Signal Processing

Naureen Ghani

December 9, 2017

Introduction

Signal processing is used to enhance signal components in noisy measurements. It is especially important in analyzing time-series data in neuroscience. Applications of signal processing include data compression and predictive algorithms.

Data analysis techniques are often subdivided into operations in the ***spatial domain*** and ***frequency domain.*** For one-dimensional time series data, we begin by ***signal averaging*** in the spatial domain. Signal averaging is a technique that allows us to uncover small amplitude signals in the noisy data. It makes the following assumptions:

1. Signal and noise are uncorrelated.
2. The timing of the signal is known.
3. A consistent signal component exists when performing repeated measurements.
4. The noise is truly random with zero mean.

In reality, all these assumptions may be violated to some degree. This technique is still useful and robust in extracting signals.

Simulation of Signal Averaging

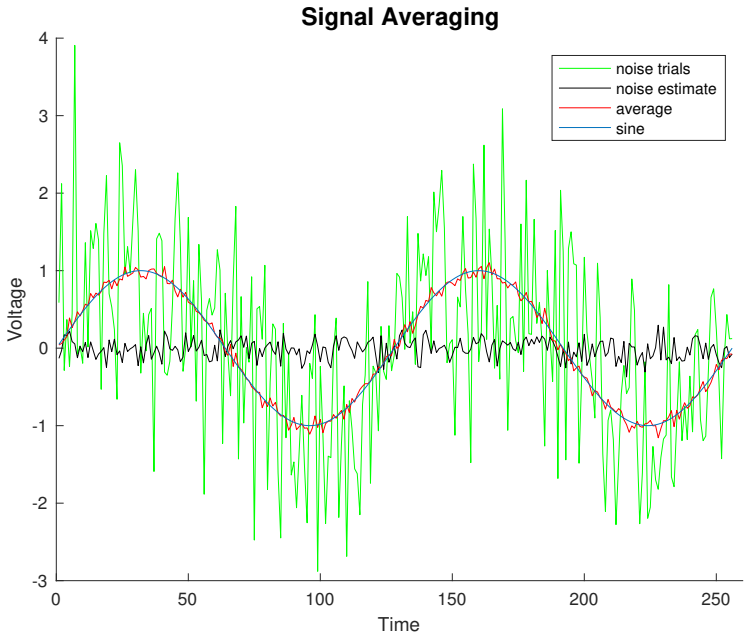
To simulate signal averaging, we will generate a measurement x that consists of a signal s and a noise component n. This is repeated over N trials. For each digitized trial, the kth sample point in the jth trial can be written as

$x_j(k) = s_j(k) + n_j(k)$

Here is the code to simulate signal averaging:

```
% Signal Averaging Simulation

% Generate 256 noisy trials
trials = 256;
noise_trials = randn(256);

% Generate sine signal
sz = 1:trials;
sz = sz/(trials/2);
S = sin(2*pi*sz);

% Add noise to 256 sine signals
for i = 1:trials
    noise_trials(i,:) = noise_trials(i,:) + S;
end

% Average of 256 noisy signals
avg = sum(noise_trials)/trials;
odd_avg = sum(noise_trials(1:2:end,:)/(trials/2));
even_avg = sum(noise_trials(2:2:end,:)/(trials/2));
```

```
noise_estimate = odd_avg-even_avg;

% Create Figure
figure;
hold;
plot(noise_trials(1,:),'g');
plot(noise_estimate, 'k');
plot(avg,'r');
plot(S);
legend('noise trials','noise estimate','average','sine');
title('Signal Averaging', 'FontSize', 14);
xlabel('Time');
ylabel('Voltage');
xlim([0 260]);
hold off;
```



This figure shows how the averaging process results in an estimate of the signal. As compared to the original signal in green, the averaged noise component is reduced in a signal average of 256 trials in red. When averaging real signals, the underlying small-amplitude component may not be as clear. For this reason, the noise estimate is repeated using different parts of the data. One common way to do this is to average all ***odd*** and all ***even*** trials in separate buffers. The average of these two values is the ***noise average***, while the difference of the two is the ***noise estimate***.

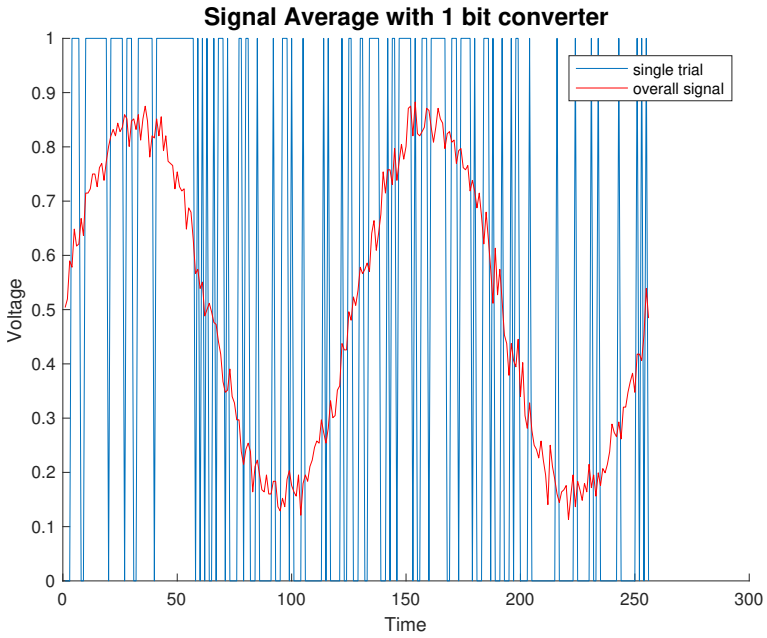
Noise

To produce a quality signal measurement, it seems intuitive that a high-precision analog-to-digital converter (ADC) would be needed. However, ADC precision is not critical to signal averaging and noise can be helpful when measuring signals through averaging. In other words, the averaging process, which aims to reduce noise, may work better if some noise is present in signals.

Assume you have a 1-bit ADC. There are only two values for the signal: 0 or 1. In this case, a small deterministic signal without added noise cannot be averaged or measured because it would result in the same uninformative series of 0s and 1s in each trial. If we now add noise to the signal, the probability of finding a 0 or 1 sample is proportional to the signal's amplitude at the time of each sample. By averaging the results of multiple trials, we can obtain a probabilistic representation of the signal.

Let us use the data set of 256 trials in the previous example to simulate signal averaging with a 1-bit ADC. Here is the code as follows:

```
% Analog to Digital Conversion

for k = 1:trials
    for m = 1:trials
        if (noise_trials(k,m) < 0 )
            noise_trials(k,m) = 0;
        else
            noise_trials(k,m)=1;
        end
    end
end

avg2 = sum(noise_trials)/trials;
figure;
hold;
plot(noise_trials(1,:));
plot(avg2, 'r');
legend('single trial', 'overall signal');
title('Signal Average with 1 bit converter', 'FontSize', 14);
xlabel('Time');
ylabel('Voltage');
hold off;
```



The averaged result using a 1 bit converter is surprisingly similar to the signal average in the previous example. This shows us how reasonable averaging results can be obtained with a low-resolution ADC using the statistical properties of the noise component. This suggests that the ADC resolution may not be a critical component in the signal averaging technique.

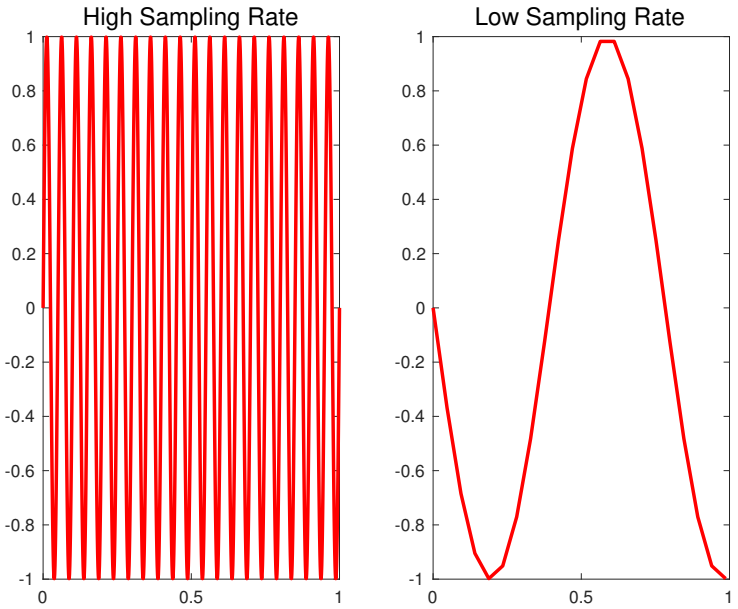
Sampling Rate and Nyquist Frequency

Sampling rate is the number of samples per second taken from a continuous signal to make a discrete or digital signal. For time-domain signals, frequencies are measured in hertz (Hz) or cycles per second. The ***Nyquist-Shannon sampling theorem*** states that perfect reconstruction of a signal is possible when the sampling frequency is greater than twice the maximum frequency of the signal being sampled. For example, if an audio signal has an upper limit of 20,000 Hz (the approximate upper limit of human hearing), a sampling frequency greater than 40,000 Hz will avoid

aliasing and enable signal reconstruction. ***Aliasing*** is a sampling effect that leads to spatial frequencies being falsely interpreted as other spatial frequencies.

Here is a simulation of waveforms at different sampling rates:

```
% Define parameters

t = 0:0.001:1;
f = 20;
signal = sin(2*pi*f*t);

% Simulate different sample rates and plots
figure;
for i = 2:5:50
    subplot(1,2,1); plot(t,signal,'r', 'Linewidth', 2); title('High
       Sampling Rate', 'FontWeight', 'normal', 'FontSize', 14);
    subplot(1,2,2); plot(t(1:i:1000),signal(1:i:1000), 'r', 'Linewidth',
       2); title('Low Sampling Rate', 'FontWeight', 'normal', 'FontSize',
        14);
end
```



Signal Filtering

A ***filter*** is any process to remove frequencies from a signal. In neurophysiological analysis, we use a linear, continuous-time filter. Data is filtered as it is collected by analog and digital systems. The frequency bands we remove depend on the hardware that is used and on settings determined in the recording software.

The type of filter we use depends on what we are interested in. If we are interested in higher frequencies, we use a ***high-pass filter***; for lower frequencies, ***a low-pass filter***. ***Band-stop filters*** remove a range of frequencies. A special case of this, the ***notch filter,*** removes only a single frequency. Notch filters are often used to remove ***60 Hz***, which is the frequency of the alternating current coming from the electrical outlets and can contaminate our recordings.

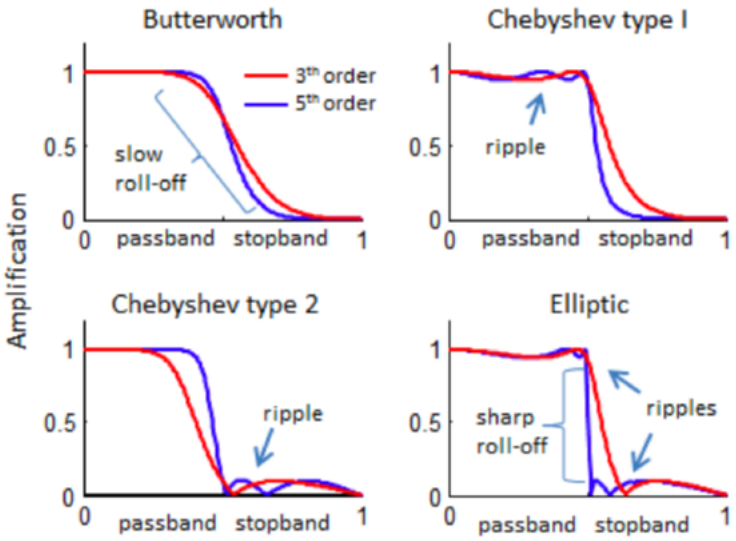
One method to filter a signal is to perform a Fourier transformation, remove coefficients within the stop-band, and then reconstruct the signal with the remaining coefficients. In the strictest sense, this is called a ***truncation filter.*** A common filtering method is to apply a differential equation:

$y(n) = (b_0x(n) + b_1x(n-1) + ... + b_Nx(n-N)) - (a_1y(n-1) + a_2y(n-2) + ... + a_My(n-M))$

Filters are often divided into two classes: ***finite impulse response (FIR)*** and ***infinite impulse response (IIR)***. FIR filters generate a filtered signal using only a fixed interval (using only the ***b coefficients*** in the equation).

In contrast, IIR filters use ongoing state changes as the filter slides across samples of the input signal (using only the ***a coefficients*** in the equation).

Bandpass filters use a range of frequencies to filter. Four major types of bandpass filters are ***Butterworth, Chebyshev type I, Chebyshev type II, and Elliptic filters.*** These techniques vary in how they balance the ***"roll-off"*** with the ***"ripple."*** The "roll-off" is the frequencies you do not want (also called the ***stop band***). The "ripple" describes how similar the range of frequencies in the passband are (how "flat" the peaks are).

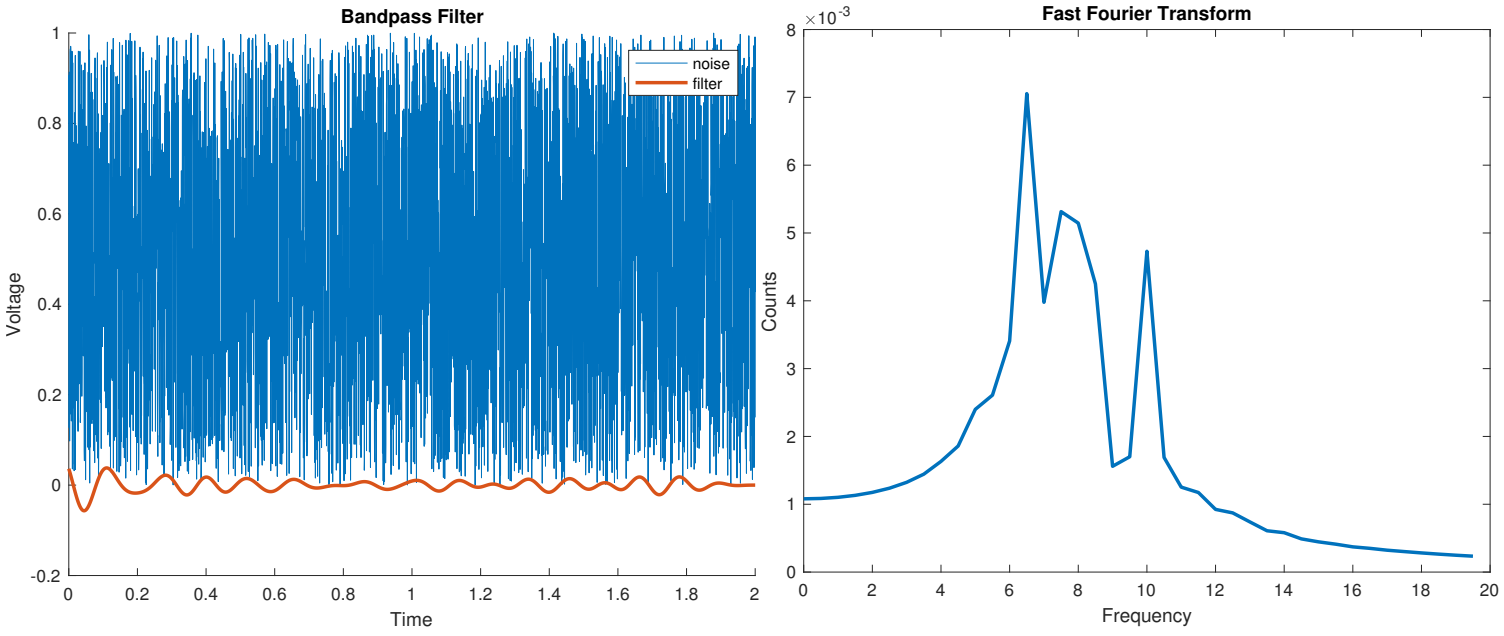


An important parameter when setting up a filter is the ***filter order.*** The order refers to the number of coefficients used in determining the filter function. It will determine the balance between time versus. frequency resolution.

In the following simulation, we filter a random signal to generate a new signal that oscillates between 6 to 10 Hz. To confirm that the filter worked, we decompose the signal with a Fourier transform. Here is the code:

```
% Generate Random Noise

noiselfp = rand(1, 4001);
figure;
hold;
plot(linspace(0,2,4001),noiselfp);

% Filter Noise at 6-10 Hz

% Nyquist freq is half the sampling rate and represents
% the highest frequency the data can accurately represent
Nyquist_freq = 2000/2;
low_freq = 6/Nyquist_freq; %lower freq bound
high_freq = 10/Nyquist_freq; %higher freq bound

filter_order = 3; % number of filter exponents
passband = [low_freq high_freq];
% Butterworth filter coefficients
[Bc Ac] = butter(filter_order, passband);
LFP_filt = filtfilt(Bc,Ac, noiselfp);
plot(linspace(0,2,4001), LFP_filt, 'Linewidth',2);


% Label figure
legend('noise','filter');
title('Bandpass Filter');
xlabel('Time'); ylabel('Voltage');
hold off;
```

```
% Decompose filtered signal with a Fourier transform
y = fft(LFP_filt)/4001; %divide by num of samples
f = 2000/2 * linspace(0,1,4001/2 + 1);
figure; plot(f(1:40), abs(y(1:40)),'Linewidth',2);
title('Fast Fourier Transform');
xlabel('Frequency'); ylabel('Counts');
```



In this example, the amplitude of each frequency between 6 to 10 Hz is quite variable. This is a result of the short time period of our sample and the chance outcome of our random number generator. The non-zero values before 6 Hz and after 10 Hz demonstrate one tradeoff in using the Butterworth filter: there will be a relatively smooth roll-off in the stopband, with the advantage that we don't introduce ripples.

Units in Fourier Transform

1. **Record raw signal:** units are ***voltage,*** values are ***real.*** We begin with a signal of time-varying voltage $v(t)$.
2. **Apply Fast Fourier Transform (FFT):** units are ***voltage,*** values are ***complex.*** A signal is always real. However, for a compact representation of the sum of sinusoids, the FFT uses complex exponentials. Since we are representing a real number by a complex-valued entity, we need to add both the complex number and its complex conjugate. For example, if we have a real signal represented as $cos(\theta) + j \times sin(\theta)$, then we also need to add its complex conjugate pair. The complex conjugate is $cos(\theta) - j \times sin(\theta) = exp(-j \times \theta)$. Since $\theta = 2 \times \pi \times f$, we have a ***negative frequency*** when we plot the FFT amplitude (counts) vs. frequency plot. *In summary, negative frequencies are a mathematical side effect of using complex exponentials as our representation of Fourier transformations.*
3. **Take the absolute value:** units are ***voltage,*** values are ***real.*** Magnitude of signal components .
4. **Square the result:** units are $voltage^2$**,** values are ***real.*** Square of magnitude of signal components. This result can be used to create a plot of ***power spectral density.***