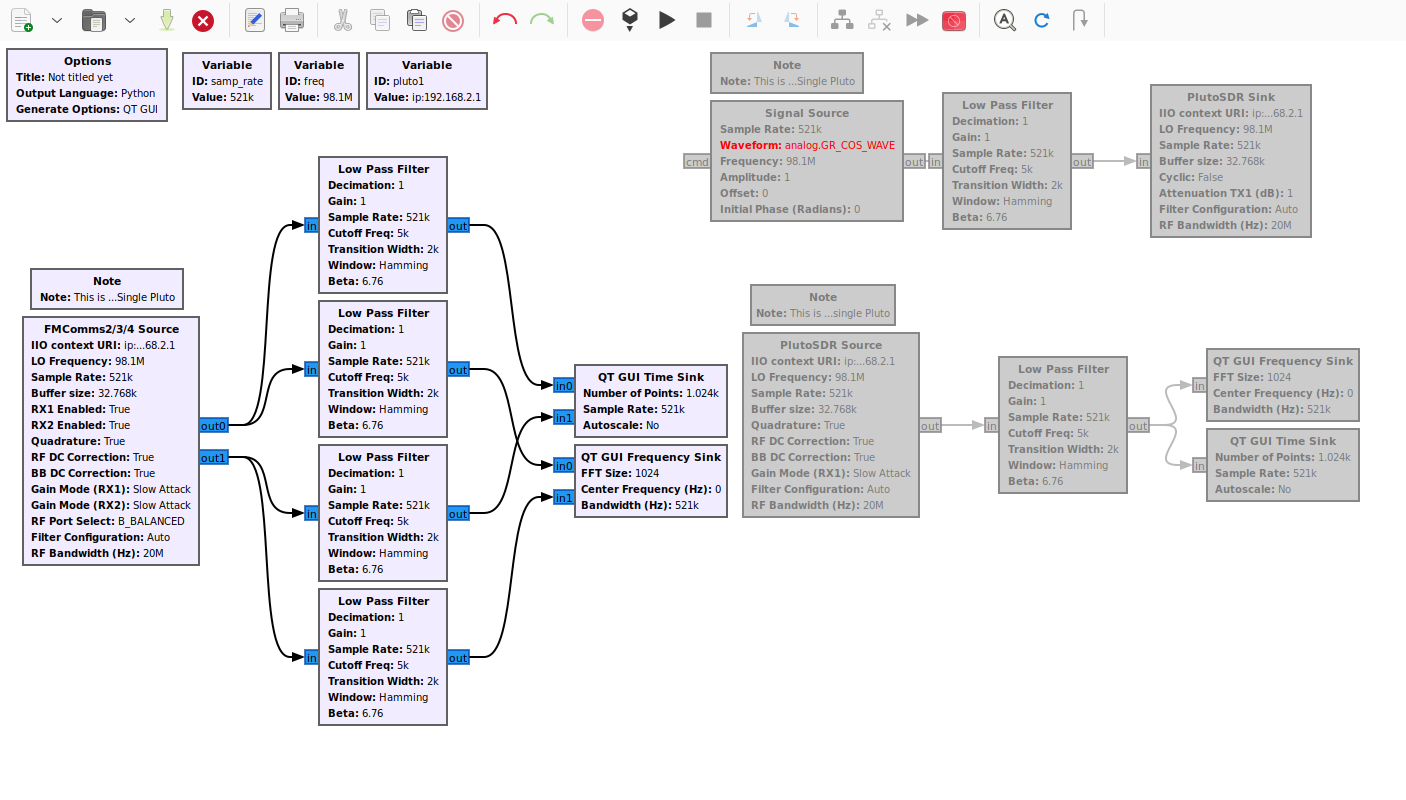
**METHOD 3: Use existing blocks to manually adjust delays.**

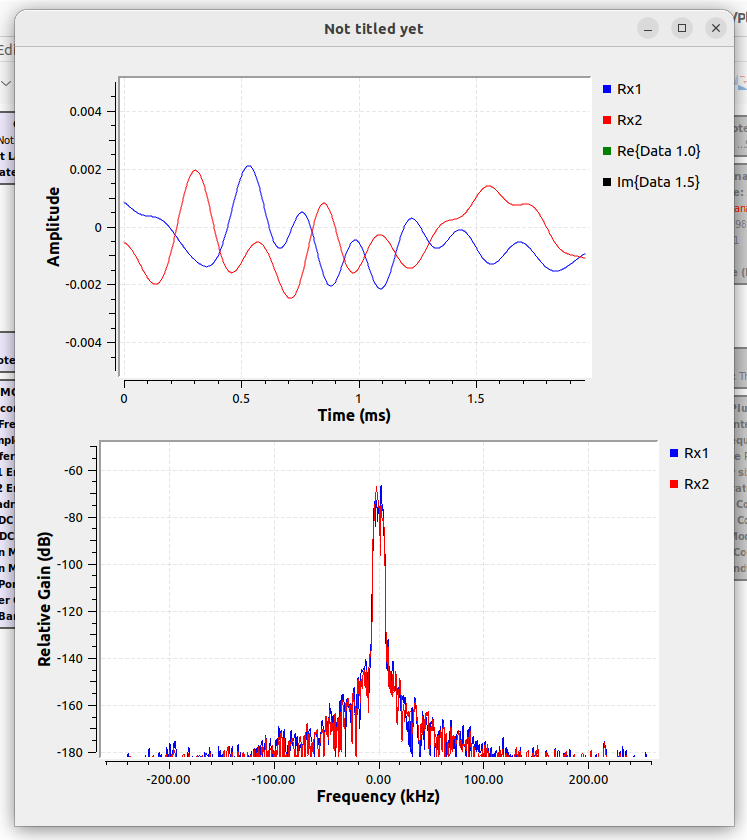
The blocks that allow the Multi-RTL block to achieve phase sync lie in its architecture. Unlike standard block variations, the Multi-RTL block is a hierarchical block, meaning it itself is built off other block components rather than existing on its own.

While it is possible to construct an heir block using the gr\_modtool the development way, there is a way to combine blocks in a GNU Radio flowgraph into an heir block for ease of use. This method may prove beneficial if we can also utilize GUI sliders/knobs to control delay blocks placed in front of desired node points.

Testing GNU Radio’s Capabilities

GNU Radio is a powerful tool for signal processing, so as we investigate software solutions for achieving phase sync, we need to exam other areas of GNU Radio that will affect our project. One such area is the modification of our Pluto SDRs in that we can use two Rx and Tx nodes. The default Pluto Sink and Pluto Source blocks do not support more than the default node ports, but there does exist a block called the FMComms2/3/4 block. Below is a test conducted with a single Pluto SDR with this block, output into both time and frequency domains [ignore the green and black marker as this is a GUI bug]:





There seems to be a slight difference in frequency between the two nodes, despite being sourced from the same Pluto SDR. I had written a script to extract IQ samples from both nodes to more closely exam their differences. Nonetheless, frequency lock is an issue we have since solved (marginally) across multiple Pluto SDRs. Below is a graph displaying the slight differences in frequency between the two Rx nodes on the same Pluto device:

