The Digital Age

Kevin Yu

March 3, 2014

Abstract

We study techniques of digital sampling and data analysis in order to demonstrate methods by which we can manipulate and study signals in both the time and frequency domains. We experiment with violating the Nyquist criteron and find that we cannot adequately represent signals with frequency components faster than half the sample rate. Additionally, we learn to use heterodyne mixing, both with an analog mixer and with the ROACH (Reconfigurable Open Architecture for Computing Hardware), which allows us to convert a signal of frequency ω to a much lower frequency δ by mixing it with a local oscillator driven at a frequency $\omega \pm \delta$. Finally, we use the ROACH's parallel processing powers to implement an eight tap FIR filter to create a digital 5/8-band filter, which we find is not capable of replicating an *ideal* filter shape but can come close with many samples. Figures and code can be found online at http://github.com/kevinyu/astro121.git.

Introduction

In this lab, we learn the advantages and limitations of digital sampling and mixing, as well as methods of analyzing the spectral compsition of signals. Digital sampling is inherently limited by the need to translate a continuous signal's characteristics into discrete values that can be understood by a computer, which represents measurements in terms of binary `1`'s and `0`'s. Thus, there are trade-offs at every turn—for example, turning up your sampling rate will improve the shape of your waveform, but your computer's memory will fill up faster and you will not be able to sample for as long.

Luckily, knowing the limitations of digital sampling also allows us to determine ways in which we can save time/energy/memory without losing important information about the radio signals we are trying to study. In cases where we are dealing with frequencies much greater than can be handled by our equipment, we will want to use heterodyne mixing to shift the signal to a much lower frequency. We can then use tools such as discrete fourier transforms and digital filters to futher manipulate our signal to extract the underlying information we are looking for.

Methods and Results

1 Fourier Analysis

The Fourier Transform, defined as

$$\hat{f}(\omega) = \int_{-\infty}^{+\infty} f(t) e^{i\omega t} dt \tag{1}$$

is the basic tool for analyzing a signal in both the time and the frequency domains. It allows one to decompose a signal into its Fourier components to see what frequencies it is "made of".

When dealing with digitally sampled data, however, we can no longer use the integral in Eq. 1. For one, we are dealing with discretely sampled points, which means we need to use a summation instead of an integral. Additionally, we cannot take samples from $t = -\infty$ to $+\infty$—and even if we could, it would already be too late to start. In the case where we can only sample at a finite rate and for a finite period of time, the integral becomes the following summation:

$$\hat{f}(\omega) = \frac{1}{N\Delta t} \sum_{n=1}^{N} f_n e^{j\omega n \Delta t} \Delta t \tag{2}$$

in which N is the total number of sampled data points and Δt is the timestep between samples ($1/\nu_{samp}$).

The naïve implementation of this algorithm—which loops through frequencies in the range $-\frac{1}{2\Delta t}$ to $\frac{1}{2\Delta t}$ in steps of $\frac{1}{\Delta t}$—runs in $O(n^2)$ time, and thus is not suited to large datasets. However, in this lab I will use the more efficient `fft` (Fast Fourier Transform) algorithm for computing discrete Fourier transforms, provided by Python's scientific computing package NumPy. This algorithm's running time scales as $O(n \log_2(n))$ and thus is suitable for large datasets.

Figures in the upcoming sections that contain plots of a power spectrum are created by first computing the `fft` of the waveform of interest, taking the magnitude of the complex output (using Python's `abs` function), and squaring the resulting values.

2 Nyquist Frequency

One of the most important concepts in digital sampling is the Nyquist criteron, which specifies the minimum frequency at which it is possible for a digitally sampled waveform to accurately reproduce the frequency composition of the actual signal. The Nyquist criteron can be put in this way: for a particular sample rate ν_{samp}, only signal frequencies at or below the Nyquist rate

$$\nu_{nyquist} = \nu_{samp}/2 \tag{3}$$

can be captured by the sample. The reason for this is fairly intuitive; if one samples points separated by a time Δt, then any fluctuations in the signal that occur between the two

samples will not be observed. To see a fluctuation in the signal, at least two points per period (one "high" and one "low") must be taken—this statement is equivalent to Eq.3.

Failure to satisfy the Nyquist criteron will result in *aliasing*, in which the sampled waveform appears to depict a slower frequency and possibly opposite phase to the original signal. An artistic interpretation of this phenomenon is shown in Figure 1.

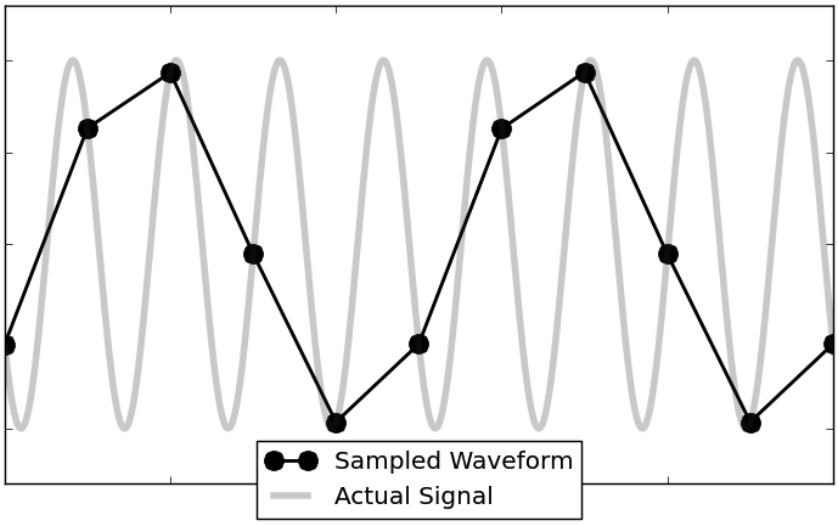

Figure 1: An example of aliasing. Because the sample rate is not fast enough to capture the high frequency information in the signal, the result appears to be of a slower frequency.

To demonsrate the effects of sampling above the Nyquist frequnecy, we use the Pulsar sampler card to take digital waveforms of incoming sine waves produced by the SRS Signal Generators. We fix our sampling rate at $\nu_{samp} = 10$ kHz* so that the Nyquist frequency is $\nu_{nyquist} = 5$ MHz.

The sampled waveforms for signal frequencies ranging from 1 kHz to 9 kHz are shown in Figure 3. The corresponding power spectra from these sampled waveforms are shown in the right column.

*This is safely below the Pulsar's maximum sampling rate of 10 MHz

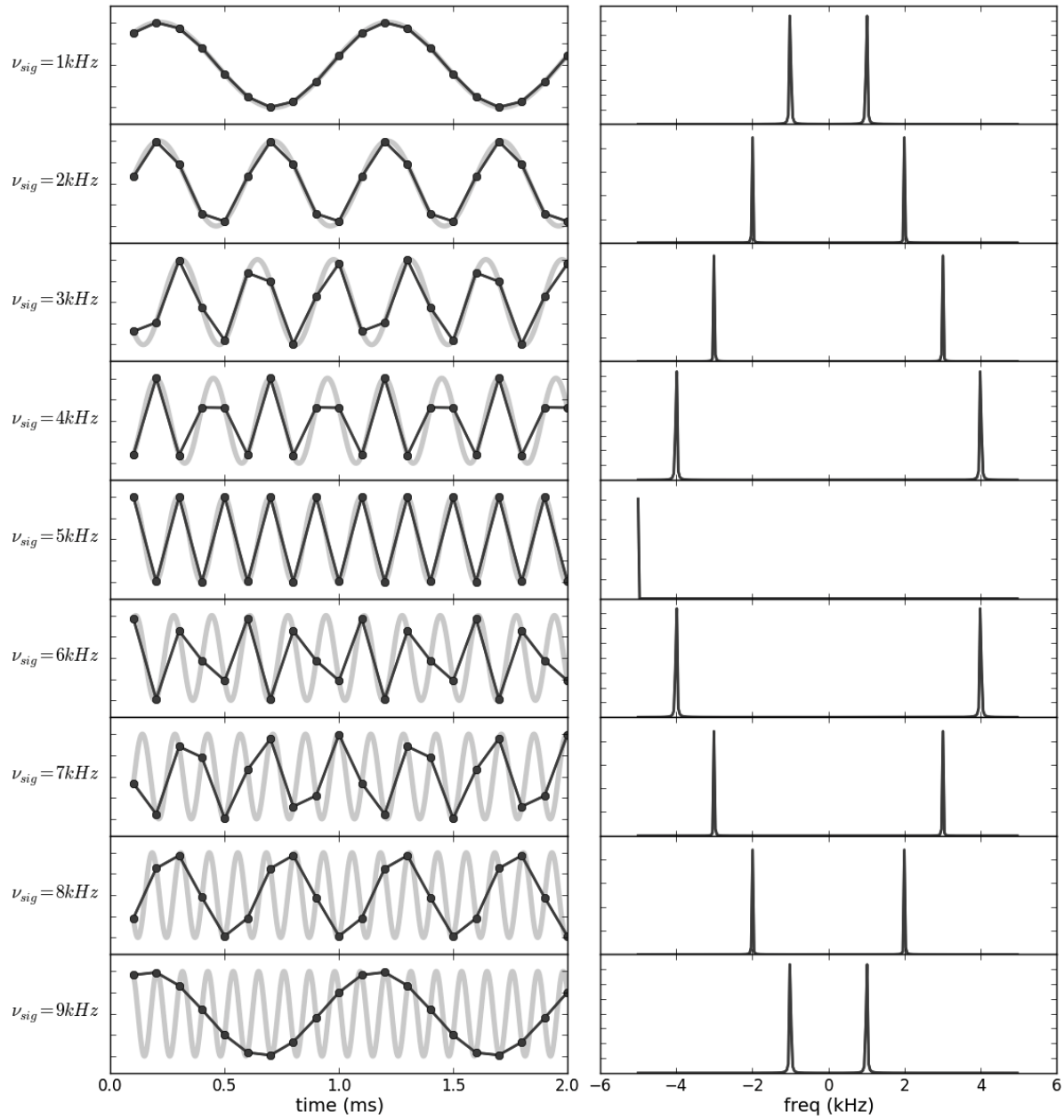

Figure 2: Left: Time domain representation of sampled waveforms, with interpolation, sampled at 10 kHz. The frequency of the original signal is shown on the left. A representation of the actual signal is overlayed in a lighter shade. Aliasing can be seen for signal frequencies greater than $\nu_{nyquist} = 5$ kHz. Right: The corresponding power spectrum for sampled waveforms. For signal frequencies greater than the Nyquist rate, the observed frequencies are the actual frequencies reflected over $\nu_{nyquist}$. The power spectrum for $\nu_{sig} = 5$ kHz is missing a spike at +5 kHz due to the nature of the `fft` program used, which is asymmetric around zero in that it omits the final frequency point on the positive side; the 5 kHz frequency does, however, exist in the sampled waveform.

As can be seen from the time domain representation of our sampled waveforms in Figure 3, the signals slower than the Nyquist rate—from 1 kHz to 4 kHz—are all represented accurately in terms of their frequency. Though the interpolated waveforms appear jagged, the underlying sinusoidal shape and frequency of the signal are clearly visible. Additionally, the peaks in the power spectrum correspond well with the true frequency of its signal. Positive and negative spikes at the signal frequency are seen due to the signal only being real-valued, consisting of both an $e^{j\omega t}$ and $e^{-j\omega t}$ components.

At 5 kHz, which is exactly the Nyquist rate, the sampled waveform is a series of "high" and "low" values, both staying at consistent levels. This is because the incoming signal is sampled *exactly* twice per period, at exactly the same point in the sinusoid's cycle each time. Though the sampled points lack detail, the underlying 5 kHz frequency is still clear in both the time domain representation and the fourier spectrum. However, we must be careful for this case; we got lucky that we sampled at approximately the top and bottom of the signal. If we had started sampling just a quarter period too late, we would only see a flat signal; we would still sample twice per period, but only at the points where the signal is crossing zero! We can conclude that although the Nyquist frequency gives us a minimum sampling frequency, we are better off sampling at least a little bit faster.

Finally, the most interesting results are for the cases in whih we violate the Nyquist criteron at signal frequencies greater than 5 kHz. For signal frequencies 6 kHz to 9 kHz, we no longer see spikes at the expected frequencies in the power spectrum but instead at the expected frequencies *reflected across* $\nu_{nyquist}$. When we look at the waveform in the time domain, it is clear that the sampled version of the signals look slower than they actually are. Additionally, these sampled waveforms violating the Nyquist criteron appear to have a 180° phase shift relative to the signal's actual phase; this can be seen by noticing that if the actual signal begins with a downward slope, the sampled waveform begins with an upward slope, or vice versa.

3 Mixing

A local oscillator, or LO, is a signal used to convert an incoming signal to a different frequency through the process of mixing. When we *mix* two signals, f and g, we multiply them in the time domain, which is equivalent to convolving them in the Fourier, or frequency, domain.

When dealing with radio signals, our equipment may not be suited to the higher frequencies. If the frequency of interest is larger than our equipment is capable of handling, we can use heterodyne mixing to shift the signal to a lower frequency where it can be appropriately analyzed.

When we mix a real-valued signal of frequency ν_{sig} with a local oscillator of the form $e^{j2\pi\nu_{LO}t}$, we will see the following frequency components in the spectrum of the output, which are calle d the intermediate frequencies (IF). They are:

$$\nu_{IF} = \nu_{LO} \pm \nu_{sig} \tag{4}$$

if our local oscillator is only real-valued, consisting of both $e^{j2\pi\nu_{LO}t}$ and $e^{-j2\pi\nu_{LO}t}$ components,

our intermediate frequencies are:

$$\nu_{IF} = \nu_{LO} \pm \nu_{sig},\ -\nu_{LO} \pm \nu_{sig} \tag{5}$$

In order to convert a high frequency signal of frequency ν to a much slower frequency δ, we can mix the signal with a local oscillator of frequency $\nu_{LO} = \nu + \delta$. This will result in intermediate frequencies of

$$\nu_{IF} = \delta,\ 2\nu + \delta \tag{6}$$

From here, we can apply a filter to isolate the frequency component at δ.

3.1 Analog Mixing (DSB)

To demonstrate a heterodyne mixer, we first use an analog mixer, the ZAD-1, to mix a "signal" and "LO"—each generated by one of the SRS Signal Generators. We choose our ν_{LO} to be 2 MHz. Using this LO frequency, we first mix it with a signal of frequency 2.1 MHz, and the second time with a signal of frequency of 1.9 MHz. In both cases, the δ is 0.1 MHz.

Our expected intermediate values are the following table, keeping in mind the fact that both signal generators only produce real-valued signals:

ν_{LO} MHz	ν_{sig} MHz	$\nu_{LO} \pm \nu_{sig}$ MHz	$-\nu_{LO} \pm \nu_{sig}$ MHz
2.0	2.1	-0.1, 4.1	-4.1, 0.1
2.0	1.9	0.1, 3.9	-3.9, -0.1

Looking at the positive ν_{LO} frequencies (it is symmetric for the negative ones), we see that the output will be split into a lower sideband—$\nu_{LO} - \nu_{sig}$—and an upper sideband—$\nu_{LO} + \nu_{sig}$. Thus, we have a double sideband (DSB) mixer. A single sideband (SSB) mixer would require mixing with pure sinusoids of the form $e^{j\omega t}$, which have both a real and imaginary part. With a complex sinusoid, we would be able to eliminate one of the two sidebands.

To sample the output of our analog heterodyne DSB mixer, we use the Pulsar card and sample at a rate of 10 MHz, the maximum frequency the Pulsar will sample at. We collect $N = 16384$ data points, thus for each waveform we sample for $N * \Delta t$, or 1.6 ms. Figure 3 shows the power spectrum of the analog mixer output.

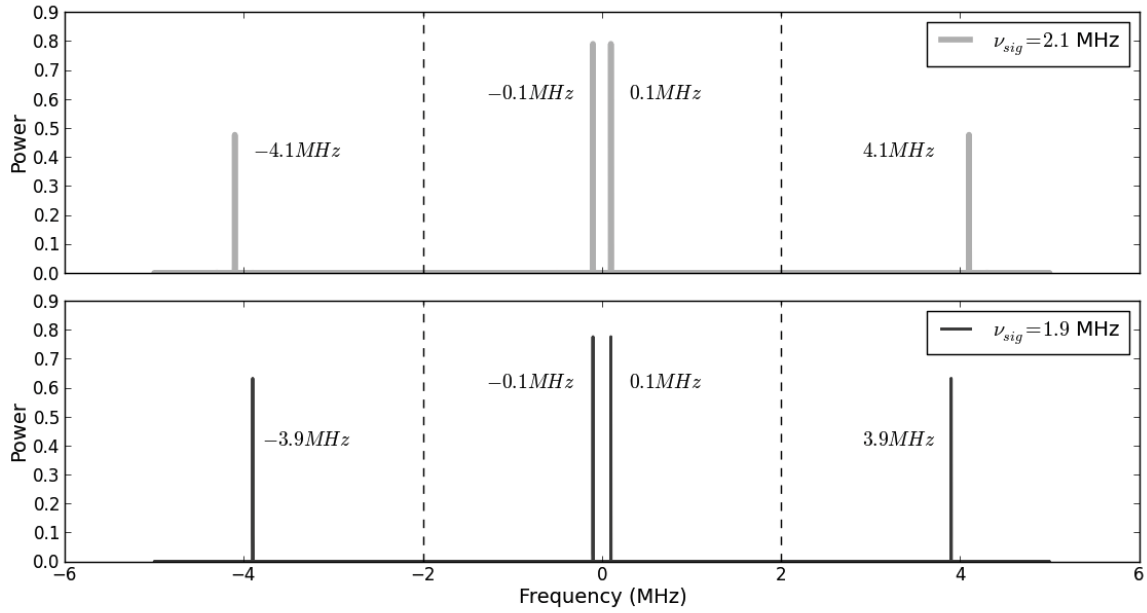

Figure 3: Power spectrums of analog mixer output for $\nu_{LO} = 2$ MHz (black dotted line), digitally sampled at 10 MHz. Notice that both results have spectral features at $\pm 0.1MHz$ due to the fact that the mixers produce frequencies at $\nu_{LO} \pm \nu_{sig}$ as well as $-\nu_{LO} \pm \nu_{sig}$

The power spectrum of the mixed signals feature upper and lower sidebands in addition to features at both positive and negative frequencies, as predicted earlier. From Eq. 6, the upper sideband (the sum frequency component) occurs at $2\nu_{sig} + \delta$, and the lower sideband (the difference frequency component) occurs at δ.

A small section of the two waveforms are shown in the top row of Figure 4. The upper and lower sidebands can be seen by the high frequency oscillations (upper sideband) on top of the slower periodic fluctuation (lower sideband). To filter out the upper sideband and keep only the lower sideband, we can use Fourier filtering.

To do this, we start by zeroing out the sum frequency values in the fourier transform data from Figure 3. For the 2.1 MHz case, we will zero out the components around ± 4.1 MHz and for the 1.9 MHz case, we will zero out the components around ± 3.9 MHz. We can then apply an inverse discrete Fourier transform and get the waveforms shown in the bottom row of Figure 4. These waveforms no longer feature a strong high frequency oscillation; only the slower frequency at 0.1 MHz remains. Unfortunately, Fourier filtering does not seem to work perfectly at the extreme ends of the time domain when the inverse transform is applied, which is why the plots begin at $100\mu s$. This is probably due to the fact that the waveform needs to suddenly terminate at the ends which requires frequency components beyond the single frequency we filtered out.

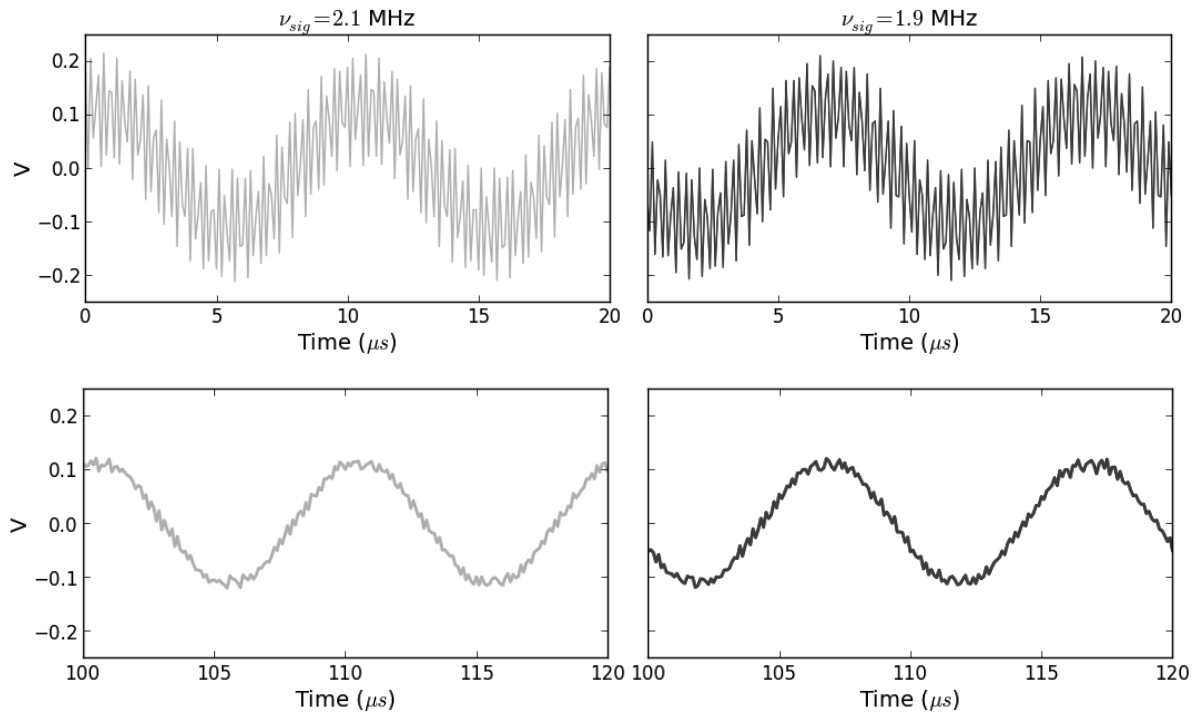

Figure 4: Top: Time domain output from analog mixer with $\nu_{LO} = 2$ MHz, digitally sampled at 10 MHz. The high frequency upper sideband can be seen on top of the slower lower sideband. Bottom: The same waveforms after removing the upper sidebands using Fourier filtering. Only the 0.1 MHz lower sideband remains.

3.2 Digital Mixing Using the ROACH

The ROACH (Reconfigurable Open Architecture for Computing Hardware), is a system consisting of a CPU running a Linux kernel and an FPGA, which can be configured to perform tasks with many logic gates running in parallel, synchronized in this lab by a 200 MHz clock. We can connect to it remotely and configure our data collection by writing to the FPGA's registers. The FPGA writes output data into block random access memory (BRAM). Once the BRAM is full, the data can be copied out of the ROACH for analysis.

3.3 Double Sideband (DSB)

In the last section, we used an analog mixer to mix our signals with frequency ν_{LO} and ν_{sig}. The ROACH is capable of digitally mixing the two signals and writing the result to BRAM. If we follow the same procedure as in Section 3.1, this time mixing digitally with the ROACH, we expect to see a similar result. The theory is the same, and the expected output frequencies are still given by Eq. 6.

For consistency, we will continue to use a local oscillator frequency of 2 MHz, and signal frequencies of 2.1 and 1.9 MHz. The only difference is that we will clock the ROACH at a rate of 200 MHz, which is the rate at which logic gates in the FPGA will execute their operations and the rate at which data will be sampled.

The results of digital mixing with these signal frequencies can be seen in Figure 5. The waveform and power spectrum are very similar to the results of the analog mixing case in Figure 3, with the same spectral features. However, there are a couple differences that can be seen in the power spectrum by noticing that the peaks seem wider than their counterparts in Figure 3.

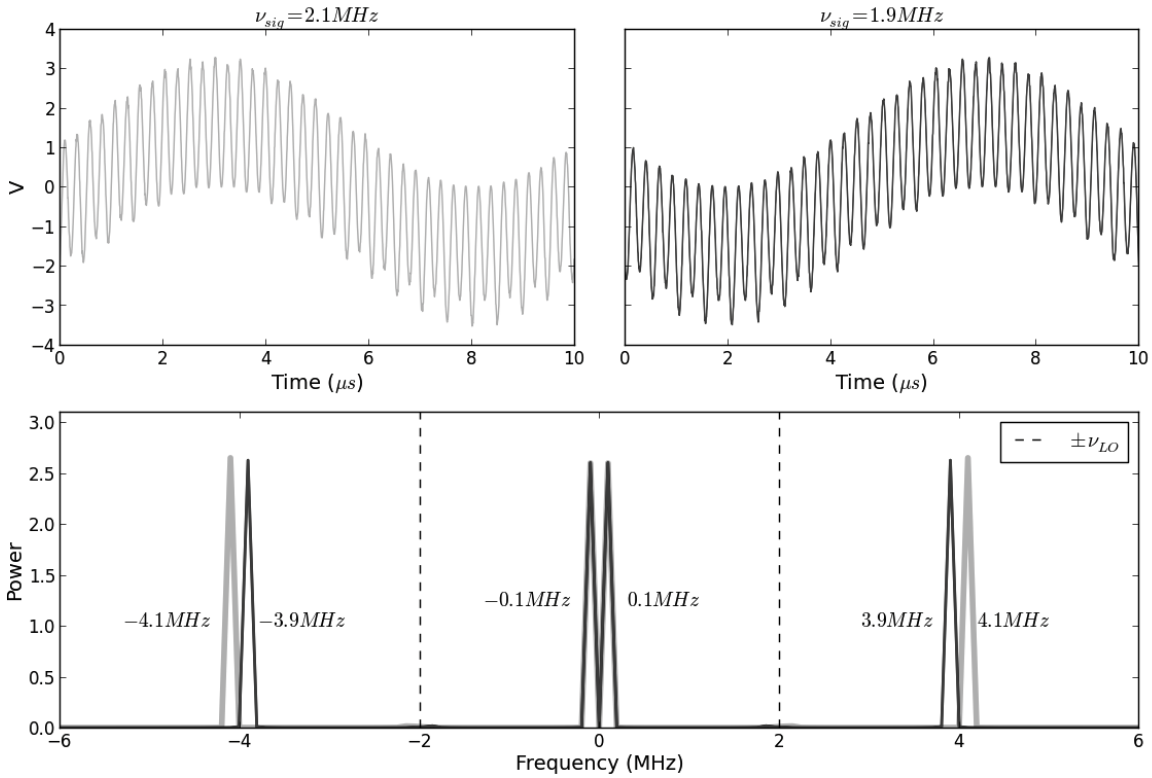

Figure 5: Similar to Figures 4 and 3, this time mixing digitally using the ROACH. Top: Output of digital mixer with $\nu_{LO} = 2$ MHz, sampled at 200 MHz. Bottom: The power spectrum for both of the outputs. The spectral features occur at the same frequencies as in the analog case of Figure 3. We do not observe a DC offset that was predicted by the lab manual. The peaks in the spectrum consist of a single datapoint each—the reason they look wider than in Figure 3 is that the limited size of the ROACH's BRAM only allowed us to sample for about one period, which limited our resolution in the frequency domain.

The reason for these apparently wider peaks is a lower frequency resolution due to a limited sample size. When data collection is triggered on the ROACH, data points are

collected and stored in BRAM until the BRAM is full, limiting the number of data points that we can take. Apparently, the BRAM can only store 2048 32-bit signed integers, meaning if we take samples at 200 MHz we can sample for 10.24 μs. In comparison, when we sampled the output of our analog mixer using the Pulsar card in Section 3.1, we took 16384 values at 10 MHz, meaning we sampled for 1 ms, one hunderd times longer. Because of this, our resolution in the frequency domain for digitally sampling on the ROACH is one hundred times lower than in the earlier section, which is why the peaks in the spectrum look so wide.

One way to solve this minor issue would be to decrease the clock rate much lower than 200 MHz, so that the BRAM does not fill up as quickly. This would allow us to take data for a longer period of time.

3.4 Single Sideband (SSB)

In the DSB cases, both analog and digital, we mixed our signal f with a local oscillator of the form $\cos(\omega t)$. Because the cos has no imaginary part, we essentially mix our signal with both $e^{j\omega t}$ and $e^{-j\omega t}$ components, creating the upper and lower sidebands, respectively. If instead we wanted to have only a single sideband, we would need to mix our signal with a complex local oscillator of the form $e^{j\omega t}$

Using the ROACH's FPGA, we will be able to mix with both a cos and sin wave simultaneously. The FPGA does not output the complex result of the mixing; instead, it mixes the signal simultaneously with samples from a sine and a cosine local oscillator (two sinusoids a quarter period out of phase). These two outputs are stored in separates BRAMs. With those sampled datapoints, we can then programmatically combine them using the following Eq. 7 to see the full, complex result.

$$\begin{aligned} Re[f(t)e^{\pm j\omega t}] &= f(t)\cos(\omega t) \\ Im[f(t)e^{\pm j\omega t}] &= \pm f(t)\sin(\omega t) \\ f(t)e^{\pm j\omega t} &= [f(t)\cos(\omega t)] \pm j[f(t)\sin(\omega t)] \end{aligned} \tag{7}$$

The frequency of the LO (ω in Eq. 7) is set by the ROACH's sampling clock and the value written to the `lo_freq` register, which I will call x. To generate samples of a cos and sin wave, the FPGA maps the interval of 0 to 2π into 256 steps; i.e. 0, $\frac{2\pi}{256}$, $\frac{2\pi}{128}$, $\frac{2\pi}{64}$, etc.

Each clock cycle, the ROACH increments its current position in the interval by x steps. The FPGA then evaluates cos and sin at that value to be mixed with the next sampled point of the incoming signal. Thus, an entire period of 2π is completed in $256/x$ steps. With a clock rate of ν_{samp}, the frequency of the LO generated in this fashion is

$$\nu_{LO} = \frac{x}{256} * \nu_{samp} \tag{8}$$

Using the above Eq. 8, we can determine the local oscillator frequency for any value we write to the FPGA's `lo_freq` register.

To demonstrate our SSB mixer, we generate a 6 MHz, 0 dBm signal with the SRS and mix it with the LO frequencies listed in the following table, for a 200 MHz clock rate. Because

we want a single sideband mixer that takes only the lower sideband, we will use the version of Eq. 6 with the minus sign. The signal generated by the SRS will contain both "positive" and "negative" frequencies, as it is simply a cosine signal, so we expect to see two frequencies in the output after combining the real and imaginary parts: $\pm\nu_{sig} - \nu_{LO}$. Our Nyquist rate is 100 MHz, so we should have no problem seeing these signals.

x	$\nu_{LO}\ [MHz]$	$\nu_{output} = \pm\nu_{sig} - \nu_{LO}\ [MHz]$
1	0.782	-6.782, 5.218
2	1.563	-7.563, 4.437
4	3.125	-9.125, 2.875
16	12.5	-18.5, 6.5

Figure 6 illustrates the result of digitally mixing for the third case, $\nu_{LO} = 3.125MHz$. As expected, this case has lower side bands at -9.125 MHz and 2.875 MHz. The upper side bands, which would have been at -2.875 MHz and 9.125 MHz, have been avoided by combining both real and imaginary parts of the mixer output.

The power spectrum also includes a small spike at -3.125 MHz, the LO frequency. It is about an order of magnitude smaller than the sidebands, so it does not feature prominently in the time domain waveform. We will remove this feature as well as the -9.125 MHz sideband using Fourier filtering.

The result of our Fourier filter, containing both real and imaginary components, is overlaid on Figure 6, isolating the 2.875 MHz component. This extracted waveform has a positive frequency, which can be seen by noticing the imaginary part (the sin component) trails the real part by a quarter phase, consistent with a positive frequency. It can also be seen by noticing that 2.875 MHz is greater than zero.

TODO DISCUSS ADVANTAGE AND DISADVANTAGE OF DSB AND SSB AND WHAT HAPPENS WHEN OUT OF PHASE

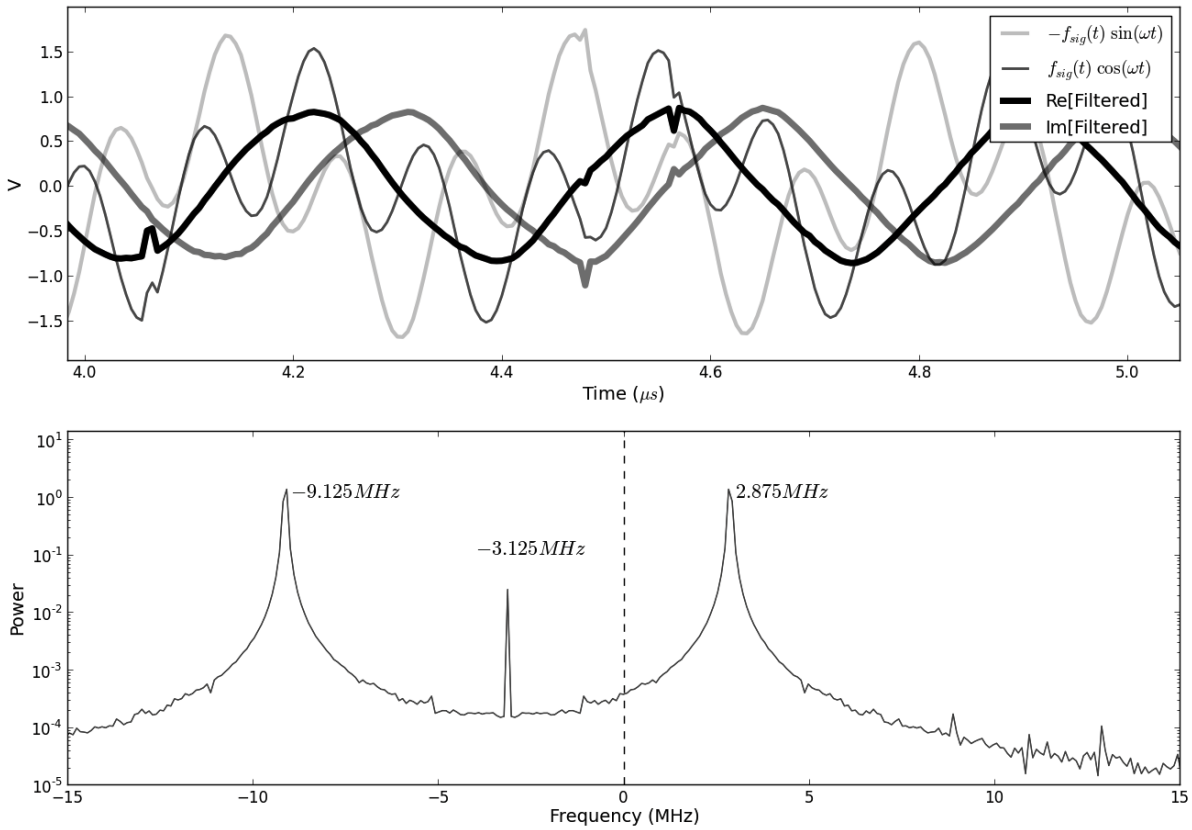

Figure 6: Top: The result of digitally mixing a $6MHz$ signal with a $3.125MHz$ local oscillator. The blue line represents the real part, $\text{Re}[f(t)e^{-j\omega t}] = f(t)\ \cos(\omega t)$, and the orange line represents the imaginary part, $\text{Im}[f(t)e^{-j\omega t}] = -f(t)\ \sin(\omega t)$. Overlaid in a thick black line is the real part of the waveform resulting from using Fourier filtering to remove the -9.125 MHz and -3.125 MHz components. The thick, lighter, line is the imaginary part of the filtered waveform. Bottom: Power spectrum of the complex waveform. There are clear peaks at $-\nu_{LO} \pm \nu_{sig}$, as well as a small peak at ν_{LO} from the local oscillator. We filter out the two negative peaks to extract the desired signal.

4 FIR Filter

A FIR, or Finite Impulse Response, filter is a way to implement a frequency domain filter by convolving in the time domain. In this section, we implement a FIR filter using the FPGA. The implementation of an FIR filter on the FPGA allows us to convolve an incoming signal with a custom waveform in the time domain *as the signal is sampled*. Our filter has 8 taps, so we choose 8 real and 8 imaginary coefficients to represent the filter, which we tell the FPGA by writing them into its `coeff` registers. As the signal is sampled, the FPGA multiplies each

of the last eight points from the ADC with their corresponding coefficients, and sums them up. Each clock cycle, a new point is sampled and the process is repeated. The sums form a new waveform, and the net effect is that we essentially drag, or *convolve*, the incoming signal across our filter. The sums are largest at times when the signal "looks like" our filter, and smallest when the sample does not resemble our filter—thus, the FIR filter picks out portions of the signal most similar to the filter shape.

4.1 Choosing Coefficients

We use this functionality to implement a bandpass filter. To choose coefficients for the FIR filter, we first create our bandpass in the frequency domain—a square spectrum centered around a certain frequency—and apply the inverse fourier transform to it to get the filter in the time domain. Since we are working digitally, we will only choose 8 points (See Figure 7) of the filter so that when we do the inverse discrete fourier transform we wind up with a complex coefficient for each of the eight `coeff` registers.

Our bandpass in this experiment is a 5/8-band filter, centered around 0 Hz. Since our clock/sample rate is 200 MHz, the Nyquist rate is 100 MHz and the frequency range that we will concern ourselves with is the range −100 MHz to 100 MHz. The 5/8-band filter will be a filter with a full response in 5/8 of that range centered around 0, or from $-62.5MHz$ to $62.5MHz$. Elsewhere, the filter's response should be zero, as depicted in Figure 7. To be able to take the discrete inverse fourier transform of this, we take eight equally spaced samples from the ideal filter and use the inverse discrete Fourier transform algorithm from NumPy's `fft` module. The points taken in the frequency domain and the corresponding complex coefficients in the time domain are enumerated in the following table:

ν [MHz]	Power		t (ns)	Real Coeff.	Imag. Coeff.
−100	0		0	`0.125`	0
−75	0		5	`-0.0517...`	0
−50	1		10	`-0.125`	0
−25	1	→ [Inverse DFT] →	15	`0.3017...`	0
0	1		20	`0.625`	0
25	1		25	`0.3017...`	0
50	1		30	`-0.125`	0
75	0		35	`-0.0517...`	0

One thing to note is that the imaginary coefficients are all zero. This can be understood by noticing that the 5/8-band filter is symmetric around zero. As we have seen earlier in the lab in our DSB mixers, this symmetry between positive and negative frequency is indicative of a real-valued function.

Since we are using a filter that only convolutes over 8 time steps, we have only fixed our filter's response at the 8 frequencies that we choose. However, the waveform entering the filter can of course contain arbitrary frequency values. To determine the filter's actual expected response as a function of frequency, we can "pad" the time domain filter with

zeroes to increase the amount of time it covers. This way, when we fourier transform our time domain waveform back into the frequency domain, we will have much higher resolution and will see what our filter looks like at frequencies in between the eight discrete points that we chose earlier. The dashed line in Figure 7 represents this response. This filter contains ripples that do not exist in the ideal filter due to the fact that our discrete inverse Fourier transform over only eight points is not capable of encoding such a high resolution in the frequency domain. To create a *perfect* square filter, one would theoretically have to create a filter that covolutes time steps starting from the beginning of time up until the end of time, and ain't nobody got time for that.

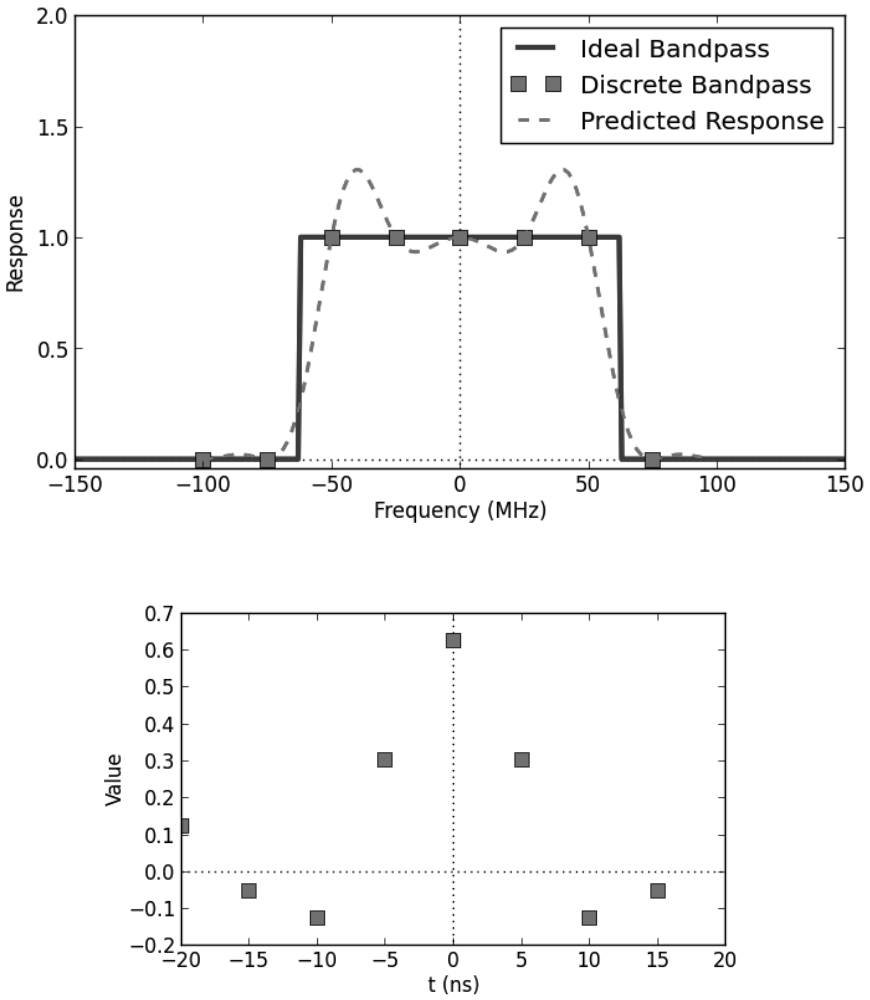

Figure 7: Top: Ideal bandpass filter with the 8 discrete values used in the FIR filter marked as squares. The actual response of a filter that uses only those 8 points is shown as a dashed line. Bottom: The inverse fourier transform of those 8 points. They follow the shape of a sinc function ($\frac{\sin x}{x}$).

PLOT DESIRED BANDPASS VS ACTUAL BANDPASS

4.2 Testing Filter Response

To test our FIR filter's response as a function of frequency, we want to compare how the power spectrum of a waveform looks with and without the filter applied. For purposes of this experiment, the response of the filter at a frequency ν is the ratio of the power at that frequency with the filter to the power at that frequency without the filter, $P_{filtered}/P_{unfiltered}$. For example, If the power spectrum has a large spike at a frequency 10 MHz without the filter, but a spike half as large at 10 MHz with the filter, then the response of the filter at 10 MHz is 0.5.

The way we can see the expected magnitude in the power spectrum is by using the FIR filter with all coefficients set to zero except for one set to the maximal possible value (a close to one as possible). This essentially reproduces the original signal as output because the FIR is simply convoluting the incoming signal with a delta function of magnitude 1—each point of the incoming signal is simply recreated in the output as it is "dragged" across the delta function filter.

To probe many frequencies in our range of -100 MHz to 100 MHz, we use the digital SSB mixing techniques of the previous section. The SRS Signal Generator can only produce signals up to 30 MHz; thus, to get larger signal frequencies we will mix our signal with LO frequencies that will produce upper and lower sidebands to span this range.

We use a 25 MHz signal at 0 dBm, mixed with LO frequencies ranging from 5 MHz to 50 MHz. Since we will combine real and imaginary parts programmatically, we can add them in both forms described by Eq. 7 to produce upper and lower sidebands.

After collecting these mixed waveforms with and without the FIR filter applied, we square the magnitude of the DFT of our convolved waveform to get the power spectrum. Each sideband that we create, enumerated in the upcoming table, represents a single datapoint at which the response is the ratio of the filtered spike's size to the unfiltered spike's size.

LO Freq. [MHz]	Upper Sidebands [MHz]	Lower Sidebands [MHz]
3.125	28.125, -21.875	21.875, -28.125
12.5	37.5, -12.5	12.5, -37.5
25.0	50.0, 0.0	0.0, -50.0
31.25	56.25, 6.25	-6.25, -56.25
35.9375	60.9375, 10.9375	-10.9375, -60.9375
37.5	62.5, 12.5	-12.5, -62.5
39.0625	64.0625, 14.0625	-14.0625, -64.0625
43.75	68.75, 18.75	-18.75, -68.75
50.0	75.0, 25.0	-25.0, -75.0

For each of these LO settings, we collect data twice—once with the coefficients for our 5/8-band filter, and again with all coefficients zero except for one (to normalize the output strength).

At each of the sideband frequencies, we take the ratio $P_{filtered}/P_{unfiltered}$ and get the results in Figure 8. They follow very closely to the predicted response, which is to be expected because the filtering and mixing was all done digitally. The fact that the response matches expectation well is still very apparent.

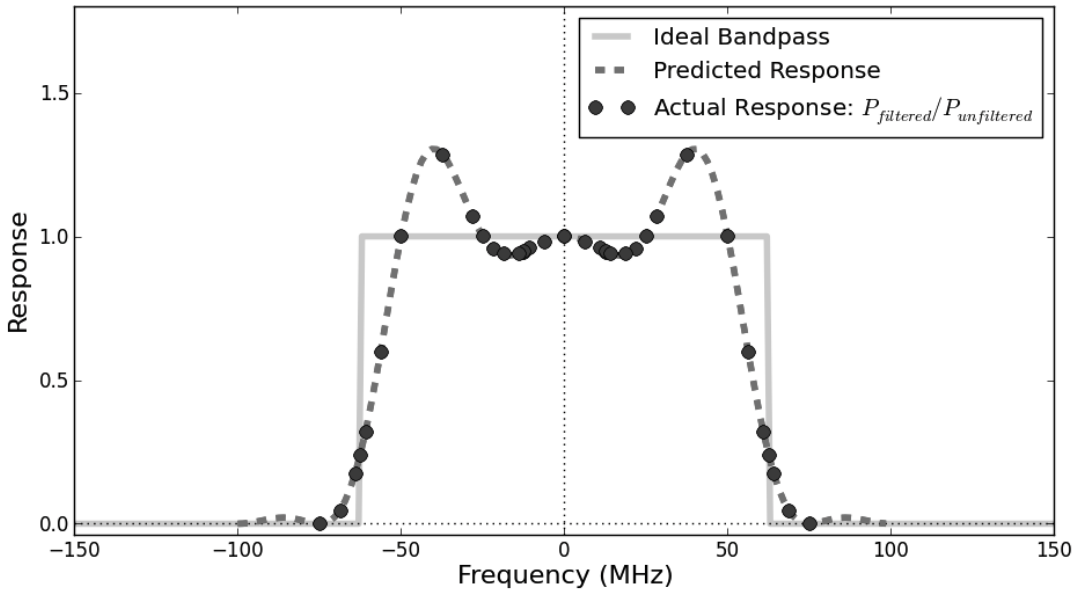

Figure 8: The response of our 5/8-band filter implemented using the 8-tap FIR filter, overlayed on the ideal 5/8-band filter in light blue as well as the theoretical response of the 8-tap filter in red.

Conclusion

I can conclude that I hate writing and I love sleep.

Acknowledgement

I would like to thank my lab partners Maissa and Sal for their work in getting us through this lab. Also thanks to the lab instructors Aaron Parsons, Karto Keating, and Baylee Bordwell for teaching us the lab material, helping us with debugging the ROACH, and sustaining us with baked goods. Especially to Karto for providing us with with the now defunct `set_srs`, which automagically just worked and made automating data collection so much easier! All figures were done in Python using the Numpy and Matplotlib libraries.

Appendix

Converting FIR Coefficients to Binary

When implementing a FIR filter, the FPGA reads 18 bit[†], fixed point, 2's complement numbers from the `coeff` registers, with 17 digits after the binary point. The smallest bit, therefore, will represent the value 2^{-17}, and the range of values that can be represented go from -1 to $1 - 2^{-17}$. A easy way to convert a decimal value into this odd representation is to divide that number by 2^{-17}, and then fill in the preceding bits with zeroes up to the full 32 bits.

This table shows the coefficients[‡] we wanted to use in our 5/8-band filter, as well as their binary representation in this representation and their hexadecimal representation.

Coeff.	Binary	Hex
`-0.0517...`	`1.11110010101111101`	`0x3e57d`
`-0.125`	`1.11100000000000000`	`0x3c000`
`0.125`	`0.00100000000000000`	`0x04000`
`0.3017...`	`0.01001101010000010`	`0x09a82`
`0.625`	`0.10100000000000000`	`0x14000`

[†]The register holds 32 bits but only 18 bits are used.

[‡]I truncated the decimal for sake of space. However, the coefficients were determined using `np.fft` with floating point precision; the binary and hex columns include as much precision as NumPy was willing to offer.