

---

## Nonfilters

Filters have a lot going for them. In the previous chapter we have seen that they are simple to design, describe and implement. So why bother devoting an entire chapter to the subject of systems that are *not* filters?

There are two good reasons to study nonfilters—systems that are either nonlinear, or not time-invariant, or both. First, no system in the real world is ever perfectly linear; all ‘linear’ analog systems are nonlinear if you look carefully enough, and digital signals become nonlinear due to round-off error and overflow. Even relatively small analog nonlinearities can lead to observable results and unexpected major nonlinearities can lead to disastrous results. A signal processing professional needs to know how to identify these nonlinearities and how to correct them. Second, linear systems are limited in their capabilities, and one often requires processing functions that simply cannot be produced using purely linear systems. Also, linear systems are predictable; a small change in the input signal will always lead to a bounded change in the output signal. Nonlinear systems, however, may behave chaotically, that is, very small changes in the input leading to completely different behavior!

We start the chapter with a discussion of the effects of small nonlinearities on otherwise linear systems. Next we discuss several ‘nonlinear filters’, a term that is definitely an oxymoron. We *defined* a ‘filter’ as a linear and time-invariant system, so how can there be a ‘nonlinear filter’? Well, once again, we are not the kind of people to be held back by our own definitions. Just as we say delta ‘function’, or talk about infinite energy ‘signals’, we allow ourselves to call systems that are obviously not filters, just that.

The mixer and the phase locked loop are two systems that are not filters due to not being time-invariant. These systems turn out to very important in signal processing for telecommunications. Our final topic, time warping, is an even more blatant example of the breakdown of time invariance.

## 8.1 Nonlinearities

Let's see what makes nonlinear systems interesting. We start by considering the simplest possible nonlinearity, a small additive quadratic term, which for analog signals reads

$$y(t) = x(t) + \epsilon x^2(t) \quad (8.1)$$

(assume  $\epsilon \ll 1$ ). The spectral consequences can be made clear by considering an arbitrary sinusoidal input

$$x(t) = A \cos(\omega t) \quad (8.2)$$

for which the system will output

$$y(t) = A \cos(\omega t) + \epsilon A^2 \cos^2(\omega t) \quad (8.3)$$

which can be simplified by substituting from equation (A.25).

$$y(t) = A \sin(\omega t) + \frac{\epsilon A^2}{2} + \frac{1}{2} \cos(2\omega t) \quad (8.4)$$

We see here three terms; the first being simply the original unscathed signal, the other two going to zero as  $\epsilon \rightarrow 0$ . The second term is a small DC component that we should have expected, since  $\cos^2$  is always positive and thus has a nonzero mean. The final term is an attenuated replica of the original signal, but at twice the original frequency! This component is known as the *second harmonic* of the signal, and the phenomenon of creating new frequencies which are integer multiples of the original is called *harmonic generation*. Harmonic generation will always take place when a nonlinearity is present, the energy of the harmonic depending directly on the strength of the nonlinearity. In some cases the harmonic is unwanted (as when a nonlinearity causes a transmitter to interfere with a receiver at a different frequency), while in other cases nonlinearities are introduced precisely to obtain the harmonic.

We see here a fundamental difference between linear and nonlinear systems. Time-invariant linear systems are limited to filtering the spectrum of the incoming signal, while nonlinear systems can generate new frequencies.

What would have happened had the nonlinearity been cubic rather than quadratic?

$$y(t) = x(t) + \epsilon x^3(t)$$

You can easily find that there is *third harmonic generation* (i.e., a signal with thrice the original frequency appears from nowhere). A fourth order nonlinearity

$$y(t) = x(t) + \epsilon x^4(t)$$

will generate both second and fourth harmonics (see equation (A.33)); and  $n^{\text{th}}$  order nonlinearities generate harmonics up to order  $n$ . Of course a general nonlinearity that can be expanded in a Taylor expansion

$$y(t) = x(t) + \epsilon_2 x^2(t) + \epsilon_3 x^3(t) + \epsilon_4 x^4(t) + \dots \quad (8.5)$$

will produce many different harmonics.

We can learn more about nonlinear systems by observing the effect of simple nonlinearities on signals composed of two different sinusoids.

$$x(t) = A_1 \cos(\omega_1 t) + A_2 \cos(\omega_2 t) \quad (8.6)$$

Inputting this signal into a system with a small quadratic nonlinearity

$$\begin{aligned} y(t) &= A_1 \cos(\omega_1 t) + A_2 \cos(\omega_2 t) \\ &\quad + A_1^2 \cos^2(\omega_1 t) + A_2^2 \cos^2(\omega_2 t) \\ &\quad + 2A_1 A_2 \cos(\omega_1 t) \cos(\omega_2 t) \\ &= A_1 \cos(\omega_1 t) + A_2 \cos(\omega_2 t) \\ &\quad + A_1^2 \cos^2(\omega_1 t) + A_2^2 \cos^2(\omega_2 t) \\ &\quad + A_1 A_2 \cos\left((\omega_1 + \omega_2)t\right) \\ &\quad + A_1 A_2 \cos\left(|\omega_1 - \omega_2|t\right) \end{aligned}$$

we see harmonic generation for both frequencies, but there is also a new nonlinear term, called the *intermodulation product*, that is responsible for the generation of sum and difference frequencies. Once again we see that nonlinearities cause energy to migrate to frequencies where there was none before.

More general nonlinearities generate higher harmonics plus more complex intermodulation frequencies such as

$$\begin{array}{ll} \omega_1 + \omega_2, & |\omega_1 - \omega_2|, \\ \omega_1 + 2\omega_2, & 2\omega_1 + \omega_2, \\ |2\omega_1 - \omega_2|, & |2\omega_2 - \omega_1|, \\ \omega_1 + 3\omega_2, & 3\omega_1 + \omega_2, \\ 2\omega_1 + 3\omega_2, & 2\omega_1 + 3\omega_2, \\ & \text{etc.} \end{array}$$

This phenomenon of intermodulation can be both useful and troublesome. We will see a use in Section 8.5; the negative side is that it can cause hard-to-locate **Radio Frequency Interference (RFI)**. For example, a television set may have never experienced any interference even though it is situated not far from a high-power radio transmitter. Then one day a taxi cab passes by a rusty fence that can act as a nonlinear device, and the combination of the cab's transmission and the radio station can cause a signal that interferes with TV reception.

## EXERCISES

- 8.1.1 Show exactly which harmonics and intermodulation products are generated by a power law nonlinearity  $y(t) = x(t) + \epsilon x^n(t)$ .
- 8.1.2 Assume that the nonlinearity is exponential  $y(t) = x(t) + \epsilon e^{x(t)}$  rather than a power law. What harmonics and intermodulation frequencies appear now?

## 8.2 Clippers and Slicers

One of the first systems we learned about was the clipping amplifier, or peak clipper, defined in equation (6.1). The peak clipper is obviously strongly nonlinear and hence generates harmonics, intermodulation products, etc. What is less obvious is that sometimes we use a clipper to *prevent* nonlinear effects. For example, if a signal to be transmitted has a strong peak value that will cause problems when input to a nonlinear medium, we may elect to artificially clip it to the maximal value that can be safely sent.

The opposite of this type of clipper is the *center clipper*, which zeros out signal values *smaller* than some threshold.

$$y = C_\theta(x) = \begin{cases} 0 & |x| < \theta \\ x & \text{else} \end{cases} \quad (8.7)$$

The center clipper is also obviously nonlinear, and although at first sight its purpose is hard to imagine, it has several uses in speech processing. The first relates to the removal of unwanted zero crossings. As we will see in Section 13.1 there are algorithms that exploit the number of times a signal crosses the time axis, and/or the time between two such successive zero crossings. These algorithms work very well on clean signals, but fail in the

presence of noise that introduces extraneous zero crossings. The problem is not severe for strong signals but when the signal amplitude is low the noise may dominate and we find many extraneous zero crossings. Center clipping can remove unwanted zero crossings, restoring the proper number of zero crossings, at the price of introducing uncertainty in the precise time between them. In fact center clipping has become so popular in this scenario that it is used even when more complex algorithms, not based on zero crossings, are employed.

A related application is motivated by something we will learn in Chapter 11, namely that our hearing system responds approximately logarithmically to signal amplitude. Thus small amounts of noise that are not noticeable when the desired signal is strong become annoying when the signal is weak or nonexistent. A case of particular interest is echo over long distance telephone connections; linear echo cancellers do a good job at removing most of the echo, but when the other party is silent we can still hear our own voice returning after the round-trip delay, even if it has been substantially suppressed. This small but noticeable residual echo can be removed by a center clipper, which in this application goes under the uninformative name of **NonLinear Processor (NLP)**. Unfortunately this leaves the line sounding too quiet, leading one to believe that the connection has been lost; this defect can be overcome by injecting artificial 'comfort noise' of the appropriate level.

The peak clipper and center clipper are just two special cases of a more general nonfilter called a *slicer*. Consider a signal known to be restricted to integer values that is received corrupted by noise. The obvious recourse is to clip each real signal value to the closest integer. This in effect slices up the space of possible received values into slices of unity width, the slice between  $n - \frac{1}{2}$  and  $n + \frac{1}{2}$  being mapped to  $n$ . The nonlinear system that performs this function is called a *slicer*.

Up to now we have discussed slicers that operate on a signal's amplitude, but more general slicers are in common use as well. For example, we may know that a signal transmitted to us is a sinusoid of given frequency but with phase of either  $+\pi$  or  $-\pi$ . When measuring this phase we will in general find some other value, and must decide on the proper phase by slicing to the closest allowed value. Even more complex slicers must make decisions based on both phase and amplitude values. Such slicers are basic building blocks of modern high-speed modems and will be discussed in Section 18.18. You may wish to peek at Figure 18.26 to see the complexity of some slicers.

## EXERCISES

- 8.2.1 Apply a center clipper with a small threshold to clean sampled speech. Do you hear any effect? What about noisy speech? What happens as you increase the threshold? At what point does the speech start to sound distorted?
- 8.2.2 Determine experimentally the type of harmonic generation performed by the clipper and the center clipper.
- 8.2.3 There is a variant of the center clipper with continuous output as a function of input, but discontinuous derivative. Plot the response of this system. What are its advantages and disadvantages?
- 8.2.4 When a slicer operates on sampled values a question arises regarding values exactly equidistant between two integer values. Discuss possible tactics.
- 8.2.5 A 'resetting filter' is a nonlinear system governed by the following equations.

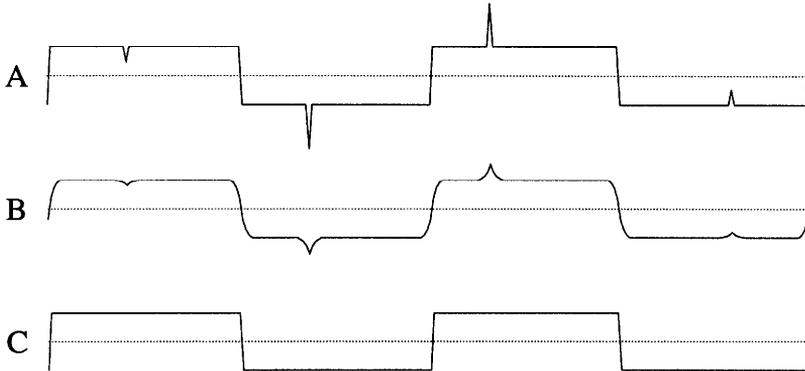
$$\begin{aligned}
 y_n &= x_n + w_n \\
 r_n &= \begin{cases} -\Theta & y_n < -\Theta \\ 0 & |y_n| < \Theta \\ \Theta & y_n > \Theta \end{cases} \\
 w_n &= y_{n-1} - r_{n-1}
 \end{aligned}$$

Explain what the resetting filter does and how it can be used.

### 8.3 Median Filters

Filters are optimal at recovery of signals masked by additive Gaussian noise, but less adept at removing other types of unwanted interference. One case of interest is that of unreliable data. Here we believe that the signal samples are generally received without additive noise, but now and then may be completely corrupted. For example, consider what happens when we send a digital signal as bits through a unreliable communications channel. Every now and then a bit is received incorrectly, corrupting some signal value. If this bit happens to correspond to the least significant bit of the signal value, this corruption may not even be detected. If, however, it corresponds to the most significant bit there is a isolated major disruption of the signal. Such isolated incorrect signal values are sometimes called *outliers*.

An instructive example of the destructive effect of outliers is depicted in Figure 8.1. The original signal was a square wave, but four isolated signal values were strongly corrupted. Using a low-pass filter indeed brings the



**Figure 8.1:** Comparison of a linear filter with a median filter. In (A) we have the original corrupted square wave signal; in (B) the signal has been filtered using a symmetric noncausal FIR low-pass filter; and in (C) we see the effect of a median filter.

corrupted signal values closer to their correct levels, but also changes signal values that were not corrupted at all. In particular, low-pass filtering smooths sharp transitions (making the square wave edges less pronounced) and disturbs the signal in the vicinity of the outlier. The closer we wish the outlier to approach its proper level, the stronger this undesirable smoothing effect will be.

An alternative to the low-pass filter is the median filter, whose effect is seen in Figure 8.1.C. At every time instant the median filter observes signal values in a region around that time, similar to a noncausal FIR filter. However, instead of multiplying the signal values in this region by coefficients, the median filter sorts the signal values (in ascending order) and selects ‘median’, i.e., the value precisely in the center of the sorted buffer. For example, if a median filter of length five overlaps the values 1, 5, 4, 3, 2, it sorts them into 1, 2, 3, 4, 5 and returns 3. In a more typical case the median filter overlaps something like 2, 2, 2, 15, 2, sorts this to 2, 2, 2, 2, 15 and returns 2; and at the next time instant the filter sees 2, 2, 15, 2, 2 and returns 2 again. Any isolated outlier in a constant or slowly varying signal is completely removed.

Why doesn’t a median filter smooth a sharp transition between two constant plateaus? As long as more than half the signal values belong to one side or the other, the median filter returns the correct value. Using an odd-order noncausal filter ensures that the changeover happens at precisely the right time.

What happens when the original signal is not constant? Were the linearly increasing signal  $\dots 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 \dots$  to become corrupted to

... 1, 2, 3, 4, 99, 6, 7, 8, 9, 10, ..., a median filter of length 5 would be able to correct this to ... 1, 2, 3, 4, 6, 7, 8, 8, 9, 10, ... by effectively skipping the corrupted value and replicating a later value in order to resynchronize. Similarly, were the corrupted signal to be ... 1, 2, 3, 4, -99, 6, 7, 8, 9, 10, ..., the median filter would return the sequence ... 1, 2, 2, 3, 4, 6, 7, 8, 9, 10, ... replicating a previous value and skipping to catch up. Although the corrupted value never explicitly appears, it leaves its mark as a phase shift that lasts for a short time interval.

What if there is additive noise in addition to outliers? The simplest thing to do is to use a median filter *and* a linear low-pass filter. If we apply these as two separate operations we should probably first median filter in order to correct the gross errors and only then low-pass to take care of the noise. However, since median filters and FIR filters are applied to the input signal in similar ways, we can combine them to achieve higher computational efficiency and perhaps more interesting effects. One such combination is the *outlier-trimmed FIR filter*. This system sorts the signal in the observation window just like a median filter, but then removes the  $m$  highest and lowest values. It then adds together the remaining values and divides by their number returning an MA-smoothed result. More generally, an *order statistic filter* first sorts the buffer and then combines the sorted values as a weighted linear sum as in an FIR filter. Usually such filters have their maximal coefficient at the center of the buffer and decrease monotonically toward the buffer ends.

The novelty of the median filter lies in the sorting operation, and we can exploit this same idea for processing other than noise removal. A *dilation filter* outputs the maximal value in the moving buffer, while an *erosion filter* returns the minimal value. These are useful for emphasizing constant positive-valued signals that appear for short time durations, over a background of zero. Dilation expands the region of the signal at the expense of the background while erosion eats away at the signal. Dilation and erosion are often applied to signals that can take on only the values 0 or 1. Dilation is used to fill in holes in long runs of 1s while erosion clips a single spike in the midst of silence. For very noisy signals with large holes or spikes dilation or erosion can be performed multiple times. We can also define two new operations. An *opening filter* is an erosion followed by a dilation while a *closing filter* is a dilation followed by an erosion. The names are meaningful for holes in 0, 1-valued signals. These four operations are most commonly used in image processing, where they are collectively called *morphological processing*.

## EXERCISES

- 8.3.1 Prove that the median filter is not linear.
- 8.3.2 Median filtering is very popular in image processing. What properties of common images make the median filter more appropriate than linear filtering?
- 8.3.3 The *conditional median filter* is similar to the median filter, but only replaces the input value with the median if the difference between the two is above a threshold, otherwise it returns the input value. Explain the motivation behind this variant.
- 8.3.4 Graphically explain the names dilation, erosion, opening, and closing by considering 0, 1-valued signals.
- 8.3.5 Explain how morphological operations are implemented for image processing of binary images (such as fax documents). Consider 'kernels' of different shapes, such as a 3\*3 square and a 5-pixel cross. Program the four operations and show their effect on simple images.

## 8.4 Multilayer Nonlinear Systems

Complex filters are often built up from simpler ones placed in series, a process known as *cascading*. For example, if we have a notch filter with 10 dB attenuation at the unwanted frequencies, but require 40 dB attenuation, the specification can be met by cascading four identical filters. Assume that each of  $N$  cascaded subfilters is a causal FIR filter of length  $L$ , then the combined filter's output at time  $n$  depends on its input at time  $n - NL$ . For example, assume that a finite duration signal  $x_n$  is input to a filter  $h$  producing  $y_n$  that is input into a second filter  $g$  resulting in  $z_n$ . Then

$$\begin{aligned}
 y_n &= h_0x_n + h_1x_{n-1} + h_2x_{n-2} + \dots + h_{L-1}x_{n-L+1} \\
 z_n &= g_0y_n + g_1y_{n-1} + g_2y_{n-2} + \dots + g_{L-1}y_{n-L+1} \\
 &= g_0(h_0x_n + h_1x_{n-1} + h_2x_{n-2} + \dots + h_{L-1}x_{n-L+1}) \\
 &\quad + g_1(h_0x_{n-1} + h_1x_{n-2} + h_2x_{n-3} + \dots + h_{L-2}x_{n-L}) \\
 &= g_0h_0x_n + (g_0h_1 + g_1h_0)x_{n-1} + (g_0h_2 + g_1h_1 + g_2h_0)x_{n-2} + \dots
 \end{aligned}$$

which is equivalent to a single FIR filter with coefficients equal to the convolution  $g * h$ .

In order for a cascaded system to be essentially different from its constituents we must introduce nonlinearity. Augmenting the FIR filter with a

hard limiter we obtain a ‘linear threshold unit’ known more commonly as the *binary perceptron*

$$y_n = \text{sgn} \left( \sum_n w_n x_n \right) \quad (8.8)$$

while using a less drastic smooth nonlinearity we obtain the *sigmoid perceptron*.

$$y_n = \tanh \left( \beta \sum_n w_n x_n \right) \quad (8.9)$$

As  $\beta$  increases the sigmoid perceptron approaches the threshold one. In some applications 0, 1 variables are preferable to  $\pm 1$  ones, and so we use the step function

$$y_n = \Theta \left( \sum_n w_n x_n \right) \quad (8.10)$$

or the smooth version

$$y_n = \sigma \left( \sum_n w_n x_n \right) \quad (8.11)$$

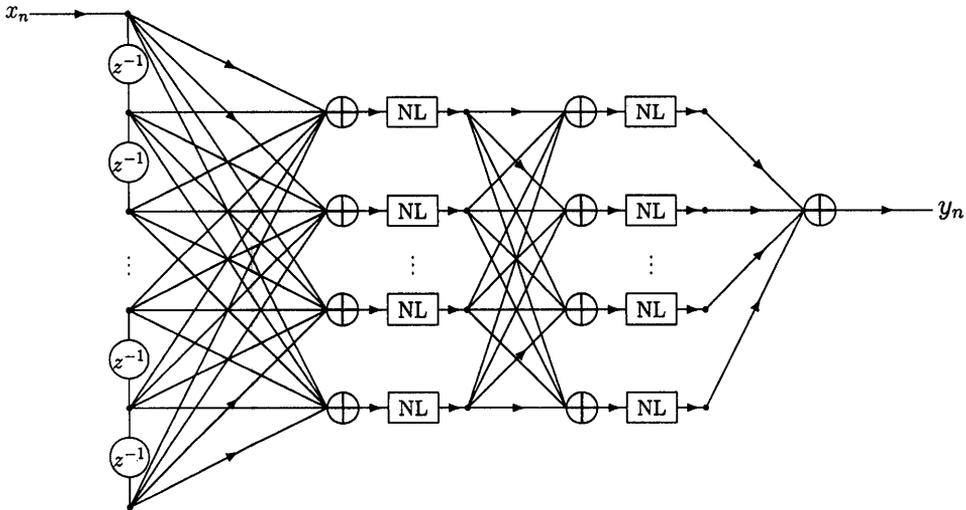
where we defined the ‘logistic sigmoid’.

$$\sigma(x) \equiv \frac{e^x}{1 + e^x} = 1 + \frac{1}{2} \tanh x \quad (8.12)$$

Cascading these nonlinear systems results in truly new systems; a single perceptron can only approximate a small fraction of all possible systems, while it can be shown that arbitrary systems can be realized as cascaded sigmoid perceptrons.

In Figure 8.2 we depict a **MultiLayer Perceptron (MLP)**. This particular MLP has two ‘layers’; the first computes  $L$  weighted sum of the  $N$  input values and then hard or soft limits these to compute the values of  $L$  ‘hidden units’, while the second immediately thereafter computes a single weighted sum over the  $L$  hidden units, creating the desired output. To create a three-layer perceptron one need only produce many second-layer sigmoid weighted sums rather than only one, and afterward combine these together using one final perceptron. A theorem due to Kolmogorov states that three layers are sufficient to realize arbitrary systems.

The perceptron was originally proposed as a classifier, that is, a system with a single signal as input and a logical output or outputs that identify the signal as either belonging to a certain class. Consider classifying spoken digits as belonging to one of the classes named 0, 1, 2... 9. Our MLP could look at all the nonzero speech signal samples, compute several layers of



**Figure 8.2:** A general nonlinear two-layer feedforward system. Although not explicitly shown, each connection arc represents a weight. NL stands for the nonlinearity, for example, the  $\text{sgn}$  or  $\text{tanh}$  function.

hidden values, and finally activate one of 10 output units, thereby expressing its opinion as to the digit that was uttered. Since humans can perform this task we are confident that there is *some* system that can implement the desired function from input samples to output logical values. Since the aforementioned theorem states that (assuming a sufficient number of hidden units) three-layer MLPs can implement arbitrary systems, there must be a three-layer MLP that imitates human behavior and properly classifies the spoken digits.

How are MLP systems designed? The discussion of this topic would lead us too far astray. Suffice it to say that there are *training algorithms* that when presented with a sufficient amount of data can accomplish the required system identification. The most popular of these algorithms is ‘backpropagation’, (‘backprop’) which iteratively presents an input, computes the present output, corrects the internal weights in order to decrease the output error, and then proceeds to the next input-output pair.

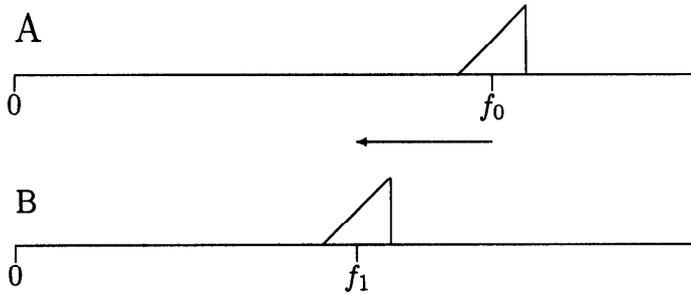
How many hidden units are needed to implement a given system? There are few practical rules here. The aforementioned theorem only says that there is some number of hidden units that allows a given system to be emulated; it does not inform us as to the minimum number needed for all specific cases, or whether one, two, or three layers are needed. In practice these architectural parameters are often determined by trial and error.

## EXERCISES

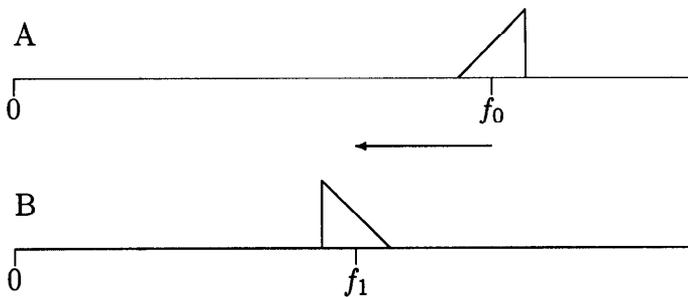
- 8.4.1 Using linear threshold units we can design systems that implement various logic operations, where signal value 0 represents ‘false’ and 1 ‘true’. Find parameters  $w_1$ ,  $w_2$ , and  $\varphi$  such that  $y = \Theta(w_1x_1 + w_2x_2 - \varphi)$  implements the logical AND and logical OR operations. Can we implement these logical operations with linear systems?
- 8.4.2 Of the 16 logical operations between two logical variables, which can and which can’t be implemented?
- 8.4.3 Find a multilayer system can implements XOR.
- 8.4.4 What is the form of curves of equal output for the perceptron of equation (8.9)? What is the form of areas of the same value of equation (8.8)? What is the form of these areas for multilayer perceptrons formed by AND or OR of different simple perceptrons? What types of sets cannot be implemented? How can this limitation be lifted?
- 8.4.5 What are the derivatives of the sigmoid functions (equations (8.11) and (8.9))? Show that  $\sigma'(x) = \sigma(x)(1 - \sigma(x))$ . Can you say something similar regarding the tanh sigmoid?
- 8.4.6 Another nonlinear system element is  $y(x) = e^{\beta \sum_n (x_n - \mu_n)^2}$ , known as the Gaussian radial unit. What is the form of curves of equal output for this unit? What can be said about implementing arbitrary decision functions by radial units?

## 8.5 Mixers

A *mixer* is a system that takes a band-pass signal centered around some frequency  $f_0$ , and moves it along the frequency axis (without otherwise changing it) until it is centered around some other frequency  $f_1$ . Some mixers may also invert the spectrum of the mixed signal. In Figures 8.3 and 8.4 we depict the situation in stylized fashion, where the triangular spectrum has become prevalent in such diagrams, mainly because spectral inversions are obvious. In older analog signal processing textbooks mixing is sometimes called ‘heterodyning’. In many audio applications the term ‘mixing’ is used when simple weighted addition of signals is intended; thus when speaking to audio professionals always say ‘frequency mixing’ when you refer to the subject of this section.



**Figure 8.3:** The effect of mixing a narrow-band analog signal without spectral inversion. In (A) we see the spectrum of the original signal centered at frequency  $f_0$ , and in (B) that of the mixed signal at frequency  $f_1$ .



**Figure 8.4:** The effect of mixing a narrow-band analog signal with spectral inversion. In (A) we see the spectrum of the original signal centered at frequency  $f_0$ , and in (B) the mixed and inverted signal at frequency  $f_1$ . Note how the triangular spectral shape assists in visualizing the inversion.

It is obvious that a mixer *cannot* be a filter, since it can create frequencies where none existed before. In Section 8.1 we saw that harmonics could be generated by introducing nonlinearity. Here there is no obvious nonlinearity; indeed we expect that shifting the frequency of a sum signal will result in the sum of the shifted components. Thus we must conclude that a mixer must be a linear but not a time-invariant system.

Mixers have so many practical applications that we can only mention a few of them here. Mixers are crucial elements in telecommunications systems which transmit signals of the form given in equation (4.66)

$$s(t) = A(t) \sin(2\pi f_c t + \phi(t))$$

where the frequency  $f_c$  is called the *carrier frequency*. The information to be sent is contained either in the amplitude component  $A(t)$ , the phase component  $\phi(t)$ , or both; the purpose of a receiver is to recover this information.

Many receivers start by mixing the received signal down by  $f_c$  to obtain the simpler form of equation (4.65)

$$s(t) = A(t) \sin(\phi(t))$$

from which the amplitude and phase can be recovered using the techniques of Section 4.12.

The phase is intimately connected with the carrier frequency so that the mixing stage is obviously required for proper phase recovery. Even were there to be a mixer but its frequency to be off by some small amount  $\Delta f$  the phase would be misinterpreted as  $2\pi\Delta ft + \phi(t)$  along with the unavoidable jumps of  $2\pi$ . The amplitude signal is apparently independent of the carrier frequency; can we conclude that no mixer is required for the recovery of amplitude-modulated signals? No, although mistuning is much less destructive. The reason a mixer is required is that the receiver sees many possible transmitted signals, each with its own carrier frequency  $f_c$ . Isolation of the desired signal is accomplished by downmixing it and injecting it into a narrow low-pass filter. The output of this filter now contains only the signal of interest and demodulation can continue without interference. When you tune an AM or FM radio in order to hear your favorite station you are actually adjusting a mixer. Older and simpler receivers allow this downmix frequency to be controlled by a continuously rotatable (i.e., analog) knob, while more modern and complex receivers use digital frequency control.

Telephone-quality speech requires less than 4 KHz of bandwidth, while telephone cables can carry a great deal more bandwidth than this. In the interest of economy the telephone network compels a single cable to simultaneously carry many speech signals, a process known as *multiplexing*. It is obvious that we cannot simply add together all the signals corresponding to the different conversations, since there would be no way to separate them at the other end of the cable. One solution, known as **F**requency **D**omain **M**ultiplexing (FDM), consists of upmixing each speech signal by a different offset frequency before adding all the signals together. This results in each signal being confined to its own frequency band, and thus simple band-pass filtering and mixing back down (or mixing first and then low-pass filtering) allows complete recovery of each signal. The operation of building the FDM signal from its components involves upmixing and addition, while the extraction of a single signal requires downmixing and filtering.

Sometimes we need a mixer to compensate for the imperfections of other mixers. For example, a modem signal transmitted via telephone may be upmixed to place it in a FDM transmission, and then downmixed before

delivery to the customer. There will inevitably be a slight difference between the frequency shifts of the mixers at the two ends, resulting in a small residual frequency shift. This tiny shift would never be noticed for speech, but modem signals use frequency and phase information to carry information and even slight shifts cannot be tolerated. For this reason the demodulator part of the modem must first detect this frequency shift and then employ a mixer to correct for it.

Mixers may even appear without our explicitly building them. We saw in Section 8.1 that transmitted signals that pass through nonlinearities may give rise to intermodulation frequencies; we now realize that this is due to unintentional mixing.

A first attempt at numerically implementing a mixer might be to Fourier analyze the signal (e.g., with the FFT), translate the signal in the frequency domain to its new place, and then return to the time domain with the iFT. Such a strategy may indeed work, but has many disadvantages. The digital implementation would be quite computationally intensive, require block processing and so not be real-time-oriented, and only admits mixing by relatively large jumps of the order  $\frac{f_s}{N}$ . What we require is a real-time-oriented time-domain algorithm that allows arbitrary frequency shifts.

As in many such cases, inspiration comes from traditional hardware implementations. Mixers are traditionally implemented by injecting the output of an oscillator (often called the local oscillator) and the signal to be mixed into a nonlinearity. This nonlinearity generates a product signal that has frequency components that are sums and differences of the frequencies of the signal to be mixed and the local oscillator. The mixer is completed by filtering out all components other than the desired one. The essential part of the technique is the forming of a product signal and then filtering.

Consider an analog complex exponential of frequency  $\omega$ .

$$s(t) = Ae^{i\omega t}$$

In order to transform it into an exponential of frequency  $\omega'$

$$s'(t) = Ae^{i\omega' t}$$

we need only multiply it by  $e^{i(\omega' - \omega)t}$ .

$$s(t)e^{i(\omega' - \omega)t} = Ae^{i\omega t}e^{i(\omega' - \omega)t} = Ae^{i\omega' t} = s'(t)$$

Note that the multiplying signal is sinusoidal at the frequency shift frequency and thus the system is not time-invariant.

Similarly, a signal that is composed of many frequency components

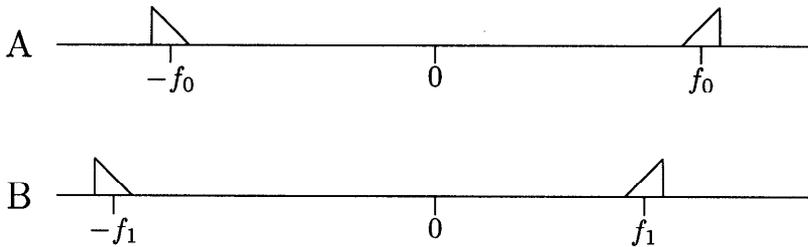
$$s(t) = \sum_k A_k e^{i\omega_k t}$$

will be rigidly translated in frequency when multiplied by a complex exponential.

$$s(t)e^{-i\Delta\omega t} = \sum_k A_k e^{i(\omega_k - \Delta\omega)t}$$

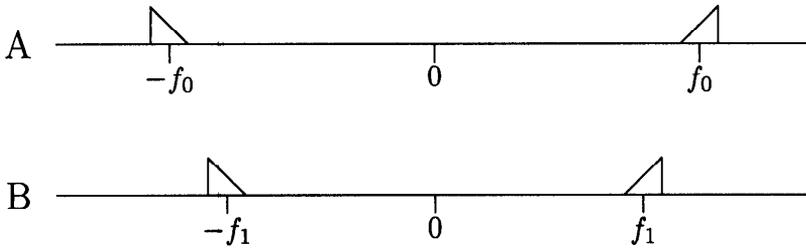
When a signal is mixed down in frequency until it occupies the range from DC up to its bandwidth, it is said to have been ‘downmixed to low-pass’. When we go even further and set the signal’s center frequency to zero, we have ‘downmixed to zero’.

So it seems that mixing is actually quite simple. The problems arise when we try to mix real-valued signals rather than analytic ones, or digital signals rather than analog ones. To illustrate the problems that arise, consider first the mixing of real signals. Since real signals have symmetric spectra, we have to look at both positive and negative frequencies to understand the whole story.

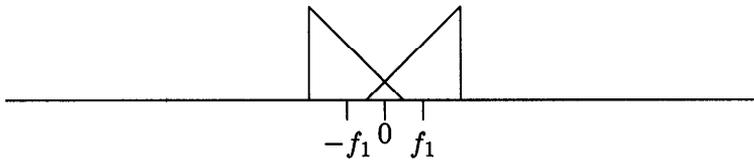


**Figure 8.5:** A real signal at frequency  $f_0$ , whose spectrum is depicted in (A), is moved to frequency  $f_1$  by complex mixing. When a signal is multiplied by a complex exponential all frequency components are shifted in the same direction, as seen in (B).

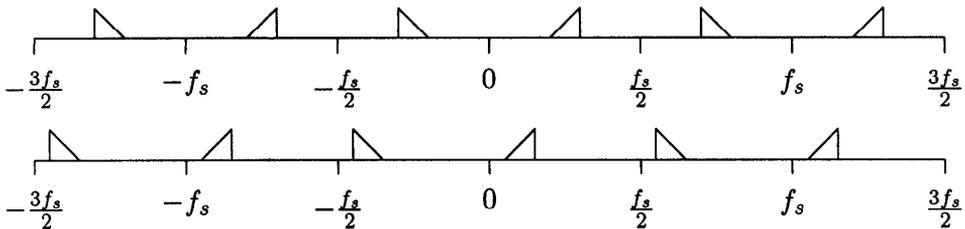
In Figure 8.5 we see the effect of mixing a real-valued signal using a complex exponential local oscillator. The mixer’s effect is precisely as before, but the resulting signal is no longer real! What we really want to do is to mix a real signal using a real oscillator, which is depicted in Figure 8.6. Here the mixer no longer rigidly moves the whole spectrum; rather it compresses or expands it around the DC. In particular we must be careful with downmixing signals past the DC to where the two sides overlap, as in Figure 8.7. Once different parts of the spectrum overlap information is irrevocably lost, and we can no longer reverse the operation by upmixing.



**Figure 8.6:** A real signal at frequency  $f_0$ , whose spectrum is depicted in (A), is moved to frequency  $f_1$  by real mixing. When a real signal is multiplied by a real sinusoid its positive and negative frequency approach each other, as seen in (B).



**Figure 8.7:** A real signal after destructive downmixing. Once the spectrum overlaps itself information is lost.



**Figure 8.8:** A digital mixer. Here a real-valued digital signal is mixed by a complex digital exponential.

What about digital signals? The spectrum of a digital signal is periodic and mixing moves all of the replicas, as depicted in Figure 8.8 for a real digital signal being mixed downward in frequency by a complex exponential. Note that there is a new phenomenon that may occur. Even when mixing with a complex oscillator downmixing to zero causes other spectral components to enter the Nyquist spectral region. This is a kind of aliasing but is both reversible and correctable by appropriate filtering.

## EXERCISES

- 8.5.1 Diagram all the cases of mixing real or complex digital signals by real or complex oscillators.
- 8.5.2 We originally claimed that a mixer generates new frequencies due to its being time-invariant but *linear*. Afterward when discussing its analog implementation we noted that the product is generated by a time-invariant *nonlinearity*. Reconcile these two statements.
- 8.5.3 There are two techniques to mix a real signal down to zero. The signal can be converted to the analytic representation and then multiplied by a complex exponential, or multiplied by the same complex exponential and then low-pass filtered. Demonstrate the equivalence of these two methods. What are the practical advantages and disadvantages of each approach?

## 8.6 Phase-Locked Loops

Another common system that fulfills a function similar to that of a filter, but is not itself a filter, is the **Phase-Locked Loop** (PLL). This is a system that can ‘lock on’ to a sinusoidal signal whose frequency is approximately known, even when this signal is only a small component of the total input. Although the basic idea is to filter out noise and retain the sinusoidal signal of interest, such ‘locking on’ is definitely a nonlinear and time-variant phenomenon and as such cannot be performed by a filter.

Why do we need such a system? One common use is clock recovery in digital communications systems. As a simple example consider someone sending you digital information at a constant rate of 1 bit every  $T$  seconds (presumably  $T$  would be some small number so that a large number of bits may be sent per second). Now the transmitter has a clock that causes a bit to be sent every  $T$  seconds. The receiver, knowing the sender’s intentions, expects a bit every  $T$  seconds. However, the receiver’s clock, being an independent electronic device, will in general run at a slightly different rate than that of the transmitter. So in effect the receiver looks for a bit every  $T'$  seconds instead of every  $T$  seconds. This problem may not be evident at first, but after enough time has passed the receiver is in effect looking for bits at the wrong times, and will either miss bits or report extraneous ones. For high bit rates it doesn’t take long for this to start happening!

In order to avoid this problem the sender can transmit a second signal, for example, a sinusoid of frequency  $f$  generated by the transmitter’s internal

clock. The receiver need only set its clock precisely according to this sinusoid and the discrepancy problem vanishes. This operation of matching clocks is called *synchronization*, often shortened to ‘synching’ (pronounced *sinking*) or ‘synching up’. Synchronization of the receiver’s clock to the transmitter’s has to be maintained continuously; even if properly initially matched, non-synched clocks will drift apart with time, introducing bit slips and insertions.

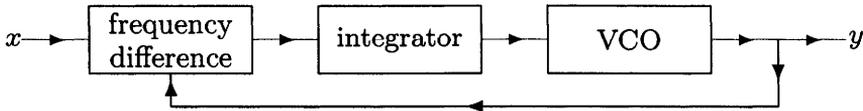
The accurate synching up of the receiver’s clock depends critically on obtaining a clean signal from the transmitter. A naive DSP approach would be to use a very narrow-band band-pass filter centered on  $f$  to recover the clock signal and reject as much noise as possible. Such an attempt is doomed to failure since we don’t know  $f$  (were we to know  $f$  there wouldn’t be anything to do). Setting an extremely sharp band-pass filter centered on the receiver’s estimate  $f'$  may leave the true  $f$  outside the filter bandwidth. Of course we could use a wider filter bandwidth, but that would increase the noise. What we really need is to find and track the received signal’s center frequency. That is what the PLL does.

In order to build a PLL we first need some basic building blocks. The first is traditionally called a **Voltage-Controlled Oscillator (VCO)**. Like an ordinary oscillator the VCO outputs a real sinusoid, but unlike the oscillators we have seen before the VCO has an input as well. With zero input the VCO oscillates at its ‘natural frequency’  $\omega_0$ , but with nonzero input  $x(t)$  the VCO output’s instantaneous frequency changes to  $\omega_0 + \nu(t)$ . It is now straightforward to express the VCO output  $y(t)$  in terms of its input  $x(t)$ .

$$y(t) = A \sin(\omega_0 t + \varphi(t)) \quad \text{where} \quad x(t) = \frac{d\varphi(t)}{dt} \quad (8.13)$$

The analog VCO is thus controlled by the voltage at its input, and hence its name; the digital version should properly be called a **Numerically-Controlled Oscillator (NCO)**, but the name VCO is often used even when no voltages are evident.

The next basic subsystem has two inputs where it expects two pure sinusoids; its output is proportional to the difference in frequency between the two. There are many ways to implement this block, e.g., one could use two frequency demodulators (Section 4.12) and an adder with one input negated. A more devious implementation uses a mixer, a special notch filter and an amplitude demodulator. The VCO output is used to downmix the input to zero; the mixer output is input to a filter with gain  $|\omega|$  so that when the input frequency matches the VCO there is no output, while as the deviation increases so does the amplitude; finally the amplitude demodulator outputs the desired frequency difference.



**Figure 8.9:** The frequency-locked loop (FLL). The output is a sinusoid that tracks the frequency of the input signal.

Using the two special blocks we have defined so far we can already make a first attempt at a system that tracks sinusoidal components (see Figure 8.9). We will call this system a **F**requency-**L**ocked **L**oop (FLL), as its feedback loop causes it to lock onto the frequency of the input signal. Consider what happens when a sinusoid with frequency  $\omega > \omega_0$  is applied to the input (previously zero). At first the frequency difference block outputs  $\omega - \omega_0$ , and were this to be input to the VCO it would change its frequency from  $\omega_0$  to  $\omega_0 + \omega - \omega_0 = \omega$ . Unfortunately, this correct response is just an instantaneous spike since the difference would then become zero and the VCO would immediately return to its natural frequency. The only escape from this predicament is to integrate the difference signal before passing it to the VCO. The integral maintains a constant value when the difference becomes zero, forcing the VCO to remain at  $\omega$ .

The FLL can be useful in some applications but it has a major drawback. Even if the input is a pure sinusoid the FLL output will not in general precisely duplicate it. The reason being that there is no direct relationship between the input and output *phases*. Thus in our bit rate recovery example the FLL would accurately report the rate at which the bits are arriving, but could not tell us precisely when to expect them. In order to track the input signal in both frequency and phase, we need the more sensitive phase-locked loop. Looking carefully at our FLL we see that the frequency difference is integrated, returning a phase difference; the PLL replaces the frequency difference block of the FLL with an explicit phase difference one.

The phase difference subsystem expects two sinusoidal inputs of approximately the same frequency and outputs the phase difference between them. One could be built similarly to the frequency difference block by using two phase demodulators and an adder with one input negated; however, there are approximations that are much easier to build for analog signals and cheaper to compute for digital ones. The most common approximate difference block shifts the phase of one input by  $90^\circ$  and multiplies the two signals.

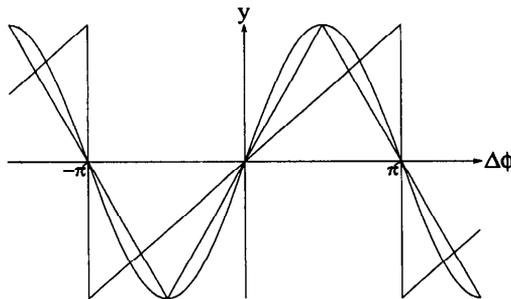
$$\begin{aligned}
 s_1(t) &= \sin(\omega_0 t + \phi_1) \\
 s_2(t) &= \sin(\omega_0 t + \phi_2) \\
 \tilde{s}_2(t) &= \cos(\omega_0 t + \phi_2) \\
 s_1(t)\tilde{s}_2(t) &= \frac{1}{2} \left( \sin(\phi_1 - \phi_2) + \sin(2\omega_0 + \phi_1 + \phi_2) \right)
 \end{aligned}$$

It the low-pass filters the output to remove the double frequency component. The filtered product is proportional to

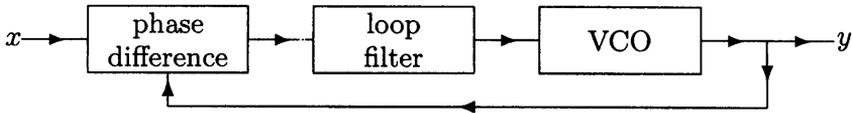
$$\sin(\phi_1 - \phi_2) \sim \phi_1 - \phi_2$$

where the approximation is good for small phase differences.

You may question the wisdom of limiting the range of the phase difference approximation to small values, but recall that even the ideal phase difference is limited to  $\pm\pi$ ! So the ideal phase difference block has a sawtooth characteristic while the approximation has a sinusoidal one. If you really prefer piecewise linear characteristics the *xor phase comparator* is implemented by hard limiting  $s_1$  and  $\tilde{s}_2$  before multiplying them and then averaging over a single cycle. When  $s_1$  and  $s_2$  are precisely in phase,  $s_1$  and  $\tilde{s}_2$  are  $90^\circ$  out of phase and thus their product is positive just as much as it is negative, and so averages to zero. When they move out of phase in either direction the duty cycle of the product becomes nonzero. The characteristics of the three phase difference blocks are contrasted in Figure 8.10.



**Figure 8.10:** Characteristics of three phase difference blocks. The ideal phase difference subsystem has its output vary like a sawtooth as a function of the phase difference  $\Delta\phi = \phi_1 - \phi_2$ . The simple product subsystem has sinusoidal characteristic, while the xor comparator has a triangular one. The important feature of all these blocks is that for small phase differences the characteristic is linear.



**Figure 8.11:** The phase-locked loop, or PLL. The output is a sinusoid that tracks the phase of the input signal.

No matter how we build the phase detector, the proper way to use it is depicted in Figure 8.11. If the input is truly a sinusoid of the VCO's natural frequency, the phase difference output causes the VCO frequency to momentarily increase in order to catch up with the input or decrease to let the input catch up. The PLL is even more useful when the input is noisy. In this case the phase difference varies erratically but the low-pass filter smooths the jumps so that the VCO only tracks the average input phase. Quite noisy signals can be applied provided the low-pass filter is sufficiently narrow.

What if the input frequency doesn't equal the VCO natural frequency? Small constant frequency differences can be thought of as constantly changing phase differences, and the phase corrections will cause the VCO to oscillate at the average input frequency. If the frequency difference is larger than the low-pass filter bandwidth the VCO will receive zero input and remain at its natural frequency, completely oblivious to the input. For input frequencies in the *capture range* the VCO does get some input and starts moving toward the input frequency. The difference then further decreases, allowing more energy through the filter, and the PLL 'snaps' into lock. Once locked the phase difference is DC and completely passed by the filter, thus maintaining lock. If the input frequency varies the VCO automatically tracks it as long as it remains in the *tracking range*.

The low-pass filter used in the PLL is usually of the IIR type. When the phase detector is of the product type a single low-pass filter can be used both for the filtering needed for the PLL's noise rejection and for rejecting the double frequency component. When the double frequency rejection is not required we may be able to skip the filter altogether. In this case there is still a feedback path provided by the PLL architecture, and so the PLL is said to be of *first order*. If the IIR filter has a single pole the additional pole-like behavior leads us to say that the PLL is of *second order*. Higher orders are seldom used because of stability problems.

## EXERCISES

- 8.6.1 Simulate the FLL's frequency behavior by assuming a VCO natural frequency, inputting some other frequency, and using simple addition to integrate. Simulate a slowly varying input frequency. How far can the input frequency be from the natural frequency?
- 8.6.2 Adding a clipping amplifier between the frequency difference and the integrator of the FLL makes the FLL have two operating regions, acquisition and tracking. Analyze the behavior of the system in these two regions.
- 8.6.3 Compare the PLL and FLL from the aspects of frequency acquisition range, steady state frequency, and steady state phase error.
- 8.6.4 Explain how to use the PLL to build a frequency synthesizer, that is, an oscillator with selectable accurate frequency.
- 8.6.5 What effect does decreasing a PLL's low-pass filter bandwidth have on the capture range, the acquisition time, and robustness to noise?

## 8.7 Time Warping

Say 'pneumonoultramicroscopicsilicovolcanoconiosis'. I bet you can't say it again! I mean pronounce precisely the same thing again. It might sound the same to you, but that is only because your brain corrects for the phenomenon to which I am referring; but were you to record both audio signals and compare them you would find that your pacing was different. In the first recording you may have dwelled on the second syllable slightly longer while in the second recording the fourth syllable may have more stress. This relative stretching and compressing of time is called 'time warping', and it is one of the main reasons that automatic speech recognition is so difficult a problem.

For sinusoidal signals making time speed up and then slow down is exactly equivalent to changing the instantaneous frequency, but for more complex signals the effect is somewhat harder to describe using the tools we have developed so far. A system that dynamically warps time is obviously not time-invariant, and hence not a filter; but we are not usually interested in building such a system anyway. The truly important problem is how to compare two signals that would be similar were it not for their undergoing somewhat different time warping.

One approach to solving this problem is called **Dynamic Time Warping** (DTW). DTW is a specific application of the more general theory of *dynamic*

*programming* and essentially equivalent to the *Viterbi algorithm* that we will discuss in Section 18.11. In order to facilitate understanding of the basic concepts of dynamic programming we will first consider the problem of spelling checking. Spelling checkers have become commonplace in word processors as a means of detecting errant words and offering the best alternatives. We will assume that the checker has a precompiled word list (dictionary) and is presented with a string of characters. If the string is a dictionary word then it is returned, otherwise an error has occurred and the *closest* word on the list should be returned.

Three types of errors may occur. First, there may be a *deletion*, that is, a character of the dictionary word may have been left out. Next there may be an *insertion*, where an extra character is added to the text. Finally there may be a *substitution* error, where an incorrect character is substituted for that in the dictionary word. As an example, the word `digital` with a single deletion (of the `a`) becomes `digitl`, and with an additional substitution of `j` for `g` becomes `dijitl`. Were there only substitution errors the number of letters would be preserved, but deletions and insertions cause the matching problem to be similar to DTW.

The *Levenshtein distance* between two character strings is defined to be the minimal number of such errors that must have occurred for one of the strings to become the other. In other words, the Levenshtein distance is the least number of deletions, insertions and substitutions that must be performed on one string to make it become the other. As we saw above `dijitl` is distance *two* from `digital`; of course we could have arrived at `dijitl` by two deletions and an insertion, but this would not have been the *minimal* number of operations. The Levenshtein distance is thus an ideal candidate for the idea of ‘closeness’ needed for our spelling checker. When the given string is not in the dictionary we return the dictionary word separated from the input string by minimal Levenshtein distance.

In order to be able to use this distance in practice, we must now produce an algorithm that efficiently computes it. To see that this is not a trivial task let’s try to find the distance between `prossesing` and `processing`. Simple counting shows that it is better to substitute a `c` for the first `s`, delete the second and then add another `s` (3 operations) rather than deleting the `es` and adding `ce` (4 operations). But how did we come up with this set of operations and how can we prove that this is the best that can be done? The problem is that the Levenshtein distance is a cost function for changing an entire string into another, and thus a global optimization seems to be required. Dynamic programming is an algorithm that reduces this global optimization to a sequence of local calculations and decisions.

Dynamic programming is best understood graphically. Write the dictionary word from left to right at the bottom of a piece of graph paper, and write the input string from bottom to top at the left of the word. For our previous example you should get something like this.

<b>g</b>										
<b>n</b>										
<b>i</b>										
<b>s</b>										
<b>e</b>										
<b>s</b>										
<b>s</b>										
<b>o</b>										
<b>r</b>										
<b>p</b>										
	<b>p</b>	<b>r</b>	<b>o</b>	<b>c</b>	<b>e</b>	<b>s</b>	<b>s</b>	<b>i</b>	<b>n</b>	<b>g</b>

Now we fill in each of the blank squares with the minimal cost to get to that square. The bottom-left square is initialized to zero since we start there, and all the rest of the squares will get values that can be computed recursively. We finally arrive at the top right square, and the value there will be the total cost, namely the Levenshtein distance.

The recursive step involves comparing three components. One can enter a square from its left, corresponding to a deletion from the dictionary word, by taking the value to its left and adding one. One can enter a square from underneath, corresponding to an insertion into the dictionary word, by taking the value underneath it and incrementing. Finally, one can enter a square from the square diagonally to the left and down; if the letter in the dictionary word at the bottom of the column is the same as the letter in the string at the beginning of the row, then there is no additional cost and we simply copy the value from the diagonal square. If the letters differ, a substitution is needed and so we increment the value diagonally beneath. In this fashion each square gets three possible values, and we always choose the minimum of these three.

Let's try this out on our example. We start with the table from above, initialize the bottom left square, and trivially fill in the lowest row and leftmost column.

<b>g</b>	9									
<b>n</b>	8									
<b>i</b>	7									
<b>s</b>	6									
<b>e</b>	5									
<b>s</b>	4									
<b>s</b>	3									
<b>o</b>	2									
<b>r</b>	1									
<b>p</b>	0	1	2	3	4	5	6	7	8	9
<b>0</b>	<b>p</b>	<b>r</b>	<b>o</b>	<b>c</b>	<b>e</b>	<b>s</b>	<b>s</b>	<b>i</b>	<b>n</b>	<b>g</b>

Now we can continue filling in the entire table, and find (as we previously discovered in a rather undisciplined fashion) that the Levenshtein distance is indeed 3.

<b>g</b>	9	8	7	7	6	5	5	5	5	<b>3</b>
<b>n</b>	8	7	6	6	5	4	4	4	<b>3</b>	5
<b>i</b>	7	6	5	5	4	3	<b>3</b>	<b>3</b>	5	5
<b>s</b>	6	5	4	4	3	<b>2</b>	3	4	4	5
<b>e</b>	5	4	3	3	<b>2</b>	3	3	3	4	5
<b>s</b>	4	3	2	<b>2</b>	2	2	2	3	4	5
<b>s</b>	3	2	<b>1</b>	<b>1</b>	2	2	3	4	5	6
<b>o</b>	2	1	<b>0</b>	1	2	3	4	5	6	7
<b>r</b>	1	<b>0</b>	1	2	3	4	5	6	7	8
<b>p</b>	<b>0</b>	1	2	3	4	5	6	7	8	9
<b>0</b>	<b>p</b>	<b>r</b>	<b>o</b>	<b>c</b>	<b>e</b>	<b>s</b>	<b>s</b>	<b>i</b>	<b>n</b>	<b>g</b>

From the table we can discover more than simply the total distance, we can actually reconstruct the optimal sequence of operations. Indeed the optimal set of deletions, insertions, and substitutions pops out to the eye as the path of minimal cost through the table. At first there seem to be many optimal paths, but quite a few of these correspond to making a deletion and insertion instead of some substitution. The true path segments are the ones that contributed the minimal cost transitions. Thus to find the true path you start at the end point and retrace your steps backward through the table; we can save redundant computation by storing in each square not only its cost but the previous square visited. The only ambiguities that remain correspond to squares where more than one transition produced the same minimal cost; in our example changing the dictionary *c* to the incorrect

ss could be accomplished by changing the c to s and then inserting an s, or by first inserting an s and then changing the c to s.

Up to now we have assumed that all errors have the same cost, but that is not always the case. Some mistaken keypresses are more prevalent than others, and there is really very little reason to assume a deletion is as likely as an insertion. However, it is not difficult to generalize the Levenshtein distance to take this into account; one need only add specific penalties rather than simply incrementing by one.

This algorithm for finding the generalized Levenshtein distance is exactly the DTW algorithm for comparing two spoken words. The word from the dictionary is placed horizontally from left to right at the bottom of a table, and the word to be compared is stretched vertically from bottom to top. We then compare short segments of the two words using some cost function (e.g., correlation, difference in spectral description, etc.) that is small for similar sounding segments. When noise contaminates a segment we may make a substitution error, while time warping causes deletions and insertions of segments. In order to identify a word we compare it to all words in the dictionary and return the word with the lowest Levenshtein distance.

## EXERCISES

- 8.7.1 The game of *doublets* was invented in 1879 by Lewis Carroll (the mathematician Charles Lutwidge Dodgson 1832-1898). The aim of the game is to convert a word into a related word in the minimal number of *substitution* steps; However, each step must leave an actual word. For example, we can change *hate* into *love*, in three steps in the following way: *hate have lave love*. Show how to make a *cat* into a *dog* in three steps, how an *ape* can *evolve* into a *man* in five steps, and how to *raise* *four* to *five* by a seven step procedure. *four foul fool foot fort fore fire five*. How many steps does it take to *drive the pig into the sty*?
- 8.7.2 In more complex implementations of spelling checkers further types of errors may be added (e.g., reversal of the order of two letters). Can the dynamical programming algorithm still be used to determine the Levenshtein distance?
- 8.7.3 An alternative method for comparing time-warped signals is the Markov model approach. Here we assume that the signal is generated by a Markov model with states  $O_1, O_2 \dots O_M$ . When the model is in state  $O_m$  it has probability  $a_{m,m}$  of staying in the same state, probability  $a_{m,m+1}$  of transitioning to state  $O_{m+1}$ , and probability  $a_{m,m+2}$  of skipping over state  $O_{m+1}$  directly to state  $O_{m+2}$ . When the model is in state  $O_m$  it outputs a characteristic signal segment  $s_m$ . Write a program that simulates a Markov model and run it several times. Do you see how the time warping arises?

- 8.7.4 An extension to the above model is the **Hidden Markov Model (HMM)**. The HMM states are hidden since they do not uniquely correspond to an output signal segment; rather when the model is in a state  $O_m$  it has probability  $b_{mi}$  of outputting signal  $s_i$ . Extend the program of the previous exercise to generate HMM signals. Why is the HMM more realistic for speech?

## Bibliographical Notes

Although there are a lot of books that deal with things that are not filters, there are very few such that happen to treat signal processing.

Median and morphological filters are mostly discussed in books on image processing, but see [68, 258, 157].

Multilayer perceptrons were introduced in [225], and popularized in the books by the same authors [168, 169], although the basic idea had been previously discovered by several researchers. A popular short introduction is [150].

Phase-locked loops are usually discussed in books on digital communications, e.g., [242, 199].

Time warping and HMM are discussed in texts on speech recognition, e.g., [204, 176].