ANALOG DEVICES
AHEAD OF WHAT'S POSSIBLE™

A Simple Way to Use Python for Analysis of Noise in Mixed-Mode Signal Chains

Mark Thoren, Systems Design/Architecture Engineer, and
Cristina Șuteu, SW System Design Engineer

Abstract

Any application involving sensitive measurements of the physical world starts with an accurate, precise, and low noise signal chain. Modern, highly integrated data acquisition devices can often be directly connected to sensor outputs, performing analog signal conditioning, digitization, and digital filtering on a single silicon device, greatly simplifying system electronics. However, a complete understanding of the signal chain's noise sources and noise limiting filters is still required to extract maximum performance from and debug these modern devices.

Introduction

This tutorial is a continuation of a converter connectivity tutorial.[1,2] It will focus on the noise of individual signal chain elements, modeling them with Python/SciPy[3] and LTspice®. It will then verify the results by using Python to drive the ADALM2000 multifunction USB test instrument via libm2k and the Linux® industrial input output (IIO) framework. All source code and additional discussion are available in the companion Active Learning lab exercise.

Mixed-mode signal chains are everywhere. Simply put, any system that transforms a real-world signal to an electrical representation, which is then digitized, can be classified as a mixed-mode signal chain. At every point along the chain, the signal is degraded in various ways that can usually be characterized either as some form of distortion or additive noise. Once in the digital domain, the processing of the digitized data is not perfect either, but at least it is, for all practical purposes, immune to many of the offenders that affect analog signals—component tolerances, temperature drift, interference from adjacent signals, or supply voltage variations.

As the industry continues to push physical limits, one thing is certain: there is always room for improvement in analog and mixed-signal components for instrumentation. If an analog-to-digital converter (ADC) or a digital-to-analog converter (DAC) appears on the market that advances the state of the art in speed, noise, power, accuracy, or price, manufacturers will happily apply it to existing problems, then ask for more improvement. However, in order to achieve the best acquisition system for your application, it is fundamental to be aware of the components' limitations and choose them accordingly.

A Generic Mixed-Mode Signal Chain

Figure 1 shows a generic signal chain typical of a precision instrumentation application, with a physical input and digital output. There are numerous background references on ADCs available,[4] and most readers will have a sense that an ADC samples an input signal at some point in time (or measures the average of a signal over some observation time) and produces a numerical representation of that signal—most often as a binary number with some value between zero and $2^{(N-1)}$ where N is the number of bits in the output word.

ADC Noise Sources

While there are several noise sources in Figure 1, one that is often either ignored, or overemphasized, is the number of bits in the ADC's digital output. Historically, an ADC's number of bits was considered the ultimate figure of merit, where a 16-bit converter was assumed to be 4 times better than a 14-bit converter.[5] But in the case of modern, high resolution converters, the number of bits can be safely ignored. Note a general principle of signal chain design:

"The input noise of one stage should be somewhat lower than the output noise of the preceding stage."

As with any signal chain, one noise source within an ADC often dominates. Thus, if a noiseless signal applied to an N-bit ADC:

- Results in either a single output code, or two adjacent output codes, then quantization noise dominates. The signal-to-noise ratio (SNR) can be no greater than (6.02 N + 1.76) dB.[6]
- Results in a Gaussian distribution of many output codes, then thermal noise source dominates. The SNR is no greater than:

$$20\ log\left(\frac{V_{IN}\ (p\text{-}p)}{\frac{\sigma}{\sqrt{8}}}\right) \quad (1)$$

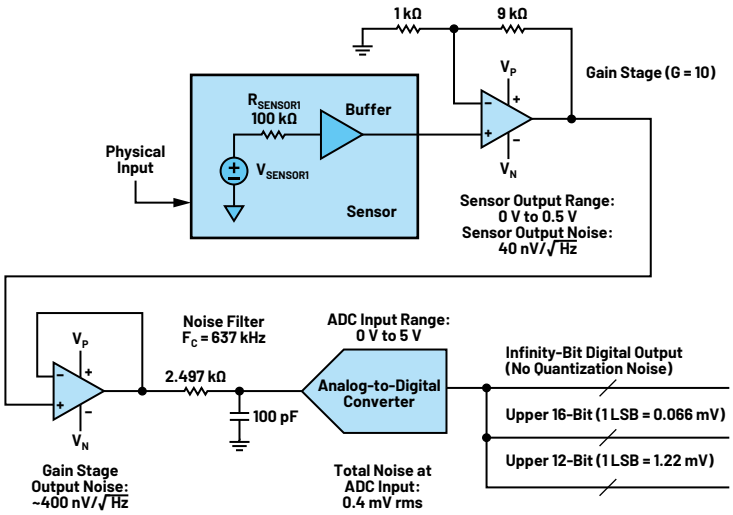

Figure 1. In a mixed-mode signal chain, some physical phenomenon such as temperature, light intensity, pH, force, or torque is converted to an electrical parameter (resistance, current, or directly to voltage). This signal is then amplified, low-pass filtered, and digitized by an ADC, which may include internal digital filtering.

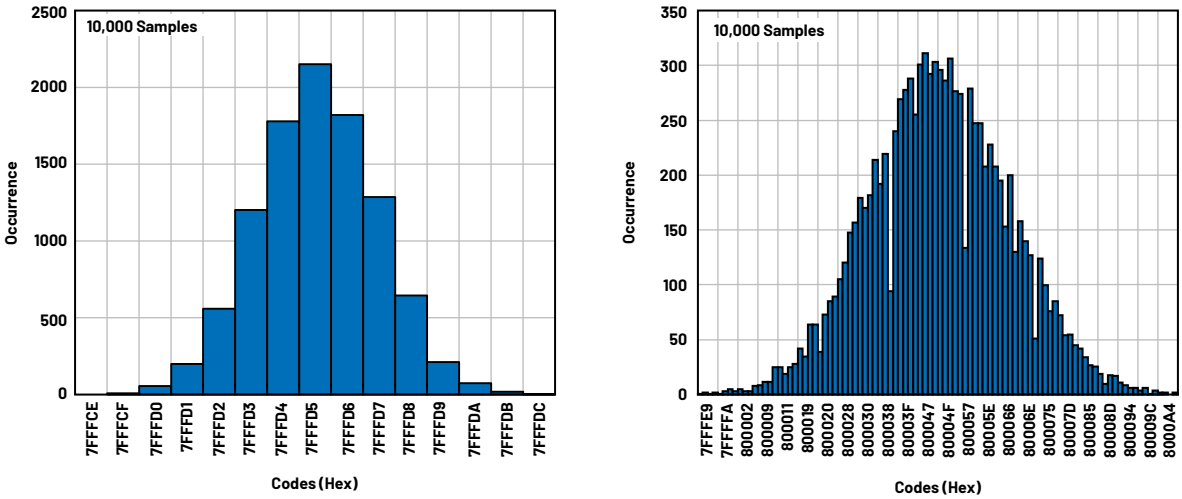

Figure 2. At a PGA gain of 1 (left), 13 codes are represented in the AD7124 output noise, and the standard deviation is about 2.5 codes. While quantization is visible, thermal noise is more significant. At a PGA gain of 128 (right), 187 codes are represented, and quantization noise is insignificant. Truncating one or two least significant bits (doubling or quadrupling quantization noise) would not result in a loss of information.

where:

V_{IN} (p-p) is the full-scale input signal.

σ is the standard deviation of the output codes in units of voltage.

Very high resolution converters, such as the AD7124-8, which will be used as an example shortly, are rarely limited by quantization noise; thermal noise dominates in all of the gain/bandwidth settings, and a shorted input will always produce a fairly Gaussian distribution of output codes. Figure 2 shows the grounded-input histogram of the AD7124-8, 24-bit sigma-delta ADC, with the internal programmable gain amplifier (PGA) set to 1 and 128, respectively.

Modeling and Measuring ADC Noise

Modeling the noise of a thermal-noise limited ADC is straightforward. If the noise is "well behaved" (Gaussian, as it is in Figure 2) and constant across the ADC's input span, the ADC's time-domain noise can be modeled using NumPy's[7] random normal function, then verified by taking the standard deviation, as seen in Figure 3.

```
# Model Gaussian Noise
# See AD7124 datasheet for noise levels per mode
import numpy as np
offset = 0.000
rmsnoise = 0.42e-6  # AD7124 noise
noise = np.random.normal(loc=offset, scale=rmsnoise,
                         size=1024)
measured_noise = np.std(noise)
print("Measured Noise: ", measured_noise)
```

Figure 3. Modeling Gaussian noise with NumPy.

The AD7124 device driver falls under the industry standard IIO framework, which has a well-established software API (including Python bindings). Application code can run locally (on the Raspberry Pi) or on a remote machine via network, serial, or USB connection. Furthermore, the pyadi-iio[8] abstraction layer takes care of much of the boilerplate setup required for interfacing with IIO devices, greatly simplifying the software interface. Figure 5 illustrates how to open a connection to the AD7124-8, configure it, capture a block of data, then close the connection.

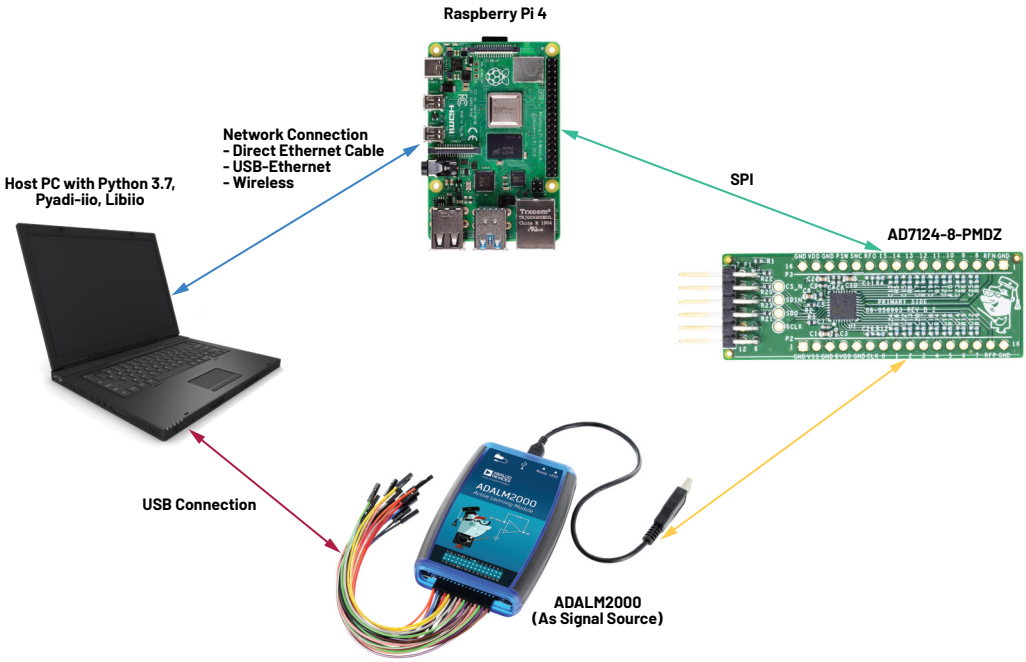

Figure 4. The ADALM2000 is a multifunction USB test instrument with two general-purpose analog inputs and two outputs, with sample rates of 100 MSPS and 150 MSPS, respectively. It can be used as a simple signal source for measuring ADC noise bandwidth and filter response. A Raspberry Pi 4 running a kernel with AD7124 device driver support acts as a simple bridge between the AD7124 and a host computer.

```
# AD7124-8 Basic Data Capture

import adi  # pyadi-iio library
# Connect to AD7124-8 via Raspberry Pi
my_ad7124 = adi.ad7124(uri="ip:analog.local")
ad_channel = 0  # Set channel
# Set PGA gain
my_ad7124.channel[ad_channel].scale = 0.0002983
my_ad7124.sample_rate = 128  # Set sample rate
# Read a single "raw" value
v0 = my_ad7124.channel[ad_channel].raw
# Buffered data capture
my_ad7124.rx_output_type = "SI"  # Report in volts
# Only one buffered channel supported for now
my_ad7124.rx_enabled_channels = [ad_channel]
my_ad7124.rx_buffer_size = 1024
my_ad7124._ctx.set_timeout(100000)  # Slow
data = my_ad7124.rx()  # Fetch buffer of samples

print("A single raw reading: ", v0)
print("A few buffered readings: ", data[:16])
del my_ad7124  # Clean up
```

Figure 5. AD7124-8 basic data capture.

With communication to the AD7124-8 established, an extremely simple, yet extremely useful test can be performed: measuring input noise directly. Simply shorting the input to an ADC and looking at the resulting distribution of ADC codes is a valuable step in characterizing a signal chain design. The AD7124 input mode is set to unipolar, so only positive values are valid; the test circuit shown in Figure 6 ensures that the input is always positive.

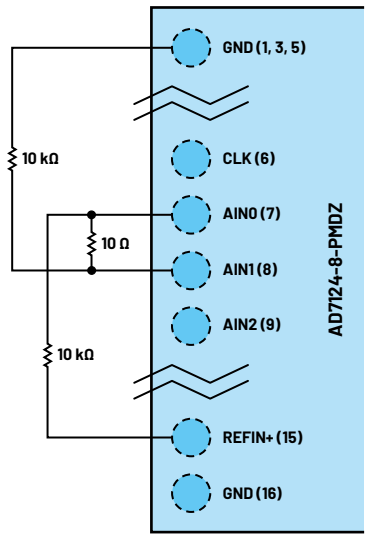

Figure 6. A resistor divider is used to generate a 1.25 mV bias across the AD7124-8's input, overcoming the 15 µV maximum offset voltage and ensuring that ADC readings are always positive.

Figure 7 shows two, 1024-point measurements. The lower (blue) trace was taken immediately after initially applying power.

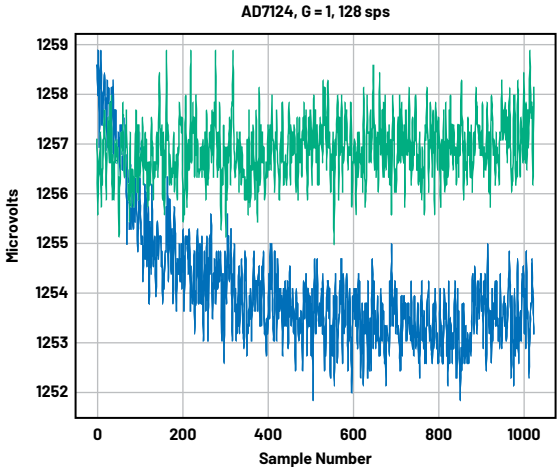

Figure 7. Two AD7124-8 data captures are taken with a 1.25 mV bias applied. The lower trace shows initial drift after power-up as the circuit warms up. The upper trace shows stable readings after a half-hour warmup time.

The "wandering" can be due to a number of factors—the internal reference warming up, the external resistors warming up (and hence drifting), or parasitic thermocouples, where slightly dissimilar metals will produce a voltage in the presence of thermal gradients. Measured noise after warmup is approximately 565 nV rms—on par with the data sheet noise specification.

Expressing ADC Noise as a Density

The general principle of analog signal chain design (that the input noise of one stage should be somewhat lower than the output noise of the preceding stage) is an easy calculation if all elements include noise density specifications—as most well-specified sensors and nearly all amplifiers do.

Unlike amplifiers and sensors, ADC data sheets typically do not include a noise density specification. Expressing the ADC's noise as a density allows it to be directly compared to the noise at the output of the last element in the analog signal chain, which may be an ADC driver stage, a gain stage, or the sensor itself.

An ADC's internal noise will necessarily appear somewhere between DC and half the sample rate. Ideally this noise is flat, or at least predictably shaped. In fact, since the ADC's total noise is spread out across a known bandwidth, it can be converted to a noise density that can be directly compared to other elements in the signal chain. Precision converters typically have total noise given directly, in volts rms:

$$e_{RMS} = \sigma \tag{2}$$

Where e_{RMS} is the total rms noise, calculated from the standard deviation of a grounded-input histogram of codes.

Higher speed converters that are tested and characterized with sinusoidal signals will typically have an SNR specification. If provided, the total rms noise can be calculated as:

$$e_{RMS} = \frac{ADCp\text{-}p}{\sqrt{8} \times 10^{\frac{SNR}{20}}} \tag{3}$$

where ADCp-p is the peak-to-peak input range of the ADC.

The equivalent noise density can then be calculated:

$$e_n = \frac{e_{RMS}}{\sqrt{\frac{f_S}{2}}} \tag{4}$$

where f_S is the ADC sample rate in samples/second.

The total noise from Figure 7 after warmup was 565 nV at a data rate of 128 SPS. The noise density is approximately:

$$\frac{565 \text{ nV}}{\sqrt{(64 \text{ Hz})}} = 70 \frac{\text{nV}}{\sqrt{\text{Hz}}} \tag{5}$$

The ADC can now be directly included in the signal chain noise analysis, which leads to a guideline for optimizing the signal chain's gain:

- Increase the gain just to the point where the noise density of the last stage before the ADC is a bit higher than that of the ADC, then stop. Don't bother increasing the signal chain gain any more—you're just amplifying noise and decreasing the allowable range of inputs.

This runs counter to the conventional wisdom of "filling" the ADC's input range. There may be a benefit to using more of an ADC's input range if there are steps or discontinuities in the ADC's transfer function, but for "well-behaved" ADCs (most sigma-delta ADCs and modern, high resolution successive approximation register (SAR) ADCs), optimizing by noise is the preferred approach.

Measuring ADC Filter Response

The AD7124-8 is a sigma-delta ADC, in which a modulator produces a high sample rate, but noisy (low resolution), representation of the analog input. This noisy data is then filtered by an internal digital filter, producing a lower rate, lower noise output. The type of filter varies from ADC to ADC, depending on the intended end application. The AD7124-8 is general-purpose, targeted at precision applications. As such, the digital filter response and output data rate are highly configurable. While the filter response is well defined in the data sheet, there are occasions when one may want to measure the impact of the filter on a given signal. The AD7124-8 filter response code block (see Figure 9) measures the filter response by applying sine waves to the ADC input and analyzing the output. This method can be easily adapted to measuring other waveforms—wavelets and simulated physical events. The ADALM2000 is connected to the AD7124-8 circuit as shown in Figure 8.

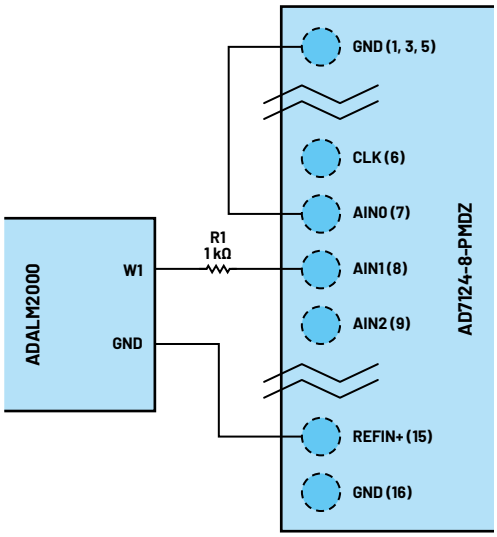

Figure 8. An ADALM2000 waveform generator is used to generate a range of sine wave frequencies, allowing the AD7124-8's filter response to be measured directly. While the script sets the sine wave amplitude and offset to a safe level, a 1 kΩ resistor protects the AD7124-8 in the event of a malfunction. (The ADALM2000 output voltage range is –5 V to +5 V, while the AD7124-8 absolute maximum limits are –0.3 V and +3.6 V.)

The AD7124-8 filter response code block (see Figure 9) will set the ADALM2000's waveform generator to generate a sine wave at 10 Hz, capture 1024 data points, calculate the rms value, then append the result to a list. The send_sinewave and capture_data are utility functions that send a sine wave to the ADALM2000 and receive a block of data from the AD7124, respectively.[2] It will then step through frequencies up to 120 Hz, then plot the result as shown in Figure 10.

```
# AD7124-8 Filter Response
import numpy as np
import matplotlib.pyplot as plt
resp = []
freqs = np.linspace(1, 121, 100, endpoint=True)
for freq in freqs:
    print("testing ", freq, " Hz")
    send_sinewave(my_siggen, freq)  # Set frequency
    time.sleep(5.0)                 # Let settle
    data = capture_data(my_ad7124)  # Grab data
    resp.append(np.std(data))  # Take RMS value
    if plt_time_domain:
        plt.plot(data)
        plt.show()
    capture_data(my_ad7124)  # Flush
# Plot log magnitude of response.
response_dB = 20.0 * np.log10(resp/0.5*np.sqrt(2))
print("\n Response [dB] \n")
print(response_dB)
plt.figure(2)
plt.plot(freqs, response_dB)
plt.title('AD7124 filter response')
plt.ylabel('attenuation')
plt.xlabel("frequency")
plt.show()
```

Figure 9. Filter response block program for the ADALM2000.

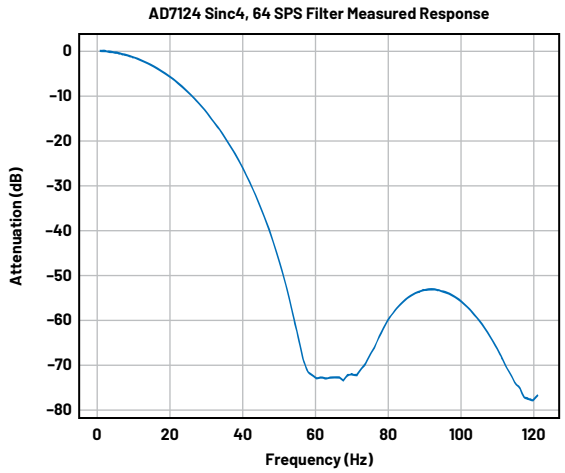

Figure 10. A measurement of the AD7124 filter response in 64 SPS, sinc4 mode shows the filter's pass band, first lobe, and first two nulls.

While measuring high attenuation values requires a quieter and lower distortion signal generator, the response of the first few major lobes is apparent with this setup.

Modeling ADC Filters

The ability to measure an ADC's filter response is a practical tool for bench verification. However, in order to fully simulate a signal chain, a model of the filter is needed. This isn't explicitly provided for many converters (including the AD7124-8), but a workable model can be reverse engineered from the information provided in the data sheet.

Note that what follows is only a model of the AD7124-8 filters; it is not a bit-accurate representation. Refer to the AD7124-8 data sheet for all guaranteed parameters.

The AD7124's filters all have frequency responses that are combinations of various sinc functions (with a frequency response proportional to $(\sin\{f\}/f)^N$). These filters are fairly easy to construct, and to reverse-engineer when nulls are known.

Figure 11 shows the AD7124-8's 10 Hz notch filters. Various combinations of higher order sinc3 and sinc4 filters are also available.

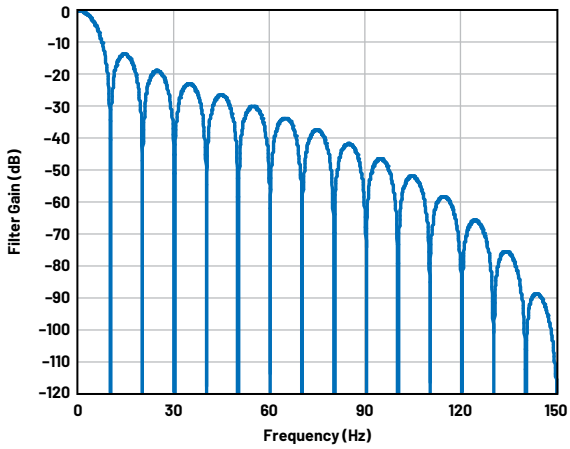

Figure 11. The AD7124-8 10 Hz notch filter has a sinc1 magnitude response; the filter's impulse response is simply an unweighted (rectangular) average of samples over a 100 ms time interval.

The simultaneous 50 Hz/60 Hz rejection filter shown in Figure 12 is a nontrivial example. This filter is intended to strongly reject noise from AC power lines, which is either 50 Hz (as in Europe) or 60 Hz (as in the United States).

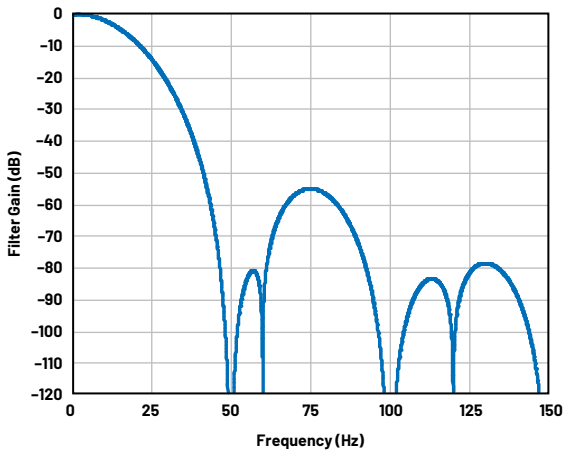

Figure 12. The AD7124-8 50 Hz/60 Hz rejection filter response is the combination of a 50 Hz, sinc3 filter and a 60 Hz, sinc1 filter.

Higher order sinc filters can be generated by convolving sinc1 filters. For example, convolving two sinc1 filters (with a rectangular impulse response in time) will result in a triangular impulse response, and a corresponding sinc2 frequency response. The AD7124 filters code block (see Figure 13) generates a sinc3 filter with a null at 50 Hz, then adds a fourth filter with a null at 60 Hz.

```
# AD7124 Filters
import numpy as np
f0 = 19200
# Calculate SINC1 oversample ratios for 50, 60Hz
osr50 = int(f0/50)  # 384
osr60 = int(f0/60)  # 320

# Create "boxcar" SINC1 filters
sinc1_50 = np.ones(osr50)
sinc1_60 = np.ones(osr60)

# Calculate higher order filters
sinc2_50 = np.convolve(sinc1_50, sinc1_50)
sinc3_50 = np.convolve(sinc2_50, sinc1_50)
sinc4_50 = np.convolve(sinc2_50, sinc2_50)

# Here's the SINC4-ish filter from datasheet
# Figure 91, with three zeros at 50Hz, one at 60Hz.
filt_50_60_rej = np.convolve(sinc3_50, sinc1_60)
```

Figure 13. AD7124-8 code example for a 50 Hz/60 Hz sinc filter.

The resulting impulse (time domain) shapes of the filters are shown in Figure 14. Filter coefficient (tap) values are normalized for unity (0 dB) gain at zero frequency.

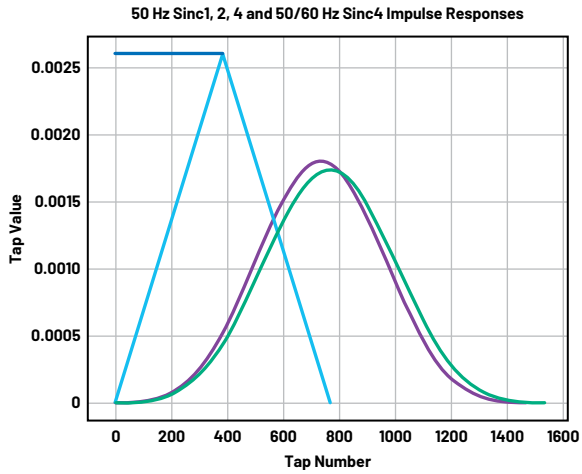

Figure 14. Repeatedly convolving rectangular impulse responses produces triangular, then Gaussian-like impulse responses.

And finally, the frequency response can be calculated using NumPy's freqz function, as seen in Figure 16. The response is shown in Figure 15.

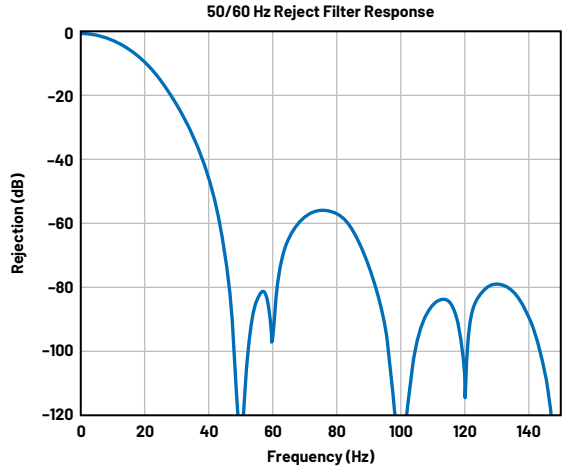

Figure 15. Convolving a sinc3, 50 Hz notch filter with a sinc1, 60 Hz filter produces a composite response that strongly rejects both 50 Hz and 60 Hz.

```
# AD7124 Frequency Response
import numpy as np
from scipy import signal
f0 = 19200
w, h = signal.freqz(filt_50_60_rej, 1, worN=16385,
                    whole = False, fs = f0)
freqs = w * f0/(2.0*np.pi)
hmax = abs(max(h))  # Normalize to unity
response_dB = 20.0 * np.log10(abs(h)/hmax)
```

Figure 16. AD7124-8 code example for sinc3 50 Hz notch filter with a sinc 60 Hz filter.

Resistance Is Futile: A Fundamental Sensor Limitation

All sensors, no matter how perfect, have some maximum input value (and a corresponding maximum output, which may be a voltage, current, resistance, or even dial position) and a finite noise floor—"wiggles" at the output that exists even if the input is perfectly still. At some point, a sensor with an electrical output will include an element with a finite resistance (or more generally, impedance) represented by R_{SENSOR} in Figure 17. This represents one fundamental

noise limit that cannot be improved upon—this resistance will produce the en(RMS) volts of noise, at a minimum:

$$e_n(RMS) = \sqrt{4 \times K \times T \times R_{SENSOR} \times (F2 - F1)} \quad (6)$$

where:

e_N (RMS) is the total noise.

K is Boltzmann's constant (1.38^{e-23} J/K).

T is the resistor's absolute temperature (Kelvin).

F2 and F1 are the upper and lower limits of the frequency band of interest.

Normalizing the bandwidth to 1 Hz expresses the noise density, in V/√Hz.

A sensor's data sheet may specify a low output impedance (often close to 0 Ω), but this is likely a buffer stage—which eases interfacing to downstream circuits but does not eliminate noise due to impedances earlier in the signal chain.

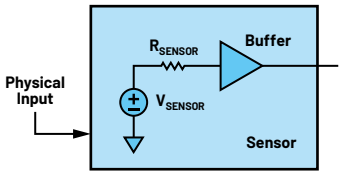

Figure 17. Sensors often include an internal buffer to simplify connection to downstream circuits. While the output impedance is low (often approaching 0 Ω), noise from high impedance sensing elements is buffered along with the signal.

There are numerous other sensor limitations—mechanical, chemical, optical—each with their own theoretical limits and whose effects can be modeled and compensated for later. But noise is the one imperfection that cannot.

A Laboratory Noise Source

A calibrated noise generator functions as a "world's worst sensor" that emulates the noise of a sensor without actually sensing anything. Such a generator allows a signal chain's response to noise to be measured directly. The circuit shown in Figure 18 uses a 1 MΩ resistor as a 127 nV/√Hz (at room temperature) noise source with "okay" accuracy and bandwidth. While the accuracy is only okay, this method has advantages:

- It is based on first principles, so in a sense can act as an uncalibrated standard.
- It is truly random, with no repeating patterns.

The OP482 is an ultralow bias current amplifier with correspondingly low current noise, and a voltage noise low enough that the noise due to a 1 MΩ input impedance is dominant. Configured with a gain of 2121, the output noise is 269 μV/√Hz.

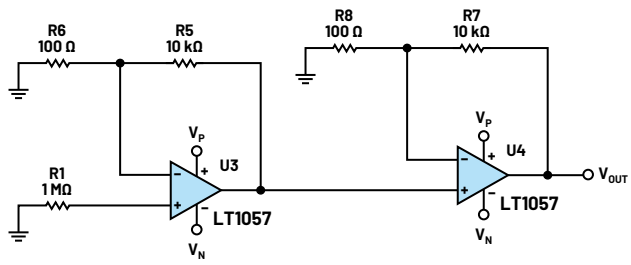

Figure 18. A 1 MΩ resistor serves as a predictable noise source, which is then amplified to a usable level by a low noise operational amplifier.

The noise source was verified with an ADALM2000 USB instrument, using the Scopy GUI's spectrum analyzer, shown in Figure 19.[9]

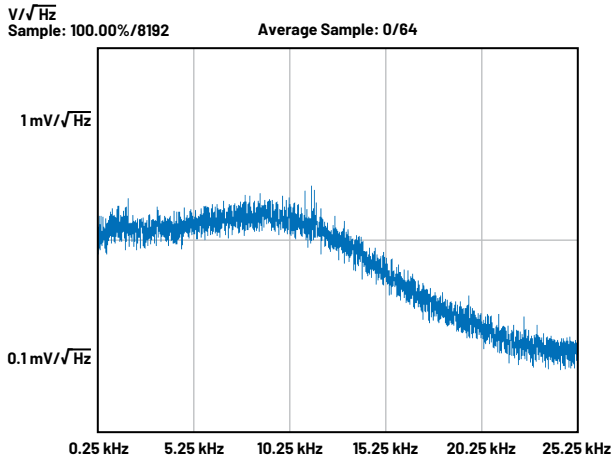

Figure 19. The output of the resistor-based laboratory noise generator has a usable bandwidth of approximately 10 kHz.

Under the analyzer settings shown, the ADALM2000 noise floor is 40 μV/√Hz, well below the 269 μV/√Hz of the noise source.

While Scopy is useful for single, visual measurements, the functionality can be replicated easily with the SciPy periodogram function. Raw data are collected from an ADALM2000 using the libm2k[10] and Python bindings, minimally processed to remove DC content (that would otherwise leak into low frequency bins) and scaled to nV/√Hz. This method, shown in Figure 20, can be applied to any data acquisition module, so long as the sample rate is fixed and known, and data can be formatted as a vector of voltages.

```
# Noise Source Measurement
import numpy as np
navgs = 32  # Avg. 32 runs to smooth out data
ns = 2**16
vsd = np.zeros(ns//2+1)  # /2 for onesided
for i in range(navgs):
  ch1 = np.asarray(data[0])  # Extract ch 1 data
  ch1 -= np.average(ch1)  # Remove DC
  fs, psd = periodogram(ch1, 1000000,
                        window="blackman",
                        return_onesided=True)
  vsd += np.sqrt(psd)
vsd /= navgs
```

Figure 20. Python noise source measurement code for the ADALM2000.

We are now armed with a known noise source and a method to measure said source, both of which can be used to validate signal chains.

Modeling Signal Chains in LTspice

LTspice® is a freely available, general-purpose analog circuit simulator that can be applied to signal chain design. It can perform transient analysis, frequency-domain analysis (AC sweep), and noise analysis, the results of which can be exported and incorporated into mixed-signal models using Python.

Figure 21 shows a noise simulation of the analog noise generator, with close agreement to experimental results. An op amp with similar properties to the OP482 was used for the simulation.

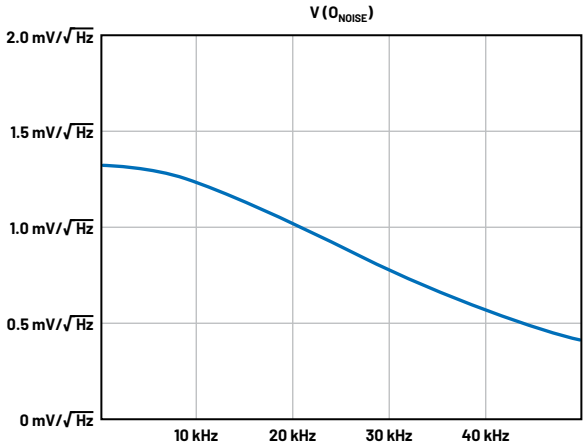

Figure 21. An LTspice simulation of the laboratory noise source shows approximately the same usable bandwidth as the measured circuit.

Figure 22's circuit noise is fairly trivial to model, given that it is constant for some bandwidth (in which a signal of interest would lie), above which it rolls off with approximately a first-order low-pass response. Where this technique comes in handy is modeling nonflat noise floors, either due to higher order analog filtering, or active elements themselves. The classic example is the noise mountain that often exists in auto-zero amplifiers such as the LTC2057, as seen in Figure 23.

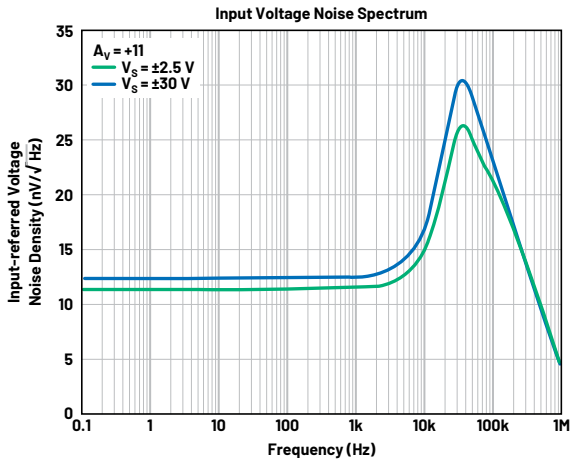

Figure 22. The LTC2057 noise density is flat at low frequencies, with a peak at 50 kHz (half of the internal oscillator's 100 kHz frequency).

Importing LTspice noise data for frequency-domain analysis in Python is a matter of setting up the simulation command such that exact frequencies in the analysis vector are simulated. In this case, the noise simulation is set up with a maximum frequency of 2.048 MHz and resolution of 62.5 Hz, corresponding to the first Nyquist zone at a sample rate of 4.096 MSPS. Figure 23 shows the simulation of the LTC2057 in a noninverting gain of 10, simulation output, and exported data format.

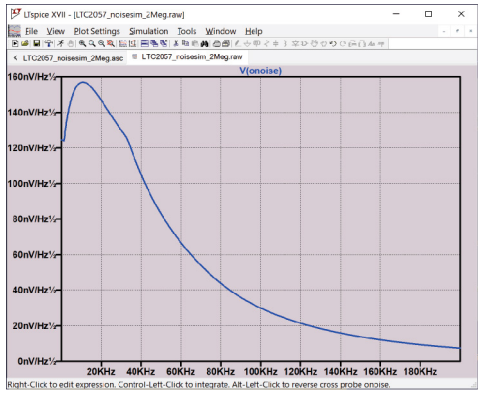

Figure 23. LTspice is used to simulate the output noise of an LTC2057 in a noninverting gain of +10 configuration. LTspice provides simple tools for integrating noise, but results of any simulation can be exported and imported into Python for further analysis.

In order to determine the impact of a given band of noise on a signal (signal-to-noise ratio) the noise is root-sum-square integrated across the bandwidth of interest. In LTspice, plotted parameters can be integrated by setting the plot limits, then control-clicking the parameter label. The total noise over the entire 2.048 MHz simulation is 32 µV rms. A function to implement this operation in Python is shown in Figure 24.

```
def integrate_psd(psd, bw):
    import numpy as np
    int_psd_sqd = np.zeros(len(psd))
    integrated_psd = np.zeros(len(psd))
    int_psd_sqd[0] = psd[0]**2.0
    for i in range(1, len(psd)):
        int_psd_sqd[i] += int_psd_sqd[i-1]\
            + psd[i-1] ** 2
        integrated_psd[i] += int_psd_sqd[i]**0.5
    integrated_psd *= bw**0.5
    return integrated_psd
```

Figure 24. Python code for a root-sum-square implementation.

Reading in the exported noise data and passing to the integrate_psd function results in a total noise of 3.21951e-05, very close to LTspice's calculation.

Generating Test Noise

Expanding on the functionality of the purely analog noise generator, it is very useful to be able to produce not only flat, but arbitrary noise profiles—flat bands of noise, pink noise, or noise mountains emulating peaking in some amplifiers. The generated time series from half-spectrum code block in Figure 25 starts with a desired noise spectral density (which can be generated manually, or taken from an LTspice simulation) and the sample rate of the time series, and then produces a time series of voltage values that can be sent to a DAC.

```
def time_points_from_freq(freq, fs=1, density=False):
    import numpy as np
    N = len(freq)
    rnd_ph_pos = (np.ones(N-1, dtype=np.complex)*
                  np.exp(1j*np.random.uniform
                         (0.0, 2.0*np.pi, N-1)))
    rnd_ph_neg = np.flip(np.conjugate(rnd_ph_pos))
    rnd_ph_full = np.concatenate(([1], rnd_ph_pos, [1],
                                  rnd_ph_neg))
    r_s_full = np.concatenate((freq, np.roll
                               (np.flip(freq), 1)))
    r_spectrum_rnd_ph = r_s_full * rnd_ph_full
    r_time_full = np.fft.ifft(r_spectrum_rnd_ph)

    if (density is True):
        # Note that this N is "predivided" by 2
        r_time_full *= N*np.sqrt(fs/(N))
    return(np.real(r_time_full))
```

Figure 25. Python code to generate arbitrary noise profiles.

This function can be verified by controlling one ADALM2000 through a libm2k script, and then verifying the noise profile with a second ADALM2000 and the spectrum analyzer in the Scopy GUI. The push noise time series to ADALM2000 code snippet (see Figure 26) generates four bands of 1 mV/√Hz noise on the ADALM2000 W2 output (with a sine wave on W1, for double-checking functionality).

```
# Push Noise Time-series to ADALM2000
import numpy as np
n = 8192
# create some "bands" of  1mV/rootHz noise
bands = np.concatenate((np.ones(n//16),
                        np.zeros(n//16),
                        np.ones(n//16),
                        np.zeros(n//16),
                        np.ones(n//16),
                        np.zeros(n//16),
                        np.ones(n//16),
                        np.zeros(n//16)))*1000e-6
bands[0] = 0.0  # Set DC content to zero
buffer2 = time_points_from_freq(bands, fs=75000,
                                density=True)
buffer = [buffer1, buffer2]
aout.setCyclic(True)
aout.push(buffer)
```

Figure 26. Verify the arbitrary noise with the ADALM2000.

Figure 27 shows four bands of 1 mV/√Hz noise being generated by one ADALM2000. The input vector is 8192 points long at a sample rate of 75 kSPS, for a bandwidth of 9.1 Hz per point. Each band is 512 points, or 4687 Hz wide. The roll-off above ~20 kHz is the sinc roll-off of the DAC. If the DAC is capable of a higher sample rate, the time series data can be upsampled and filtered by an interpolating filter.[11]

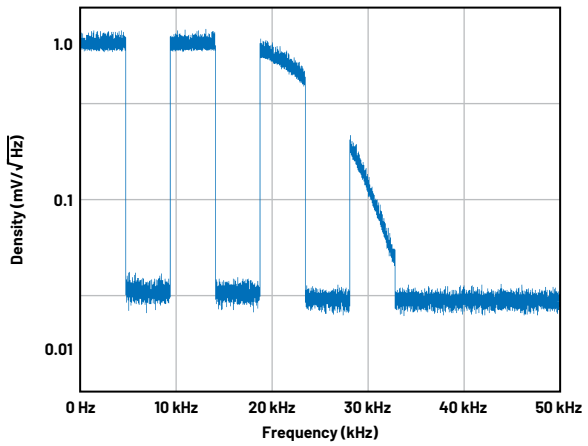

Figure 27. The Scopy spectrum analyzer is used to verify the arbitrary noise generator. Deep notches between noise bands expose the analyzer's noise floor, showing that an arbitrary noise profile can be accurately generated.

This noise generator can be used in conjunction with the pure analog generator for verifying the rejection properties of a signal chain.

Modeling and Verifying ADC Noise Bandwidth

External noise sources and spurious tones above $f_s/2$ will fold back (alias) into the DC to $f_s/2$ region and a converter may be sensitive to noise far beyond $f_s/2$. Consider the LTC2378-20, which has a sample rate of 1 MSPS and a -3 dB input bandwidth of 34 MHz. While performance may not be the best at such high frequencies, this converter will digitize more than 68 Nyquist zones of noise and fold them back on top of your signal. This illustrates the importance of antialiasing filters for wideband ADCs. Converters for precision applications are typically sigma-delta (like the AD7124-8) or oversampling SAR architectures, in which the input bandwidth is limited by design.

It is often useful to think of the equivalent noise bandwidth (ENBW) of a filter, including an ADC's built-in filter. The ENBW is the bandwidth of a flat pass-band "brick wall" filter that lets through the same amount of noise as the nonflat filter. A common example is the ENBW of a first-order RC filter, which is:

$$ENBW = \frac{f_c \times \pi}{2} \quad (7)$$

Where f_c is the cutoff frequency of the filter. If broadband noise, from "DC to daylight," is applied to the inputs of both a 1 kHz, first-order low-pass filter and a 1.57 kHz brick wall low-pass filter, the total noise power at the outputs will be the same.

The ENBW example code block in Figure 28 accepts a filter magnitude response and returns the effective noise bandwidth. A single-pole filter's magnitude response is calculated and used to verify the ENBW = $f_c \times \pi/2$ relationship.

```
import numpy as np
def arb_enbw(fresp, bw):
    int_frsp_sqd = np.zeros(len(fresp))
    int_frsp_sqd[0] = fresp[0]**2.0
    for i in range(1, len(fresp)):
        int_frsp_sqd[i] += (int_frsp_sqd[i-1] +
                            fresp[i-1] ** 2)
    return int_frsp_sqd[len(int_frsp_sqd)-1]*bw

fc = 1  # Hz
bw_per_point = 200/65536  # Hz/record length
frst_ord = np.ndarray(65536, dtype=float)
# Magnitude = 1/SQRT(1 + (f/fc)^2))
for i in range(65536):
    frst_ord[i] = (1.0 /
                   (1.0 +
                    (i*bw_per_point)**2.0)**0.5)
fo_enbw = arb_enbw(frst_ord, bw_per_point)
```

Figure 28. Python code example to calculate the effective noise bandwidth.

This function can be used to calculate the ENBW of an arbitrary filter response, including the AD7124's internal filters. The frequency response of the AD7124 sinc4 filter, 128 SPS sample rate can be calculated by a method similar to the previous 50 Hz/60 Hz rejection filter example. The arb_anbw function returns an ENBW of about 31 Hz.

The ADALM2000 noise generator can be used to validate this result. Setting the test noise generator to generate a band of 1000 µV/√Hz should result in a total noise of about 5.69 mV rms, and measured results are approximately 5.1 mV rms total noise. The oscilloscope capture of the ADC input signal is plotted next to the ADC output data, in Figure 29. Note the measured peak-to-peak noise of 426 mV, while the ADC peak-to-peak noise is about 26 mV. While such a high

noise level is (hopefully) unrealistic in an actual precision signal chain, this exercise demonstrates that the ADC's internal filter can be relied on to act as the primary bandwidth limiting, and hence noise reducing, element in a signal chain.

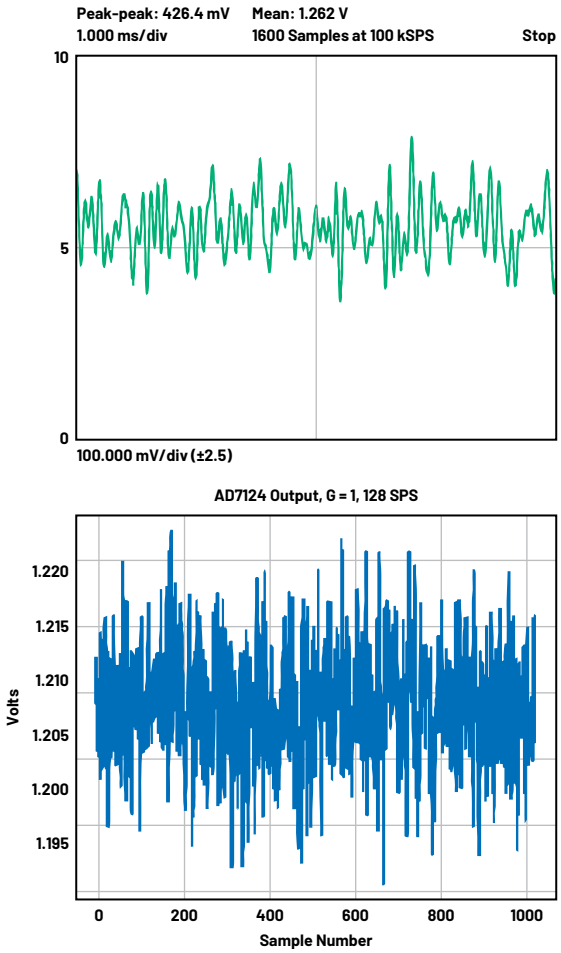

Figure 29. A 1 mV/√Hz noise band is driven into the AD7124-8 input. A qualitative reduction in noise is apparent; 426 mV peak-to-peak noise at the ADC input results in approximately 25 mV peak-to-peak noise at the ADC output. The 5.1 mV rms total output noise is close to the predicted 5.69 mV rms, given the 1 mV/√Hz noise density and 31 Hz ENBW of the ADC's filter.

Conclusion

Noise is a limiting factor in any signal chain; once noise contaminates a signal, information is lost. Before building a signal acquisition system, the application requirements must be understood, components selected accordingly, and the prototype circuit tested. This tutorial offers a collection of methods that accurately model and measure sensor and signal chain noise that can be used during the design and testing process.

The techniques detailed in this tutorial are, individually, nothing new. However, in order to achieve an adequate system, it is valuable to have a collection of fundamental, easy to implement, and low cost techniques to enable signal chain modeling and verification. Even though manufacturers continue to offer parts with increased performance, there will always be a certain limitation that one must be aware of. These techniques can not only be used to validate parts before building a mixed-mode signal chain, but also to identify design faults in an existing one.

Acknowledgements

- Jesper Steensgaard, who enabled/forced a paradigm shift in thinking about signal chain design, starting with the LTC2378-20.
- Travis Collins, Architect of Pyadi-iio (among many other things).
- Adrian Suciu, Software Team Manager and contributor to libm2k.

References

[1] "Converter Connectivity Tutorial." Analog Devices Wiki, January 2021.

[2] Analog Devices Education Tools Repository. Zenodo, July 2021.

[3] Pauli Virtanen, Ralf Gommers, et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python." Nature Methods, 17(3), February 2020.

[4] Steven W. Smith. *The Scientist & Engineer's Guide to Digital Signal Processing*. California Technical Publishing, 1999.

[5] Ching Man. "MT-229: Quantization Noise: An Expanded Derivation of the Equation, SNR = 6.02 N + 1.76." Analog Devices, Inc., August 2012.

[6] Walt Kester. "MT-001: Taking the Mystery out of the Infamous Formula, 'SNR = 6.02N + 1.76dB,' and Why You Should Care." Analog Devices, Inc., 2009.

[7] Charles R. Harris, K. Jarrod Millman, et al. "Array Programming with NumPy." Nature, 585, September 2020.

[8] "pyadi-iio: Device Specific Python Interfaces for IIO Drivers." Analog Devices Wiki, May 2021.

[9] "Scopy." Analog Devices Wiki, February 2021.

[10] "What Is Libm2k?" Analog Devices Wiki, October 2021.

[11] Walt Kester. "MT-017: Oversampling Interpolating DACs." Analog Devices, Inc., 2009.

Attribution

This article first appeared in the proceedings of the 2021 Scientific Computing with Python conference under the title "Using Python for Analysis and Verification of Mixed-Mode Signal Chains." DOI: 10.25080/majora-1b6fd038-001.

About the Authors

Mark Thoren joined Linear Technology (now part of Analog Devices) in 2001 as an applications engineer supporting precision data converters. He's since held various roles in mixed-signal applications including developing evaluation systems, training, technical publications, and customer support. Mark is now a systems engineer in ADI's System Development Group, where he works on reference designs and the ADI University Program. Mark has a B.S. in agricultural mechanical engineering and an M.S. in electrical engineering, both from University of Maine, Orono. He can be reached at mark.thoren@analog.com.

Cristina Șuteu joined Analog Devices' System Development Group in 2019 as a systems application engineer. During her time with ADI, she has contributed to software improvements, had different roles in technical publications and trainings, and provided educational material for the ADI University Program, including a video series for the ADALM2000. Cristina has a B.S. in electrical engineering from the Technical University of Cluj-Napoca, Romania, and an M.S. in signal and image processing, from the Technical University of Cluj-Napoca in collaboration with the University of Bordeaux, France. She can be reached at cristina.suteu@analog.com.