Digital Lab: Sampling, Fourier Transforms, Mixers, and Down-Converters

February 11, 2014

The purpose of this lab is to experimentally investigate digital sampling, digital Fourier transforms, and mixers. Mixers are the basis of heterodyne spectroscopy. Heterodyne spectroscopy, in turn, is what you use every day that you listen to a radio, use a cell phone, or watch TV—or do radio astronomy.

Most heterodyne applications use the more common double-sideband (DSB) mixer, for which the upper and lower sidebands produce identical mixer output. Here we also explore single-sideband (SSB) mixers, which display the remarkable property that upper and lower sidebands produce outputs at negative and positive baseband frequencies, respectively. In real life, SSB mixers allow such things as the transmission and reception of stereo FM signals; in olden days, FM transmission were all monophonic. Later in the course, we will use an SSB mixer for our first measurements of the 21-cm line of interstellar atomic hydrogen (HI).

In his lab, you will be performing several experiments, analyzing the data and generating a number of different data files. You will need to keep careful notes in your **lab notebook**! Or, pay the penalty, and forget what you did, do things twice, and be completely disorganized. Your choice!

Goals:

- Learn how to sample electronic signals—here, one or more sine waves—digitally using our computers.
- Become acquainted with the basic law of sampling: the Nyquist criterion.
- Learn how to use Digital Fourier Transforms (or Discrete Fourier Transform; DFT) to determine the frequency spectrum of a signal.
- Learn about the Fast Fourier Transform (FFT) as a particular, and particularly fast, implementation of the DFT.
- Learn the basics of mixing for frequency conversion (that's the *heterodyne* technique) and for measuring phase.
- Construct a two-output mixer, composed of two mixers, that can be operated as either a DSB or an SSB mixer.
- Use the two-output mixer in DSB and SSB modes and understand the difference.
- Learn how complex inputs to a FT break the negative/positive frequency degeneracy.
- Develop proficiency in Python, using it for the mathematical analysis, signal processing, and making nice plots.
- Learn enough Latex to write up your results in a formal lab report, including nice plots and graphs.

Schedule: As before, there's a lot to do in this lab! If you don't understand the Nyquist criterion by the end of the first week, you're behind. Here's how it should be:

1. Finish §1.1, §1.3, §1.4 below. Be prepared to show your results to the class, making real-time plots in Python.
2. Finish §2.1 and §2.2 below. Again, be prepared to show your results to the class.
3. Finish §3.1 and §3.2 Write your formal report! It should follow the standard format, consisting of an introduction, discussion of experimental activities and results, description of the analysis technique, presentation of analysis results, and discussion/interpretation. With all this, you should hand in a reasonable number of plots together with commentary to illustrate your work, your thought processes, and your conclusions.

1 Week 1: Sampling and Analog Mixing

Prerequisites

- Discrete Fourier Transform
- Convolution Theorem
- Nyquist Sampling

Materials

- 2 local oscillators
- digitizer (either ROACH or Pulsar sampler card)

1.1 Nyquist Sampling and Aliasing: A Single Sine Wave

Here we explore the all-important realms of the Nyquist criterion and aliasing in digital sampling. Clearly, if you sample too slowly the signal won't be well-reproduced. But if you sample really fast, then you generate large data files that take a long time to process. Just how slowly can you sample the signal without completely losing its basic properties (such as, for example, the fact that it oscillates with frequency ν_{sig})?

The fundamental parameter here is the ratio of sampling frequency ν_{smpl} to signal frequency ν_{sig}. With our equipment we can set ν_{smpl} to only selected, quantized values. However, we can set ν_{sig} with almost arbitrarily high precision. So to explore these issues we will pick a sampling frequency ν_{smpl} and take data at several signal frequencies ν_{sig}. Be sure to use a coax T so that you can look at the sampled signal on the oscilloscope. Set the peak-to-peak voltage appropriately so that it doesn't saturate the Analog-to-Digital Converter (known as the ADC).

1.1.1 Your First Digital Sampling

We want to explore sampling rate issues, so to that end we will begin by...

1. Pick a convenient sampling frequency ν_{smpl}.
2. Set the synthesizer to frequency $\nu_{sig} = (0.1, 0.2, 0.3, \ldots, 0.9)\nu_{smpl}$ and take data.

Take N contiguous samples with N an integral power of 2, say $N = 256$. Throughout the datataking, you should always be monitoring the signal with the oscilloscope. These are sine waves, so it's easy to measure the period by looking at the oscilloscope; each time you digitally sample the signal, you should write down the period (maybe in your lab notebook?).

For each dataset, plot the digitally sampled waveform versus time. In particular, make the digital plot informative, meaning that you can clearly see the signal shape; if necessary, plot only a part of the data so you can clearly see the signal shape (e.g., a few cycles of the sine wave); compare this with the oscilloscope trace. Also, for all the datasets derive and plot the Fourier power spectrum. Make sure that you *label the axes* with proper values of time and frequency—and choose convenient units, such as microsec and megaHz, to avoid huge and tiny numbers. In deriving the Fourier spectra, use our homegrown DFT procedure (see §1.3 below).

Now, look at both sets of these plots and note any funny business. Think about your results and draw your own conclusion: just what is the minimum sampling rate that you can get away with? (That's Nyquist's criterion).

1.1.2 Let's Go to Extremes...

By now you might have an idea of what's going on. Test yourself: try the following two experiments. But before analyzing the results, predict to yourself what they will look like. How? Use good old-fashioned paper and pencil to make some diagrams. If you can successfully predict the following, then you really understand the Nyquist criterion! So here we go:

1. Repeat the above for $\nu_{sig} = \nu_{smpl}$.
2. Now make $\frac{\nu_{sig}}{\nu_{smpl}}$ really large! In other words, blatantly violate Nyquist's criterion! Our oscillators won't run faster than 30 MHz, so to accomplish this you'll have to use not only a large ν_{sig} but also change ν_{smpl} to be very slow. Use as large as a ratio as you can, but make sure that the ratio is not an integral or half-integral number. Take lots of samples. Look at the sampled waveform. What do you get? Why?

1.2 For your Lab Report

In your report, select a well-considered set of plots to illustrate what you've learned, and compose a well-written commentary that convinces me that you really understand what's going on. Also clearly state what you have concluded regarding Nyquist's criterion.

With your plots, you can save paper by fitting multiple plots on a page by using Pylab's `subplot` function.

1.3 Using FFT's to Calculate a Power Spectrum

1.3.1 The Analytic Fourier Transform

The input to the Fourier transform is voltage versus time, say $E(t)$; the output is voltage versus frequency, say $E(\nu)$. The Fourier transform is the integral

$$E(\nu) = \frac{1}{T} \int_{-T/2}^{T/2} E(t) e^{2\pi j \nu t} dt \; . \tag{1}$$

The input voltage is real; it is multiplied by the complex exponential and integrated, so the output is complex. Of particular importance is that he Fourier Transform is invertible: you can go from the time to the frequency domain, and from the frequency domain you can get back to the time domain using the inverse transform

$$E(t) = \frac{1}{F} \int_{-F/2}^{F/2} E(\nu) e^{-2\pi j \nu t} dt \ . \tag{2}$$

Note: If you're paying attention, you would wonder how F and T are defined above. In the proper analytic formulation, they are both infinity. We emphasize their boundedness here because, in practice, i.e. when you do numerical calculations, neither can be infinity!

1.3.2 The Discrete Fourier Transform (DFT)

Our voltage versus time is not continuous, but rather it is discrete samples. With the digital transform, the integral becomes a sum. In this sum, you need to specify:

1. The set of sample times. I strongly suggest:
   (a) Using N samples, where N is even (and even better: a power of 2).
   (b) Use the center channel as the zero point. With N even, there is no center channel, so make the times run from $\frac{-N}{2}/\nu_{smpl}$ to $(\frac{N}{2} - 1)/\nu_{smpl}$.
2. The output is a function of frequency, so you have to specify the frequencies for which you want the output $E(\nu)$. I strongly suggest that, at first, you calculate the the output for N frequencies running from $-\frac{\nu_{smpl}}{2}$ to $+\frac{\nu_{smpl}}{2}\left(1 - \frac{2}{N}\right)$. This makes the frequency increment equal to $\Delta\nu = \nu_{smpl}/N$. Thus, you calculate a voltage spectrum running from $-\frac{\nu_{smpl}}{2}$ to about $\frac{\nu_{smpl}}{2}$ using our in-house DFT procedure.
   Later on, if you are intellectually daring and curious, try doubling or tripling the frequency range, keeping the separation $\Delta\nu$ the same (i.e., by increasing the number of output frequencies to $2N$ or $3N$).

$$P(\nu) = E(\nu)E(\nu)^* \ . \tag{3}$$

In Python, there are two ways to get this product. One is to use the `numpy.conj` function, i.e. `PF = EF * np.conj(EF)`. Should the imaginary part of `PF` be zero? (answer: yes! Why is this?) Is it? (answer: no! Why not?) To get rid of this annoying and extraneous imaginary part, you can use the `float` function: `PF = PF.astype(np.float)`.

The other (more convenient and suggested) way is to square the complex vector, i.e. `PF = np.abs(EF)**2`. The result is automatically real.

1.3.3 OPTIONAL: The Fast Fourier Transform (FFT)

Above in §1.3.2, you had N time samples and evaluated the DFT for N well-chosen frequencies. These were "well-chosen" because for these particular values of frequency—and only these particular values—you can get back to the time domain by using the inverse transform.

It so happens that, for these particular combinations of frequency and time, there is a very fast algorithmic implementation called the Fast Fourier Transform, the FFT. What do we mean by "Fast"? Well, normally when you do a DFT, you have N input numbers and N output numbers and the number of calculations $\propto N^2$. When N gets large, this takes a long time to calculate! For the FFT, on the contrary, the number of calculations $\propto N \ln_2(N)$, and this makes it possible to do large-N transforms.

If you have the time and energy, try Numpy's FFT and compare it to your DFT calculation above. The FFT output is ordered in what you might think is a funny and awkward way; however, it's really not awkward for most applications. See our "DFT's with DFT's" handout for details.

1.4 Leakage Power and Frequency Resolution

1.4.1 Leakage Power

Above, you calculated a power spectrum for each input signal at N distinct frequencies separated by $\Delta\nu = \nu_{smpl}/N$. In each, you found a spike corresponding to the input signal's frequency. Here, focus on just one of the properly-sampled signals ν_{sig}. Calculate the power spectrum for many more than N output frequencies over the Nyquist range $\left[-\frac{\nu_{smpl}}{2} \text{ to } +\frac{\nu_{smpl}}{2}\left(1-\frac{2}{N}\right)\right]$; i.e., make the frequency increment much smaller than $\Delta\nu = \nu_{smpl}/N$. Making the output frequencies closer together gives a more nearly continuous frequency coverage in the plot of the output spectrum. Turn up the vertical scale a lot to see if there is any nonzero power at frequencies other than ν_{sig}. You do see such power! This is Spectral Leakage. It affects all power spectra calculated using Fourier techniques.

Can you understand what's going on from a mathematical viewpoint?

1.4.2 Frequency Resolution

If you had two sharp spectral lines, how closely spaced in frequency could they be and still resolve them? Roughly, this is just the apparent width of the line when plotted against frequency. Look at the width of the line for your plot of §1.4 above. Compare this width to $\frac{1}{T}$, where T is the total time span over which the samples were taken. If you have the inclination, try taking other time series with varying number of samples (and thus, varying T) and confirm any relationship between line width (that is, frequency resolution) and T.

Can you understand this from a mathematical viewpoint?

2 Week 2

Prerequisites

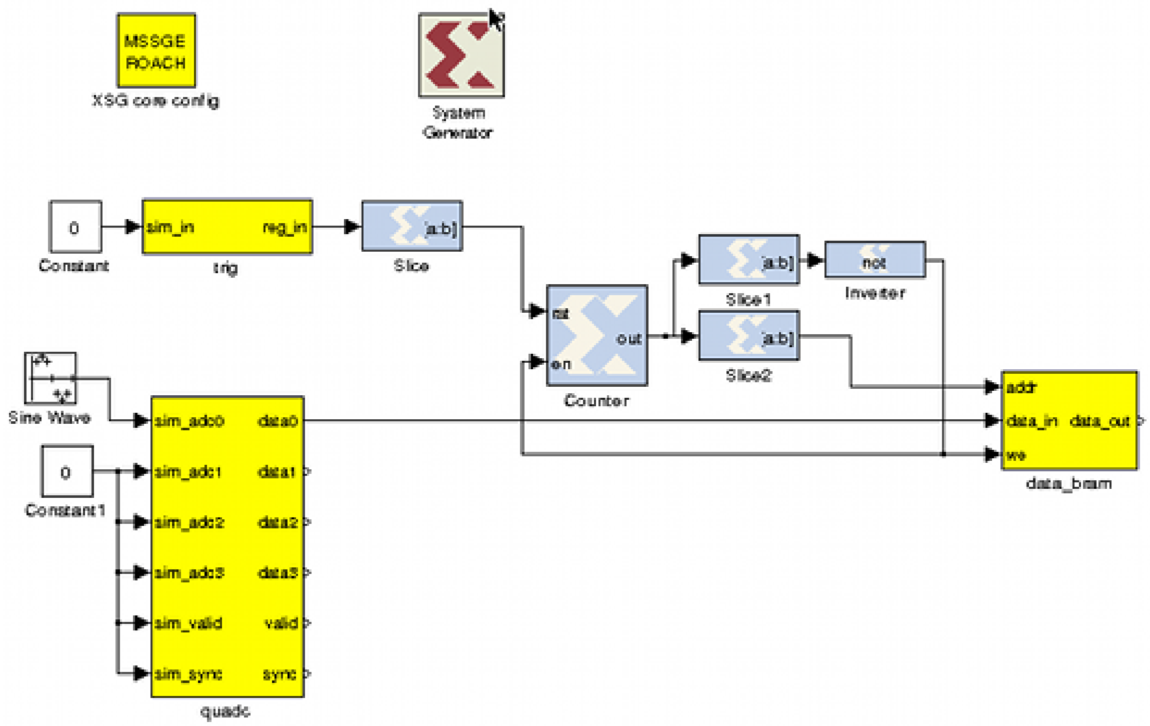
- Heterodyne Mixers
- Data Representations
- Digital Down Conversion

Materials

- 3 local oscillators
- digitizer (both ROACH and Pulsar sampler card)

Some Thoughts

This week we will mix some sine waves together to observe their beat frequencies. We will first do this in analog to learn a bit about heterodyne mixing, but then we will migrate the mixing part into a powerful digital processor, setting up the work next week, where we will control a down-conversion signal chain implemented entirely on this digital processor.



FPGAs and Simulink

We're just getting started on our digital part of the class, and we haven't really had a chance to talk about Field Programmable Gate Arrays (FPGAs) or how they are programmed — that's going to come next week. However, we're going to use one today, so let's cover what you need to know.

The ROACH

The Reconfigurable Open Architecture for Computing Hardware (ROACH) is an open-source board (e.g. all schematics, board design, etc. are free and open) developed by the CASPER collaboration, which was started here at Berkeley (by Dan Werthimer and Aaron Parsons) and is now pan-global. On this board is a Virtex-5 FPGA made by Xilinx, along with a CPU. The CPU runs linux (you can ssh into it), using a special linux kernel cooked up by CASPER that allows it to support automatic interfaces to some of the logic you program the FPGA with.

FPGAs are programmed with "bit files" that you load onto the chip, and tell the circuitry inside what wires to connect to what logic components. To load a bit file onto the ROACH FPGA, you execute a *.bof file from the CPU. The BOF file contains both a bit file, as well as some other information that allows linux to interface to the FPGA. After running a BOF file (leave it running), you need to find the process ID of that program. The linux command "jobs" should do the trick.

If you try to run a BOF file and receive a "Resource Busy" error, that means another BOF file is already running on the FPGA (FPGAs can't have multiple personalities simultaneously). That BOF file must be killed before a new one can be run.

Finally, if you go to a secret directory /proc/PID/hw/ioreg (where PID is the process ID), you'll find a bunch of files that have names that correspond to some of the blocks in the schematic diagram above. The schematic above is an actual FPGA program. It tells the FPGA to read samples from an interface to the quad-ADC (described below), and place them into a BRAM (block random access memory) until it is full. The address to which samples are written is supplied by a counter that stops itself at the final address of the BRAM, and waits for a signal to come in over the register "trig" to reset the counter back to 0 and allow it to write new samples in into the memory. You'll notice that "trig" and "data_bram" are files in the ioreg directory; those are your interfaces to the register that triggers a data capture, and the data itself that is written into the BRAM. You can write a value to the file "trig", and you can read values out of the file "data_bram", just as if they actually were files. But they are not. They are interfaces to the FPGA. But Python doesn't know that.

So all you need to know now is this: that "trig" is an unsigned 32-bit integer, and the least-significant bit controls the counter reset button. The BRAM holds signed 32-bit samples for each ADC sample. The ADC is natively an 8-bit sampler, and its sample clock is controlled by a function generator feeding the SMA port marked "clk". Happy sampling.

2.1 Basic Double Sideband (DSB) Mixer Operation: Upper and Lower Sidebands

For this, use two SRS synthesizer oscillators as inputs to a mixer to explore the spectra and waveforms in the DSB mixing process. The SRS synthesizers work up to 30 MHz. Assign one of the SRS synthesizers to be your "local oscillator" (LO) with frequency ν_{lo}, and the other your "signal" with frequencies $\nu_{sig} = \nu_{lo} \pm \delta\nu$. Here, you choose the frequency difference $\delta\nu$ and you set the two synthesizers, one to the lo frequency and the other to the signal frequency. There are two cases for the signal frequency, $\nu_{sig} = \nu_{lo} + \delta\nu$ and $\nu_{sig} = \nu_{lo} - \delta\nu$. Make $\delta\nu$ somewhat small compared to ν_{lo}, maybe 5% of ν_{lo}. For the input power level, a good choice is 0 dbm[1] for both synthesizers.

We will want to digitally sample the mixer output and explore both the sum and difference frequencies. As you learned in the Fourier lab, there are extremely important issues regarding sampling rate. The most basic is the Nyquist criterion. For this lab, we also want enough samples per period to give you a reasonable facsimile of the sine wave when you plot it; from this standpoint, it's not unreasonable to sample at twice Nyquist, or even faster. Another issue is the number of points you sample, which must be large enough to give you at least a few periods of the slowest sine wave.

From what you know about mixers, what is the fastest sine wave in the output? This, combined with our above comment and the upper limit on our sampling frequency (20 MHz for single channel, 10 MHz for dual channel), would determine the upper limit on ν_{lo}.

2.1.1 The Mixer

Combine the two signals, ν_{lo} and ν_{sig}, in a mixer for the two values of ν_{sig}. For the mixer use a Mini-Circuits ZAD-1, which has three BNC connectors (three *ports*) and works well at these frequencies. The ZAD-1, like nearly all mixers, has its ports labeled "R" (the "RF" or "signal"); "L" (the "local oscillator"); and "X" (the "mixing product") or "I" (the "intermediate frequency"). However, as we explain below, these labels are misnomers. They are based on the usual use for a

[1]What does this "dbm" mean? It's the power relative to 1 milliwatt, expressed in decibels (db). For our system the cable impedance is 50 ohms; what's the rms voltage for a signal with power level 0 dbm?

mixer, which is to take two high frequency signals as the inputs to the R and L ports and produce a low frequency difference frequency as the output at the I port.

The ZAD-1 is a balanced mixer, so the "R" and "L" ports are identical, and in particular will not couple to DC or very low frequencies. In contrast, the "I" port is coupled differently and will handle voltages all the way down to, and including, DC. The mixing process functions no matter which two ports are used as inputs. For example, if you are using a mixer to modulate a high frequency (say, a few MHz) with a low frequency (say, a few kHz), you should use the "I" port for the low frequency and either of the other two for the high frequency; take the output from the third port.

We will want to look at the output, which consists of both the sum and difference frequencies, so choose the ports appropriately. Digitally sample the mixer output for both cases ($\nu_{sig} = \nu_{lo} \pm \delta\nu$).

2.1.2 For Your Lab Report

For the two cases, plot the power spectra versus frequency. Explain why the plots look the way they do. In your explanation include the terms "upper sideband" and "lower sideband".

For one of the cases, plot the waveform. Does it look like the oscilloscope trace? Also, take the Fourier transform (not the power spectrum) of the waveform and remove the sum frequency component by zeroing both the real and imaginary portions (this is "Fourier filtering"). Recreate the signal from the filtered transform by taking the inverse transform; see §2.1.3 to see how this is done. Plot the filtered signal versus time. Explain what you see.

2.1.3 On Fourier Filtering

When you use DFT to go from the time to the frequency domain, you specify the times and the sampled voltages as input and calculate the output for a well-chosen set of frequencies. These times and frequencies should be symmetric around zero, as we strongly suggested above in §1.3. To filter out the high-frequency mixed signal, you have to zero both the real and imaginary high-frequency components, and you must zero both the negative and positive frequencies. In frequency space, these zeroed values must be symmetric.

To go back from the frequency domain to the time domain, use the filtered frequency Fourier components (which are complex) and their associated frequencies as inputs and calculate the output for the original times. You should also use `numpy.ifft`, which keeps the amplitude scale correct (check the documentation in IPython). The output will be a time series, and because you've eliminated the sum frequency component, the only thing that remains should be the difference frequency component.

One more thing. The output of the inverse transform had better be real—after all, your original input was real! You'd better check this! If it isn't real, then either (1) you didn't treat the negative and positive frequencies symmetrically when you zeroed the signal, or (2) you didn't use our suggested input times and output frequencies. If you're having trouble, check your basic technique by doing the inverse transform on the non-filtered Fourier components; you should recover the original time samples.

2.2 Digital Mixing (DSB and SSB) on an FPGA

Now we are going to do the exact same thing as in §2.1, but digitally.

First, let's make sure we can interact with the ROACH board (see "Some Thoughts", above) to digitize an analog signal and read out the digital data. In contrast to the CPU-hosted ADC used in Week 1, this week we will need to acquire data directly from the FPGA hosted on the ROACH. This entails:

- connecting the signal to be digitized to port 1 of the ADC card (please keep signal amplitudes around -10 dBm)
- connecting the sample clock to the clock input of the ADC card (choose an appropriate sample rate)
- logging onto the ROACH board
- launching the BOF file for the sampler/mixer FPGA design
- writing the correct values to the file interface to the "trig" register
- reading the binary data out of the file interface to the "adc_bram" memory onto another computer where you will do your analysis

Once you have acquired some data, write a Python script that interpreting the binary data as numbers using the correct binary format and endian-ness and plots your data to make sure it makes sense. So... convince yourself that your data makes sense. What is the time interval between each sample? What is the signal amplitude relative to full-scale for the ADC?

Once you understand your data, connect the output of your mixer from §2.1 to the same ADC input and collect some more data. Interpret the resulting waveform and decide if it makes sense. Optionally, you might compare the samples you collect on the FPGA to those obtained from the CPU-hosted ADC.

Now here's the exciting part. Ports 2 and 3 on the ADC card are also digitized, but then are multiplied post-digitization before the result is written to "mix_bram" (triggering the data capture is still done through "trig"). Now you can take each of the signals that were mixed together in analog, connect them to Ports 2 and 3, and compare the digitally mixed output to the result you obtained with an analog mixer.

Finally, instead of mixing two input signals digitally, we will mix a single signal with an LO that is derived from the sample clock. In this case, connect the signal that we want to mix with to Port 4 on the ADC card. Next, before triggering to acquire your data, write a carefully chosen value into the file interface to the "lo_freq" register. This register maps the 0 to 2π interval of an oscillator to addresses from 0 to 255. For each ADC sample that comes in, the number stored in "lo_freq" is added to the current address (wrapping around to 0 after 255), and the sine/cosine evaluated at the corresponding point between 0 and 2π is output for mixing with the incoming signal. The result, as many ADC samples come in, is that the input waveform is multiplied with sine/cosine waveform of a frequency that is determined by "lo_freq". Choose an appropriate value to write in this register, and be sure to record what the corresponding frequency of your digital LO is.

Now you may trigger your data capture. The result of mixing the input signal with a cosine wave of your chosen frequency is stored in "cos_bram", while the output of mixing with a sine wave is recorded in "sin_bram". Use the data stored in each of BRAMs to reconstruct the *complex valued* sinusoid that this digital Single-SideBand (SSB) mixer outputs. Compare this recorded waveform to what you expect for a SSB mixer, given the frequency of the LO and the frequency of your input signal.

2.2.1 For Your Lab Report

Plot the power spectra versus frequency for the digital and analog DSB mixing cases. Explain why the plots look the way they do. Identify key features. For the digital mixing case, plot the waveform.

Also plot the power spectrum of the output of the digital SSB mixer. As before, explain why the plot looks as it does, and in particular, explain the difference between the SSB mixing case and the DSB mixing case that you explored. Also plot the waveform (both real and imaginary components on one plot), and determine whether the waveform has a positive or negative frequency.

In all cases, make sure that the frequency/time axes have appropriate physical units (Hz, μs, etc.). It is not acceptable to just have these axes as sample counts.

Finally, explain some of the advantages and disadvantages of DSB and SSB mixers for both digital and analog implementations. In comparing digital and analog implementations of SSB mixers, consider what the effects would be if the sine/cosine components were slightly out of phase with each other (you can simulate this to find out). Which case would be more likely to have such phase errors?

3 Week 3

Prerequisites

- Synchronous and Asynchronous Logic
- Processor Architectures
- FIR Filters

Materials

- 1 local oscillator
- ROACH board
- broad-band noise source
- 1 analog filter

3.1 Coefficients for an FIR Filter

As you should have seen in lecture videos, Finite Impulse Response (FIR) filters are able to implement frequency-domain filters with a tunable shape that is determined by the coefficients that they convolve with an input waveform. Also, now that you've had a bit of a preview of the capabilities of FPGAs, and how they function, you might appreciate how their architecture makes them particularly adept at implementing FIR filters, which have a natural pipelined flow to them.

In the first part of this lab, we will choose the coefficients for an FIR filter that is running on the ROACH as part of a larger Digital Down-Converter:

- Choose coefficients that implement a 5/8-band filter (a filter that, if you divided the band into 8 complex channels, would extract the 5 channels centered around 0):

  Choose the frequency-domain response you want for your filter (i.e. the function you would like to multiply the frequency spectrum of your noise by to "filter" it)

  Compute the real-valued, time-domain coefficients that implement that filter, keeping in mind that:

Multiplication in the frequency domain is a convolution in the time domain

The Fourier transform of a real-valued signal has the property that: $\hat{f}(\omega) = \hat{f}^*(-\omega)$

You will want to think carefully about where the 0 frequency bin is

The FFT puts negative frequencies after positive frequencies in your array. Similarly, when you take the inverse, it will put negative times after positive times. When implementing your coefficients in an FIR, though, negative times *have* to operate on samples that arrive before positive times.

- In what order are you going to write these coefficient into the software registers of your FIR filter?
- Plot the predicted filter response for your coefficients at a frequency resolution much finer than an 8-channel DFT produces. The best way to do this will be to add more time-domain samples to your coefficients. We don't want to introduce any new signal, though, so just add zeros to pad your coefficients out to 64 samples, and then transform the result back into the time domain.
- Why aren't the passband and stopband flat?

3.1.1 For Your Lab Report

Produce a table that indicates the (floating point) coefficients you intend to use. Next, we'll need to convert these coefficients to fixed-point numbers that we can write to the FPGA. Assuming that software registers will be interpreted on the FPGA as 32-bit signed integers with 31 bits after the binary point, what binary values will you use to represent these coefficients?

Also, plot the desired (ideal) bandpass of your filter, along with the actual response that (as determined above) is not quite what you ideally wanted. Explain why this filter deviates from your ideal. Is it possible to perfect this filter?

3.2 Putting it all together for a Digital Down-Converter

Now you are going to use your coefficients to program the coefficients of an FIR filter running on the FPGA which has been chained onto the end of the digital mixer from the previous week. You can now control the LO and the shape of a filter that follows to implement a Digital Down-Converter (DDC) that mixes and filters an input signal according to your every whim.

What we'd like to do now is to empirically characterize the shape of the filter you have implemented. One way to do this is to input sine waves of a fixed amplitude and different periods and use the amplitude of the recorded waveform to determine the filter response at that frequency. However, a faster way to do this is to input a noise source (which has noise at a range of frequencies), record the output noise, and use the output amplitude of the noise at each frequency to determine how much the input signal was attenuated at that frequency.

So set up a noise source input to the DDC on Port 1 of the ADC. Program your FIR coefficients and choose an appropriate LO frequency. Trigger a capture. On the BRAM entitled "input_bram" will be the input noise, recorded immediately after it was digitized. On the BRAM entitled "ddc_bram" will be the filtered noise. Use the data in both of these files to determine the attenuation between the input and output signal as a function of frequency. This will give you, empirically, the shape of your FIR filter (and its placement in the band as a result of your LO).

To reduce the noise in your estimate of the filter shape, it helps to average over time. There are a lot of data-points recorded in the BRAM. You don't need to do an FFT over all of them—that's more frequency resolution than you need. Instead, compute the power spectrum using the first N points, and average that spectrum with the power spectra you meaasure on each of the next N-point blocks of samples, until you run out of samples. The averaged power spectrum you compute will not have as much frequency resolution, but it will have a lot less noise in each measurement.

3.2.1 For Your Lab Report

Compare your empirically determined filter shape with the (non-ideal) filter shape you calculated in the previous section. Are they the same? If not, you should figure out why not. This is a digital system, so you can expect exactness, down to the level of the residual noise.

Speaking of residual noise, what do you expect the noise in your estimate of the filter shape to be? How many samples did you average in each frequency bin? What was the standard deviation of samples prior to averaging? Can you use those two bits of information to calculate the error in each measurement?