LAB 1: Exploring Digital Sampling, Fourier Filtering, and Heterodyne Mixers

Contents

1. Introduction

In this lab, we experimentally investigate digital samplers, discrete Fourier Transforms, and mixers. These are tools of the trade in radio astronomy, and you use them every day when you use a cell phone, listen to the radio, or watch television. Did you know that WiFi was invented by a radio astronomer?

To get firm footing in the methods of radio astronomy, you will perform a series of experiments illustrating the utility and hazards of these various techniques. You will configure equipment in the lab, acquire and analyze data, and store and illustrate the results. The signal processing pipeline you build in this lab will be re-used in Lab 2 to observe 21-cm line emission from hydrogen in the Milky Way, so put in the time to understand it and make it work. The effort you invest now will pay dividends later in the class.

This first lab is an excellent time to establish good research and laboratory practices. Remember that your grade is determined by the report you turn in three weeks from now. All of the exercises that follow are aimed at building your understanding and giving you the raw materials (and data) from which to build that report. You don't have to finish everything or proceed in the order presented, but you do have to write a quality report that addresses the goals in §2.

Here are some specific suggestions for turning these lab instructions into a successful report:

- Keep notes in a **lab notebook** as you hook things up. Good notes document the experimental setup corresponding to your data with sufficient detail to allow you to describe it accurately and track down mistakes. If you show your knowledgable GSI a head-scratcher of a plot but cannot remember which port of the mixer you plugged the noise into, how can they diagnose your problem?
- Back up (or revision control) your data, code, and plots. Nothing is more demoralizing than redoing work or being unable to reproduce a key result the night before a deadline.
- Start your report early. If you plan for which equipment, data, and plots you need for your report, you will spend your lab time more efficiently and avoid panic later. Write descriptions and captions as you go to save yourself the existential horror of the blank page.
- Formal lectures, the pages linked on AstroBaki, and these lab instructions are all resources for helping you write a strong report. As in real-life research, there is no one textbook containing everything you need to know. Explore, take notes, and show me what you learned in your report. Remember, I can only grade you on what you write down. Make sure what you write down is your own work, in your own words. Cite sources external to this class (no need to cite AstroBaki, lecture, or the lab instructions).
- Your reports will be in the style of scientific publications. They are narratives of explanation and discovery that bring the reader—a science colleague—to the point of understanding and having confidence in your results. You are presenting your work, but this report is about the *reader's* journey, not yours. It is a presentation of synthesized results with the background and salient details necessary to understand them. It is *not* a play-by-play of everything you did, all the plots you made, and all of the wrong turns along the way.

2. Goals

The goals of this lab—the results your report should demonstrate you have achieved—are as follows.

- Sample electronic signals and convert them into digital signals. Demonstrate the phenomenon of aliasing and quantitatively relate your results to the Nyquist criterion.
- Correctly use and display Discrete Fourier Transforms (DFTs) to determine the frequency power spectrum of time-ordered voltage data. Correctly calculate and label frequency and power axes in plots and demonstrate understanding of how frequency ranges and resolutions are determined by the duration and cadence of samples in a time series.
- Demonstrate understanding of negative frequencies. Motivate and measure how complex inputs to a Fourier Transform break the positive/negative frequency degeneracy.
- Identify and characterize noise in electronic measurements and explore how it behaves under the Fourier Transform.
- Apply the convolution/correlation theorem to explain spectral leakage in power spectra and test the relationship between power spectra and autocorrelation functions.
- Apply heterodyne mixing for frequency conversion and quantify how electronic mixers differ from ideal ones. Construct double- and single-sideband mixers and use measurements to demonstrate the theoretical and practical differences in their operation.

Your report will be judged on the basis of your presentation of the theoretical background, experimental setup, data analysis, and quantitative interpretation of results for each of these areas. Along the way, you will develop many new technical skills which need not be written about explicitly in your report; the quality of the report itself will demonstrate your growing proficiency. These technical skills include:

- configuring laboratory equipment and navigating the Linux computing environment on your Raspberry Pi,
- using the Python programming language for experimental control, data acquisition, analysis, signal processing, and plotting, and
- writing a technical report, including an abstract, captioned figures, tables, and citations, typeset in LaTeX.

3. Schedule

This is an ambitious lab with a steep learning curve. Do not to get behind or backload your work schedule. Writing the report takes a lot longer than you might think.

1. *Week 1.* Finish §5 and read the accompanying material in §6. *Be prepared to show work, software, and results to the class.*
2. *Week 2.* Finish §7 and read §8. Again, be prepared to present to the class.
3. *Week 3.* Read the handouts in §9 and write your formal report. Refer to our handouts and linked material for tips on how to structure a scientific paper. You will be graded on the degree to which your report addresses the specific goals in §2 and conforms to standards of a quality scientific publication.

4. Software Engineering

The programming required to complete these labs increases in scale throughout the semester. As it does, you will need to learn to organize, document, test, and stabilize your code. The process of building sound code (software engineering) may be one of the most marketable skills you can take

from this class. Learning to engineer good software can take a lifetime, but let us start with a few principles for this lab.

- *Package your code.* Many students these days only have experience coding inside of a Jupyter notebook. Jupyter, while a fantastic resource for developmental coding, has many drawbacks when it comes to software engineering, particularly as you start changing code and running cells out of order. Build infrastructure code (e.g. functions and classes) and store it in **modules** and **packages**[1]. Python modules are simply text files containing code. Put them in a directory and add a `setup.py` file out front, and you have a package. Modules separate the definition of the code from the scripts and notebooks used to execute code with particular parameters. Packages bundle modules, scripts, and tests into installable bundles. `numpy`, `scipy`, and `ugradio` are all examples of packages. Packaging code vastly improves organization, testability, and code reusability.
- *Code (at least) twice.* It is nearly impossible to write well-organized code while figuring things out for the first time. You need room to hack at the problem, to try and fail, to learn and iterate. Code once to figure out how your program should work, then re-code it to get the organization and interfaces right. Good software engineers think carefully about interfaces (the boundaries between functions, classes, and modules, along with the information that flows across them). Time spent organizing and testing code pays dividends later as you are debugging.
- *Test your code.* How do you know your code is doing what it should? Running without raising an exception doesn't necessarily mean it did what you intended. In software as in science, you can only *expect* what you *inspect*, which is to say that you should only trust something work to the extent that you have tested that it does so. Once you write a test, keep it with your code so that if you change your code, you can verify it still works. Modular testing (a.k.a **unit testing**) is the key to success in large software projects, particularly ones involving multiple people. The `unittest` module is commonly used in Python, but other modules have their own merits.
- *Control your revisions.* **Revision control** uses a specialized program (`git` is popular) to track the changes made to your code. They allow you to edit code with impunity, knowing that you can always backtrack to a previously committed state of the code if you screw something up. Open a (free) GitHub account, initialize a project for this class, and commit to it regularly. GitHub works with `git` to back up your work remotely, and it also allows you to easily synchronize your work across multiple computers, and with collaborating members of your group. It is 100% worth the learning curve.

Advanced software engineering combines the steps above, using online repositories to hold branches of a software project. A community of coders push their changes, a continuous integration system runs unit tests on the code to make sure nothing breaks, that new code is covered by tests, and that all the code conforms to reasonable documentation and legibility standards. Code reviews examine changes proposed in pull requests, after which changes are merged into the main branch of the project.

We don't need this advanced machinery for this class, but it is good to be aware of how coding communities work and to start down the path of good coding practices yourself.

[1] See `https://docs.python.org/3/tutorial/modules.html`

At a minimum, firmly separate data acquisition (code controlling equipment in the lab and at the telescope) from data analysis. For data acquisition, scripts that run from the command line are most appropriate. For data analysis, `jupyter` notebooks are a fantastic tool. For code needed in scripts and notebooks, write modules that you can import into either.

5. Digitally Sampling a Sine Wave (Lab Activity, Week 1)

5.1. Handouts and Software

Your work this first week will require you to navigate the Linux operating system, using a (Vi/Emacs/3rd party) text editor to write Python. On Astrobaki[2] we have linked primers on Linux/Unix, Python Installation and Basic Programming, Unix Text Editors, and Revision Control. You can also review the key course content for this week: *Nyquist Sampling* and *The Fourier Transform.*

5.1.1. The `ugradio` *Python Package*

As mentioned above, modules and packages help us distribute and reuse Python code. For this class, we will use the `ugradio` package to provide supporting code for your labs. This package is already installed on the SD card we provide you for your Raspberry Pi (you can test this by opening `ipython` or a `jupyter notebook` and typing `import ugradio`). If you need to install it on another computer, all of the code is on GitHub and linked to from the course website.

For the activities below, you will want to use several modules inside the `ugradio` package, particularly `ugradio.sdr`, and `ugradio.dft`. Take a minute to browse the code in these modules, which you can do on GitHub, in the `ugradio/ugradio_code/src` directory on your Raspberry Pi, or within Jupyter/IPython using `??` after any module, function, or class name.

- `ugradio.sdr` — provides code for interfacing to the Software-Defined Radio (SDR) receiver attached to your Raspberry Pi.
- `ugradio.dft` — provides code for doing arbitrary Discrete Fourier Transforms. This contrasts the functionality of `numpy.fft`, which can only do certain kinds of Discrete Fourier Transforms, but can do them very fast.

5.2. Digital Sampling and the Nyquist Criterion

In class, we learned about the Nyquist criterion for sampling discretely in time. Sampling too slowly results in a signal being aliased, but oversampling is inefficient and often impossible because of hardware limitations. Investigate the phemenon using an SDR module connected to a Raspberry Pi, and numerical simulations in Python. In both cases, try different ratios of signal frequency (ν_0) and sampling frequency (ν_s) and compare the result to theoretical expectations.

Remember, your goal is to write a section of your report on aliasing that includes theory, data analysis, and (optionally) numerical simulation. This is best done by posing and quantitatively answering a question. Generally, the more specific the question, the better your report will read. For example, "*what is aliasing*" is too broad. "*How many Nyquist zones can I demonstrate using my RPi SDR module*" is a tighter framing with a quantifiable answer that you can provide evidence to

[2] `https://casper.astro.berkeley.edu/astrobaki/index.php/Undergraduate_Radio_Lab`

support. Even better would be "*using (purposeful) aliasing, can I characterize the spectral response of my SDR module?*" Feel free to pose and answer your own unique question.

5.2.1. Sampling with an SDR and a Raspberry Pi

We assume you have followed the instructions on AstroBaki for *Setting Up Your Raspberry Pi*. If you imaged your SD Card from a lab copy, you should be good to go. Connect a keyboard, monitor, and power cable, bench, and plug in an SDR module (e.g. the Nooelec NESDR SMART) into your RPi's USB port and pull up a terminal on your RPi. Remember to update and reinstall `ugradio` periodically to ensure it is up to date. To do this:

1. `cd /ugradio/ugradio_code`
2. `git pull`
3. `pip install .`

For a fun demo, hook up a cheap antenna to the SDR SMA input, run `gqrx` at the terminal, press the "Play" button, and then fiddle with the settings (e.g., use Hardware Automatic Gain Control, WFM Stereo demodulation). With audio output hooked up, you should be able to listen to Frequency Modulated (FM) radio by setting an appropriate center frequency (in the US, FM spans 87.8–108 MHz).

Listening to FM radio broadcasts is fun, but we need to figure out what is actually happening between the antenna and the output. The SDR modules we have chosen for this lab are based off of two chips: the R820T2 (a tuner chip), and the RTL2832U (a digital sampling, filtering, and demodulation chip). These SDR modules are quite versatile and can be coaxed into doing all sorts of useful things. For starters, we would like to bypass most everything and just use the analog-to-digital (ADC) sampler inside the RTL2832U. This is called **direct sampling**, and is one of the modes supported in the `ugradio.sdr` module.

Using a function generator, generate a sine wave around $\nu = 500$ kHz with a $V_{\rm PP} = 100$ mV peak-to-peak voltage and send it to the input of the SDR module plugged into your RPi. We suggest using a co-axial T joint so you can put a copy of the signal you are sampling into an oscilloscope. Good science is founded on inspection, verification, and documentation. "Mistrust everything," is one of the mantras of this class. Inspect the signal you are sampling with the oscilloscope, verify that it matches that amplitude and period you set on the function generator, and document the settings used.

Now, using python on your RPi (either interactively with `ipython` or by writing a script), make an `SDR` object (e.g. `sdr = ugradio.sdr.SDR(...)`) and use `sdr.capture_data` to collect some data and plot it. You can set the `sample_rate` in the range of $\nu_s = 1.0$–3.2 MHz. Experiment with various combinations of ν_s and input frequency and see what frequency sine waves you observe in your data.

You will notice that, for a fixed ν_s, the amplitude of the sine wave you measure will change for different values of ν input. This happens as a result of some filters inside of the RTL2832U chip. A major role of these filters is to suppress aliasing! In other words, these SDR modules are smarter than we want them to be. In order to better illustrate aliasing, it helps to override the default filter, which took your intrepid instructor some amount of hacking (and bugfixing in the SDR driver!) to accomplish. Suffice it to say, your ability to see aliasing will be dramatically improved if you provide `fir_coeffs=np.array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,`

`0, 2047])` as an argument to your `SDR` object. (That is an array of length 16, with the last number set to $2^{12} - 1$) [3]

For a given sample frequency, explore different values of ν, using aliasing to probe different Nyquist zones to examine the filter response of the RTL2832U. Document the approach you are using, the hardware involved (brand, part number), and the time of signal acquisition. If you want, you can also explore the difference in aliased amplitudes you get for `fir_coeffs=None` (default) versus the above suggested value for enhanced aliasing.

Once you have acquired an array of data, you should save it to a file with `numpy.savez`. Your analysis scripts can then load data from this file with `numpy.load`. These files represent the interface between your data acquisition code and your analysis code, and good design often boils down to good interfaces. In this case, that could mean adding a bunch of meta-data to your `numpy.savez` call (it accepts multiple arrays/scalars) so that you have all the information you need for analysis and presentation sitting right there with your data.

For each dataset, use the `matplotlib` package (typically imported as `import matplotlib.pyplot as plt` to plot the digitally sampled waveform versus time. For plotting, it is easier to visualize fewer samples, so if you load a waveform from file into a variable `data`, you can slice off the first, say, 200 samples with the command `data = data[:200]`. Make the plot informative by putting the x axis in time units (e.g. `plt.plot(times, data)` instead of `plt.plot(data)`). Does the period match what you expected?

If you would like to use this (or any) plot in your report, make sure you label both axes (with appropriate units to avoid tiny or huge numbers) give it a title, and restrict the range to an appropriate scale where you can clearly see the signal shape and frequency. You will also need to come up with an appropriate caption that stands alone as a description of the plot and what it demonstrates.

For each of the datasets you take, derive and plot the Fourier power spectrum (e.g. the square magnitude of the voltage spectrum; see §6.2 and §6.4). You can use `numpy.fft` to do the Fourier transform, but feel free to use our homegrown `ugradio.dft` version until you have mastered dealing with Fourier units and the subtleties of `fftshift`.

Now, examining the results, draw your own conclusion: what is the minimum sampling rate ($\nu_{s,min}$) that accurately reproduces the spectral frequency ν_0 in the sampled data? That is **Nyquist's criterion**.

5.3. Voltage Spectra and Power Spectra

Power spectra tell us about the frequency content of our signal with the magnitudes of the complex numbers in the Fourier transform. Voltage spectra, by contrast, give us phase information, with real and imaginary parts.

What does it mean, that the voltage spectra are complex? What do the real and imaginary parts represent? Is the imaginary part less 'real' than the real part? What does it mean, for frequencies to be negative versus positive? These are questions your lab report could try to answer.

[3] If you are curious, I expanded the `librtsdr` driver and `pyrtlsdr` Python wrapper to support setting the coefficients of a Finite Impulse Response (FIR) filter. FIR filters implement convolutional filters, which we will learn about next week.

For one of your data captures in the previous section, plot the real and imaginary parts (e.g. `spec.real` and `spec.imag`) of the voltage spectrum on the same panel in different colors. Do the plotted points exhibit any symmetry between negative and positive frequencies?

Repeat this process on independent data streams to ensure the result is not a fluke. When you compare the plots for several independent data captures of the same sine wave, do the voltage spectra repeat identically? Why not? What is happening when sometimes the real portions are positive or negative? When the imaginary portions have more amplitude than the real ones?

For the power spectra, repeat this symmetry examination and the test for repeatability. What kind of symmetry do the power spectral points exhibit? Why might we use power spectra instead of voltage spectra, and vice versa?

Choose a power spectrum and take its inverse Fourier transform. For this to work, you need to make sure `dft.idft` correctly infers the frequencies corresponding to each bins in your power spectrum array, Separately, calculate the autocorrelation function (ACF) directly from the voltage time series manually with `dft/idft`, with `numpy.correlate`, and with `scipy.signal.correlate`. According to the correlation theorem, the Fourier transform of the power spectrum should equal the ACF. Does it? Explain any differences.

5.4. Leakage Power

By default, `dft` calculates a power spectrum at N frequencies for a signal with N time samples, each frequency sepearated by $\Delta\nu = \nu_s/N$. (For Fast Fourier Transform (FFT) operations like `numpy.fft`, this correspondence between time and frequency samples is hard-coded.) For some of the waveforms you captured above, calculate the power spectrum with $N_{\rm freq} \gg N$, contrary to the recommendations in §6.2. Use `dft` and choose frequency increments much smaller than $\Delta\nu = \nu_s/N$. Use a logrithmic vertical axis to see if there is nonzero power at frequencies other than ν_0. This is evidence of **spectral leakage**, which affects all power spectra calculated using Fourier techniques.

Can you explain mathematically why you might find power at $\nu \neq \nu_0$ using a Discrete Fourier Transform?

5.5. Frequency Resolution

If you had two spectral lines (sine waves), how closely spaced in frequency could they be and still be resolvable? Investigate this experimentally by combining the output of two function generators in a power splitter. (A splitter run backward is a combiner.) Use two frequencies very close together and plot the power spectrum again using points more closely spaced in frequency than the $\Delta\nu = \nu_s/N$.

How close together can the two frequencies be for you to still distinguish them? This is called the **frequency resolution.** How does it depend on the number of samples used in the DFT? In particular, how does it compare to time interval that those samples span?

Can you explain your findings mathematically?

5.6. Nyquist Windows

Above, we calculated spectra for frequencies in the range $\pm\nu_s/2$. What happens if we increase this range?

Explore by taking a Nyquist-sampled time series and calculating the power spectrum for a much larger frequency range, $\pm W\nu_s/2$, where W is at least 4, centered on the original frequency interval. Each value of W gives you a spectrum in a different **Nyquist window**. How do the spectra in different Nyquist windows compare? Note that, for $W > 1$, you are calculating power spectra for frequencies that violate the Nyquist criterion, yet the results are sensible, in their way. In Lab 4, we will use a digital spectrometer that samples the 12^{th} Nyquist window.

This shows that the strictly correct statement of the Nyquist criterion is that the bandwidth—the frequency range of the signal *including negative frequencies*—must not exceed ν_s. For the first Nyquist window this is equivalent to the simpler statement of the Nyquist criterion we first explored.

5.7. Fourier Transforms of Noise

Noise comes into our data from a variety of sources. Here, we restrict the term noise to refer to random fluctuations with a known statistical distribution (in our cases, this is almost always a Gaussian with mean zero). We are careful to distinguish this from such things as: systematic contributions to error, bias, ambiguity, and posterior likelihood. Unlike how we use the term colloquially, noise has a specific meaning in science, so be careful with it.

Many of the astronomical signals we observe are, in fact, noise. Blackbodies are noise sources: a blackbody at temperature T emits statistically random electromagnetic waves with specific intensity (power per area per Hz per solid angle) given by the Planck function

$$B_\nu = \frac{2h\nu^3}{c^2}\frac{1}{e^{h\nu/kT}-1}. \tag{1}$$

As radio astronomers, we operate in the regime where $h\nu/kT \ll 1$ (the Rayleigh-Jeans limit), so the blackbody formula simplifies to:

$$B_\nu \approx \frac{2kT}{\lambda^2}. \tag{2}$$

Notice how the noise power depends linearly on T.

For a number of good reasons, radio astronomers choose to measure noise power in units of temperature and define a brightness temperature T_B such that $I \equiv 2kT_B/\lambda^2$ for an observed specific intensity, even if the radiation source (for example, galactic synchrotron emission) is not even remotely thermal.

Using this framework, all sources of power in our measurement can be related to a temperature, including the receiver noise that is added to our incoming astronomical signal by the thermal motion of electrons in our amplifiers and resistors (otherwise known as **Johnson** noise). This noise (which, by the Central Limit Theorem, is almost always Gaussian) has a variance that is characterized by receiver temperature, T_{rx}, and adds onto the brightness temperature, T_B, that characterizes the variance of signal coming through the antenna feed of the telescope.

For calibration and testing, we have laboratory sources of noise. Below, we outline several options for exploring the properties of digitally sampled noise.

5.7.1. Option 1: Use the Laboratory Noise Generator

We have a noise generator in the lab that uses a diode to generate broadband, statistically random voltages. In order to avoid aliasing, you should connect a filter to the output to limit the noise to a band that you can sample either with the SDR module on your RPi. The Minicircuits SBP-21.4 filter, for example, passes a $\sim$ 6-MHz wide band centered at 21.4 MHz. We have another custom one that might be more suited for the SDR. What sample rate would you need to avoid aliasing this output?

- Connect the noise generator to a filter and take several tens of thousands of samples with the SDR's ADC. These samples are voltages. What is the mean voltage over this data block? What is the variance? The standard deviation (which, for a zero-mean signal, is the same as the root-mean-square, or rms)?
- Plot a histogram of the sampled voltages (see `numpy.histogram`—for documentation, type `numpy.histogram?` in IPython). The histogram should look Gaussian, with a width equal to the rms voltage. Overplot this theoretically-expected Gaussian. Does it look like your observed distribution?
- Take many (tens to hundreds) blocks of thousands of samples with the ADC. Compute the power spectrum of each block, as well as the average power spectrum over all blocks.
- Plot the power spectrum for a single block and compare to the above average. Do the same for the average of N blocks, where $N = (2, 4, 8, 16)$ What you are doing here is looking at how integration time affects the signal-to-noise ratio (SNR): the 'signal' is the relatively smooth distribution that emerges with long integration times. The 'noise' is the scatter you see around that value. If the SNR goes as N^x, can you characterize what x is? How certain are you about this value, and how does it compare to what central-limit theory suggests?
- Calculate the ACF (by manually computing Fourier transforms and by using `numpy/scipy`'s version) for all 16000 samples in a block. Zoom in on delays of ≤ 2000 samples. Also derive the power spectrum from this ACF and compare with the Fourier-transform-derived power spectrum for the same block. Are they identical? Compare the width (full-width half-max, or FWHM) of the ACF ($\Delta\tau_{FWHM}$) with the FWHM of the power spectrum (ΔF_{FWHM}). Are they related?

6. Fourier Transforms, Analytic and Discrete (At Home, Week 1)

6.1. The Analytic Fourier Transform

A Fourier transform maps a function between two complementary coordinates which for now are usually time, t, and spectral frequency, ν. The input to a forward Fourier transform is a signal versus time, $E(t)$; the output is a signal versus frequency, $\tilde{E}(\nu)$, which is computed as

$$\tilde{E}(\nu) = \int_{-T/2}^{T/2} E(t) e^{-2\pi i \nu t} dt \ . \tag{3}$$

The input signal $E(t)$ is multiplied by a complex exponential and integrated, so the $\tilde{E}(\nu)$ is a complex-valued function. Of particular importance is that the Fourier Transform is **invertible**:

you can get back to the time domain using the inverse transform

$$E(t) = \frac{1}{B}\int_{-B/2}^{B/2} \tilde{E}(\nu) e^{2\pi i \nu t} d\nu \ . \tag{4}$$

This works because (complex) sine waves form a **basis** over the space of functions. Any function can be expressed exactly and uniquely by its Fourier coefficients [4]

For those of you looking for applications of that linear algebra class you took, the Fourier transform (FT) is a linear matrix operation akin to a rotation. As such, it has nice properties like:

1. linearity: the FT of a sum is the sum of the FTs,
2. invertibility: there is no information loss in the FT; it can be undone, and
3. unitarity: the FT is power-preserving.

6.2. The Discrete Fourier Transform (DFT)

The difference between an analytic Fourier transform and a discrete Fourier transform is that signals sampled at discrete intervals are no longer continuous. This means that integrals become somes, and the Nyquist criterion applies.

You have multiple discrete Fourier transform implementations at your disposal. Eventually, we would like you to use `numpy.fft`, which is fast (it can handle millions of points), but does not allow you to choose which frequencies are in your output. It also has the (sensible but confusing) feature that, in the output spectrum puts the zero frequency in the 0th array index, counts upward through the positive frequencies, then switches in the middle to the most negative frequency and continues counting in the positive direction toward zero. The inverse Fourier transform `numpy.fft.ifft` requires the input spectrum to be in this order to work properly.

However sensible, this arrangement of frequencies is not ideal for plotting. There is a function `fftshift` that will swap the order to a plot-friendly negative-to-positive order. However, once you apply this shift to your array, `ifft` no longer works as you expect. For this reason, *I suggest only using* `fftshift` *in plotting calls to avoid confusion.* See our "DFT's with DFT's" handout for details.

You should also take a look at `fftfreq`, which calculates the frequencies of the `numpy.fft` output.

For this class, we also provide the `ugradio.dft` module. It is slow, but it allows you to manually specify which output frequencies are desired, and it accepts and returns coordinate arrays (time and frequency, respectively) to facilitate clarity and transparency. We recommend using this module in the first week and for applications where you need to oversample the output frequencies. Otherwise, migrate to `numpy.fft`.

To use `ugradio.dft`, here are some recommendations:

- The set of sample times, N. Modern FFT libraries allow for arbitrary N, but performance improves if N has small prime factors. It is also advantageous to use even values of N, so

[4] You may wonder how the integration limits B and T are defined above. In the proper analytic formulation, they are both infinity. We emphasize their boundedness here because, in practice, neither can be infinity, and this necessarily introduces spectral leakage. Also note that, according to the Fourier conventions we've written here, our forward FT does not divide by the integration interval, but the inverse FT does. These conventions match the `numpy.fft` and `ifft` conventions.

that the zero frequency falls in the center of a bin. Powers of two are common. Our `dft` module is slow, though, and this won't help.

- Define the time range so that $t = 0$ falls at the center of the range. For N even, there is no center time, so make the times run from $-\frac{N}{2}/\nu_s$ to $(\frac{N}{2} - 1)/\nu_s$.
- In specifying the frequencies for which you want the output $\tilde{E}(\nu)$, I suggest that you first calculate the output for N frequencies running from $-\frac{\nu_s}{2}$ to $+\frac{\nu_s}{2}\left(1 - \frac{2}{N}\right)$. This makes the frequency increment equal to $\Delta\nu = \nu_s/N$ over a total range of just under ν_s.

To find out how to use the DFT, you can type `ugradio.dft.<tab>` to see an auto-complete of what is available in the module. You can also type `ugradio.dft??` to see the code, and of course, you can type `ugradio.dft.dft?` to see the documentation for the DFT function inside the `dft` module.

6.3. Fourier Filtering and a Secret Message

As an optional (fun) side investigation you can do on your own at home with Fourier filtering, we have provided an audio message encoded as a `numpy` array in the `secret_message.npz` file. This file holds the waveform (`'data'`) and the sample frequency (`'fs'`). If you install the python module `sounddevice` on **your home computer**:

```
$ pip install sounddevice
```

you can record (from *your home* microphone):

```
>>> import sounddevice as sd
>>> import scipy.io.wavfile
>>> seconds = 3 # s, recording duration
>>> fs = 44100 # Hz, sample rate
>>> recording = sd.rec(int(seconds * fs), samplerate=fs, channels=1) # record from microphone
>>> sd.wait() # wait for recording to finish
>>> scipy.io.wavfile.write('output.wav', fs, recording)
```

and play data (from `numpy` arrays!):

```
>>> fs, data = scipy.io.wavfile.read('output.wav') # read wav file to numpy array
>>> sd.play(data, fs) # play to speaker
>>> sd.wait() # wait until finished playing
```

In this case, however, we've stored the data in a `npz` file, not a `wav` file, so you don't need to use `scipy`.

The data we have provided contains a secret audio message, but with tons of high-frequency noise overlaying it that makes it hard to hear. See if you can use a power spectrum to characterize what frequencies cointain the high-frequency noise, then filter the voltage spectrum, transform back to a time-domain function (because Fourier transforms are invertible!) and play the secret message (again, on *your home computer*).

6.4. Power Spectra and Discrete Fourier Transforms

We are often interested in the output **power spectrum**, P_ν. Power is proportional to voltage squared. For complex quantities, the squaring operation means we want the sum of the squares of the real and imaginary parts. We obtain this by multiplying the voltage by its complex conjugate (denoted by '*'),

$$P_\nu = \tilde{E}(\nu)\tilde{E}^*(\nu) \ . \tag{5}$$

In Python, there are two ways to get this product. One is to use the `conj` function, i.e. `P = E * E.conj()`. Should the imaginary part of `P` be zero? (answer: yes! Why is this?) Is it? (answer: not always! Why not?) To get rid of this annoying and extraneous imaginary part, try casting your array as a float.

The other (more convenient and suggested) way is to square the length of the complex vector, i.e. `P = numpy.abs(E)**2`. The result is automatically real.

6.5. The Power Spectrum and the Autocorrelation Function (ACF)

Facility with the **convolution theorem** and its cousin, the **correlation theorem**, is a requirement for radio astronomy.

The convolution of two functions E and F (here arbitrarily taken to be functions of time) is

$$[E * F](\tau) = \int_{-T/2}^{+T/2} E(t)F(\tau - t)\ dt, \tag{6}$$

and the correlation of the two functions is

$$[E \star F](\tau) = \int_{-T/2}^{+T/2} E(t)F(\tau + t)\ dt. \tag{7}$$

Conceptually, these two functions describe sliding F over E by changing the delay parameter, τ, which describes a time shift between the two functions. The only difference between convolution and correlation is the sign of t in the argument of F.

Using our definition of the Fourier transform from Equation 3, the convolution theorem states:

$$\widetilde{[E * F]}(\nu) \equiv \int_{-T/2}^{T/2} [E * F](\tau)\ e^{2\pi i\tau\nu} d\tau = \tilde{E}(\nu) \cdot \tilde{F}(\nu) \tag{8}$$

and the correlation theorem:

$$\widetilde{[E \star F]}(\nu) \equiv \int_{-T/2}^{T/2} [E \star F](\tau)\ e^{2\pi i\tau\nu} d\tau = \tilde{E}(\nu) \cdot \tilde{F}^*(\nu) \tag{9}$$

In words, the theorem states that the Fourier transform of the convolution/correlation of two functions is equal to the product of their Fourier transforms (with one complex-conjugated for correlation). If $F(t)$ is symmetric, then the imaginary part of its Fourier transform is zero, which means $\tilde{F}^*(\nu) = \tilde{F}^(\nu)$, and two theorems become identical.

An important application of this theorem is the case when $E(t) = F(t)$, in which the correlation function becomes the **Autocorrelation function** $ACF(\tau)$, and equation 9 states that the power spectrum is equal to the Fourier transform of the ACF.

Because of end/wrap-around effects, these theorems apply strictly only in the limit $T \to \infty$. When calculating a digital version of the correlation function, you have to worry about end effects. Suppose you are calculating an ACF for N samples with delays ΔN ranging up to $N/2$. Then the number of terms in the sum is always smaller than N because the delays spill over the edge of the available samples.

7. Mixers (Lab Activity, Week 2)

7.1. The Double-SideBand (DSB) Mixer

In this section, you will build a double sideband (DSB) mixer (Figure 1). A mixer is an electronic device that multiplies the two input signals. It is simple: the radio frequency (RF) signal goes into one mixer port, the local oscillator (LO) goes into the second mixer port, and the intermediate frequency (IF) product is output through the third port.

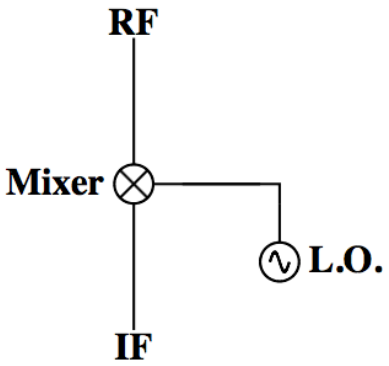

Fig. 1.— A DSB mixer. In the text, we sometimes refer to the RF input as the 'signal'.

In our lab, we have Mini-Circuits ZAD-1 and ZFM-15 mixers. They have three BNC connectors labeled R, L, and X or I, corresponding to the RF, LO, and IF, respectively. Both the ZAD-1 and the ZFM-15 are balanced mixers, so the R and L ports are identical but *will not couple to DC or very low frequencies*. To find out the frequency ranges these mixers support on each input, look up the datasheet online.

Use a mixer to build a DSB. Assign a signal generator to be your LO with frequency ν_{LO} and another to be your RF signal with frequency $\nu_{RF} = \nu_{LO} \pm \Delta\nu$. Here, you choose the frequency difference $\Delta\nu$ and you set the two signal generators for each of two cases: $\nu_{RF} = \nu_{LO} + \Delta\nu$ and $\nu_{RF} = \nu_{LO} - \nu$. Make $\Delta\nu$ small ($\sim 5\%$) compared to ν_{LO}. For the input power level, a good choice is 0 dBm[5] for both synthesizers. The output consists of both the sum and difference frequencies, so choose the ports appropriately.

Digitally sample the mixer output and identify the sum and difference frequencies. Consider the Nyquist criterion and choose a sample rate. If you want enough samples per period to produce a reasonable facsimile of the analog sine wave, you may wish to sample at twice Nyquist, or even faster. Another parameter to consider is the number of points sampled, which must be large enough

[5]What does this "dBm" mean? It is the power relative to 1 milliwatt, expressed in decibels (dB).

to capture a few periods of the slowest sine wave.

For the two cases $\nu_{RF} = \nu_{LO} \pm \Delta\nu$, 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 and plot the filtered signal versus time. Explain what you see.

7.2. Intermodulation Products in Real Mixers

An ideal mixer multiplies two input signals, but real mixers are not ideal. Inside, nonlinear diodes are used to perform an approximate multiplication, but deviations from ideal behavior produce harmonics of the input signals and harmonics of the sum, and harmonics of harmonics. These undesired products are called **intermodulation products**. When a well-designed mixer is operated with the proper input signal levels, the intermods have much less power than the main product, but they can nevertheless ruin sensitive measurements.

Examine one of the power spectra obtained above using a logarithmic vertical axis. Do you see the forest of lines? See if you can identify how some of the stronger ones originate as harmonics of the main lines.

7.3. The Single-Sideband Mixer (SSB Mixer)

A single sideband (SSB) mixer (Figure 2) is a little more complicated than the DSB mixer. It consists of two identical DSB mixers fed by an LO that is 90° phase shifted in the right-hand mixer. Hence, we can regard the left-hand output as being mixed with a cosine while the right-hand side extracts the sine component. together, they produce the real and imaginary components of a complex function that can now produce power spectra that contain different information at postive and negative frequencies. Engineers call this IQ sampling. For now, connect the two inputs (I and Q) to an oscilloscope that can display both simultaneously.

From the block diagram in Figure 2, construct an SSB mixer that achieves the phase delay with a cable[6]. We will use it to experiment with no phase delay (a short cable) and a 90° phase delay (a long cable). For experimentation with this two-output mixer, use two frequency synthesizers, as before.

7.3.1. Reverting to a DSB Mixer

First see what happens when the phase delay cable is short (ideally zero), so that the two halves of the SSB are essentially identical. Pick a value for $\pm\Delta\nu$ and take time series data for the two

[6]In vaccuum, light travels 1 foot per nanosecond—the only legitimate use of imperial units in astronomy. In a cable, light travels about 70% slower than in vaccuum. You should be able to use this information to get a ballpark estimate of a cable suits your needs for a chosen frequency.

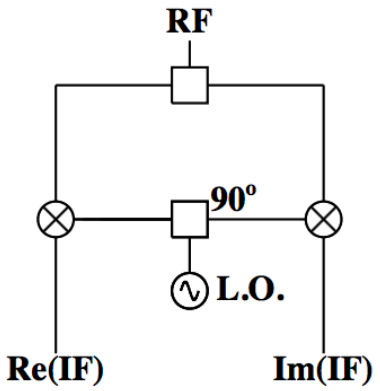

Fig. 2.— An SSB mixer. The important part is the 90° phase delay in the right-hand LO, which is normally produced with a quadrature splitter, but we can make it with a $\lambda/4$ piece of cable.

corresponding values of ν_{RF} (i.e., the upper and lower sidebands). Examine the phase of the I and Q components for mixing settings that should yield positive and negative IF frequencies. Do you see any difference?

7.3.2. The SSB Mixer

Now see what happens when the phase delay cable introduces a phase delay of 90° in the LO going to the right mixer. Repeating the steps in §7.3.1, can you distinguish between positive and negative $\Delta\nu$ in the phase of I and Q now? Why does it behave this way?

7.3.3. The R820T chip

As it turns out, our SDR module has a built-in mixer in front of the sampling chip, and you can tune the LO in software, provided you turn of `direct` sampling. This should allow you to sample I and Q data and store the results as `numpy` arrays. Calculate power spectra for your sampled data. When taking the Fourier transform, be sure to make assign the inputs to the real and imaginary parts of a complex `numpy` array. Looking at the power spectra alone, can you distinguish between positive and negative $\Delta\nu$?

How to the waveforms from one side of your R820T mixer compare to what you get using an external mixer? Which one produces the best results? Why?

8. On Mixers and the Heterodyne Process (At Home, Week 2)

8.1. The Heterodyne Process

Mixers allow us to shift the frequency of the whole input spectrum by a uniform amount. They do this by multiplying the input signal by a sine-wave **local oscillator** (LO) with frequency ν_{LO} (though we will often use angular frequency, ω_{LO}, for cleaner notation).

This is an important tool for RF signal chains (astronomical and otherwise) because *antennas* must be tuned to the frequency of the incoming radiation they target, but our filters, amplifiers, and samplers often must work at far lower frequencies.

AM (amplitude-modulated) radio stations, for example, broadcast signals in the 1-MHz range (for frequency-modulated FM stations, it is 100 MHz), but the information content is music at audio (kHz) frequencies. A mixer is used to shift the frequencies of the AM station down to the audio region, where they are filtered off, amplified, and sent to a speaker that converts the voltage fluctuations into pressure-wave fluctuations that your ear is sensitive to. When you tune in to a station, you are simply changing the LO frequency; the rest of the signal chain can stay the same.

Such receivers are called **heterodyne** receivers, and they are used in consumer radios, television (even modern digital ones), and cellphones, as well as in more noble pursuits, like radio astronomy.

8.2. Single Sideband (SSB) Mixer Theory

Even though we construct the SSB Mixer after the DSB in the lab, the theory of it is easier to understand, provided we use complex numbers and **negative frequencies**. Negative frequencies might seem weird. How can something oscillate a negative number of times per second?

To understand, let us use Euler's formula to write a complex sinusoid as

$$Ae^{iwt} = A\cos(\omega t) + i \cdot A\sin(\omega t). \tag{10}$$

In some ways, this complex sinusoid is the "true" sine wave. The real-valued versions are built out of pairs of complex sine waves:

$$\cos(\omega t) = \frac{1}{2}(e^{i\omega t} + e^{-i\omega t}) \tag{11}$$

$$\sin(\omega t) = \frac{1}{2i}(e^{i\omega t} - e^{-i\omega t}) \tag{12}$$

Now, once we have defined a complex sine wave (which henceforth will just be called a "sine wave"), we can switch $-\omega$ for ω,

$$Ae^{i(-w)t} = A\cos(\omega t) - i \cdot A\sin(\omega t), \tag{13}$$

which is to say that the negation of a frequency swaps the sign of the imaginary component. So rather than thinking of a negative frequency as "negative Hertz", let us instead think of it as a phase relationship between the sine and cosine components.

Now let's take an idealized SSB mixer that has a LO that is a complex sinusoid $e^{-i\omega_0 t}$ of unity amplitude with a negative frequency. This LO is mixed (multiplied) by an input signal. As an example, let's take an input signal that is the sum of two real-valued sine waves,

$$E(t) = A\sin(\omega_0 - \Delta\omega)t + B\sin(\omega_0 + \Delta\omega)t. \tag{14}$$

We can then use Euler's formula to express the product output by the mixer as

$$E(t) \cdot e^{-i\omega_0 t} = A\sin(\omega_0 - \Delta\omega)t \cdot e^{-i\omega_0 t} + B\sin(\omega_0 + \Delta\omega)t \cdot e^{-i\omega_0 t} \tag{15}$$

$$= \frac{A}{2i}\left[e^{i(\omega_0-\Delta\omega)t} - e^{-i(\omega_0-\Delta\omega)t}\right]e^{-i\omega_0 t} + \frac{B}{2i}\left[e^{i(\omega_0+\Delta\omega)t} - e^{-i(\omega_0+\Delta\omega)t}\right]e^{-i\omega_0 t} \tag{16}$$

$$= \frac{A}{2i}\left[e^{-i\Delta\omega t} - e^{-i(2\omega_0-\Delta\omega)t}\right] + \frac{B}{2i}\left[e^{i\Delta\omega t} - e^{-i(2\omega_0+\Delta\omega)t}\right]. \tag{17}$$

After mixing, each component sine wave in $E(t)$ has a beat-frequency term $e^{\pm i\Delta\omega t}$, as well as a component at much higher frequency ($2\omega_0$). These higher-frequency components are typically filtered off using a low-pass filter (LPF), leaving just the beat-frequency terms

$$\mathrm{LPF}\left[E(t)\cdot e^{-i\omega_0 t}\right] = \frac{A}{2i}e^{-i\Delta\omega t} + \frac{B}{2i}e^{i\Delta\omega t}. \tag{18}$$

These terms retain the amplitude and frequency offset of their original signal, but have been shifted to lower frequencies where they can be easily sampled and processed[7].

Furthermore, so long as we retain both the real and imaginary components (which, in reality, are just the components of the original signal that were multiplied by the cosine and sine components of the LO, respectively), we can distinguish between the A and B signal components by whether they appear at positive or negative frequencies. This ability to distinguish between positive and negative **sidebands** is why we call this a single-sideband mixer: we can look at each sideband separately. If we didn't have both the cosine and sine components, we would not be able to distinguish positive and negative frequencies. Signals A and B would then sit on top of each other, and we would have no choice but to look at both sidebands simultaneously.

Figure 2 shows a block diagram of the SSB mixer. The RF input and the LO are each split by a power splitter so that we have two identical mixers, one on the left and one on the right, whose outputs are labelled **Re(IF)** and **Im(IF)**, respectively. The one on the left is identical to the DSB mixer in figure 1. The one on the right differs in only one way, which is crucial: its LO is delayed by 90° relative to that on the left. With this, the **Im(IF)** output lags the **Re(IF)**, becoming a sine wave to the **Re(IF)**'s cosine.

8.3. Double Sideband (DSB) Mixer Theory

We now turn to the theory of the DSB mixer, which is very straightforward to build (the LO is just a real-valued sine wave, as is the RF, so we only require one mixing circuit), but is more complicated to understand.

Let us repeat the SSB derivation, but instead of using $e^{-i\omega_0 t}$ as our LO, we will use $\sin\omega_0 t$. If we take our signal to be $E(t) = A\sin(\omega_0+\Delta\omega)t$. In this case, the output of our mixer becomes

$$\begin{aligned} E(t)\cdot\sin\omega_0 t &= A\sin(\omega_0+\Delta\omega)t\cdot\sin(\omega_0 t) && (19)\\ &= \frac{A}{2i}\left[e^{i(\omega_0+\Delta\omega)t} - e^{-i(\omega_0+\Delta\omega)t}\right]\cdot\frac{1}{2i}\left[e^{i\omega_0 t} - e^{-i\omega_0 t}\right] && (20)\\ &= \frac{A}{2}\left[e^{i\Delta\omega t} + e^{-i\Delta\omega_0 t} - e^{i(2\omega_0+\Delta\omega)t} - e^{-i(2\omega_0+\Delta\omega)t}\right] && (21)\\ &= A\left[\cos\Delta\omega t - \cos(2\omega_0+\Delta)t\right]. && (22) \end{aligned}$$

As in the SSB, the mixer output has two components: a beat frequency (our desired output) and a component near $2\omega_0$ (which we remove using a LPF).

[7]For a radio station, the signal is speech or music which spans a range of $\Delta\omega$. In astronomy, (e.g. the 21-cm line), the signal is a Doppler broadened line, which again has a broad range of $\Delta\omega$. In both cases, a mixer with a well-chosen LO frequency can be used to mix the signal down to lower frequencies where it can be sent to a speaker (if you are listening to the radio, or if you are Jodie Foster's character *Ellie* in *Contact*.

The unfortunate part about the DSB mixer becomes obvious if we consider the case we used for the SSB mixer, where $E(t) = A\sin(\omega_0 + \Delta\omega)t + B\sin(\omega_0 - \Delta\omega)t$. Repeating our algebra above, we find that

$$E(t) \cdot \sin\omega_0 t = A\left[\cos\Delta\omega t - \cos(2\omega_0 + \Delta)t\right] + B\left[\cos(-\Delta\omega)t - \cos(2\omega_0 - \Delta)t\right]. \tag{23}$$

But $\cos(-\Delta\omega)t = \cos\Delta\omega t$, so after filtering off the high-frequency components, we end up with

$$\mathrm{LPF}\left[E(t) \cdot \sin\omega_0 t\right] = (A + B)\cos\Delta\omega t, \tag{24}$$

which has signals that were at two different frequencies ($\omega_0 + \Delta\omega$ and $\omega_0 - \Delta\omega$) stacked on top of each other.

The DSB mixer, though simpler to build, cannot distinguish between positive and negative deviations around the LO frequency. It stacks them right on top of each other, so that any frequency you look at in the output can be the sum of two different signals. This is what lends it the name "double sideband".

Figure 3 illustrates the result. The top panel shows the original RF spectrum, which consists of signals above the LO (the upper sideband; USB signal) and below (the lower sideband; LSB). Suppose you use a filter to eliminate the LSB, leaving only the USB. The second panel shows the IF spectrum after DSB mixing: the USB appears at both negative and positive frequencies and the spectrum is symmetric, meaning that the negative frequencies give exactly the same result as the positive ones.

Now use a filter to eliminate the USB, leaving only the LSB; the third panel shows the resulting IF spectrum.

Without any filters, then both the LSB and the USB would appear in the IF spectrum, as in the fourth panel. With a DSB mixer, you can't distinguish between LSB and USB. They are inextricably mixed into a sum in the power spectrum. However, the bottom panel shows that SSB mixing retains the sideband separation.

9. Writing the Lab Report (Week 3)

Please review the resources linked on AstroBaki and the guidelines offered in class for generating your report. Remember that your write-up is the basis for the majority of your grade in this class; make sure to include high-quality plots and description, providing adequate background and discussion such that one of your peers who has not taken this class would be able to understand your findings and conclusions. Take the opportunity to show off what you have learned so far!

Your report should directly address the goals listed in §2, but please ensure that you do so in a narrative that is directed toward the goal of conveying high-level results. It would be a mistake to make this report a laundry-list of the things you did during the last couple of weeks. Not every plot you made needs to be in this report, nor should every wrong turn be documented. A scientific publication documents process, but only insofar as is necessary to generate the results presented.

Finally, heed the warning that write-up takes longer than you think it will. Get started early and allow yourself plenty of time.

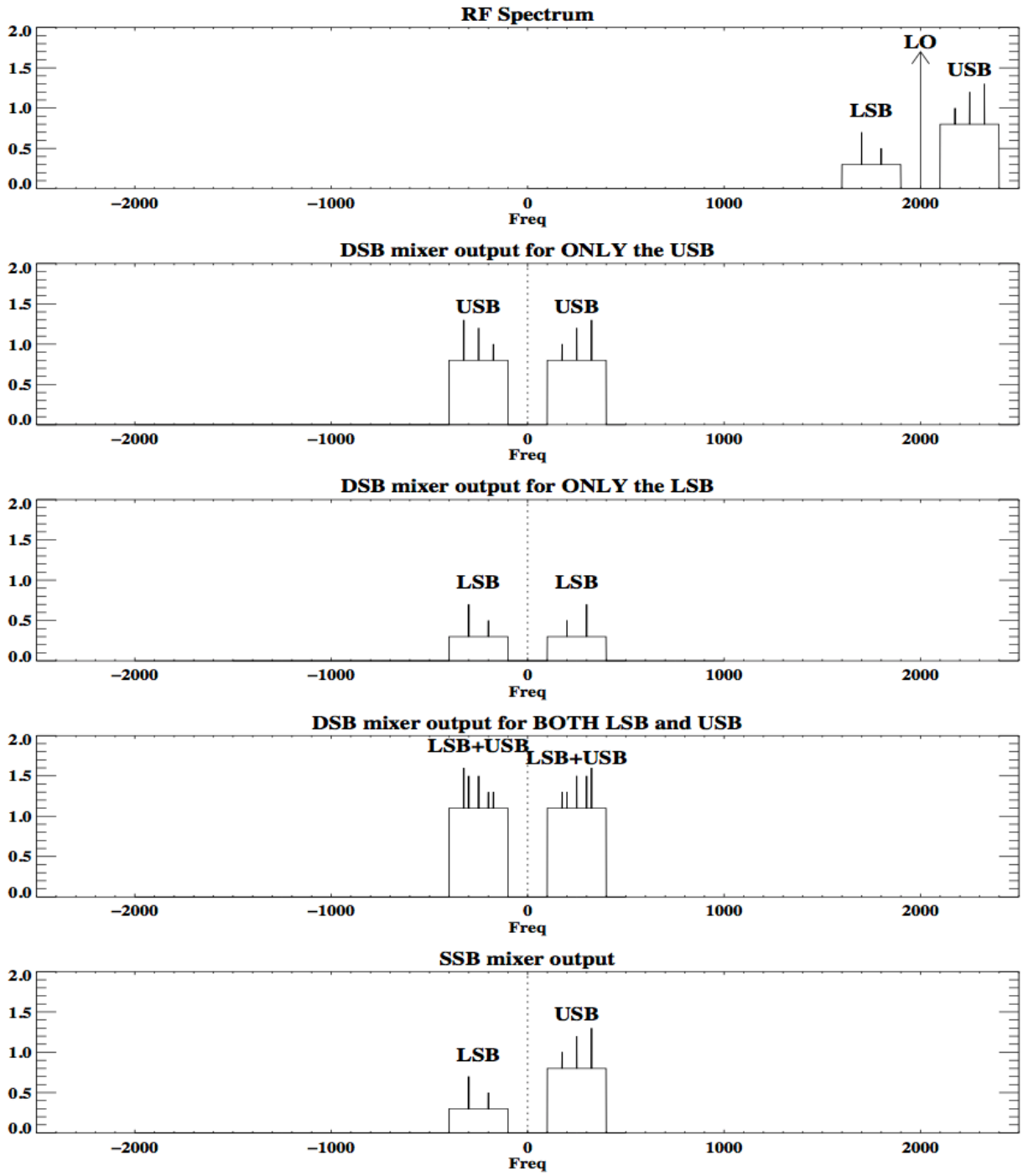

Fig. 3.— Upper (USB) and lower (LSB) sidebands in DSB and SSB mixers for a set of test tones on top of broad level noise spectra. Panel 1 illustrates the RF spectrum; panels 2 and 3 show the USB and LSB individually when they undergo the DSB mixing process; panel 4 shows how they both add together; panel 5 shows how the SSB mixer keeps them separate.