Chapter 10: Rudiments of Adaptive Filters

10.1 Introduction

All of the filters discussed up to this point have been strictly linear, with frequency responses that are time invariant. In many cases this is fine, as attested by the myriad circumstances in which they are applied. But consider this problem: what if we had a broadband source, such as a speech or music signal, which was degraded by narrowband interference with a frequency within the bandwidth of the signal – for example, a whistle? Simple, you might say – just design a notch filter to remove it. This would work, as long as we could live with the fact that the filter would inevitably remove some of the signal. If the interference were more broadband in nature – such as engine noise – then the bandwidth of the filter required to remove it would suppress most of the signal, which of course would be unacceptable. Another situation that the linear filter could not handle would be narrowband interference that drifted across the spectrum of the audio source, such as a tone that rose or fell in pitch. Ostensibly, these problems appear intractable but they may be solved with the use of a special class of filter called the *adaptive filter*. As its name suggests, the filter adapts to the input signals (it has more than one, as we'll see in a moment), and learns to distinguish between the noise and the wanted input. Hence it can be used to recover a narrowband signal degraded by broadband noise, or vice versa, or non-stationary noise or a combination of these, as shown in Figure 10.1. The adaptive filter is, in short, quite a remarkable thing.

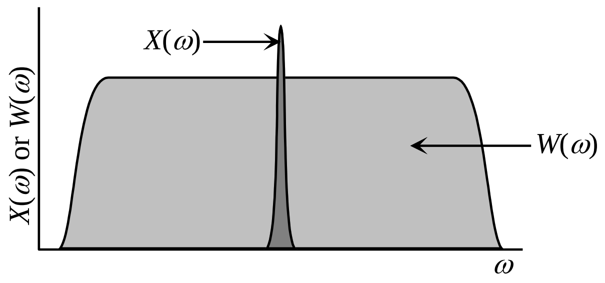


Figure 10.1 A narrowband signal embedded in broadband noise.

Now there are various classes of adaptive filter; for example, there are adaptive FIR filters and adaptive IIR filters. Additionally, there are various algorithms employed for their implementation. The full theory behind adaptive filters is complex, and we shall cover only the basic principles here. As is often the way with such matters however, adaptive filters are often *very* simple to code – much easier than say designing FIR or IIR types. The nice thing about them is that at no point do we have to explicitly calculate the filter coefficients, or the poles and zeros. This sounds almost too good to be true, but true it is, as the following pages will show.

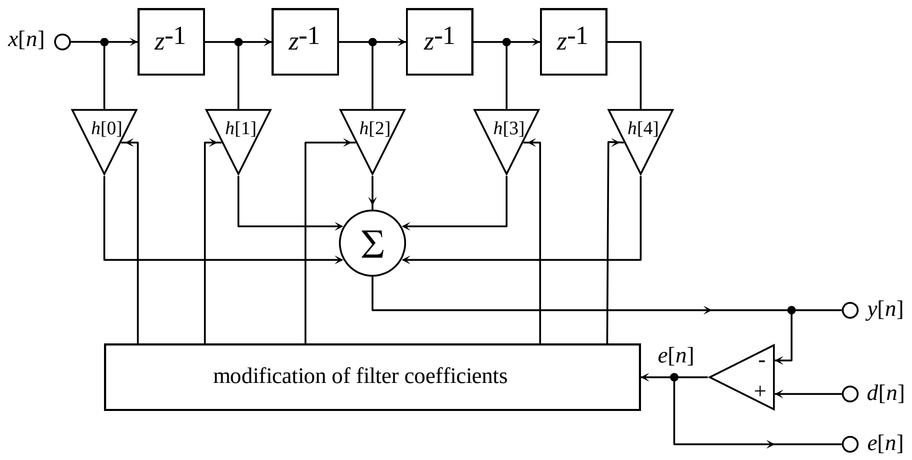


Figure 10.2. Schematic of the adaptive FIR filter. It comprises an ordinary FIR subsection and a coefficient modification unit (described below using an LMS algorithm).

10.2 Brief theory of adaptive FIR filters

Here we will cover the rudimentary theory and implementation of the adaptive FIR filter based on the *least mean square* (LMS) algorithm, for reasons both of its popularity and simplicity. The basic way an adaptive FIR filter works may be understood with respect to Figure 10.2. Note that in this figure, the process represented by the triangle is subtraction (differencing) and not an analogue multiplication. It consists of an ordinary FIR subsection, which generates an output $y[n]$ in response to an input $x[n]$. The output $y[n]$ is then compared to a reference signal called the *desired signal*, $d[n]$. The difference between these two signals generates an error signal, $e[n]$, a proportion of which is fed back to a routine that modifies the values of the FIR coefficients in an attempt to minimise the future value of the error signal. The smaller the error becomes, the more closely $y[n]$ approximates $d[n]$. As a result of this feedback, the filter may become unstable if it is not designed correctly, as we shall see below. Furthermore, because the coefficients change over time, it is clearly a nonlinear system. Now a question you might reasonably ask is: if we already have access to a desired signal, why do we need a filter in the first place? Well, this desired signal is a mathematical description, not a statement of what we actually need as an output. The theory is as follows: we start with the output signal $y[n]$, which as we know is given by the conventional convolution expression

$$y[n] = \sum_{k=0}^{M-1} h[k]x[n-k] \qquad (10.1)$$

where the number of filter coefficients is given by M. The error signal is

$$e[n] = d[n] - y[n] \qquad (10.2)$$

Now we are going to try to minimise the *sum of the squared errors, E*. If there are N points in the input/output signals, then the sum of the squared errors, E, is defined as

$$E = \sum_{n=0}^{N} e^2[n] = \sum_{n=0}^{N}\left[d[n] - \sum_{k=0}^{M-1} h[k]x[n-k]\right]^2 \qquad (10.3)$$

Expanding this, we get

$$E = \sum_{n=0}^{N}\left(\left[d[n] - \sum_{k=0}^{M-1} h[k]x[n-k]\right]\left[d[n] - \sum_{k=0}^{M-1} h[k]x[n-k]\right]\right) \qquad (10.4)$$

ie

$$E = \sum_{n=0}^{N} d^2[n] - \sum_{n=0}^{N} 2d[n]\sum_{k=0}^{M-1} h[k]x[n-k] + \sum_{n=0}^{N}\left[\sum_{k=0}^{M-1} h[k]x[n-k]\right]^2 \qquad (10.5)$$

If, for arguments sake, we say that

$$r_c[k] = \sum_{n=0}^{N} d[n]x[n-k] \qquad (10.6)$$

then Equation 10.5 becomes

$$E = \sum_{n=0}^{N} d^2[n] - 2\sum_{k=0}^{M-1} h[k]r_c[k] + \sum_{n=0}^{N}\left[\sum_{k=0}^{M-1} h[k]x[n-k]\right]^2 \qquad (10.7)$$

Taking the last term on the RHS, we have

$$\sum_{n=0}^{N}\left[\sum_{k=0}^{M-1} h[k]x[n-k]\right]^2 = \sum_{n=0}^{N}\sum_{k=0}^{M-1} h[k]x[n-k]\sum_{k=0}^{M-1} h[k]x[n-k] \qquad (10.8)$$

Now in general linear summations may be rearranged thus:

$$\sum_{n=1}^{N} x[n]\sum_{n=1}^{N} y[n] = \sum_{n=1}^{N}\sum_{l=1}^{N} x[n]y[l] \qquad (10.9)$$

So expressing the term given in Equation 10.8 in the above form, we have

$$\sum_{n=0}^{N}\left[\sum_{k=0}^{M-1} h[k]x[n-k]\right]^2 = \sum_{k=0}^{M-1}\sum_{l=0}^{M-1} h[k]h[l]\sum_{n=0}^{N} x[n-k]x[n-l] \qquad (10.10)$$

If we also say that

$$r_a[k-l] = \sum_{n=0}^{N} x[n-k]x[n-l] \qquad (10.11)$$

then after a little rearranging, Equation 10.7 becomes

$$E = \sum_{k=0}^{M-1}\sum_{l=0}^{M-1} h[k]h[l]r_a[k-l] - 2\sum_{k=0}^{M-1} h[k]r_c[k] + \sum_{n=0}^{N} d^2[n] \qquad (10.12)$$

where $r_a[k]$ is the auto-correlation function of $x[n]$ and $r_c[k]$ is the cross-correlation function between the desired output $d[n]$ and the actual input $x[n]$. In other words, the sum of the squared errors E is a quadratic function *of the form*

$$E = ah^2 - 2bh + c^2 \qquad (10.13)$$

where, in Equation 10.13, a represents the shifted auto-correlation function, b represents the cross-correlation function, c^2 represents the ideal signal and h is a given coefficient of the FIR filter. The purpose of the adaptive filter is to modify h so that the value of E is minimised for the given coefficients a, b and c. By definition, we can achieve this as follows: the quadratic equation is differentiated with respect to h, which in the case of Equation 10.13 becomes

$$2ah - 2b = 0, \quad ie \quad h = \frac{b}{a} \qquad (10.14)$$

The RHS of Equation 10.14 indicates that the new value of h is found by transposition of the linear equation that results from the differentiation of a quadratic function. This new value of h is substituted back into Equation 10.13, and we find that for any given value of a, b or c, E is minimised.

The procedure outlined above suggests that it is possible to obtain a filter with coefficients optimised such that $y[n]$ is identical with, or close to, $d[n]$. However, Equation 10.12 also reveals that because we have M values for h, to obtain the minimal error we need to perform a series of M partial differential operations; it also assumes that we know, or can calculate, the auto-correlation function and the cross-correlation function. These identities are in practical circumstances unavailable, or at least prohibitively costly to obtain from a computational perspective, so we need to approach the problem in a different way.

10.3 The least mean square (LMS) adaptive FIR algorithm

The LMS algorithm is a successive-approximation technique that obtains the optimal filter coefficients required for the minimisation of E. It is implemented as follows. Initially, the coefficients of the filter will be any arbitrary value; by convention, they are usually set to zero. Each new input signal value $x[n]$ generates an output signal value $y[n]$, from which we generate the error signal value, $e[n]$, by subtracting it from an ideal signal value, $d[n]$. The value of $h[k]$ is then modified by the expression

$$h_{new}[k] = h_{old}[k] + \Delta e[n]x[n-k] \qquad (10.15)$$

In other words, the new value of $h[k]$ is calculated by adding to its present value a fraction of the error signal $\Delta e[n]$, multiplied by the input signal value $x[n\text{-}k]$, where $x[n\text{-}k]$ is the input signal located at the k^{th} tap of the filter at time n. Equation 10.15 may be explained as follows (for the sake of clarity we will assume all values are positive). First, if the error $e[n]$ is zero, we obviously do not want to modify the value of $h[k]$, so the new value is simply equal to the old value. However, if it is not zero, it is modified it by adding to it a fraction of the error signal multiplied by $x[n\text{-}k]$. Since $y[n]$ is the convolution of $h[k]$ with $x[n\text{-}k]$, if we increase $h[k]$ we will therefore increase $y[n]$ and decrease $e[n]$, since $e[n] = d[n] - y[n]$. For reasons of stability, it is important that we only add a fraction of $e[n]$. If the rate of convergence is too rapid, the algorithm will over-shoot and become unstable. To ensure stability, Δ must be in the region

$$0 < \Delta < \frac{1}{10MP_x} \tag{10.16}$$

where M represents the number of coefficients in the filter and P_x is the average power of the input signal, approximated by

$$P_x = \frac{1}{N+1}\sum_{n=0}^{N} x^2[n] \tag{10.17}$$

Where again, N represents the number of points in the input signal. The LMS algorithm is not the only method employed in the implementation of adaptive filters; indeed, neither is it the fastest to converge to the optimum value for a given filter coefficient. The recursive least squares (RLS) technique, for example, is faster and is also commonly encountered (Ifeachor and Jervis, 1993). However, the LMS method is probably the most widely used for two reasons: first, it very easy to code, as we shall see in a moment. Second, it is very stable, as long as we make sure we don't feed back too much of the error signal, i.e. we adhere to the stipulation of Equation 10.16.

10.4 Use of the adaptive filter in system modelling

To commence our discussion on how we can usefully employ the adaptive filter in practical situations, we will first look at using it to identify the impulse response of an unknown (linear) system. The concept is illustrated in Figure 10.3.

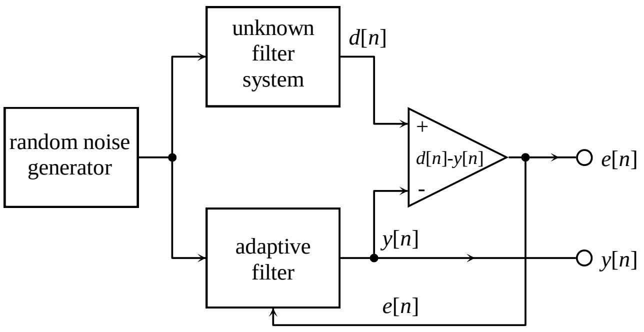


Figure 10.3. Using an adaptive filter to obtain the impulse response of an unknown linear system.

The first component that we need is an input test signal, which as Figure 10.3 indicates is a broadband source generated using a random noise sequence; a broadband source is very appropriate in this circumstance, since we do not know the bandwidth of the frequency response of the system we are modelling. This signal is input both to the unknown system and to our adaptive filter, the coefficients of which are initially all set to zero, as we established above. The unknown system will filter the input by means of its (hidden) impulse response, generating an output we will define as the desired signal, $d[n]$. Together with the output from the adaptive filter $y[n]$, it is used to calculate the error signal $e[n]$, part of which is fed back in the LMS process to update the filter coefficients. Over time, the output signal $y[n]$ will converge towards $d[n]$. In other words, the impulse response of unknown system will have been identified since it will be the same, or very similar to, the impulse response of the adaptive filter.

A screenshot of a program can be seen in Figure 10.4 which generates an impulse response in the form of a triangular function, stored in an array; this is at first unknown to the adaptive filter section. An adaptive filter routine in the code will, in the fullness of time, replicate the impulse response of the unknown system and store it in an equivalent array. Referring to Figure 10.4, the upper right trace shows the unknown impulse response, and the lower right trace shows the one estimated by the adaptive filter. The trace shown in the upper left shows the convergence of a given tap, which we will discuss below.

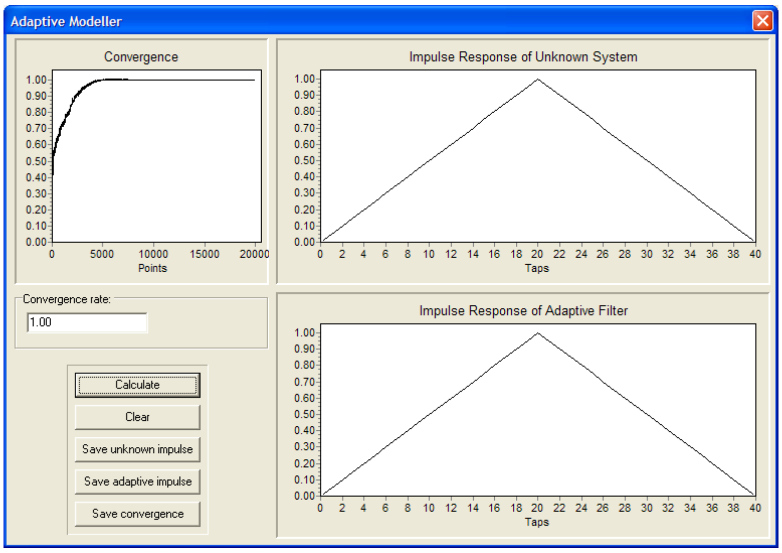


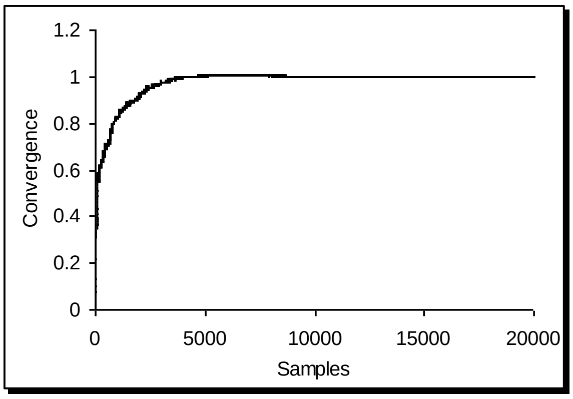
Figure 10.4. Screenshot of *adaptive_model.exe*.

```
{Generate unknown response}
  for n:=0 to j div 2 do
  begin
    h0[n]:=n/20;
    h0[j-n]:=n/20;
  end;
  for n:=0 to j do h1[n]:=0;
{Calculate delta}
  delta:=strtofloat(edit1.Text);
  power:=0;
  for n:=0 to m do
  begin
    x[n]:=random;
    power:=power+sqr(x[n]);
  end;
  power:=power/(m+1);
  delta:=delta/(10*j*power);
{Adaptive filter kernel}
  for n:=0 to m do
  begin
    y:=0;
    d:=0;
    for k:=0 to j do
    begin
      d:=d+h0[k]*x[n-k];
      y:=y+h1[k]*x[n-k];
    end;
    e:=d-y;
    for k:=0 to j do h1[k]:=h1[k]+delta*e*x[n-k];
    converge[n]:=h1[20];
  end;
```

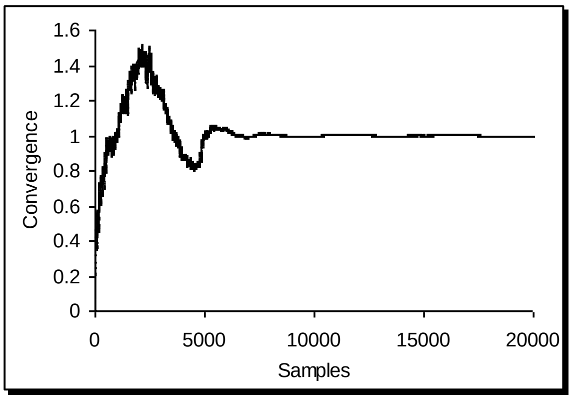
Listing 10.1.

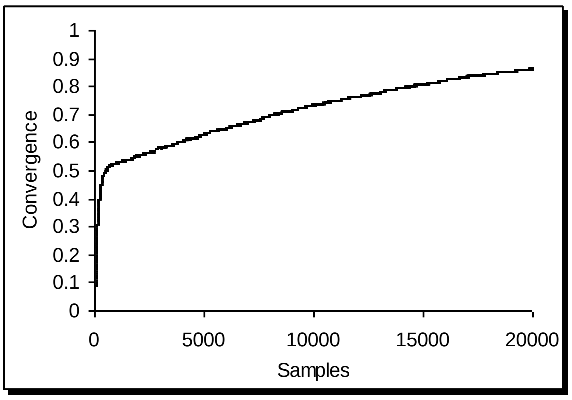
To understand how the program works, take a look at Listing 10.1. Initially, the code simply generates a triangular function from 0 to 1 and back down again in steps of 0.1. The values are loaded into `h0[n]`, i.e. this represents the impulse response of the unknown system that the adaptive filter will estimate. After this, it initialises the adaptive filter impulse response array, `h1[n]`, to zero. The next stage involves generating a set of random numbers, stored in the array `x[n]`, which will be used as the input signal. The value of the feedback fraction Δ is now computed, using Equations 10.16 and

10.7, and a gain factor that the user may supply from the data entry area labelled *Convergence rate*. Finally, we enter the adaptive filter kernel; the input signal is convolved both with `h0[n]` and `h1[n]`, and as each new pair of signal points is produced, an error signal is calculated by subtracting one from the other. This is then used, along with the value of Δ, to modify the values of the coefficients held in `h1[n]` according to the LMS algorithm given by Equation 10.15. The program also stores, in an array called `converge[n]`, the value of the 20th tap of the estimated impulse response at each stage of the iteration. Ideally, after convergence this should equal 1.0. (i.e. `h0[20] = h1[20]`).



(a)





(b) (c)

Figure 10.5. Three convergence curves taken from the program *adaptive_model.exe*, using convergence parameters of (a) 1.0, which is optimal, (b) 10.0, causing overshoot and (c) 0.1, causing undershoot. If the parameter is increased too much, the filter becomes unstable.

Using the preset value of convergence (1.0), you can see that after 20000 cycles the convergence is near enough perfect. As mentioned above, the little plot in the top left corner shows how the value of `h[20]` approaches unity. With a convergence of 1.0, it reaches this after approximately 7000 cycles. If you reduce the convergence parameter, you can see that the curve takes longer to flatten out. Conversely, if you increase it beyond 1.0, the curve overshoots before reaching equilibrium (i.e. this also takes longer). If you increase it too far, the system becomes unstable and fails. Figure 10.5 illustrates three different convergence curves taken from this program, using convergence parameters of 1.0, 0.1 and 10.0.

10.5 Delayed (single) input adaptive LMS filters for noise removal

The above program illustrated the mechanics of how the adaptive filter operates; in practice however, it would be far simpler to obtain the impulse and frequency response of an unknown system simply by probing it with an impulse function, recording its response and taking its Fourier transform. Moreover, it still left unanswered the issue of the desired signal, and how we might generate this if we intended to employ the adaptive filter for real signal filtering. Therefore, in this and the following section we will discuss some modifications to the basic adaptive design, and show how it can be used to extract a swept narrowband source embedded in broadband noise, or vice versa (Gaydecki, 1997). We start with the delayed input adaptive filter, also known as the single input adaptive filter, a schematic of which is shown in Figure 10.6.

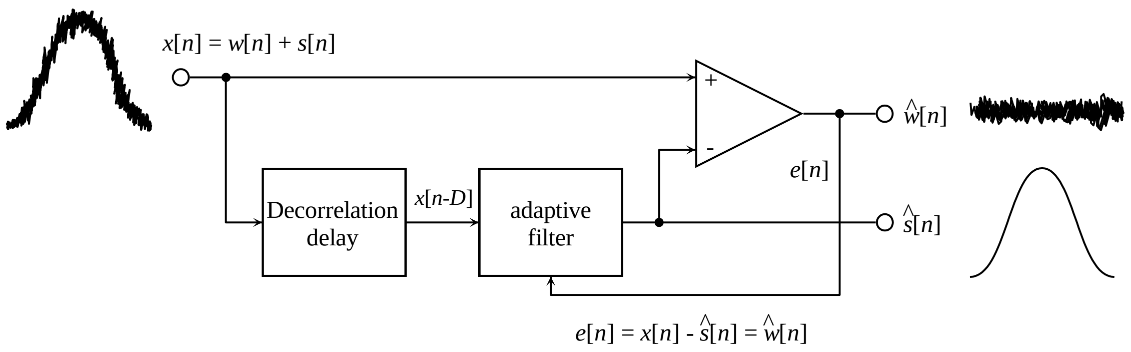


Figure 10.6. A delayed input adaptive filter, used here to separate wideband and narrowband signals.

In this scheme, we assume that the bandwidth of the narrowband signal is unknown, and that it may also vary with time. Formally, we can say that the signal corrupted with noise, x[*n*] is given by

$$x[n] = w[n] + s[n] \tag{10.18}$$

where *w*[*n*] is the broadband noise source and *s*[*n*] is the narrowband signal. In order to remove *w*[*n*], we must introduce into the adaptive filter a version of the signal *x*[*n*] that has been delayed by *D* intervals, where *D* is defined as the *decorrelation delay*. If the noise has a very wide bandwidth, such as a series of random values, then each point in the signal will have no correlation with its neighbour. *D* must be chosen such that the signal value *x*[*n*] has no correlation with *x*[*n-D*], with respect to *w*[*n*]. Conversely, with a narrowband signal, neighbouring signal values will be highly correlated. Hence the probability is that *x*[*n*] will be highly correlated with *x*[*n-D*], with respect to *s*[*n*].

When we used the adaptive filter for system modelling, the desired signal represented an output from some unknown system. *In this case, strange as it may seem, the desired signal is considered to be* *x*[*n*]. As Figure 10.6 shows, the input signal to the adaptive filter is *x*[*n-D*] (as opposed to *w*[*n*]). The feedback error signal is given by

$$e[n] = x[n] - \hat{s}[n] \tag{10.19}$$

In other words, *e*[*n*] is an approximated version of *w*[*n*], the original broadband noise; we will therefore call it $\hat{w}[n]$. It must follow that $\hat{s}[n]$ is an approximated version of the original narrowband signal, *s*[*n*]. Before we think about how this works, there is a very important observation we should now make: this filter has two outputs, which are the error signal, given by $\hat{w}[n]$, and the output from the adaptive block, given by $\hat{s}[n]$. Whether we use it to extract the narrowband or the broadband signal depends on our application and what we define as noise.

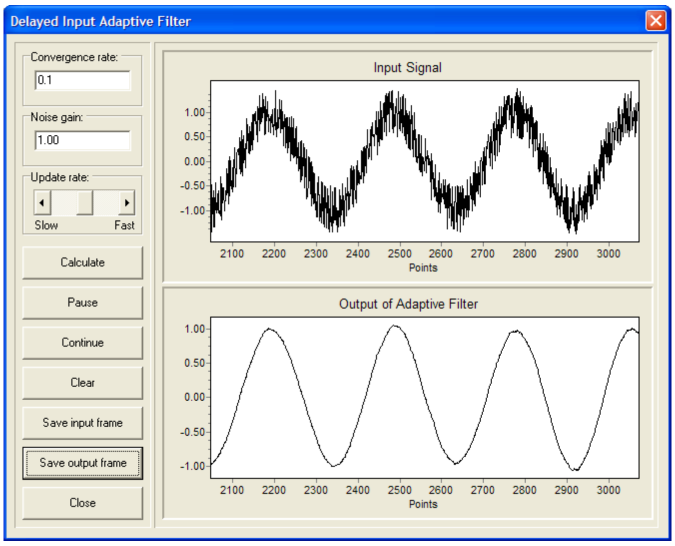
So how does it work? Suppose the original noise *w*[*n*] comprised a series of random numbers. In this case, we could set *D* equal to 1, since no noise value would correlate with its immediate neighbour. An adaptive filter operates by producing an output (here $\hat{s}[n]$) that is maximally correlated with some desired signal *d*[*n*], in this case *x*[*n*], and the input, in this case a filtered version of *x*[*n-D*]. Because of this delay, *x*[*n*] and *x*[*n-D*] are uncorrelated with respect to *w*[*n*] but highly correlated with respect to *s*[*n*]. The error signal will therefore be minimal when it resembles as closely as possible the signal *s*[*n*]. Since it uses an LMS algorithm, the approximation improves with accuracy over time. When the signal $\hat{s}[n]$ is subtracted from *x*[*n*], it results in the error signal *e*[*n*] being an approximation of *w*[*n*]. Now, because $\hat{s}[n]$ is obtained via

$$\hat{s}[n] = \sum_{k=0}^{N-1} h[k]x[n-k-D] \tag{10.20}$$

the steepest-descent algorithm becomes

$$h_{new}[k] = h_{old}[k] + \Delta e[n] x[n-k-D] \qquad (10.21)$$

A screenshot is shown in Figure 10.7 of a program which runs a delayed input adaptive filter. The program will first generate a narrowband, swept frequency signal degraded with broadband noise. It will then attempt to remove this noise using the LMS algorithm that we have just described.



Convergence rate: 0.1

Noise gain: 1.00

Figure 10.7. Screenshot of *adaptive_delay.exe*.

The program also allows the user to alter the convergence rate and the noise gain factor. Using the default values, the adaptive filter makes a pretty good job of extracting the signal from the noise, even as its sweeps upwards in frequency. The reconstruction is not perfect, as attested by the wobble in the signal, but the situation may be improved by increasing the number of coefficients in the FIR section (this one has 300). Listing 10.2 shows the kernel of the delayed input adaptive filter, and also serves to illustrate how it differs from that used in the modelling system we developed above.

```
for n:=0 to m do
begin
  y[n]:=0;
  for k:=0 to taps-1 do
  begin
    y[n]:=y[n]+h[k]*x[n-k-d];
  end;
  e:=x[n]-y[n];
  for k:=0 to taps-1 do h[k]:=h[k]+delta*e*x[n-k-d];
end;
```

Listing 10.2

In the program, the degraded narrowband signal is held in the array `x[n]`. It was computed by mixing together the outputs from a swept sine function and a random number generator (not shown in Listing 10.2) As the code shows, the implementation of the adaptive filter is simple in the extreme, following Equations 10.20 and 10.21 exactly. The signal arrays comprise 20691 values, so the system has plenty of room to reach convergence. Perhaps the most surprising thing about the filter is how effective it is. This is illustrated in Figure 10.8, which shows signal input and output traces taken directly from the program.

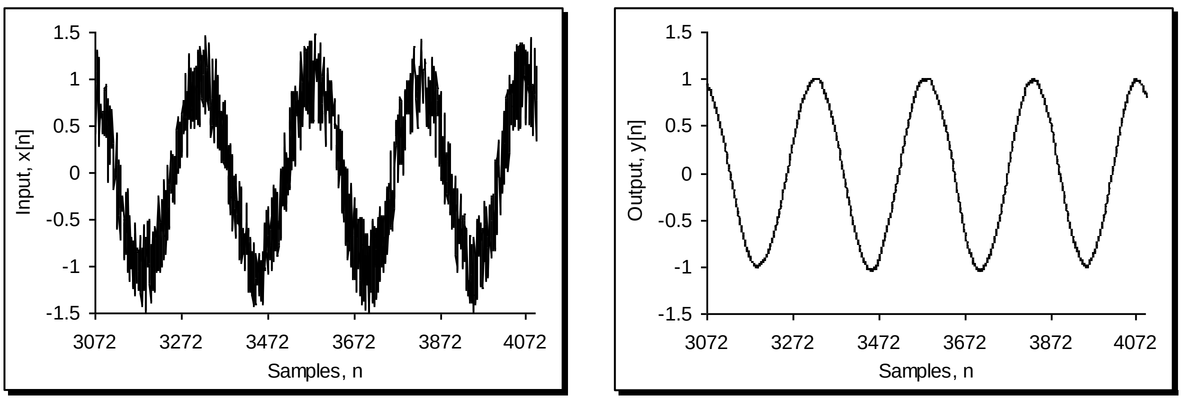


Figure 10.8. (a) Input and (b) outputs from the program *adaptive_delay.exe*. The input narrowband signal is normalised to unity and degraded using a random noise with a gain of 1.0.

10.6. The true (dual input) adaptive LMS filter

The delayed input adaptive filter as described here is impressive, but also limited by the magnitude of the noise it can accommodate. You can readily demonstrate this for yourself by increasing the noise gain to say 10, and running the filter. It makes a brave attempt at quenching the broadband interference, but no matter how the convergence factor is adjusted, it is still there at the end of the signal. Dramatic improvements can be realised if we have access to both the degraded signal *and* the noise, as independent inputs. If we do, we can design an adaptive filter based around the architecture shown in Figure 10.9.

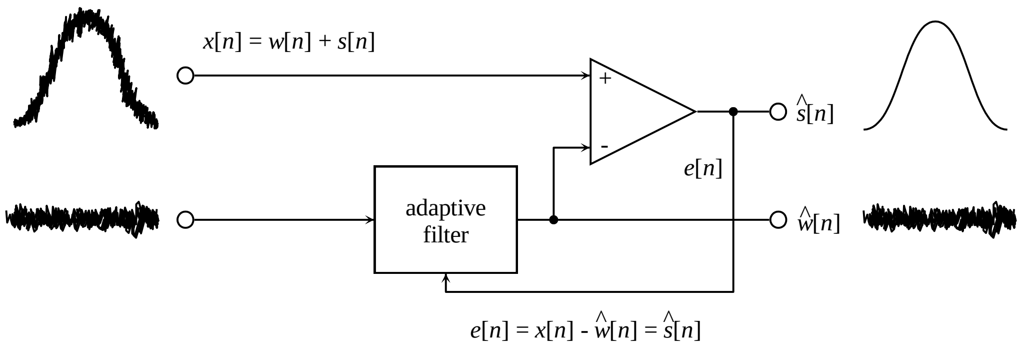


Figure 10.9. The true (dual input) adaptive filter.

This kind of configuration is known as a true (or dual input) adaptive filter. Because we have access to the noise source, we don't have to introduce a decorrelation delay *D* to synthesise it. Once more, the degraded signal $x[n]$ is given by Equation 10.18, and this we also define as the desired signal, $d[n]$. The broadband noise source, $w[n]$, is fed into the adaptive filter unit, and after convolution with the filter's impulse response, it is subtracted from $x[n]$. Since there is no delay in the system, the error signal $e[n]$ will be minimal when the filtered noise signal, $\hat{w}[n]$, matches as closely as possible the original noise given by $w[n]$. It therefore follows that the error signal will be given by

$$e[n] = x[n] - \hat{w}[n] = \hat{s}[n] \qquad (3.22)$$

In other words, $e[n]$ is equal to $\hat{s}[n]$, an approximated version of the narrowband input signal. The code fragment given in Listing 10.3 is for a true adaptive filter. The information flow is organised around the filter architecture depicted in Figure 10.9. Here, `w[n]` is an array that holds a sequence of random numbers, representing the broadband noise source. The array defined by x[n] holds the degraded narrowband signal, computed here (as in the delayed input adaptive filter) by adding a swept sine, point for point, with the random values held in `w[n]`. In this case however, `w[n]` is convolved with `h[k]`, and since we wish to retain $e[n]$ rather than $y[n]$, a single variable `y` is used to hold the result of each convolution operation. This is subtracted from the degraded signal `x[n]`, which, from a mathematical perspective, is the ideal signal. The program retains the error signal, or feedback signal, in the array `e[n]`, using it to modify the filter coefficients.

```
for n:=0 to m do
begin
  y:=0;
  for k:=0 to taps-1 do
  begin
    y:=y+h[k]*w[n-k];
  end;
  e[n]:=x[n]-y;
  for k:=0 to taps-1 do h[k]:=h[k]+delta*e[n]*w[n-k];
end;
```

Listing 10.3

With a normalised convergence rate of 1.0, the system quickly identifies the signal, since the error fraction, Δ, is approximately optimal. If this is reduced, the system takes longer to remove the noise, as you would expect. Conversely, if the convergence rate is set greater than 1.0, say to 5, then initially, the noise disappears more quickly, but it returns or the output becomes erratic as the system convergence overshoots. Figure 10.10 shows the input and output signal traces (after convergence) with a peak-to-peak signal-to-noise ratio of 1:100.

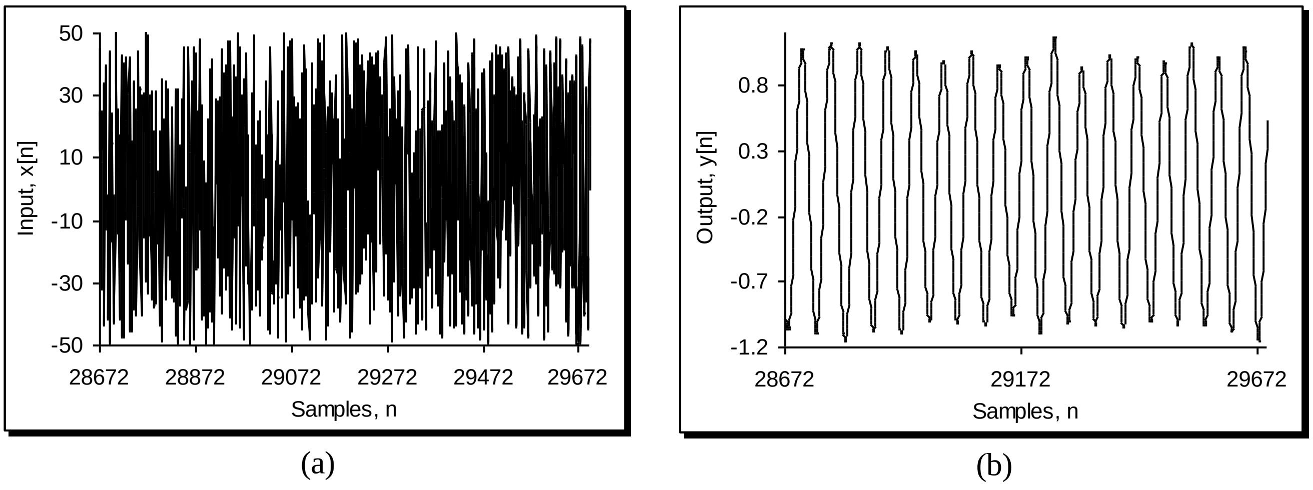


Figure 10.10. The power of the true adaptive filter. (a) Input and (b) output signals from the program *adaptive_true.exe*. The input narrowband signal is normalised to unity and degraded using a random noise with a gain of 100.

As with the delayed input adaptive filter, the true adaptive filter can select for the narrowband or the broadband signal, depending on whether we take the output as the error signal or from the filter unit.

10.7 Observations on real-time applications of adaptive filters

Adaptive filters are at their most useful when applied in real-time situations. For example, think about a helicopter pilot communicating with air traffic control, using a microphone mounted within his or her helmet. Helicopter cockpits can be noisy places, so inevitably the microphone will pick up not just the pilot's voice, but also noise from the engine and the rotor blades. A true, dual input adaptive filter would be ideal in this circumstance. One input would be connected to the pilot's microphone; this would represent the desired signal. The second input would be connected to a microphone outside of the helmet, listening to the engine/rotor noise. The algorithm would be identical to the one employed here, but run on a real-time DSP device. If you have been following this line of reasoning closely, then you may have formulated two questions that so far have not been addressed, and which relate to the successful operation of a real-time adaptive filter. The first concerns the calculation of the error fraction, Δ. Equations 10.16 and 10.17 suggest that it is related to the duration of the signal, since here, N represents the number of points. However, if you look at these closely you also discover that the signal squared is normalised by the length - as we said previously, P_x is the average signal power, and is independent of time. Therefore, with real-time adaptive filters, a conservative estimate of Δ is made, based on the nature of the signal. If the convergence rate is too long, this can be revised upwards towards some optimal value. The second question concerns the timing of the two signals coming into the true adaptive filter. In our software, each noise value entering the adaptive unit was perfectly synchronous with the same value added to the narrowband source; moreover, it had the same magnitude. In practical circumstance (think about the helicopter example), the pure engine/rotor noise will not be identical in each microphone, respecting both magnitude and timing. Is this a problem? The short answer is (fortunately), no. The greater the disparity, the longer the filter will take to converge, but matters have to be very extreme before the filters fails. Take a look at listing 10.4, which shows the code that generates the swept sine wave and the broadband noise, and then combines them.

```
for n:=-30 to m do
begin
  sweep:=sweep*1.00004;
  w[n]:=noise*(random-0.5);
  x[n]:=w[n]*+sin(sweep*5*pi*n/r);
end;
```

Listing 10.3=4

The variable called `sweep` gradually increases in value within the loop, and is responsible for increasing the centre frequency of the signal. Each value of `w[n]` is added to `x[n]`, for the same value of `n`. Furthermore, the amplitude relationships are maintained. However, we could add to `x[n]` an advanced version of `w[n]`, with a greater magnitude, as shown by Listing 10.5

```
for n:=-30 to m do
begin
  sweep:=sweep*1.00004;
  w[n]:=noise*(random-0.5);
  x[n]:=w[n-10]*5+sin(sweep*5*pi*n/r);
end;
```

Listing 10.5