LMS Adaptation Using a Recursive Second-Order Circuit

in partial fulfillment of the requirements for
EE373A
Adaptive Signal Processing
Prof. Bernard Widrow
Winter, 1998
Stanford University

Jon Dattorro

Introduction

The purpose of this project is to detect the frequency of a lone sinusoid, of unknown phase and amplitude, existing in uncorrelated WSS noise. The problem is made difficult by demanding that there exist no reference signal; i.e., no "desired" signal $d[n]$. Only the signal $x[n]$ is known which is composed of one sinusoid and a fair amount of noise.

To make the computation efficient we adapt the coefficients of a recursive structure, an IIR filter as opposed to the traditional FIR. As is well known, the performance surface using an IIR structure is multi-modal. [1,ch.8] Hence there is no unique minimum, in general, and we would be forced to search the entire coefficient space. But in our particular application of adaptation, there exists one unique global minimum under the assumptions that the signal is a lone sinusoid with additive uncorrelated noise, and that the noise power is spectrally widespread.

Viable applications of the technique proposed in this project might include fundamental frequency detection, phase-locked loop, or sonar. We demonstrate the dynamic characteristics of our detector by subjecting it to frequency modulated inputs of two types: frequency sweeps and FM.

C Program

We include as an Appendix the simple C program simulation which is the verification of this method. Finite register-length effects are not considered. We generate one record of length 32767 samples of sinusoidal signal plus noise. Then the linear combination is fed to the recursive adaptive filter and the adaptation takes place. The record length is chosen as one period of a pseudo-random number sequence that we selected.

Noise Generation

The noise generator subroutine is listed in the C program. The method we employ to generate the uncorrelated noise is known in the literature as a maximal length pseudo-random number sequence; [2] rather, a PN sequence. A realization of this noise is shown in Figure 1 which is zoomed all the way in. Figure 2 shows the envelope of the realization.

Because the PN generator is a marginally stable recursive circuit, the pseudo-noise is periodic. The period is determined by the generator register width. The generator register wordlength is 15 bits, hence a sequence of period 32767 ($=2^{15}-1$) is produced. Each 15-bit word of the sequence is unique. This multi-bit sequence has a spectrum as shown in Figure 3. The spectrum is colored as a consequence of the fact that we are extracting a multi-bit output from the noise generator which was originally designed for a single-bit output. [2] Hence the resulting sequence is not perfectly uncorrelated. But the circulant covariance matrix can be shown to be banded-diagonal having one narrow band along the main diagonal of width approximately equal to 15 samples in our case. The circular autocovariance is relatively brief indeed.

Sinusoid Generator
The sinusoid generation subroutine in our C program simulation is capable of FM, logarithmic sweeps in frequency, and amplitude damping. We demonstrate only tone modulated FM and sweeping.

Throughout this project we maintain a sample rate of $F_s = 22050$ Hz which is a standard audio rate; albeit, a low one.

Signal to Noise Ratio
In this project, we leave the **SNR** constant at 20 dB. We determined the proper amplitude of the noise by noting that our pseudo-random noise generator is designed to create realizations having uniform pmf $\in$ (-1/2,1/2), hence $\sigma_\eta^2 = 1/12$. In order to achieve the proper **SNR** we must scale the noise sequence by the constant c that satisfies the following specification:

$$\mathbf{SNR} = 10\log\left(\frac{\sigma_s^2}{c^2\,\sigma_\eta^2}\right)$$

$$\sigma_s^2 = \frac{A^2}{2}$$

where A is the sinusoid $s[n]$ amplitude.

The Recursive Notch Filter
The coefficients of the second order recursive filter we use are set to produce a notch having an absolute zero somewhere along the Fourier frequency axis. The notch bandwidth is always fixed and does not change during adaptation.

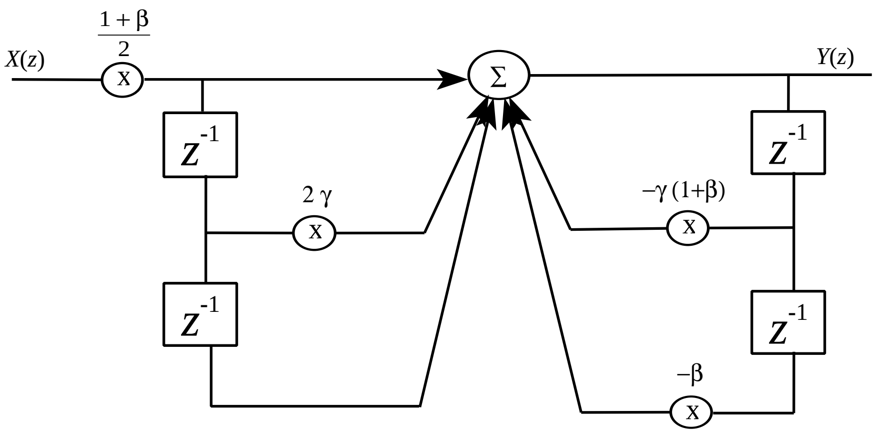

Figure 4. The recursive notch filter, $H_n(z)$.

The filter coefficients in Figure 4 are chosen so as to maintain a fixed specified bandwidth and to maintain a zero of transmission right on the unit circle.

$$\gamma = -\cos(\omega_c) \tag{1}$$

$$\beta = \frac{1 - \tan(\Delta\omega/2)}{1 + \tan(\Delta\omega/2)}$$

where $\Delta\omega$ is the filter bandwidth in radians normalized to 2π, and ω_c is the normalized radian frequency of the notch center. Note that β is not a function of ω_c. The transfer function of the notch filter is

$$H_n(z) = \frac{1}{2}(1+\beta)\frac{1 + 2\gamma z^{-1} + z^{-2}}{1 + \gamma(1+\beta)z^{-1} + \beta z^{-2}} \tag{2}$$

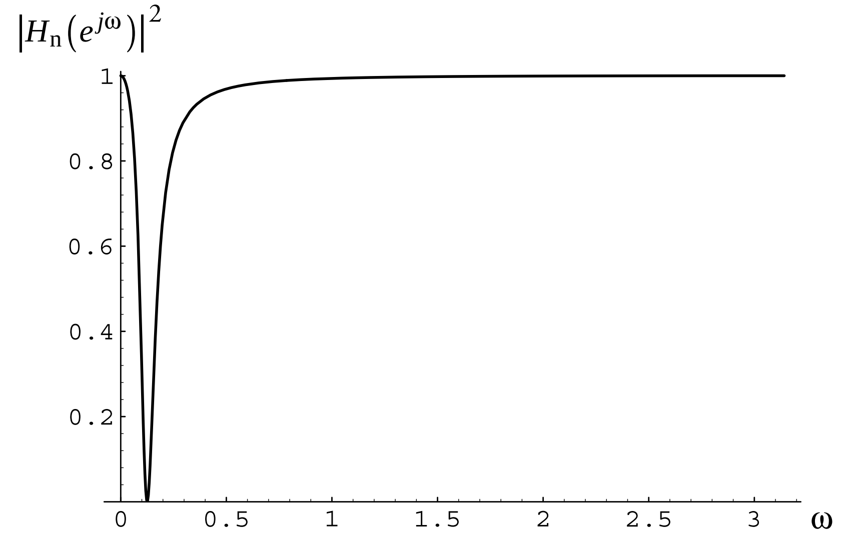

Figure 5. Typical shape of a low-frequency notch.

Note the characteristic shape of the notch in Figure 5 which is pretty flat (for a second-order filter) in the region outside the stopband.

The Adaptive System

One of the reasons that this notch filter is likely to adapt well is that the bandwidth is not subject to adaptation. This fact counteracts the performance surface' potential increase of modality due to the IIR structure, and makes our problem simpler. Empirically we have learned that the absolute filter bandwidth plays some kind of role in determining whether the adaptive system can acquire lock. In other words, if the initial filter center frequency is too far away from that of the incoming sinusoid, then the system is not guaranteed to converge to the signal frequency. We do not explore this filter bandwidth dependency problem in this project, but we later conjecture as to its cause. The bandwidth is fixed at 300 Hz unless otherwise specified.

During adaptation, the filter zero never strays from the Fourier frequency axis. This is insured by the monic-numerator coefficient 1 in Equ. (2) multiplying z^{-2}.

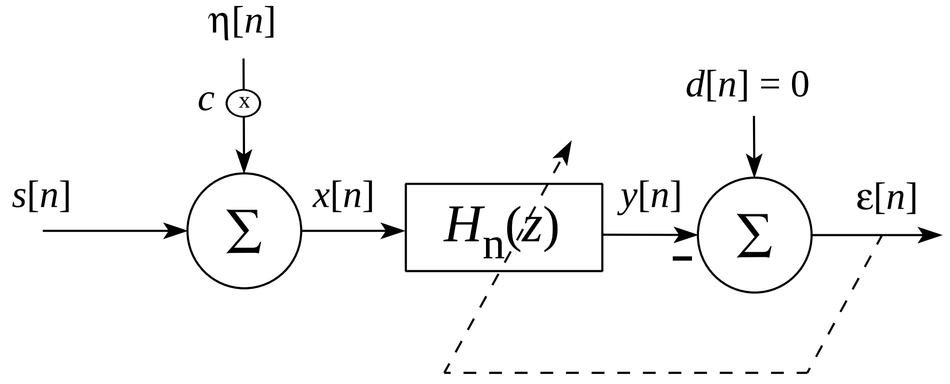

Figure 6. The complete adaptive system.

Figure 6 shows the scheme we explore in this project. In the figure, $s[n]$ is the unknown sinusoidal signal, $\eta[n]$ is the pseudo-noise, $d[n]$ is the traditional desired signal, and $\varepsilon[n]$ is the error signal. The desired signal $d[n]$ is assumed not provided in this application. Hence we have blind adaptation.

We use LMS-type adaptation of the γ coefficient as it controls notch-center frequency ω_c directly as in Equ. (1). γ controls ω_c independently of the notch bandwidth $\Delta\omega$. The γ coefficient is the only coefficient that requires adaptation in our system.

$$\varepsilon[n] = -y[n]$$

$$\hat{\xi}_n \equiv \varepsilon^2[n] \qquad (3)$$

This time-varying estimate of MSE is from [1,ch.6,pg.100]. From the definition for the gradient and Equ. (1) we may also say

$$\hat{\nabla}_n \equiv -\frac{d\hat{\xi}_n}{d\gamma}$$

$$\cos(\omega_c)_{n+1} = \cos(\omega_c)_n - \mu\,\hat{\nabla}_n \qquad (4)$$

The gradient estimate is a scalar in our application. We write the LMS coefficient update [ibid.] equation [Equ. (4)] in terms of ω_c instead of γ because it is more meaningful to the notch filter.

From Equ. (2) we determine

$$y[n] = \frac{1}{2}(1+\beta)(x[n] + 2\gamma\, x[n-1] + x[n-2]) - \gamma(1+\beta)\,y[n-1] - \beta\, y[n-2]$$

and using Equ. (3) we say

$$\frac{d\hat{\xi}_n}{d\gamma} = \frac{d\left(y^2[n]\right)}{d\gamma} = 2\,y[n]\,\frac{dy[n]}{d\gamma}$$

and so we get

$$\frac{dy[n]}{d\gamma} = (1+\beta)\left(x[n-1] - y[n-1] - \gamma\frac{dy[n-1]}{d\gamma}\right) - \beta\frac{dy[n-2]}{d\gamma}$$

$$\hat{\nabla}_n = -2\,y[n]\,\frac{dy[n]}{d\gamma}$$

This nice recursion allows us to calculate the derivative of the filter output by retaining two older values of it. [1,ch.8,pg.155] These last two equations along with Equ. (4) may actually be found implemented in the C program in the Appendix.

The Performance Surface
From [1,Equ.(7.65),pg.133] we determine that

$$\xi = \frac{1}{2\pi}\int_{-\pi}^{\pi} \left|H_{\mathrm{n}}\left(e^{j\omega}\right)\right|^2 \Phi_{xx}\left(e^{j\omega}\right) d\omega$$

In our particular case we have

$$x[n] = s[n] + c\,\eta[n]$$

which are independent and zero mean, hence

$$\Phi_{xx}\left(e^{j\omega}\right) = \Phi_{ss}\left(e^{j\omega}\right) + c^2\,\Phi_{\eta\eta}\left(e^{j\omega}\right)$$

$$\Phi_{ss}\left(e^{j\omega}\right) = \frac{A^2\pi}{2}\left(\delta(\omega-\omega_o) + \delta(\omega+\omega_o)\right)$$

$$\Phi_{\eta\eta}\left(e^{j\omega}\right) \approx \sigma_\eta^2$$

where we have approximated the noise as being white (Figure 3), and where ω_o is the frequency of the incoming sinusoid $s[n]$. So ξ becomes

$$\xi = \frac{A^2}{2}\left|H_{\mathrm{n}}\left(e^{j\omega_o}\right)\right|^2 + \frac{c^2\,\sigma_\eta^2}{\pi}\int_0^{\pi} \left|H_{\mathrm{n}}\left(e^{j\omega}\right)\right|^2 d\omega$$

$$\xi = \frac{A^2}{2}\left|H_{\mathrm{n}}\left(e^{j\omega_o}\right)\right|^2 + c^2\,\sigma_\eta^2\,\frac{(1+\beta)}{2}$$

The succinct result of the integration was discovered with the aid of *Mathematica*. The performance surface is only two-dimensional because there is only one independent variable, γ. Because β is not a function of γ, the performance surface directly follows the shape of $H_{\mathrm{n}}(e^{j\omega})$ (Figure 5) which *is* a function of γ, the adaptation coefficient. Hence in the case of a single input sinusoid plus white noise input, the performance surface is indeed uni-modal; i.e., there is one unique global minimum, hence no local minima.

We conjecture that the difficulty in achieving lock in practice may be due to the shape of the notch being too flat when the center frequency is far away from the signal frequency. The noisy gradient bobbles about zero for what seems to be an endless period.

Simulation

We found it necessary to limit the γ coefficient to the range (-1,1). That is the function of the subroutine called *saturate*(). Otherwise, the filter would cease being a notch and the adaptive system would go unstable.

We find that μ is unusually small for convergence in this application. Otherwise, the adaptive system easily goes unstable. Typical values are in the region $\mu = 0.000001$.

In Figure 7 we show the negative error signal $y[n]$ for the case of adaptation when the sinusoid frequency is constant and the sinusoid is covered by noise 20 dB down. The system has pretty much adapted in about 5000 samples. The relevant parameters are listed in the figure. After adaptation, the sinusoid cannot be heard in the background noise. Figure 8 shows the adaptation trajectory of center frequency for this example.

Figure 9 shows the negative error signal for a sinusoidal sweep in -20 dB noise. The sweep goes from 440 to 220 Hz in about 1.5 seconds. After acquiring lock, the adaptation holds onto the dynamic signal well and the sinusoid is not audible. Adaptation occurs in about 7000 samples for this example. The other relevant parameters are listed in the figure. Figure 10 shows the adaptation trajectory and tracking of center frequency for this example.

Figure 11 shows the negative error signal for a 440 Hz carrier modulated by a 6 Hz sinusoid with a depth of ± 83 Hz (+3 semitones). Like before, the additive noise is 20 dB down. After acquisition, the adaptive system has some trouble tracking this fast moving undulating tone as evidenced by the audible modulated tone along with the background noise at the filter output. Adaptation occurs in about 4000 samples for this example. The other relevant parameters are listed in the figure. Figure 12 shows the adaptation trajectory and tracking of center frequency for this FM example. Figure 13 and Figure 14 show the improved result of increasing μ to 0.000005, but we find empirically that there is a very narrow range of μ over which the system will acquire lock. We call this *finicky* behavior. The sinusoid is no longer audible after acquisition.

Conclusions

The theoretical existence of one unique global minimum in the performance surface is not borne out by our experiments. We have conjectured that the reason is due to the too-flat shape of the notch filter itself. We find the adaptation to be finicky and very easily pushed into instability. When the system acquires lock, it holds steadfast, however.

References

[1] B. Widrow, S.D. Stearns, *Adaptive Signal Processing*, 1985, Prentice-Hall

[2] F. Jessie MacWilliams, Neil J. A. Sloane, ‘Pseudo-Random Sequences and Arrays’, *Proceedings* of the IEEE, vol.64, no.12, pg.1715, 1976 December

APPENDIX - C Program

```
//  Sinusoid tracking in noise simulation - Jon Dattorro

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/file.h>

#define TRUE         1
#define SWEEPING     !TRUE
#define INTERVAL     0.5         //Musical interval of sinusoid SWEEP;
                                 //e.g., 8ve, semitone.
#define DAMPING      0.          //exponential decay of sinusoid
#define SINE_PHASE   0.
#define Fs           22050.      //sample rate
#define SAMPLES      32767

#define FM           TRUE
#define f_modulator  6.          //Hz, FM
#define F_DELTA      3.0         //how many semitones up, FM
#define A            0.5         //sinusoid amplitude
#define fo           440.        //sinusoid frequency in Hz
#define CENTER       800.        //initial center frequency, Hz
#define BANDWIDTH    300.        //bw of filter in Hz
#define SNR          20.         //signal to noise in dB
#define Mu           0.000005

char string1[48] = "sndconvert -c 1 -r -o noise.snd noise";
char string2[48] = "sndconvert -c 1 -r -o sinusoid.snd sinusoid";
char string3[48] = "sndconvert -c 1 -r -o y.snd y";
char string4[48] = "sndconvert -c 1 -r -o fc.snd fc";
int tap, wordlength;
unsigned noisereg[32];                       //time origin at n=2
double pow2to[48+1], noise[SAMPLES+2], sinusoid[SAMPLES+2],
                                 x[SAMPLES+2], y[SAMPLES+2];
short shortnoise[SAMPLES+2], shortsinusoid[SAMPLES+2];
short shortfc[SAMPLES+2], shortfiltered[SAMPLES+2];
double sinamp, twopi, f_delta, derive[SAMPLES+2], saturate(double),
                                 fc[SAMPLES+2];

void main(argc, argv)
int argc; char **argv;
{
int i, n, mask, fd1, fd2, fd3, fd4;
int creat(), write(), close(), fclose(), open(), read(), strcat(),
                                 strcmp();
```

```
unsigned PNoise(int);
double generator(int), domega, c, signal_power, noise_power;
double coswc, beta, gamman, gradient;

/************/
/*** initialization ***/
for(i=0; i<=48; i++)
   pow2to[i] = pow(2.,(double)i);
/*** noise generation ***/
wordlength = 15;
tap=4;
mask = (unsigned)(pow2to[wordlength]-1.)<<(32-wordlength);
fd1 = creat("noise", 0666);
noisereg[0] = 0xc3d64ccc; //seed for generator 0
/*** sinusoid generation ***/
sinamp = A;
signal_power = (sinamp*sinamp)/2.;
twopi = 8.*atan(1.);
fd2 = creat("sinusoid", 0666);
f_delta = fo*(pow(2., (1./12.)*F_DELTA) - 1.);
/*** adaptive filter init ***/
domega = twopi*BANDWIDTH/Fs;
beta = (1. - tan(domega/2.))/(1. + tan(domega/2.));
fc[1] = CENTER;
coswc = cos(twopi*fc[1]/Fs);
fd4 = creat("fc", 0666);
gamman = -coswc;
fd3 = creat("y", 0666);
/*** end initialization ***/
/************/

/*** generate floating-point noise samples ***/
for(i=2; i<SAMPLES+2; i++)
   noise[i] = PNoise(0)/pow2to[32] - 0.5;
noise_power = 1./12.;
c = sqrt(signal_power*pow(10.,-SNR/10.)/noise_power); //used later for
                                                       //noise amplitude
/*** create integer file for viewing and listening to noise alone ***/
for(i=2; i<SAMPLES+2; i++)
   shortnoise[i] = rint(noise[i]*pow2to[15]);
write(fd1, shortnoise + 2, 2*SAMPLES);
close(fd1);
system(string1);
system("rm noise");
```

```
/*** generate floating-point sinusoid samples ***/
for(i=2; i<SAMPLES+2; i++)
   sinusoid[i] = generator(i-2);
/*** create integer file for viewing and listening to sinusoid alone
***/
for(i=2; i<SAMPLES+2; i++)
   shortsinusoid[i] = rint(sinusoid[i]*(pow2to[15]-1.));
write(fd2, shortsinusoid + 2, 2*SAMPLES);
close(fd2);
system(string2);
system("rm sinusoid");

/*** feed signal plus noise into adaptive filter ***/
for(n=2; n<SAMPLES+2; n++) { //time origin @n=2
   x[n] = sinusoid[n] + c*noise[n];
   y[n] = (1. + beta)*(x[n] + 2.*gamman*x[n-1] + x[n-2])/2.
                      - gamman*(1. + beta)*y[n-1] - beta*y[n-2];
   /*** adapt gamma coefficient ***/
   derive[n] = (1. + beta)*(x[n-1] - y[n-1] - gamman*derive[n-1]) -
                 beta*derive[n-2];
   gradient = -2.*y[n]*derive[n];
   coswc -= Mu*gradient;
   gamman = saturate(-coswc);
   fc[n] = acos(-gamman)*Fs/twopi;
// printf("center freq[%d] is %f\n", n-2, fc[n]);
                          }
/*** create integer file for viewing and listening to filtered sinusoid;
                                                          -epsilon_n ***/
for(i=2; i<SAMPLES+2; i++)
   shortfiltered[i] = rint(y[i]*(pow2to[15]-1.));
write(fd3, shortfiltered + 2, 2*SAMPLES);
close(fd3);
system(string3);
system("rm y");
/*** create integer file for viewing filter center frequency ***/
for(i=2; i<SAMPLES+2; i++)
   shortfc[i] = rint(40.*fc[i]);
write(fd4, shortfc + 2, 2*SAMPLES);
close(fd4);
system(string4);
system("rm fc");

/*****************************************************************/
}//end main()
```

```
/****************************************************************/
/**************** subroutines ****************/
/************** Pseudo-Random maximal length sequence noise generator
**************/
/*** This routine returns 32 bit unsigned integers of which the most
     significant ***/
/*** "wordlength" bits constitute samples of the noise sequence. ***/
/*** Up to 32 independent generators are supported. ***/
unsigned PNoise(int gen_num)
{
unsigned getbit(unsigned, int), setbit(unsigned, unsigned);

noisereg[gen_num] = setbit(noisereg[gen_num]>>1,
   getbit(noisereg[gen_num],32-wordlength+tap) ^
              getbit(noisereg[gen_num],32-wordlength));

return(noisereg[gen_num]);
}
/************************* sinusoid generator
**************************************/
/*** This routine is can produce damped, swept, or frequency modulated
     sinusoids. ***/
double generator(int i)
{
double freq, phase, arg, amp, z;

if(!SWEEPING) {
   freq  =        fo;
   phase = (twopi*fo*i)/Fs;
   if(FM) {
          arg = (twopi*f_modulator*i)/Fs;
          freq +=                f_delta*cos(arg);
          phase += (f_delta/f_modulator)*sin(arg);
          }
              }
else {                                //one musical interval in samples
   freq = fo*pow(INTERVAL,(double)i/(SAMPLES-1))/Fs;
   phase = (twopi*freq*(SAMPLES-1))/log(INTERVAL);
     }

arg = phase + SINE_PHASE;
amp = (1. - pow(DAMPING, (double)i));
if(DAMPING == 0.0 && i == 0) amp = 1.0;
z = sinamp*amp*cos(arg);
return(z);
}
/*********************** get/setbit ******************************/
unsigned getbit(unsigned x, int position)
{
   return((x >> position) & 1);
}

unsigned setbit(unsigned x, unsigned state)
{
if(state) (x |= 0x80000000);
else      (x &= 0x7fffffff);
return(x);
}
/******************************* saturate ************************/
#define SATVAL 1.0

double saturate(double o)
{
if(o >  SATVAL) return( SATVAL);
if(o < -SATVAL) return(-SATVAL);
return(o);
}
```