ECE4305: Software-Defined Radio Systems and Analysis

Laboratory 0: Introduction To SDR

Objective

This laboratory will introduce the concept of a communication systems and introduce channel noise. Specifically, a simplified error model will be discussed along with the theoretical connections to random processes discussed in Chapters III and IV. Moreover, MATLAB® will also be introduced as development tools for digital communication systems, especially the construction of a prototype software-defined radio (SDR) based simulation. An overview of the PLUTO radio will also be provided.

Contents

1 Theoretical Preparation

The fundamental concepts of digital communication systems and related theoretical background material covered in this section will serve as a basis for the implementation and design of prototype systems throughout the rest of this course.

1.1 Connection To Theory

Calculation of power is a non-trivial exercise in practice and in many situations loosely defined. This is especially true when determining SNR. Let us first consider a unique example where we have scaled exponentials ($s(t) = \alpha\, exp(j\pi f_c t)$) in AWGN. A resulting FFT provided in Figure 1 generated from some simple code for two different source signals:

```
r = signal+noise;
% View averaged spectrum
freq = linspace(-bandwidth/2,bandwidth/2,fftLen);
R = reshape(r,fftLen,frames);
R = fftshift(fft(R));
R_mean = mean(abs(R),2)/fftLen;
plot(freq,10*log10(R_mean));
```

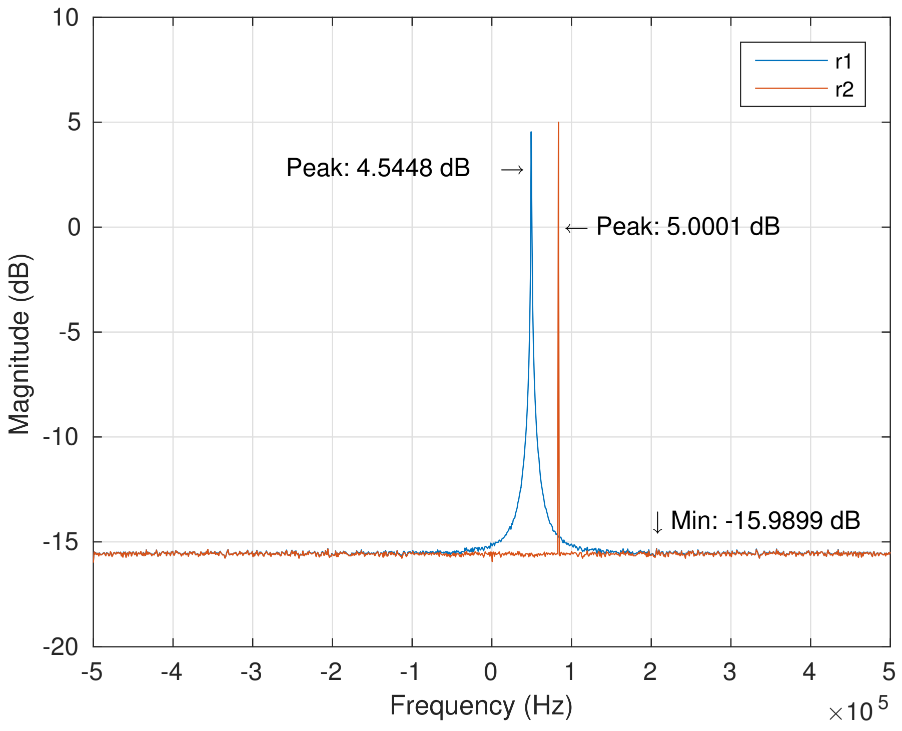


Figure 1: Averaged FFT of two different received signals with identical noise.

An obvious question to ask would be "What is the SNR of these signals?". However, we cannot directly derive the SNR from this plot alone. If we are using a spectrum or vector analyzer we need knowledge of the Resolution Bandwidth (RBW), which is a function of the FFT bin count and observation bandwidth. Since we already know the observation bandwidth (1 MHz), if we used a FFT of size 1024, then the true SNR is $\sim$ 10 dB. An important aspect to note from Figure 1 is the shape of the signals r_1 and r_2. In generation these two signals only differ in f_c, although they appear quite different in the figure. r_2 was strictly chosen to be within a FFT bin and r_2 strattles bins. This difference is due to the window effect of the FFT applied to the signals. This is commonly known as *scalloping* and will be dependent on the window chosen. fred harris (yes that capitalization is correct) provides a detailed overview of different windows and how to compensate for their effects in [3]. When performing digital signal processing it is alway important to understand effects of discrete computations over the infinite resolution of our written equations.

Now let us connect this to the theoretical concepts provided in the lectures and book. From Section 3.3.4 in the textbook we know that the power spectral density (PSD) is obtained through a Fourier Transform of a signal's autocorrelation function. Formally written as:

$$S_{XX}(f) = \int_{-\infty}^{\infty} R_{XX}(\tau) e^{-j2\pi f \tau} \, d\tau \tag{1}$$

where the autocorrelation of our process or signal $X(t)$ is:

$$R_{XX}(t_1, t_2) = E[X(t_1)X^*(t_2)]. \tag{2}$$

In the case of AWGN this autocorrelation is simply:

$$R_{XX}(t_1, t_2) = \begin{cases} \sigma^2 & \text{if } t_1 = t_2 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

where σ^2 is the variance of the noise. Therefore, $S_{XX} = \sigma^2$ for all frequencies within the observation bandwidth. and the power of the AWGN signal is simply $P_N = \sigma^2 \times B$. This result provides a mechanism for determining power from the PSD or frequency domain of a signal. However, it can be useful to calculate signal power from the time domain, and from Parseval's theorem we know power is identical between domains. For an AWGN signal calculating the variance directly provides P_N, but this is not true for non-zero mean processes. In that case, power can be simply calculated based on the squared RMS or squared mean of the process samples. However, a significant amount of data should be collected to get a representative power value of the signal.

To generate white noise with a specific PSD in MATLAB is a straight-forward process, but we need to be cautious of our units and concepts of time since time is abitrary in MATLAB. We will walk through a simple example where we generate white noise both complex and real for a given PSD and bandwidth. If we are given a specification sheet of a RF product it should provide one of two values: noise power over a given bandwidth (Usually defined at DANL), or possibly average PSD over a given band. When noise PSD numbers are given they are usually provide in dBm/Hz, which is what we will use here. Provided in the lab is script *generateNoise.m* which is also provided in the Figure below.

There are three important ideas that should be pointed out in this script. First, when generating noise from `randn` it is scaled to maintain unit magnitude. Second, two different ways are provided to determine the power of the noise of the generated signal using both the `var` and `rms` functions. Finally, when verifying the PSD of the produced signal we simply utilize the direct Einstein-Wiener-Khinchin (EWK) relation as discussed above.

Within MATLAB there are also several other tools that can be used to generate PSDs of signals such as `pwelch`, `periodogram`, and `pmtm` among others. When using make sure you understand their

```matlab
%% Generate noise and calculate Noise Power
% Choose parameters
noiseFloorDBMHZ = -30; % dBm/Hz
bandwidth = 1e6; % Hz (-bandwidth/2 -> bandwidth/2)
% Convert to linear units
noiseFloorWHZ = 10^((noiseFloorDBMHZ-30)/10); % Watts/Hz
% Calculate noise power
NoisePower = noiseFloorWHZ*bandwidth; % Watts
% Generate AWGN signal with desired noise power
fftLen = 2^10; frames = 1e3;
noiseC = sqrt(NoisePower)/sqrt(2).*...
    (randn(fftLen*frames,1)+randn(fftLen*frames,1).*1i); % Complex noise
noise = sqrt(NoisePower).*(randn(fftLen*frames,1)); % Real Noise
% Check
disp([NoisePower var(noise) var(noiseC) rms(noise)^2 rms(noiseC)^2]);

%% Veryify PSD
noiseFramesC = reshape(noiseC,fftLen,frames);
noiseFrames = reshape(noise,fftLen,frames);
% Determine PSD in Watts/(Frequency Bin)
noiseFramesFreqC = fft(noiseFramesC,fftLen);
noiseFramesFreq = fft(noiseFrames,fftLen);
% Autocorrelate and calculate mean power
S_kC = mean(abs(noiseFramesFreqC).^2,2)/fftLen;
S_k = mean(abs(noiseFramesFreq).^2,2)/fftLen;
% Convert to dBm/Hz
S_k_DBMHZC = 10*log10(S_kC/bandwidth) + 30;
S_k_DBMHZ = 10*log10(S_k/bandwidth) + 30;
% Plot
freqs = linspace(-bandwidth/2,bandwidth/2,fftLen);
figure;plot(freqs,S_k_DBMHZ,freqs,S_k_DBMHZC);
xlabel('Hz');ylabel('dBm/Hz');grid on;
ylim(noiseFloorDBMHZ+[-20 20]);
legend('Real Noise','Complex');
```

Figure 2: MATLAB code for generating white noise from specific PSD metric.

parameterization, since these function can be rather complex. However, they can provide more useful results beyond the simple application of EWK used above.

- Question 0: What is the SNR in dB for the red signal of Figure 1 if the resolution bandwidth is reduced by half?
- Question 1: Generate a complex noise signal with 100 μWatts of power and provide a PSD plot (in dBm/Hz) over 1 MHz of bandwidth.
- Question 2: Generate a sinusoidal signal with power 0 dB of power and provide a PSD plot (in dBm/Hz) over 1 MHz of bandwidth.
- Question 3: Calculate the SNR of the signals and noise from Questions 1 and 2 over 500 kHz of bandwidth.

1.2 Complex Baseband & Signal Representation

Understanding how a signal is represented can greatly enhance one's ability to analyze and design baseband digital communication systems. We need a convenient mathematical framework to represent signal and noise. We usually have two: Envelope/Phase and In-phase/Quadrature.

A bandpass signal can be represented by the sum of its in-phase (I) and quadrature (Q) components:

$$x(t) = R_I(t)\cos(2\pi f_c t) - R_Q(t)\sin(2\pi f_c t). \tag{4}$$

where $R_I(t)$ is in-phase amplitude, $R_Q(t)$ is quadrature amplitude and f_c is carrier frequency. It can also be represented as a sinusoid by its envelope and phase:

$$x(t) = R(t)\cos(\omega t + \phi(t)), \tag{5}$$

where $R(t)$ is the amplitude and $\phi(t)$ is phase offset.

Consider a 4-QAM signal constellation and how it would be represented by I and Q or its envelope and phase. If you needed to compute the distance between two points, which representation would be easier to work with? What about with other signal constellations? Table 1 shows these relationships graphically.

Given the importance of representing signals in an efficient format, consider how you would compute the distance between two constellation points in QPSK whose four points given in I/Q format are:

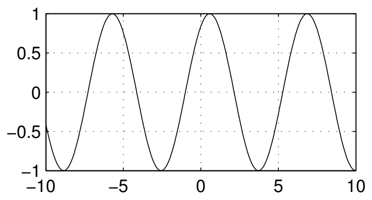
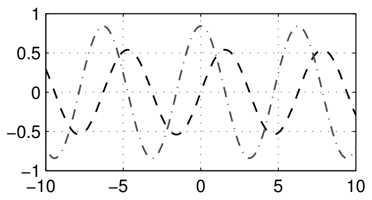
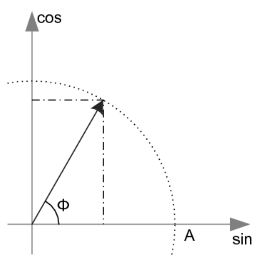
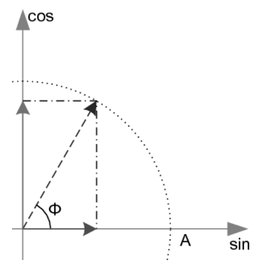
$$S_1(t) = +A\cos(2\pi f_c t) + A\sin(2\pi f_c t),$$

$$S_2(t) = -A\cos(2\pi f_c t) + A\sin(2\pi f_c t),$$
$$S_3(t) = -A\cos(2\pi f_c t) - A\sin(2\pi f_c t),$$
$$S_4(t) = +A\cos(2\pi f_c t) - A\sin(2\pi f_c t).$$

Using basic trigonometry, the distance between any two points can be quickly computed using the law of cosines, namely:

$$C^2 = A^2 + B^2 - 2ABcos(\theta). \tag{6}$$

Table 1: Signal representations of $A\sin(\omega t + \phi)$.

	Envelope/Phase $\mathbf{A}, \phi$	In-phase/Quadrature $(\mathbf{A}\sin\phi), (\mathbf{A}\cos\phi)$
Time	$\mathbf{A}\sin(\omega t + \phi)$	$(\mathbf{Acos}\phi)\sin(\omega t)$ $+(\mathbf{Asin}\phi)\cos(\omega t)$
Waveform		
Vector		

This solution yields the distance between the two points. The same principle holds true for other signal constellations including M-QAM, M-PSK, M-PAM, 7-around-1, BOX, etc. Although the trigonometry can be different for various constellations, it is almost always simpler to evaluate the math if you represent the signals in complex baseband.

1.3 Suggested Readings

Although this laboratory handout provides some information about the fundamentals of digital communications, the reader is encouraged to review the material from the following references in order to gain further insight on these topics.

- Overview of Signals and Systems:
  - Chapter 2 in Reference [6].
- Overview of Probability Theory:
  - Chapter 4 in Reference [6].
- Introduction to Communication Simulation Techniques:
  - Chapter 1 in Reference [7].

2 Radio Setup and Environmental Noise Observations

In these laboratory experiments an Analog Devices ADALM-PLUTO SDR with AD9363 transceiver will be used. This SDR in the most basic sense is a System-on-Chip (SoC) with an attached RF module providing complex baseband data. The MATLAB interface utilize the libiio kernel driver to talk with the SDR through a MATLAB class called `iio_sys_obj_matlab`. The two system objects provided in the hardware support package (HSP) for PlutoSDR are:

- comm.SDRRxPluto: PlutoSDR Receiver System object
- comm.SDRTxPluto: PlutoSDR Transmitter System object

These objects are typically constructed through the `sdrrx` or `sdrtx` function calls as in Code 2.

```
rx = sdrrx('Pluto')
tx = sdrtx('Pluto')
```

However, these objects can also be directly instantiated directly. The resulting object of sdrrx either way will have the following basic properties, which will be directly printed to the terminal when not using the semicolon as:

```
rx = sdrrx('Pluto')
rx =
    comm.SDRRxPluto with properties:
    DeviceName: 'Pluto'
    RadioID: 'usb:0'
    CenterFrequency: 2.4000e+09
    GainSource: 'AGC Slow Attack'
    ChannelMapping: 1
    BasebandSampleRate: 1000000
    OutputDataType: 'int16'
    SamplesPerFrame: 3660
    ShowAdvancedProperties: false
```

The transmitter System object `comm.SDRTxPluto` has near identical properties except for *GainSource*, *SamplesPerFrame*, and *OutputDataType* which do not make sense in the transmitter context. If you want to examine the available parameters simply type **doc comm.SDRRxPluto** or **doc comm.SDRTxPluto** into the MATLAB command prompt.

Before moving further with the radio it is important to outline how the radio works from the perspective of MATLAB using the `sdrrx` and `sdrtx` objects. After an object associated to the SDR has been created, the necessary methods can be run.

When configured in receive mode and the received method is called, with help from the driver, temporary buffers are created of size *SamplesPerFrame*, as set by the class parameter. Then the device will proceed to fill these buffer with contiguous data from the ADCs.

For each receive chain there are technically two ADCs, one for the in-phase portion of the signal and one for the quadrature. This is the reasoning behind the multiple buffers. However, these ADCs are time aligned and sampling of the dual chains happens at the same instances in time. Once the buffers are filled that data is provided to MATLAB and the link between the device and MATLAB is halted. As a result, when the receive method is called again from MATLAB the resulting buffer will be disjoint in time from the preceding buffer. For the overall structure of this communication Analog Devices provides a block diagram presented in Figure 3. The `PlutoSDR` class for convenience provides the output in a single complex vector, and accepts complex vectors as well.

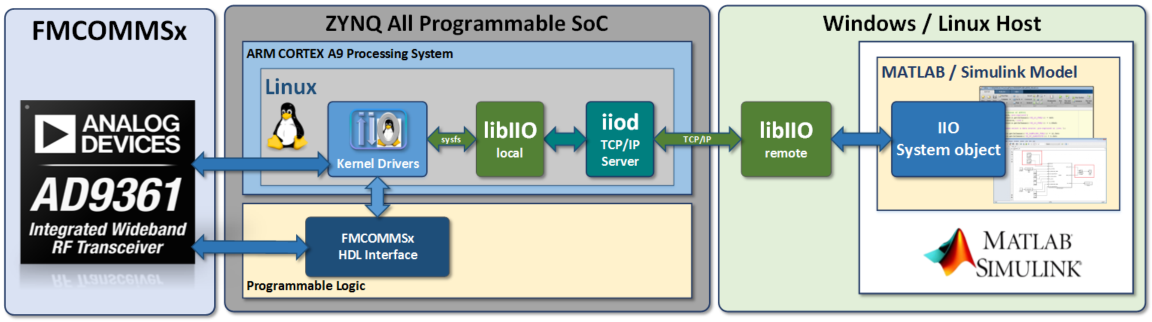


Figure 3: Structure of libiio and SDR hardware interconnection with MATLAB software [2].

Within the FMCOMMS/AD9361 direct downconversion is used to translate signals from RF to baseband frequencies. This is different than traditional heterodyne designs which utilize intermediate frequencies. Therefore, all observed signals passed to MATLAB are at baseband which were originally centered around *CenterFrequency* with a bandwidth of *BasebandSampleRate* parameterized by the `sdrrx` object.

2.1 Signal Loopback

Now that we have a basic understanding of how the radio operates and have it setup we can perform some simple experiments. First we will run a simple loopback test which transmits a sinusoidal tone out the transmitter and is simultaneously received at the receiver. To perform this we will utilize the provided `loopback.m` script. This script collects multiple buffers of data, which can be necessary since there is an unknown startup lag for the transmitter and the desired signal can be missed. This script should generate a plot similar to the one shown in Figure 4.

- Modify the amplitude of the sinusoid, along with the PLUTO's *GainSource* and observe their effects.
  - The three `GainSource` (AGC) settings are: *Manual*, *AGC Slow Attack*, and *AGC Fast Attack*.
  - When *Manual* is selected another option called *Gain* becomes available and this parameter can be modified.
  - It maybe useful to use signals with varying amplitudes or zeros.
  - If enough gain is applied the sinusoid should appear as a square wave at the receiver.

2.2 IEEE 802.11 Wireless Local Area Networks

Another interesting experiment is using the PLUTO to observe the spectrum of wireless local area networks within the vicinity (WLANs). Most high population density areas employs numerous wire-

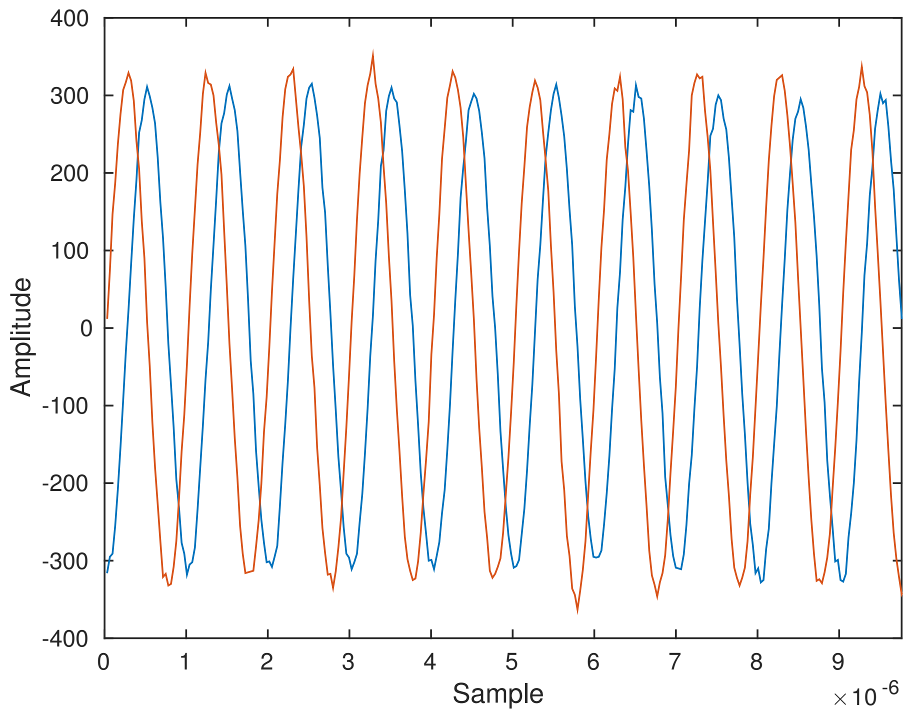


Figure 4: Looped back sinusoid observed from the Pluto receiver.

less communication networks for a variety of applications, such as the WPI wireless network. Consequently, you can use your PLUTO experimentation platform to plot their magnitude spectrum. IEEE 802.11 [4] is one type of WLAN standard that possesses a list of carrier frequencies for a collection of Wi-Fi channels. For example, The IEEE 802.11 standard defines Channel 1 of the 2.4 GHz band to be centered at 2.412 GHz.

- Specify the *CenterFrequency* parameter of `sdrrx` with this carrier frequency, use the `FFTs` or `dsp.SpectrumAnalyzer` objects to plot their magnitude spectrums.
- Since your AD9364 transceiver supports the 6 GHz band as well, also use the `dsp.SpectrumAnalyzer` to plot spectrum in the 5 GHz band.

You might want to turn on “Spectral Averaging” or “PlotMaxHoldTrace” and adjust the amplitude, since the peaks of these wireless signals could be rather low. It might also be useful to change the *BasebandSampleRate* parameter of `sdrrx` to adjust the frequency resolution for a better graph of the spectrum.

2.3 Measurements and the Radio

What a SDR device like the PLUTO does is give you digital samples. What these are is nothing more or less than what the ADC makes out of the voltages it observes. Then, those numbers are subject to the receiver processing chain which includes frequency translation (Mixing), decimation and filtering. Analog Devices provides a great overview here [1] for the *ad936x*. Altogether, the complex signal’s envelope coming from the PLUTO should be proportional to the voltages observed by the ADC. Therefore, the magnitude square of these samples should be proportional to the signal power as seen by the ADC. However, it must be stress that these are in relation to the range of the ADC. Therefore, these values are of an arbitrary measure relative to the full scale of the ADC and the remaining receive chain, commonly denoted as dBFS [5]. Scopes provided by MATLAB may denote the signal amplitude in dB but this is just an arbitrary engineering unit.

With this knowledge we cannot directly determine power of an input signal to the SDR, unless we have performed some calibration and can relate individual samples to received energy. However, SNR can still be determined since signal and noise we be subjected to the same ADC effects and receive chain. This can be done with techniques already presented previously. For Figure 5 we again used the transceiver setup, but transmitted signal vectors which contain half zeros and half signal. After extracting the noise and signal plus noise sections the SNR can be manually calculated simply by:

$$SNR_{dB} = 10\,log_{10}\Big(\frac{P_{SN} - P_N}{P_N}\Big) \tag{7}$$

- Repeat this process for different `GainSource` settings and different source amplitudes by modifying the supplied `loopback.m` script. Comment on the changes in SNR.

This is a rough method for estimating SNR, but does provides a reasonable metric without additional equipment. To provide better results we would need to remove some of the adaption performed in the receiver conditioning the samples.

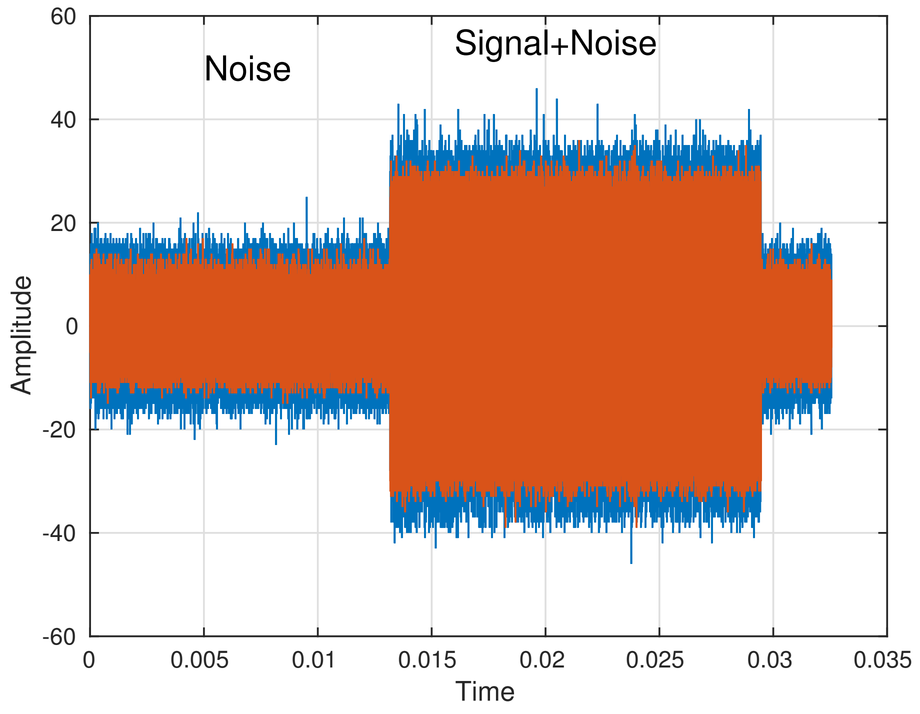


Figure 5: Receive loopback signal with signal half nulled.

3 Additional Probability Simulation Experiments

The MATLAB software uses a matrix language, which means it is designed for vector and matrix operations. You can often speed up your code by using vectorizing algorithms that take advantage of this design. *Vectorization* means converting `for` and `while` loops to equivalent vector or matrix operations. In this laboratory, MATLAB is used as a digital communication system design and evaluation tool. Due to the nature of many the mathematical operations used in communications, it is also important to vectorize operations in MATLAB as much as possible. Avoiding loops will save you many computational cycles and ultimately result in much shorter simulation times.

Read the online documentation *Techniques for Improving Performance* for more information about vectorization, as well as some other techniques for improving performance.

3.1 Random Number Generators

We will now focus on the generation of additive Ricean noise via the manipulation of uniform and Gaussian random number generators. The goal of this laboratory exercise is to refresh your knowledge about probability and prepare you for conducting communication simulations in MATLAB.

3.1.1 Generation Process

Generate two uniform random variables, x = $\{x_1, x_2, ...x_n\}$ and y = $\{y_1, y_2, ...y_n\}$, on the interval [0,1]. Use `rand()` function of MATLAB. Let n = 20,000.

3.1.2 Random Variable Transformation

Apply the following transformations to obtain two new random variables, $x_{std-normal}$ and $y_{std-normal}$:

$$x_{std-normal} = \mu_1 + \sqrt{-2log(x)}cos(2\pi y) \tag{8}$$

$$y_{std-normal} = \mu_2 + \sqrt{-2log(x)}sin(2\pi y) \tag{9}$$

where μ_1 and μ_2 are equal to zero. Plot the histograms of $x_{std-normal}$ and $y_{std-normal}$. Describe and explain your observations.

3.1.3 New Random Variables

Apply the following transformation to obtain a new random variable, $z_{rayleigh}$:

$$z_{rayleigh} = \sqrt{x^2_{std-normal} + y^2_{std-normal}} \tag{10}$$

Make μ_1=1 and μ_2=2 and apply the same transformation to obtain another random variable, z_{rician}:

$$z_{rician} = \sqrt{x^2_{std-normal} + y^2_{std-normal}} \tag{11}$$

Plot the histograms of $z_{rayleigh}$ and z_{rician}. Describe and explain your observations.

3.1.4 Noise Generation

We will now use the random variables $x_{std-normal}$ (or $y_{std-normal}$) and $z_{rayleigh}$ to simulate the performance of an idealized binary phase shift keying (BPSK) transceiver system generate a symbol vector consisting of 20,000 random "1" and "-1" values.

Additive Gaussian Noise: Add the elements of $x_{std-normal}$ to the symbol vector and round the obtained noisy symbols to the nearest constellation point (i.e., "1" or "-1"). Repeat the procedure by generating zero mean Gaussian random variables of different variances. Use the following variance values:

$$\sigma^2 = \{0.1, 0.1259, 0.1585, 0.1995, 0.2512, 0.3162, 0.3981, 0.5012, 0.6310, 0.7943, 1\} \tag{12}$$

for generating different Gaussian random variables. Note that a standard normal random variable, i.e., zero mean and unit variance, can be converted to a random variable of desired mean $\mu_{desired}$ and variance $\sigma^2_{desired}$, namely x$\sim$ N($\mu_{desired}$, $\sigma^2_{desired}$), by the transformation:

$$x = \mu_{desired} + \sigma_{desired} \times x_{std-normal}. \tag{13}$$

Count the number of errors resulting for each value of the variance and generate a plot, where the y-axis is the ratio of the number of errors to the total number of symbols transmitted, while the x-axis consists of the variance values chosen. Use `semilogy()` function from MATLAB. For the x-axis, use the vector obtained by converting each element of the above variance vector into a vector N = 10*$log_{10}(1/\sigma^2)$. What do you observe?

Additive Rayleigh Noise: Now add the elements of $z_{rayleigh}$ to the symbol vector and round the obtained noisy symbols to the nearest constellation point (i.e., "1" or "-1"). Repeat the procedure by generating Rayleigh random variables of different variances (use the Gaussian random variables obtained previously and apply the procedure described in Section 3.1.3.

Count the number of errors resulting for each value of the variance and generate a plot, where the y-axis consists of number of errors divided by the total number of symbols, and the y-axis is the variance values chosen. Use `semilogy()` function from MATLAB. What do you observe?

References

[1] Analog Devices. AD9361, AD9364 and AD9363. [Online]: https://wiki.analog.com/resources/eval/user-guides/ad-fmcomms2-ebz/ad9361.

[2] Analog Devices. IIO System Object. [Online]: https://wiki.analog.com/_detail/resources/tools-software/linux-software/libiio/clients/sys_obj.png?id=resources

[3] F. J. Harris. On the use of windows for harmonic analysis with the discrete fourier transform. *Proceedings of the IEEE*, 66(1):51–83, Jan 1978.

[4] IEEE 802.11 Working Group. IEEE 802.11: The working group setting the standards for wireless LANs. [Online]: http://www.ieee802.org/11/.

[5] Marcus Mller. How to Calculate power spectral density using USRP data? [Online]: http://stackoverflow.com/questions/40523362/how-to-calculate-power-spectral-density-using-usrp-data.

[6] Michael Rice. *Digital Communications: A Discrete-Time Approach*. Pearson/Prentice Hall, Upper Saddle River, New Jersey, 2009.

[7] Dennis Silage. *Digital Communication Systems using MATLAB and Simulink*. Bookstand Publishing, Gilroy, CA, 2009.