Time and Frequency Corrections in a Distributed Network Using GNURadio

Sam Whiting SAM@WHITINGS.ORG
Electrical and Computer Engineering Department, Utah State University, 4120 Old Main Hill, Logan, UT 84322

Dana Sorensen DANA.R.SORENSEN@GMAIL.COM
Electrical and Computer Engineering Department, Utah State University, 4120 Old Main Hill, Logan, UT 84322

Todd K. Moon, and Jacob H. Gunther {TODD.MOON, JAKE.GUNTHER}@USU.EDU
Electrical and Computer Engineering Department, Utah State University, 4120 Old Main Hill, Logan, UT 84322

Abstract

When using low-cost analog to digital converters (ADCs), synchronization between multiple ADCs is often difficult to achieve but is desirable for applications such as direction-of-arrival or time-of-arrival processing. A lack of synchronization between multiple ADCs can result in three offsets that are introduced when the sampling begins: a sample timing offset, a frequency offset, and a phase offset. In this paper, an adaptive method for correcting these offsets using software feedback loops is presented. The system is implemented in GNURadio with low-cost RTL-SDR receivers as a proof-of-concept.

1. Introduction

Synchronization between receivers in a distributed network is often required to perform analysis such as direction-of-arrival or time-of-arrival processing. With a clock shared between ADCs, synchronization can and has been obtained in previous studies (Krysik, 2016), (RTL2832u, 2014). Even in this case, the software system and operating system may deliver sampled packets at staggered times, so there are still synchronization issues. More generally, tying clocks together may be impractical (for example, with widely separated ADCs), so hardware methods may be employed such as atomic clocks or GPS-disciplined oscillators. In many cases, this hardware solution is impractical and a software solution is desired. This software-based coherency fits into the general movement of software defined radio by requiring less hardware in the RF front end.

Establishing synchronization between low cost ADCs driven by different clocks is complicated beyond determining initial timing delays, because the clocks in different ADCs experience drift. For inexpensive ADCs, sample drift can be substantial, as illustrated in figure 1 which depicts the sample drift between two RTL-SDR receivers with oscillators rated at 1-2 ppm drift, sampling at 2.048 MHz, where the sample drift was measured using cross correlation.

To produce this data, two RTL-SDR dongles were tuned to a local FM radio station and their streams of samples were cross correlated using a 16384 point FFT. This cross correlation produced a peak, located at an index that represents how many samples apart the two sets of samples are. This value drifts over time, as can be seen in figure 1 due to the clock difference in the ADC oscillators.

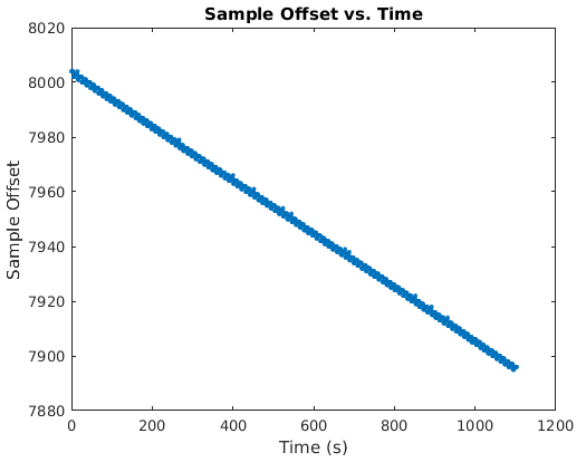

Figure 1. Sample Drift for two ADCs with independent clocks

This paper discusses how to treat four different aspects of synchronization between different ADCs. The first is bulk delay offset, resulting from differences in when the sampling actually starts. The second is fine clock offset and clock timing drift. The third and fourth are frequency and phase offsets. Similar projects have tried to correct for synchronization problems in software, and have had some success with post-processing methods (Junming Wei). The

Proceedings of the 7^{th} GNU Radio Conference, Copyright 2017 by the author(s).

main objective of this paper, however, is to present a real-time method for making these corrections for two ADCs with no common clock or other hard connection.

Many of the GNURadio blocks used in this paper are from a custom out-of-tree (OOT) module and can be found at https://github.com/samwhiting/gnuradio-doa .

2. Bulk Delays

Even in systems with two ADCs sharing a clock, sampling does not typically start at the same time, resulting in large delays. This initial "bulk delay" can be detected using cross correlation to find a rough alignment (within a sample) of the two signals. Detecting this bulk timing offset is accomplished with the "sample offset" block shown in figure 2. In figure 2, the block Sample Offset computes a cross correlation between the two signals. Given the potential uncertainty in sample conversion start time, this is a long cross correlation (our system is pictured using 262,144 points in the FFT). Since computing long cross correlations is expensive, this is done only for a fixed number of iterations, defaulting to 10 in the Sample Offset block, but resettable using a message if it is desired to re-initiate the bulk delay estimation. The output value is the median of the peak values from the cross correlations. The delay determined by the Sample Offset block is applied to a Delay block, applied to whichever signal is ahead in time, as shown in figure 3.

The bulk delay determined by this cross correlation may be considered to be the y-intercept of the sample offset plot in figure 1, with a resolution of one sample.

3. Clock Timing Drift

Over time, the clock frequency difference in the receivers will change the sample offset, as shown by the slope in the sample offset in figure 1. In order to adapt to these changes, a feedback loop is used.

A cross correlator (defaulting to 8192 width) is used to estimate offsets after the bulk delay correction. A quadratic function is fit around the peak of the correlation function from which the delay is measured to subsample resolution. This is called the fractional delay, and is denoted by mu (μ). This is performed by the blocks shown in figure 4. The way that μ changes in time determines the slope in figure 1.

The fractional delay μ is used in a feedback loop which drives μ and its slope to zero. This is shown in figure 5. The fractional delay changes with time approximately linearly. By using linear regression, a slope can be obtained that represents the sample clock frequency difference between the two receivers.

The value of μ is unwrapped (so that it does not merely go from -0.5 to 0.5 of a sample). The slope of μ is obtained by a linear fit. This slope (scaled by β below) is then integrated in the box "Cumulative Sum". Also, the value of μ itself (scaled by α below) is integrated in another "Cumulative Sum" box (the upper one in figure 5). These two accumulated values are added together and used to update an error term (ϵ below) to drive the fractional offset and the change in fractional offset to zero.

The objective in this feedback loop, as has been stated, is to drive our error term to zero. As we continue to receive samples, we continue to update this error term by adding the value of μ itself, and its slope, or derivative. Each value can be scaled. This update operation could be written as follows, with the error term written as ϵ

$$\epsilon = \epsilon_{old} + \alpha\mu + \beta\frac{d}{dt}\mu$$

Where α and β are scalars that weight each term some amount which affect the rate of convergence for the feedback loop. Typical values for α were between .1 and .01, while typical values for β were between 100 and 500. To estimate the slope, data points were grouped into GNURadio vectors of 1024 points to get a decent estimate of the slope. The direct term, μ, was downsampled by 1024 to match this rate. These rates and constants were chosen because they sucessfully drove the error term to zero within about 30 seconds without oscillations.

The implementation of this updating error term requires cumulative summation (similar to the standard integrator block in GNURadio) as well as a way to approximate a derivative for given points. These custom blocks are also found in the noted github repository.

The result of the fractional offset accumulations determined as in figure 5 is used to drive the "Cubic Alignment" operation in figure 7. This block interpolates whichever stream has the faster sample rate to be at the rate of the stream with the slower rate, as illustrated in figure 6. Interpolation is done using a cubic interpolator in the box "Cubic Align" in figure 7.

Cubic interpolation requires four points of known data, so this block is implemented with a queue. One stream of data (the slowest) is designated as the reference, and is passed through the block with no change. The other stream of data enters the queue, where the four relevant samples are used to interpolate the desired sample. The desired sample is some fractional distance away from the base, or original sample. This distance is controlled by the error value passed in to the block's Delay parameter. The operation is summarized as follows.

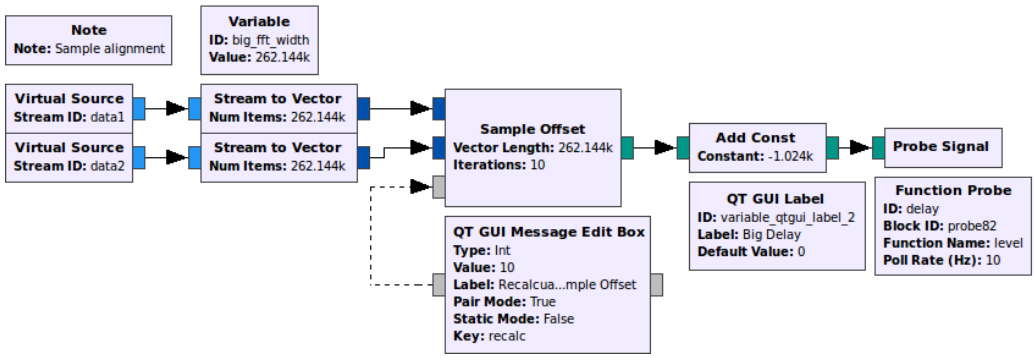

Figure 2. Detecting the Bulk Delay with Cross Correlation

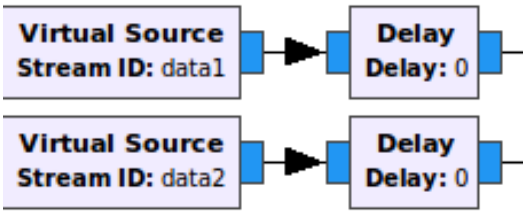

Figure 3. Applying the Bulk Delay Correction

For each received sample:

1. Push the sample onto the queue of points
2. Determine which four samples should be used
3. Interpolate using the four relevant samples
4. Update the current sample offset using the given delay parameter, which allows us to make decisions about which samples to use next

In figure 6, the four filled circles are the four samples to be used in order to obtain a new sample at the open square. The leading filled circle, labeled as the base sample, is the point two samples ahead of the desired interpolated sample. Usually this base sample increases, or moves ahead in time by one sample for every interpolated sample. However, after enough samples have gone by, the base sample may need to 'skip' a sample in order to maintain its position of two samples in front of the sample to be interpolated. This is the logic required by step four in the list above, where we determine if our base sample must skip a sample or not.

The full code for the block can be seen on the provided github repository. The cubic interpolation is modeled after Paul Bourke's primer on intepolation. (Bourke, 1999) A linear interpolation block is also included in the github repository.

In summary, the timing correction performs the following operations: We interpolate between the points of the cross correlation to get sub-sample results. Once the offset is known, we can approximate the sample drift rate between the two receivers with linear regression. This slope is then used to determine how much each sample should be interpolated in order to cancel out the sample drift. The performance of this feedback loop can be seen in figure 8. There is an initial period of wide variation in the sample offset (until about 12 seconds in this example). This is due to the time for the bulk offset to be determined and the delays to take effect. Then the timing loop starts converging. After about 25 seconds, it converges to where the sample offset is near 0 and stays there.

4. Frequency and Phase Offsets

The timing interpolation of the last section accounts for timing offsets, but does not account for frequency offsets between the two ADCs. Like the timing offset, which has a bulk offset and a changing time offset, the frequency offset has both a phase offset term and a frequency offset term.

In order to correct for this offset, the frequency offset must first be measured or estimated using the phase difference.

In our configuration, the phase difference is estimated using an eigenvector method. This is computed as follows. Let $s_1(t)$ be the signal at one receiver taken as the reference receiver. This signal is assumed to be a narrowband signal $m(t)$ mixed by some frequency component, expressed in

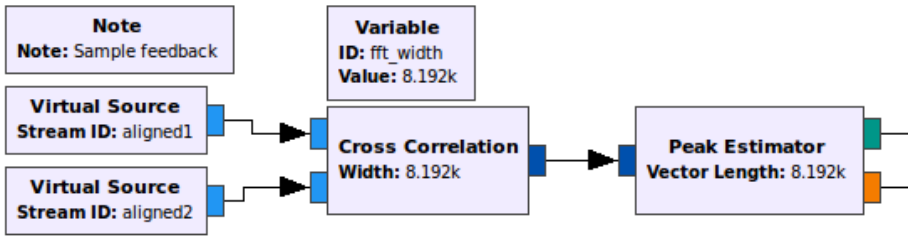

Figure 4. Finding the Fractional Delay

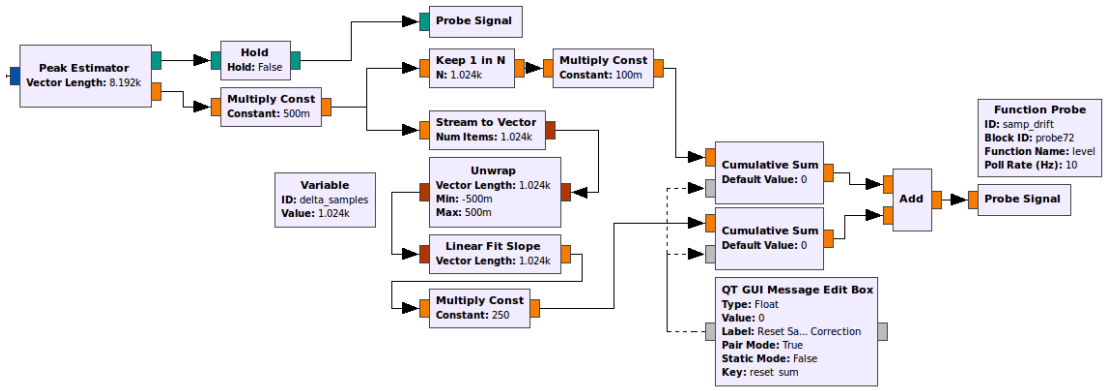

Figure 5. Linear Regression to Produce an Error Term

terms of its complex representation as

$$s_1(t) = m(t)e^{j\omega_c t}.$$

The signal $m(t)$ is not assumed to be known. The signal $s_1(t)$ is basebanded to produce $x_1(t) = m(t)$.

A delayed version of the signal is received at a second receiver, with delay τ. Assuming $m(t)$ is sufficiently narrowband and the delay is small enough that that $m(t-\tau) \approx m(t)$, we have

$$s_2(t) = s_1(t-\tau) = m(t)e^{j\omega_c(t-\tau)} = m(t)e^{j\omega_c t - \phi}$$

with $\phi = \omega_c \tau$. This signal is also basebanded to produce $x_2(t) = m(t)e^{-j\phi}$. Stacking the two received signals into a vector $\mathbf{x}(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix}$ we can write

$$\mathbf{x}(t) = m(t) \begin{bmatrix} 1 \\ e^{-j\phi} \end{bmatrix}.$$

Let $\mathbf{h} = \begin{bmatrix} 1 \\ e^{-j\hat{\phi}} \end{bmatrix}$ be a steering vector applied to $\mathbf{x}(t)$ to form $y(t;\hat{\phi})$ as

$$y(t;\hat{\phi}) = \mathbf{h}^H \mathbf{x}(t) = \begin{bmatrix} 1 \\ e^{-j\hat{\phi}} \end{bmatrix}^H m(t) \begin{bmatrix} 1 \\ e^{-j\phi} \end{bmatrix}$$
$$= m(t)(1 + e^{j(\hat{\phi}-\phi)}).$$

The average power in $y(t;\hat{\phi})$ is maximized when $\hat{\phi} = \phi$. This average power is computed as

$$E[|y(t;\hat{\phi})|^2] = \mathbf{h}^H E[\mathbf{x}(t)\mathbf{x}(t)^H]\mathbf{h} \triangleq \mathbf{h}^H R_x \mathbf{h}. \quad (1)$$

Here, R_x is the 2×2 correlation matrix of the measured signals, which is estimated as

$$R_x \approx \frac{1}{N} \sum_{i=i_0}^{N+i_0} \mathbf{x}(i)\mathbf{x}(i)^H.$$

To maximize the average power in (1) for a steering vector of given norm, it is well known that the maximum is obtained when the steering vector is proportional to the eigenvector of R_x corresponding to the largest eigenvalue. Since R_x is a 2×2 matrix, the largest eigenvalue and corresponding eigenvector are easily computed.

A block to compute the phase difference by this method is shown in figure 9. The vector autocorrelation and eigendecomposition are done in the block PCA DOA. Also in this figure, note that the signals are downsampled by 50. This reduces the computation, and does not affect the phase estimate resolution.

The phase estimate produced in figure 9 is passed to the system in figure 10. The phase is unwrapped so that

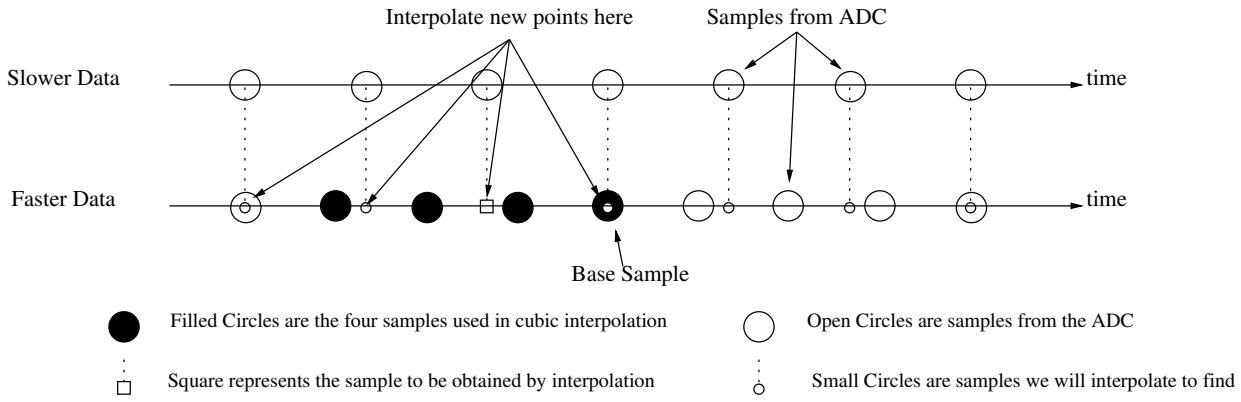

Figure 6. Adjusting the timing by interpolation

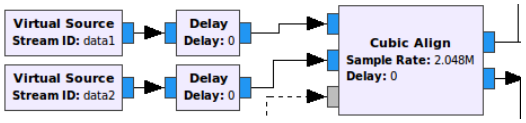

Figure 7. Applying the Fractional Sample Correction

changes in phase fall approximately linearly. The slope of the phase change line is estimate by linear regression ("Linear Fit Slope"). The slope is accumulated, along with the accumulated phase itself. These two accumulated errors are added, and used to drive the error term to zero.

This generation of an error term is exactly analogous to the error term used in correcting the clock timing drift. As stated earlier, the error term, represented as ε can be updated as samples are available as follows:

$$\varepsilon = \varepsilon_{old} + \alpha_\phi \phi + \beta_\phi \frac{d}{dt}\phi$$

Where α_ϕ and β_ϕ are again scalars that weight each term some amount which affect the rate of convergence for the feedback loop. Typical values for α_ϕ were around .1 and for β_ϕ, around 400. Like with the clock timing drift loop, the data was converted to GNURadio vectors of length 100 to fit a slope to the data, and the direct stream of ϕ values from the PCA block were downsampled to match this rate.

The only other difference in this GNURadio implementation from the clock timing drift loop is the addition of a moving average block to smooth out the error term, with a scaling value around .04 and length of 25. This averaged error term is then used directly as the frequency of the correcting complex exponential mentioned previously.

The phase/frequency correction term so obtained is applied to the signal after the timing has been adjusted by the Cubic Align step. In figure 11, a complex exponential Signal Source is produced which goes into a Multiply box to perform the frequency adjustment. These corrections use the standard Signal Source and Multiply blocks in GNURadio. Figure 12 shows the results of the phase/frequency correction. After the timing loops settle down (at about 13 seconds), the phase difference converges to near zero and stays there.

5. Putting the Pieces Together

With both feedback loops running, the system corrects for sample, frequency, and phase offsets. As ADCs continue to drift, the system adapts to correct for the new offsets. The complete system, implemented as a GNURadio flowgraph, can be seen in figure 13

6. Conclusions

Synchronizing timing between two RTL-SDR ADCs involves steps to account for bulk timing delay, clock drift, and phase and frequency offsets. In this paper, signal processing methods implementable using the GNURadio system have been presented, with some results showing the performance of these methods. These all combine into a system with multiple feedback loops that work together to achieve the desired synchronization.

The authors have made their work available on github, at https://github.com/samwhiting/gnuradio-doa .

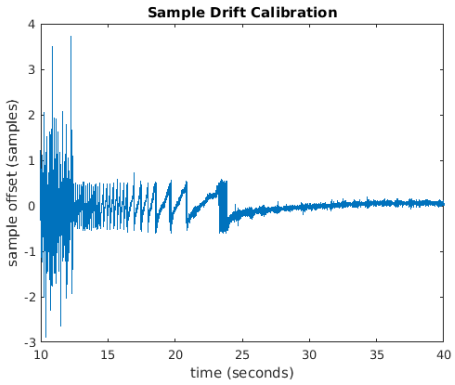

Figure 8. Sample Offset Corrections

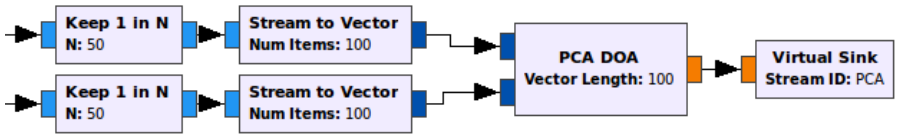

Figure 9. Calculating Phase Difference with Principal Component Analysis

References

Bourke, Paul. *Interpolation Methods*, 1999. http://paulbourke.net/miscellaneous/interpolation/.

Junming Wei, Changbin Yu. Improvement of software defined radio based TDOA source localization.

Krysik, Piotr. *Multi-RTL*, 2016. https://ptrkrysik.github.io/.

RTL2832u. *RTL2832u based coherent multichannel receiver*, 2014. http://yo3iiu.ro/blog/?p=1450.

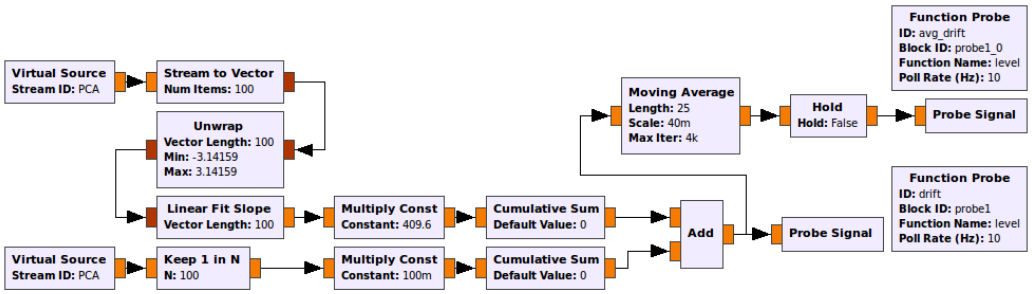

Figure 10. Linear Regression to find a Frequency Error term

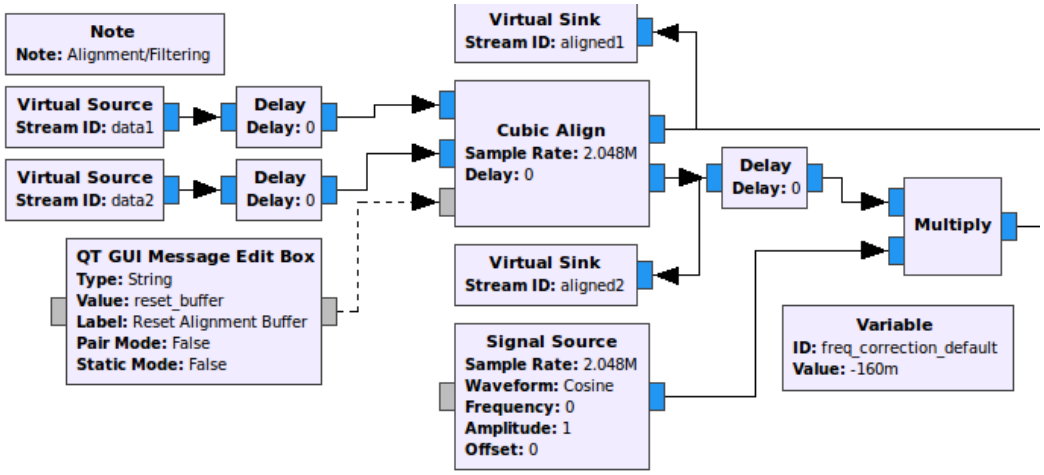

Figure 11. Applying the Complex Frequency Correction

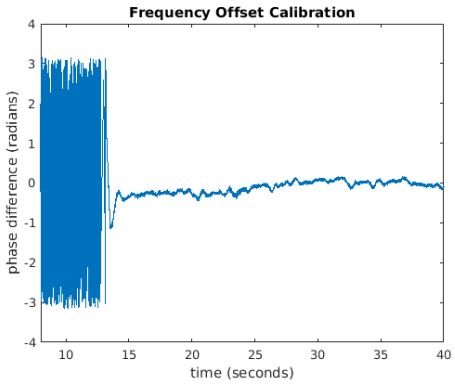

Figure 12. Frequency Difference Corrections

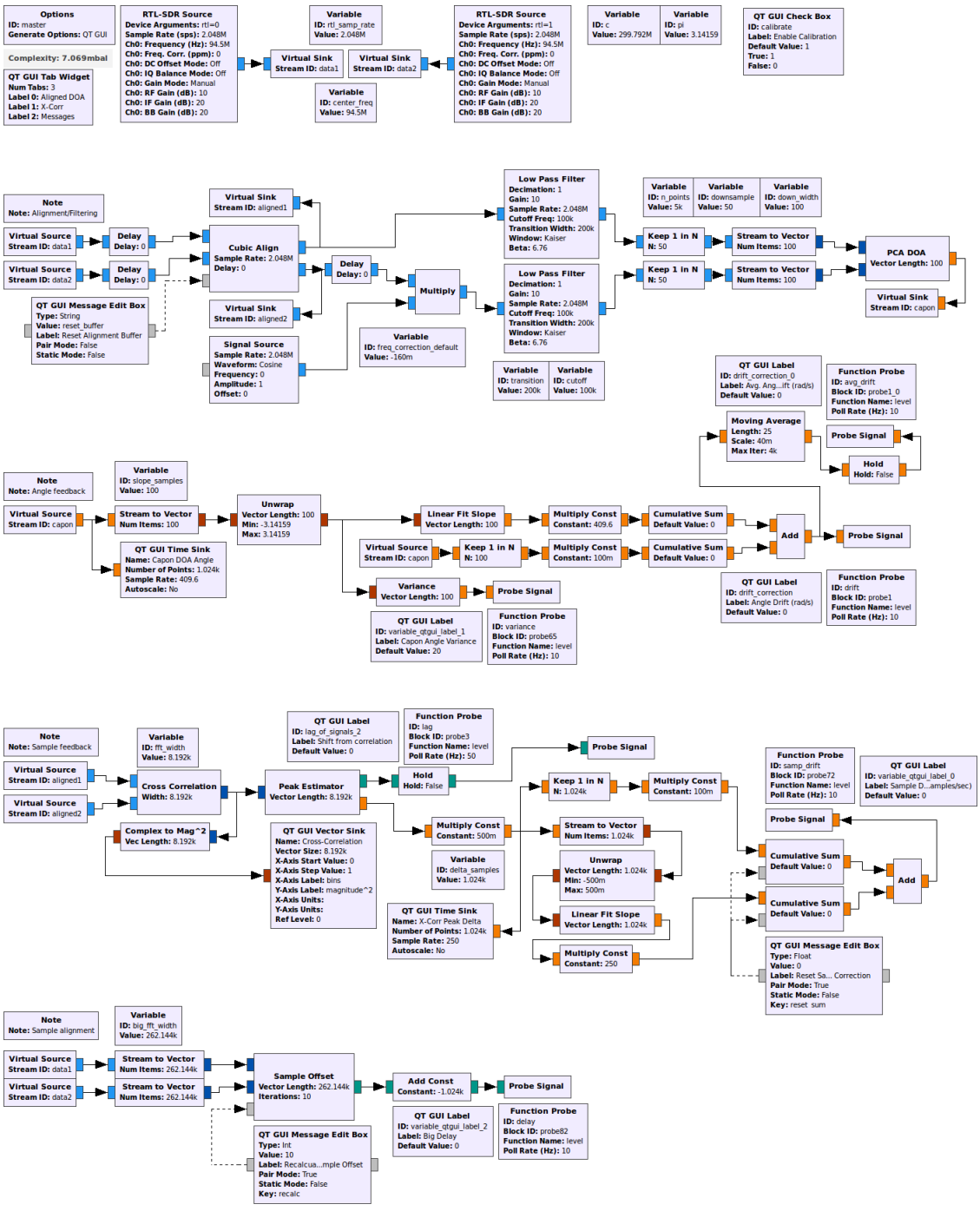

Figure 13. GNURadio Flowgraph