Some signal-processing common chores to be implemented

Juan David González Cobas

August 5, 2010

1 Motivation and documentation

For getting into these, you'll need a certain mastery of two Python packages: `numpy` and `matplotlib`. The best way to have a grasp of what they provide is to read the tutorials *Tentative Numpy Tutorial* and *Tentative Numpy Tutorial* and *PyPlot Tutorial* (both barely scratch 1% of the surface of these immense libraries).

You will need to implement the figures of merit you have studied in the Analog Devices document and other sources:

Static DNL, INL (by the histogram method)

Dynamic ENOB, SNR, SINAD, THD

A necessary step from this to work properly is to test your programs with artificial signals that will give results know beforehand (e.g., we make up a noisy sine with a well-defined SNR and use it to test the algorithms implemented). So some of these exercises aim at providing such kind of signal. Then, we go for the implementation of the true measures that we will apply to true data coming from the devices to characterize.

2 Artificial signals to provide

1. We basically need a function

   ```
   def pure_sine(amplitude, frequency, phase,
   sampling_rate, samples)
   ```

   returning an array (from now on, we'll refer to a `numpy` vector as such) that results from sampling

   $$A\sin(2\pi ft+\phi) \tag{1}$$

   at the given sampling rate.

2. In the same vein, you will provide a noise generator like

```
def noise(amplitude, samples)
```

that generates `samples` samples of random noise of given amplitude.

3. Combining both, it is easy to create

```
def noisy_sine(amplitude, frequency, phase,
sampling_rate, samples, SNR)
```

to make a sine wave with a prescribed SNR. Work out a numeric example to check that the vector returned is OK.

4. Create a small routine

```
def plot(arr)
```

that plots the contents of the array `arr`.

5. For DNL and INL computations, we will also need to implement a simple function like

```
def adc_transfer(v, frs, bits)
```

that will return the integer (digital) value corresponding to an ideal ADC with full-range scale and number of bits given, when fed with `v` volts input.

6. Plot adc_transfer for $v = 0..frs$.

7. Quiz: how would you modifiy in the simplest way possible the function above to produce a transfer function with given DNL and INL?

3 Implementation of static measures

As described in your document, you'll make a (dynamic) measurement of static figures of merit such as DNL and INL by feeding the ADC with a "perfect" sinewave and histogramming the digital values obtained after a lot of cycles. So now you have to implement the big ones

1. Implement a function that takes a vector of digital samples and the number of bits and computes the histogram (i.e., the frequencies of occurence) of each sample. To wit

```
def histo(bits, samples):
```

should return an array (all arrays are assumed to be `numpy` arrays from now on) of length 2^{bits} that contains in poisition i the number of occurrences of i in the samples vector. Hint: with `numpy.histogram`, this can be a one-liner, or almost.

2. Implement a function that computes the array of DNL(n) values according to the formula in equations 5.16–5.18, given a huge vector of samples and the number of bits.

3. Ditto for the INL, using formula of figure 5.40. Hint: use `numpy.cumsum`.

4. Test the above. You generate a perfect sinewave and pass it through your perfect ADC transfer function. The samples obtained should give DNLs and INLs near to zero. Then do the same to a sinewave passed through your other ADC transfer function, i.e., the one in which you prescribe a DNL or INL. Then, DNL's and INL's obtained should match closely what you prescribed. Do not expect this to work at the first stroke: now things are getting quite complex and this will take some effort to fit.

5. Take a real set of samples from a real ADC and measure its DNL and DNL once you got the above right

4 Implementation of dynamic measures

Here you'll have to implement some functions to compute the SNR, SINAD, ENOB and THD. Essentially, we need functions that take as argument an possibly huge array of samples, do the FFT of it and measure on it the different parameters. So, for example

1. Implement a function that takes an array of samples (integer digital values presumably coming from an ADC sampling a sinewave) and produces its FFT (this is already given by `numpy`)

2. Ditto, with a given window function (Hamming, Bartlett, etc.) Hint: look up the functions `bartlett` or `hamming` in numpy's documentation.

3. Building on the preceding, write a function that takes an array of samples, a frequency value (it is supposed to be the frequency of the pure sine wave the samples came from) and extracts a prescribed number of values for the peaks of the harmonics. To wit:

   ```
   def harmonics(samples, frequency, n):
   ```

   should return an array giving the position of the peaks corresponding to the first n harmonics of the pure sine wave.

4. Same exercise, but this time the function has to return the value of the noise floor (quick and dirty way: the average value of the FFT components that do not exceed a certain threshold). The right way to compute it, though, is to average the FFT when no input is applied.

5. Test interlude: generate a sine, a noisy sine with a known SNR, and a sine with the addition of some other small harmonics of it (say something like

$$\sin 2\pi ft + a_3 \sin 2\pi 3ft + a_5 \sin 2\pi 5ft \tag{2}$$

with a_3, a_5 small coefficients to provide harmonic distortion). Now you know what the position of the peaks for the harmonics should be, and the THD as well. Check that your functions work correctly.

6. Here comes the difficult part: same exercise about harmonic detection, but taking into account (fig. 5.67) that harmonics can be spread if the number of sine periods spanned by the size M of the Fourier transform is not an exact integer. This is mostly the creative part of peak detection Javier mentioned. This is quite an open exercise: it is not easy to define what exactly a peak is. Take as a hint that the harmonic peaks will be at predefined positions in the frequency axis, and a bit spread. Using a windowed FFT will make the peaks sharper.

7. From the preceding functions, you can compose now one that gives, for a sampled array of digital values and a given frequency, the total energy in the fundamental, the distorted harmonics and the floor noise, and you can compute all the measures of SNR, SINAD, ENOB and THD.

8. More tests: with predefined perfect sine waves, perfect sine with prescribed noise, and perfect sine with slight harmonic perturbation, check that the functions you wrote provide the expected results. This, again, will take some effort. It wil *never* work at first time, so do not get discouraged at the fifth iteration. It will come out well at the end.

5 Some style indications

After a period of experimentation with the language and the task at hand, it is the moment to stick to some design style indications, so that the product is as useful as possible.

- The functions described as exercise above should always behave in such a way that do not perform I/O. It is up to the main program, or the caller, to input and/or output the values that a function gets passed. Later, printouts or plots are simply a means to check that everything is OK.

- Try to get as much as possible from `numpy`. Whenever you are looping to construct something, think twice because it is very likely that some pre-canned function (either in the standard library or in `numpy` is giving you what you want for free. A cruel example: if you're given a voltage level v and need to find its place in a vector of voltage levels $[v_0, v_1, v_2, \ldots, v_N]$, you can loop to find it, but `numpy.searchsorted` does it for you in a single stroke.

- Most functions should be ten-liners at most. If a function gets too long, think about splitting it in smaller functions or think about the point above.