Simulation: Digital Receivers and Filtering

Sebastian Magierowski

***Abstract*—Brief notes on a way to construct a simulation of a digital receiver accounting for an analog front-end.**

I. Introduction

Its often the case that we want to simulate the behaviour of some digital components (filter, detectors, etc.), but need to account for the impact of the analog-front end preceding that digital component. This note considers one way of doing this in the context of a receiver handling a simple binary waveform. The approach should be generalizable to other situations.

II. The Signal

The signal that I will be considering is the randomly varying binary signal ideally assuming the form

$$x(t) = \sum_{k=-\infty}^{\infty} Ap(t - kT_{sym}) \tag{1}$$

where $p(t)$ is a rectangular function with amplitude of ± 1 and temporal extent T_{sym} and A is any number you want to represent the amplitude of your signal. Thus we have a randomly varying binary waveform switching between $+A$ and $-A$ with the minimum time at any level equal to T_{sym}.

We can show that the power spectral density (PSD) of this signal (if the data symbols are independent) is

$$S_{xx}(f) = \frac{A^2 |P(f)|^2}{T_{sym}} \tag{2}$$

where $P(f)$ is the Fourier transform (FT) of our symbol $p(t)$.

Grinding through this calculation reveals a PSD of

$$S_{xx}(f) = A^2 T_{sym} \left[\frac{\sin(\pi f T_{sym})}{\pi f T_{sym}} \right]^2 \tag{3}$$

$$= A^2 T_{sym} \mathrm{sinc}^2(f T_{sym}) \tag{4}$$

The total power of this signal is

$$P_x = \int_{-\infty}^{\infty} S_{xx}(f) \mathrm{d}f = \frac{A^2}{T_{sym}} \int_{-\infty}^{\infty} |P(f)|^2 \mathrm{d}f \tag{5}$$

where we can invoke Parseval's theorem

$$\int_{-\infty}^{\infty} |P(f)|^2 \mathrm{d}f = \int_{-\infty}^{\infty} p(t)^2 \mathrm{d}t = T_{sym} \tag{6}$$

to conclude that

$$P_x = A^2. \tag{7}$$

Many thanks to EMIL's friends.

III. The Perfect Front-End

The absolutely simplest scenario is the case where you have a "perfect" (i.e. invisible) analog front-end that does absolutely nothing (good or bad) to your signal. If the digital receiver that you are building is only interested in the level of your received waveform (and does not care to accumulate signal statistics like say a matched filter or a CUSUM device) then you can do an adequate simulation on it, by grabbing only one sample per T_{sym}.

In MATLAB your input signal could then be described with

```
L = 2^16; % Number of symbols to simulate
A = 1; % The symbol amplitude
x = A*sign(rand(1,L)-0.5);
```

If you do this you have essentially created a world where your random binary process **is no longer a rectangular wave**, but essentially consists of sinc pulses. The PSD of this process is as pictured in Fig. 1. Note that our total signal power is still

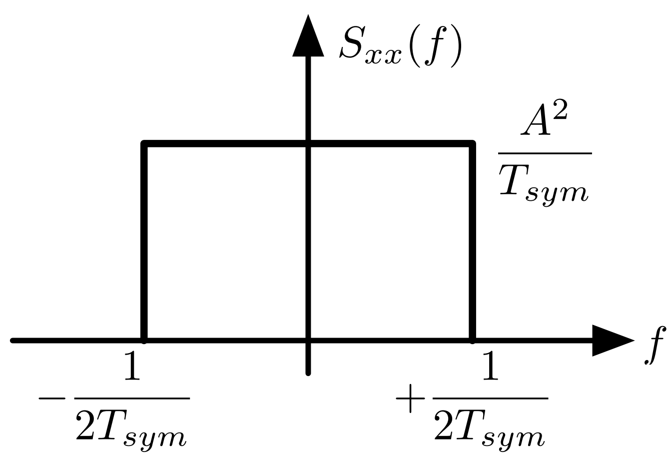

Fig. 1. The effective PSD of your random binary process (in simulation) if you only grab one sample per T_{sym}.

A^2 (i.e. integrate Fig. 1) so our contrivance (i.e. just taking one sample every T_{sym}) captured all the power. Intuitively this should make sense; yes we take only one sample, but we use it to represent a signal over T_{sym} (just like our original continuous-time signal). This changes the power spectrum, but our totals have not been compromised.

If you want to add some random noise to the signal to study the impact of SNR on your system just calculate it by noting that

$$\mathrm{SNR} = \frac{P_x}{P_n} = \frac{A^2}{A_n^2} \tag{8}$$

where P_n is the noise power and A_n is the average noise amplitude of your signal sample (i.e. the square-root of its variance, $\sqrt{\sigma_n^2}$. Remember your simulation will effectively only be taking one sample of noise per T_{sym} as well. Thus the PSD of the noise in this simulation will be identical to that of the signal (albeit at a different level A_n^2/T_{sym}). Based on this, if you stipulate your desired SNR in terms of dB you can execute the following code (or something like it) to generate an appropriate signal and noise sequence

```
L = 2^12; % Number of symbols to simulate
A = 1; % The symbol amplitude
x = A*sign(rand(1,L)-0.5); % Signal
SNRdB = 5; % SNR in dB
An = A*10^(-SNRdB/20) % Noise amp
n = An*randn(1,L); % Noise
xn = x + n; % Signal + Noise
```

Recall that MATLAB's `randn` function returns a sequence of random numbers with $N(0,1)$ (i.e. Normal distribution with average of zero and variance of 1). Using the relationship

$$\alpha + \beta N(\mu, \sigma^2) = N(\alpha + \beta\mu, \beta^2\sigma^2) \tag{9}$$

which shows the impact of scaling on the first and second moments of a Gaussian distribution we can appreciate that the calculation `An*randn(1,L)` above achieves the desired noise scaling needed for the stipulated SNR.

IV. Nonuniform Noise PSD

In the ideal case considered above you might wonder about the impact of nonuniform (i.e. not white!) noise distributions. For example in nanopores we've talked about noise that whose PSD, $S_{nn}(f)$, increases as a function of frequency (e.g. flicker and capacitive noise).

In the simulation procedure above you **cannot directly retain** the impact of non-uniform PSD as the single-sample-per-T_{sym} approach makes all spectra white. But of course you **can indirectly retain** the impact of non-uniform PSD. The simplest way to do this is to imagine that an ideal (brickwall) filter with cutoff frequency (f_c) is used to filter both the original signal and the noise and then you sample these filtered signals every T_{sym}. You use a filter because you need some standard for the amount of non-uniform noise PSD that you are going to take. Since the PSD is non-uniform the total power you capture depends on f_c. To be fair and consistent though you need to capture a like amount of signal spectrum too (recall from (3) that your signal's PSD is **also non-uniform**!!!!).

Following this strategy effectively extracts

$$P_{x,fc} = \int_{-f_c}^{f_c} S_{xx}(f)\mathrm{d}f \tag{10}$$

and

$$P_{n,fc} = \int_{-f_c}^{f_c} S_{nn}(f)\mathrm{d}f. \tag{11}$$

Using $A_{fc}^2 = P_{x,fc}$ and $A_{nfc}^2 = P_{n,fc}$ you can reform your simulation sequences in ways similar to those given above. Below is an example of how I might do it (I use particular expressions and constants for my signal, but this is of course just meant to be an example, use whatever expressions are appropriate for the signals that you are dealing with)

```
% ===== Constants and Settings =====
L = 2^12; % Number of time points
fsym = 2e6; % The symbol rate
Tsym = 1/fsym; % Inv. of symbol rate
fc   = 0.8*fsym; % Filter cutoff freq.
N    = 2^10; % Number freq. points
freq = fc*linspace(-1,1,N); % freq. axis
% ===== Signal PSD & Power =====
Sxx  = (0.001)^2*Tsym*(sinc(freq/fsym)).^2;
Pxfc = trapz(freq,Sxx); % Integrate PSD
A = sqrt(Pxfc);
x = A*sign(rand(1,L)-0.5); % Signal
% ===== Noise PSD & Power =====
vn2  = (5e-9)^2; % Amp noise, nV^2/Hz
Rs   = 60e6; % Sensor resistance
Ct   = 2.5e-12; % Amp capacitance
Snn  = vn2*(1 + (freq*Ct*Rs/2/pi).^2);
Pnfc = trapz(freq,Snn); % Integrate PSD
An   = sqrt(Pnfc);
n    = An*randn(1,L); %Noise
% ===== Noise and Signal =====
xn = x + n;
% ===== Effective SNR =====
SNR   = Pxfc/Pnfc; % Your SNR for this noise
SNRdB = 10*log10(SNR).
```

V. Multi-Level Signal

If your signal consists of multiple levels, $a_1, \ldots, a_N$ (i.e. rather than just $\pm A$ as discussed above) you can still pretty much employ the same procedure discussed until now.

First off, generating your multiple level signal sequence should be pretty straightforward. Here's an example that uses logical indexing to generate a signal sequence whose amplitude is uniformly distributed among four levels.

```
Levels = [-1,-0.2,0.3,1.1]; % Possible levels
L = 10; % Number of symbols to simulate
vec = rand(1,L); % Raw random sequence
% === Logical Indexing to Map Sequence ===
x(vec <=0.25) = Levels(1);
x(vec > 0.25 & vec <= 0.50) = Levels(2);
x(vec > 0.50 & vec <= 0.75) = Levels(3);
x(vec > 0.75 & vec <= 1.00) = Levels(4);
```

To carry out the other calculations above you just need an equivalent expression for A and then you treat you multi-level signal just as above (i.e. a 2-level signal with amplitude A). That equivalent (for a uniform amplitude distribution) is

$$A = \sqrt{\frac{1}{N}\sum_{i=1}^{N} a_i^2}. \tag{12}$$

VI. Realistic Filtering in Time

And what if you actually have some more realistic analog (or digital) filtering between your signal and digital detector as shown in Fig. 2.

It is certainly possible to handle such a scenario directly in the time domain as sketched out with the MATLAB code below.

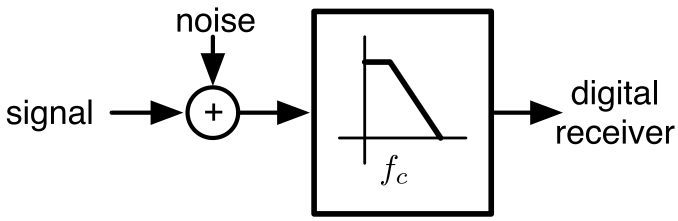

Fig. 2. What if the signal is processed by some realistic filter?

```
1  L = 2^14; % Number of symbols to simulate
2  % ==== Signal Params ====
3  fsym = 1e6; % Symbol rate (event rate)
4  Tsym = 1/fsym; % Symbol period
5  R    = 10; % Oversample rate
6  x = 1.0*sign(rand(1,L)-0.5); % Raw sig
7  xRi = rectpulse(x,R); % Oversampled sig
8  % ==== Simulation Params ====
9  fsamp = fsym*R; % Sampling rate
10 fnyq = fsamp/2; % Nyquist rate
11 % ==== Noise Filter ====
12 fcn = 0.5*fsym; % Noise filter cutoff
13 ordernoise = 20; % Noise filter order
14 f = linspace(0,1,24); % Noise filter freq
15 fcnnorm = fcn/fnyq; % Normalized cutoff
16 amp = 1+(f/fcnnorm).^2; % Freq shape
17 % Noise filter coefficients
18 [bn,an] = yulewalk(ordernoise,f,amp);
19 % freqz(b,a); % Check freq response
20 % ==== Noise ====
21 nwhite = 0.1*randn(1,R*L); % White noise
22 n = filter(bn,an,nwhite); % Colored noise
23 % ==== Signal + Noise ====
24 xRin = xRi + n;
25 % ==== Filter Params ====
26 fc   = fsym/2; % Filter cutoff frequency
27 fcnorm = fc/fnyq; % Normalized cutoff
28 order = 1; % Filter order
29 [b,a] = butter(order,fcnorm,'low'); %
      Filter taps
30 % ==== Filtered Signal + Noise
31 xRo = filter(b,a,xRin,-1);
```

As I hope is clear this code simulates a random binary waveform flipping between ± 1 at an average rate of $f_{sym} = 1$ MHz. Note that in line 5 we define an oversample rate with $R = 10$. This is very important. We now need to capture a potentially sophisticated filter and this will probably have a frequency above f_{sym} so we create a sampling frequency of $f_{samp} = Rf_{sym}$. Note now how the input signal is created in lines 6 and 7. It start off the same as before (line 6), but then I used the `rectpulse` function (alas only available in MATLAB's Communications Toolbox) to oversample the original signal without compromising the rectangular nature of each pulse at all (if you blindly just use a MATLAB function line `interp1` to try the same thing then you will unwittingly distort your signal due to incorrect interpolation between points).

Notice that I first create another filter (using `yulewalk` in line 18) to create a filter representing what the effective shape of the input noise will be. I then create samples pertaining to this noise in line 22 and thus realize our colored noise if need be (e.g. capacitive/violet noise in the nanopore system).

Then in lines 26-29 I create the filter whose effect we want to study (I just use a simple 1st order Butterworth filter in this example, alas note that in order to use the `butter` function you need MATLAB's Signal Processing Toolbox). Running my net signal through this filter (line 31) produces the desired output. If you are following this with a digital processor that is only interested in one sample per T_{sym} then you can safely decimate the output and grab one sample (i.e. grab every Rth sample). **Just be careful** that when you decimate you grab a sample from about the middle to T_{sym} otherwise you might be grabbing samples more subject by transient effects.

VII. Realistic Filtering in Frequency

If you are only ever interested in one sample per T_{sym} then you can achieve a result equivalent to the above without having to oversample at all. You achieve this, by simply working in the frequency domain. You find the power spectral expressions for your input signal ($S_{xx,i}$) and noise ($S_{nn,i}$). You then utilize an expression for the transfer function of the filter you are studying $|H(f)|$ and you use these to find the output power spectral densities

$$S_{xx,o} = S_{xx,i}|H(f)|^2 \tag{13}$$
$$S_{nn,o} = S_{nn,i}|H(f)|^2 \tag{14}$$
$$\tag{15}$$

Taking the integral of $S_{xx,o}$ and $S_{nn,o}$ as in (5) (again, use `trapz` in MATLAB) will give you the P_x and P_n values that you can use in your simple one-sample-per-T_{sym} simulations that we described above. Example code that accomplishes these calculations is shown below.

```
% ==== Signal Params ====
fsym = 1e6; % Symbol rate (event rate)
Tsym = 1/fsym; % The symbol period
R    = 10; % The oversample rate
freq = logspace(0,6+log10(R),1e6);
% ==== The Signal ====
A    = 1.0; % Signal amplitude
Sxxi = A*2*Tsym*(sinc(freq/fsym)).^2;
% ==== The Noise ====
An   = 1e-4; % Noise std. dev
fcn  = 0.5*fsym; % Noise filter cutoff
Snni = An^2*(1+(freq/fcn).^2);
% ==== The Filter ====
order=1;
fc   =fsym/2; % Filter's cutoff frequency
Hfilt=1./(sqrt(1+(freq/fc).^(2*order)));
% ==== The Output Signal & Noise ====
Sxxo = Sxxi.*((abs(Hfilt)).^2);
Snno = Snni.*((abs(Hfilt)).^2);
Px = trapz(freq,Sxxo); % Sig power
Pn = trapz(freq,Snno); % Noise pwr
```