of $x(n)$ would require two vectors, one each for x and n. For example, a sequence $x(n) = \{2, 1, -1, \underset{\uparrow}{0}, 1, 4, 3, 7\}$ can be represented in MATLAB by

```
>> n=[-3,-2,-1,0,1,2,3,4];  x=[2,1,-1,0,1,4,3,7];
```

Generally, we will use the x-vector representation alone when the sample position information is not required or when such information is trivial (e.g. when the sequence begins at $n = 0$). An arbitrary *infinite-duration* sequence cannot be represented in MATLAB due to the finite memory limitations.

TYPES OF SEQUENCES

We use several elementary sequences in digital signal processing for analysis purposes. Their definitions and MATLAB representations are given below.

1. *Unit sample sequence*:

$$\delta(n) = \begin{cases} 1, & n = 0 \\ 0, & n \neq 0 \end{cases} = \left\{ \cdots, 0, 0, \underset{\uparrow}{1}, 0, 0, \cdots \right\}$$

In MATLAB the function `zeros(1,N)` generates a row vector of N zeros, which can be used to implement $\delta(n)$ over a finite interval. However, the logical relation `n==0` is an elegant way of implementing $\delta(n)$. For example, to implement

$$\delta(n - n_0) = \begin{cases} 1, & n = n_0 \\ 0, & n \neq n_0 \end{cases}$$

over the $n_1 \leq n_0 \leq n_2$ interval, we will use the following MATLAB function.

```
function [x,n] = impseq(n0,n1,n2)
% Generates x(n) = delta(n-n0); n1 <= n <= n2
% ---------------------------------------------
% [x,n] = impseq(n0,n1,n2)
%
n = [n1:n2]; x = [(n-n0) == 0];
```

2. *Unit step sequence*:

$$u(n) = \begin{cases} 1, & n \geq 0 \\ 0, & n < 0 \end{cases} = \left\{ \cdots, 0, 0, \underset{\uparrow}{1}, 1, 1, \cdots \right\}$$

In MATLAB the function `ones(1,N)` generates a row vector of N ones. It can be used to generate $u(n)$ over a finite interval. Once again an elegant

approach is to use the logical relation `n>=0`. To implement

$$u(n - n_0) = \begin{cases} 1, & n \geq n_0 \\ 0, & n < n_0 \end{cases}$$

over the $n_1 \leq n_0 \leq n_2$ interval, we will use the following MATLAB function.

```
function [x,n] = stepseq(n0,n1,n2)
% Generates x(n) = u(n-n0); n1 <= n <= n2
% ---------------------------------------
% [x,n] = stepseq(n0,n1,n2)
%
n = [n1:n2]; x = [(n-n0) >= 0];
```

3. *Real-valued exponential sequence*:

$$x(n) = a^n, \forall n;\ a \in \mathbb{R}$$

In MATLAB an array operator ".^" is required to implement a real exponential sequence. For example, to generate $x(n) = (0.9)^n$, $0 \leq n \leq 10$, we will need the following MATLAB script:

```
>> n = [0:10]; x = (0.9).^n;
```

4. *Complex-valued exponential sequence*:

$$x(n) = e^{(\sigma + j\omega_0)n}, \forall n$$

where σ is called an attenuation and ω_0 is the frequency in radians. A MATLAB function `exp` is used to generate exponential sequences. For example, to generate $x(n) = \exp[(2 + j3)n]$, $0 \leq n \leq 10$, we will need the following MATLAB script:

```
>> n = [0:10]; x = exp((2+3j)*n);
```

5. *Sinusoidal sequence*:

$$x(n) = \cos(\omega_0 n + \theta), \forall n$$

where θ is the phase in radians. A MATLAB function `cos` (or `sin`) is used to generate sinusoidal sequences. For example, to generate $x(n) = 3\cos(0.1\pi n + \pi/3) + 2\sin(0.5\pi n)$, $0 \leq n \leq 10$, we will need the following MATLAB script:

```
>> n = [0:10]; x = 3*cos(0.1*pi*n+pi/3) + 2*sin(0.5*pi*n);
```

6. *Random sequences*: Many practical sequences cannot be described by mathematical expressions like those above. These sequences are called random (or stochastic) sequences and are characterized by parameters of the associated probability density functions or their statistical moments. In MATLAB two types of (pseudo-) random sequences are available. The `rand(1,N)` generates a length N random sequence whose elements are uniformly distributed between $[0, 1]$. The `randn(1,N)` generates a length N Gaussian random sequence with mean 0 and variance 1. Other random sequences can be generated using transformations of the above functions.

7. *Periodic sequence*: A sequence $x(n)$ is periodic if $x(n) = x(n+N)$, $\forall n$. The smallest integer N that satisfies the above relation is called the *fundamental* period. We will use $\tilde{x}(n)$ to denote a periodic sequence. To generate P periods of $\tilde{x}(n)$ from one period $\{x(n), \quad 0 \le n \le N-1\}$, we can copy $x(n)$ P times:

```
>> xtilde = [x,x,...,x];
```

But an elegant approach is to use MATLAB's powerful indexing capabilities. First we generate a matrix containing P rows of $x(n)$ values. Then we can concatenate P rows into a long row vector using the construct `(:)`. However, this construct works only on columns. Hence we will have to use the matrix transposition operator `'` to provide the same effect on rows.

```
>> xtilde = x' * ones(1,P);   % P columns of x; x is a row vector
>> xtilde = xtilde(:);        % long column vector
>> xtilde = xtilde';          % long row vector
```

Note that the last two lines can be combined into one for compact coding. This is shown in Example 2.1.

OPERATIONS ON SEQUENCES

Here we briefly describe basic sequence operations and their MATLAB equivalents.

1. *Signal addition*: This is a sample-by-sample addition given by

$$\{x_1(n)\} + \{x_2(n)\} = \{x_1(n) + x_2(n)\}$$

It is implemented in MATLAB by the arithmetic operator "+". However, the lengths of $x_1(n)$ and $x_2(n)$ must be the same. If sequences are of unequal lengths, or if the sample positions are different for equal-length sequences, then we cannot directly use the operator +. We have to first augment $x_1(n)$ and $x_2(n)$ so that they have the same position vector `n` (and hence the same length). This requires careful attention to MATLAB's indexing operations. In particular, logical operation of intersection "`&`", relational operations like "`<=`" and "`==`", and the `find` function are required to make $x_1(n)$ and $x_2(n)$ of equal length. The following function, called the `sigadd` function, demonstrates these operations.

```
function [y,n] = sigadd(x1,n1,x2,n2)
% implements y(n) = x1(n)+x2(n)
% ------------------------------
% [y,n] = sigadd(x1,n1,x2,n2)
%   y = sum sequence over n, which includes n1 and n2
%  x1 = first sequence over n1
%  x2 = second sequence over n2 (n2 can be different from n1)
%
n = min(min(n1),min(n2)):max(max(n1),max(n2)); % duration of y(n)
y1 = zeros(1,length(n)); y2 = y1;              % initialization
y1(find((n>=min(n1))&(n<=max(n1))==1))=x1;     % x1 with duration of y
y2(find((n>=min(n2))&(n<=max(n2))==1))=x2;     % x2 with duration of y
y = y1+y2;                                     % sequence addition
```

Its use is illustrated in Example 2.2.

2. *Signal multiplication*: This is a sample-by-sample multiplication (or "dot" multiplication) given by

$$\{x_1(n)\} \cdot \{x_2(n)\} = \{x_1(n)x_2(n)\}$$

It is implemented in MATLAB by the array operator ".*". Once again the similar restrictions apply for the `.*` operator as for the `+` operator. Therefore we have developed the `sigmult` function, which is similar to the `sigadd` function.

```
function [y,n] = sigmult(x1,n1,x2,n2)
% implements y(n) = x1(n)*x2(n)
% ------------------------------
% [y,n] = sigmult(x1,n1,x2,n2)
%   y = product sequence over n, which includes n1 and n2
%  x1 = first sequence over n1
%  x2 = second sequence over n2 (n2 can be different from n1)
%
n = min(min(n1),min(n2)):max(max(n1),max(n2)); % duration of y(n)
y1 = zeros(1,length(n)); y2 = y1;              %
y1(find((n>=min(n1))&(n<=max(n1))==1))=x1;     % x1 with duration of y
y2(find((n>=min(n2))&(n<=max(n2))==1))=x2;     % x2 with duration of y
y = y1 .* y2;                                  % sequence multiplication
```

Its use is also given in Example 2.2.

3. *Scaling*: In this operation each sample is multiplied by a scalar α.

$$\alpha\{x(n)\} = \{\alpha x(n)\}$$

An arithmetic operator "*" is used to implement the scaling operation in MATLAB.

4. *Shifting*: In this operation each sample of $x(n)$ is shifted by an amount k to obtain a shifted sequence $y(n)$.

$$y(n) = \{x(n-k)\}$$

If we let $m = n - k$, then $n = m + k$ and the above operation is given by

$$y(m+k) = \{x(m)\}$$

Hence this operation has no effect on the vector `x`, but the vector `n` is changed by adding k to each element. This is shown in the function `sigshift`.

```
function [y,n] = sigshift(x,m,n0)
% implements y(n) = x(n-n0)
% -------------------------
% [y,n] = sigshift(x,m,n0)
%
n = m+n0; y = x;
```

Its use is given in Example 2.2.

5. *Folding*: In this operation each sample of $x(n)$ is flipped around $n = 0$ to obtain a folded sequence $y(n)$.

$$y(n) = \{x(-n)\}$$

In MATLAB this operation is implemented by `fliplr(x)`function for sample values and by `-fliplr(n)` function for sample positions as shown in the `sigfold` function.

```
function [y,n] = sigfold(x,n)
% implements y(n) = x(-n)
% -----------------------
% [y,n] = sigfold(x,n)
%
y = fliplr(x); n = -fliplr(n);
```

6. *Sample summation*: This operation differs from signal addition operation. It adds all sample values of $x(n)$ between n_1 and n_2.

$$\sum_{n=n_1}^{n_2} x(n) = x(n_1) + \cdots + x(n_2)$$

It is implemented by the `sum(x(n1:n2))` function.

7. *Sample products*: This operation also differs from signal multiplication operation. It multiplies all sample values of $x(n)$ between n_1 and n_2.

$$\prod_{n_1}^{n_2} x(n) = x(n_1) \times \cdots \times x(n_2)$$

It is implemented by the `prod(x(n1:n2))` function.

8. *Signal energy*: The energy of a sequence $x(n)$ is given by

$$\mathcal{E}_x = \sum_{-\infty}^{\infty} x(n)x^*(n) = \sum_{-\infty}^{\infty} |x(n)|^2$$

where superscript * denotes the operation of complex conjugation[1]. The energy of a finite-duration sequence $x(n)$ can be computed in MATLAB using

```
>> Ex = sum(x .* conj(x)); % one approach
>> Ex = sum(abs(x) .^ 2); % another approach
```

9. *Signal power*: The average power of a periodic sequence with fundamental period N is given by

$$\mathcal{P}_x = \frac{1}{N} \sum_{0}^{N-1} |x(n)|^2$$

□ **EXAMPLE 2.1** Generate and plot each of the following sequences over the indicated interval.

a. $x(n) = 2\delta(n+2) - \delta(n-4), \quad -5 \le n \le 5.$

b. $x(n) = n\,[u(n) - u(n-10)] + 10e^{-0.3(n-10)}\,[u(n-10) - u(n-20)], \quad 0 \le n \le 20.$

c. $x(n) = \cos(0.04\pi n) + 0.2w(n), \quad 0 \le n \le 50,$ where $w(n)$ is a Gaussian random sequence with zero mean and unit variance.

d. $\tilde{x}(n) = \{\ldots, 5, 4, 3, 2, 1, \underset{\uparrow}{5}, 4, 3, 2, 1, 5, 4, 3, 2, 1, \ldots\}; \quad -10 \le n \le 9.$

Solution

a. $x(n) = 2\delta(n+2) - \delta(n-4), \quad -5 \le n \le 5.$

```
>> n = [-5:5];
>> x = 2*impseq(-2,-5,5) - impseq(4,-5,5);
>> stem(n,x); title('Sequence in Problem 2.1a')
>> xlabel('n'); ylabel('x(n)');
```

The plot of the sequence is shown in Figure 2.1a.

[1]The symbol * denotes many operations in digital signal processing. Its font (roman or computer) and its position (normal or superscript) will distinguish each operation.

b. $x(n) = n[u(n) - u(n-10)] + 10e^{-0.3(n-10)}[u(n-10) - u(n-20)]$, $0 \le n \le 20$.

```
>> n = [0:20];
>> x1 = n.*(stepseq(0,0,20)-stepseq(10,0,20));
>> x2 = 10*exp(-0.3*(n-10)).*(stepseq(10,0,20)-stepseq(20,0,20));
>> x = x1+x2;
>> subplot(2,2,3); stem(n,x); title('Sequence in Problem 2.1b')
>> xlabel('n'); ylabel('x(n)');
```

The plot of the sequence is shown in Figure 2.1b.

c. $x(n) = \cos(0.04\pi n) + 0.2w(n), \quad 0 \le n \le 50$.

```
>> n = [0:50];
>> x = cos(0.04*pi*n)+0.2*randn(size(n));
>> subplot(2,2,2); stem(n,x); title('Sequence in Problem 2.1c')
>> xlabel('n'); ylabel('x(n)');
```

The plot of the sequence is shown in Figure 2.1c.

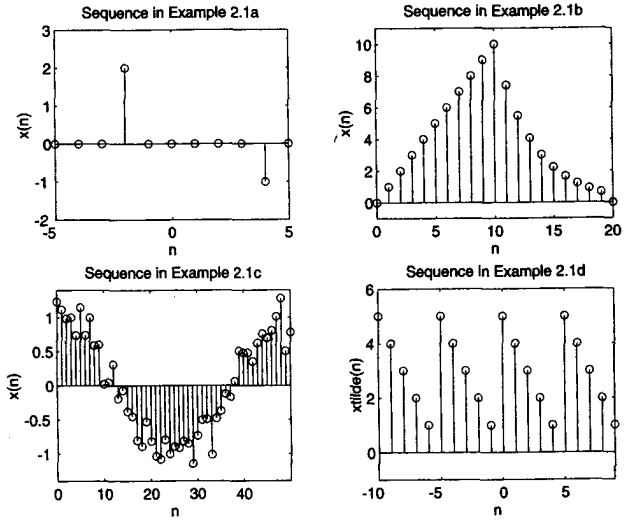

FIGURE 2.1 *Sequences in Example 2.1*

d. $\tilde{x}(n) = \{\ldots, 5, 4, 3, 2, 1, \underset{\uparrow}{5}, 4, 3, 2, 1, 5, 4, 3, 2, 1, \ldots\}$; $-10 \le n \le 9$.

Note that over the given interval, the sequence $\tilde{x}(n)$ has four periods.

```
>> n = [-10:9]; x = [5,4,3,2,1];
>> xtilde = x' * ones(1,4);
>> xtilde = (xtilde(:))';
>> subplot(2,2,4); stem(n,xtilde); title('Sequence in Problem 2.1d')
>> xlabel('n'); ylabel('xtilde(n)');
```

The plot of the sequence is shown in Figure 2.1d. □

□ **EXAMPLE 2.2** Let $x(n) = \{1, 2, 3, \underset{\uparrow}{4}, 5, 6, 7, 6, 5, 4, 3, 2, 1\}$. Determine and plot the following sequences.

a. $x_1(n) = 2x(n-5) - 3x(n+4)$
b. $x_2(n) = x(3-n) + x(n)x(n-2)$

Solution The sequence $x(n)$ is nonzero over $-2 \le n \le 10$. Hence

```
>> n = -2:10; x = [1:7,6:-1:1];
```

will generate $x(n)$.

a. $x_1(n) = 2x(n-5) - 3x(n+4)$.

The first part is obtained by shifting $x(n)$ by 5 and the second part by shifting $x(n)$ by -4. This shifting and the addition can be easily done using the `sigshift` and the `sigadd` functions.

```
>> [x11,n11] = sigshift(x,n,5); [x12,n12] = sigshift(x,n,-4);
>> [x1,n1] = sigadd(2*x11,n11,-3*x12,n12);
>> subplot(2,1,1); stem(n1,x1); title('Sequence in Example 2.2a')
>> xlabel('n'); ylabel('x1(n)');
```

The plot of $x_1(n)$ is shown in Figure 2.2a.

b. $x_2(n) = x(3-n) + x(n)x(n-2)$.

The first term can be written as $x(-(n-3))$. Hence it is obtained by first folding $x(n)$ and then shifting the result by 3. The second part is a multiplication of $x(n)$ and $x(n-2)$, both of which have the same length but different support (or sample positions). These operations can be easily done using the `sigfold` and the `sigmult` functions.

```
>> [x21,n21] = sigfold(x,n); [x21,n21] = sigshift(x21,n21,3);
>> [x22,n22] = sigshift(x,n,2); [x22,n22] = sigmult(x,n,x22,n22);
>> [x2,n2] = sigadd(x21,n21,x22,n22);
>> subplot(2,1,2); stem(n2,x2); title('Sequence in Example 2.2b')
>> xlabel('n'); ylabel('x2(n)');
```

The plot of $x_2(n)$ is shown in Figure 2.2b. □

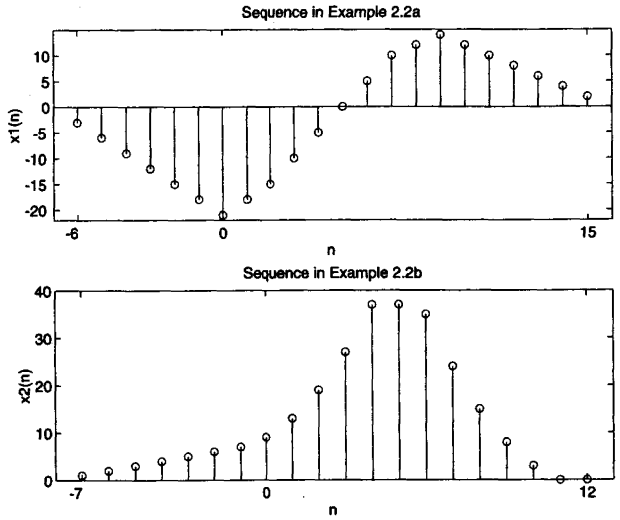

FIGURE 2.2 *Sequences in Example 2.2*

This example shows that the four `sig*` functions developed in this section provide a convenient approach for sequence manipulations.

□ **EXAMPLE 2.3** Generate the complex-valued signal

$$x(n) = e^{(-0.1+j0.3)n}, \quad -10 \le n \le 10$$

and plot its magnitude, phase, the real part, and the imaginary part in four separate subplots.

Solution

MATLAB Script

```
>> n = [-10:1:10]; alpha = -0.1+0.3j;
>> x = exp(alpha*n);
>> subplot(2,2,1); stem(n,real(x));title('real part');xlabel('n')
>> subplot(2,2,2); stem(n,imag(x));title('imaginary part');xlabel('n')
>> subplot(2,2,3); stem(n,abs(x));title('magnitude part');xlabel('n')
>> subplot(2,2,4); stem(n,(180/pi)*angle(x));title('phase part');xlabel('n')
```

The plot of the sequence is shown in Figure 2.3. □

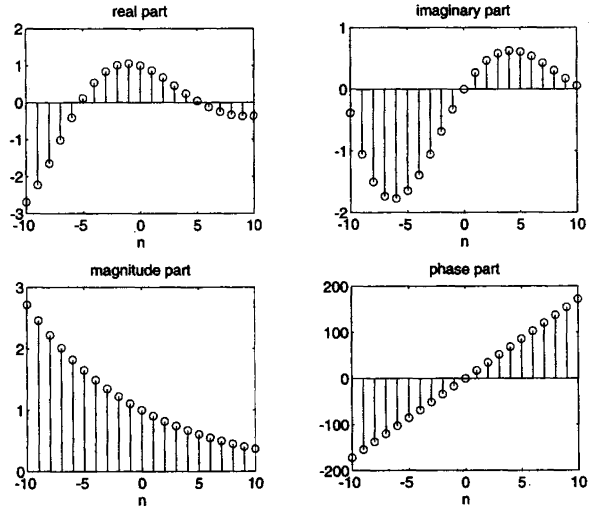

FIGURE 2.3 *Complex-valued sequence plots in Example 2.3*

SOME USEFUL RESULTS

There are several important results in discrete-time signal theory. We will discuss some that are useful in digital signal processing.

Unit sample synthesis Any arbitrary sequence $x(n)$ can be synthesized as a weighted sum of delayed and scaled unit sample sequences, such as

$$x(n) = \sum_{k=-\infty}^{\infty} x(k)\delta(n-k) \tag{2.1}$$

We will use this result in the next section.

Even and odd synthesis A real-valued sequence $x_e(n)$ is called even (symmetric) if

$$x_e(-n) = x_e(n)$$

Similarly, a real-valued sequence $x_o(n)$ is called odd (antisymmetric) if

$$x_o(-n) = -x_o(n)$$

Then any arbitrary real-valued sequence $x(n)$ can be decomposed into its even and odd components

$$x(n) = x_e(n) + x_o(n) \tag{2.2}$$

where the even and odd parts are given by

$$x_e(n) = \frac{1}{2}[x(n) + x(-n)] \quad \text{and} \quad x_o(n) = \frac{1}{2}[x(n) - x(-n)] \tag{2.3}$$

respectively. We will use this decomposition in studying properties of the Fourier transform. Therefore it is a good exercise to develop a simple MATLAB function to decompose a given sequence into its even and odd components. Using MATLAB operations discussed so far, we can obtain the following evenodd function.

```
function [xe, xo, m] = evenodd(x,n)
% Real signal decomposition into even and odd parts
% ------------------------------------------------
% [xe, xo, m] = evenodd(x,n)
%
if any(imag(x) ~= 0)
      error('x is not a real sequence')
end
m = -fliplr(n);
m1 = min([m,n]); m2 = max([m,n]); m = m1:m2;
nm = n(1)-m(1); n1 = 1:length(n);
x1 = zeros(1,length(m));
x1(n1+nm) = x; x = x1;
xe = 0.5*(x + fliplr(x));
xo = 0.5*(x - fliplr(x));
```

The sequence and its support are supplied in x and n arrays, respectively. It first checks if the given sequence is real and determines the support of the even and odd components in m array. It then implements (2.3) with special attention to the MATLAB indexing operation. The resulting components are stored in xe and xo arrays.

□ **EXAMPLE 2.4** Let $x(n) = u(n) - u(n-10)$. Decompose $x(n)$ into even and odd components.

Solution The sequence $x(n)$, which is nonzero over $0 \le n \le 9$, is called a *rectangular pulse*. We will use MATLAB to determine and plot its even and odd parts.

```
>> n = [0:10]; x = stepseq(0,0,10)-stepseq(10,0,10);
>> [xe,xo,m] = evenodd(x,n);
>> figure(1); clf
>> subplot(2,2,1); stem(n,x); title('Rectangular pulse')
>> xlabel('n'); ylabel('x(n)'); axis([-10,10,0,1.2])
>> subplot(2,2,2); stem(m,xe); title('Even Part')
>> xlabel('n'); ylabel('xe(n)'); axis([-10,10,0,1.2])
>> subplot(2,2,4); stem(m,xo); title('Odd Part')
>> xlabel('n'); ylabel('xe(n)'); axis([-10,10,-0.6,0.6])
```

The plots shown in Figure 2.4 clearly demonstrate the decomposition. □

A similar decomposition for complex-valued sequences is explored in Problem 2.5.

The geometric series A one-sided exponential sequence of the form $\{\alpha^n, \quad n \ge 0\}$, where α is an arbitrary constant, is called a geometric series. In digital signal processing, the convergence and expression for the sum of this series are used in many applications. The series converges for $|\alpha| < 1$, while the sum of its components converges to

$$\sum_{n=0}^{\infty} \alpha^n \longrightarrow \frac{1}{1-\alpha}, \quad \text{for } |\alpha| < 1 \tag{2.4}$$

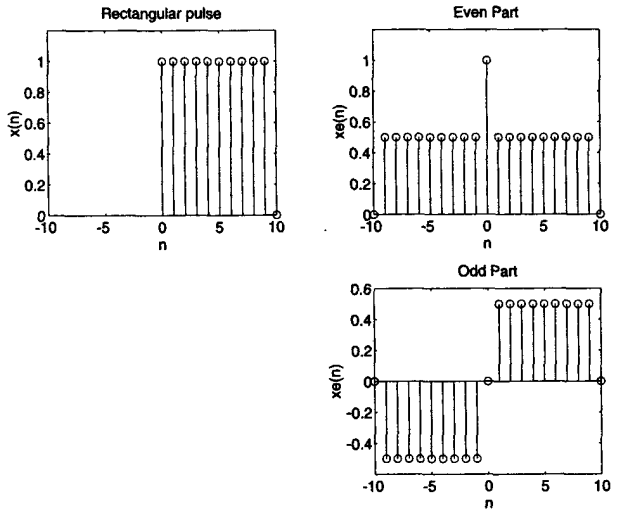

FIGURE 2.4 *Even-odd decomposition in Example 2.4*

We will also need an expression for the sum of any finite number of terms of the series given by

$$\sum_{n=0}^{N-1} \alpha^n = \frac{1-\alpha^N}{1-\alpha}, \ \forall \alpha \tag{2.5}$$

These two results will be used throughout this book.

Correlations of sequences Correlation is an operation used in many applications in digital signal processing. It is a measure of the degree to which two sequences are similar. Given two real-valued sequences $x(n)$ and $y(n)$ of finite energy, the *crosscorrelation* of $x(n)$ and $y(n)$ is a sequence $r_{xy}(\ell)$ defined as

$$r_{x,y}(\ell) = \sum_{n=-\infty}^{\infty} x(n)y(n-\ell) \tag{2.6}$$

The index ℓ is called the shift or lag parameter. The special case of (2.6) when $y(n) = x(n)$ is called *autocorrelation* and is defined by

$$r_{xx}(\ell) = \sum_{n=-\infty}^{\infty} x(n)x(n-\ell) \tag{2.7}$$

It provides a measure of self-similarity between different alignments of the sequence. MATLAB functions to compute auto- and crosscorrelations are discussed later in the chapter.

DISCRETE SYSTEMS

Mathematically, a discrete-time system (or *discrete system* for short) is described as an operator $T[\cdot]$ that takes a sequence $x(n)$ (called *excitation*) and transforms it into another sequence $y(n)$ (called *response*). That is,

$$y(n) = T[x(n)]$$

In DSP we will say that the system processes an *input* signal into an *output* signal. Discrete systems are broadly classified into *linear* and *nonlinear* systems. We will deal mostly with linear systems.

LINEAR SYSTEMS

A discrete system $T[\cdot]$ is a linear operator $L[\cdot]$ if and only if $L[\cdot]$ satisfies the principle of superposition, namely,

$$L[a_1x_1(n) + a_2x_2(n)] = a_1L[x_1(n)] + a_2L[x_2(n)], \forall a_1, a_2, x_1(n), x_2(n) \tag{2.8}$$

Using (2.1) and (2.8), the output $y(n)$ of a linear system to an arbitrary input $x(n)$ is given by

$$y(n) = L[x(n)] = L\left[\sum_{n=-\infty}^{\infty} x(k)\,\delta(n-k)\right] = \sum_{n=-\infty}^{\infty} x(k)\,L[\delta(n-k)]$$

The response $L[\delta(n-k)]$ can be interpreted as the response of a linear system at time n due to a unit sample (a well-known sequence) at time k. It is called an *impulse response* and is denoted by $h(n,k)$. The output then is given by the *superposition summation*

$$y(n) = \sum_{n=-\infty}^{\infty} x(k)\,h(n,k) \tag{2.9}$$

The computation of (2.9) requires the *time-varying* impulse response $h(n,k)$, which in practice is not very convenient. Therefore time-invariant systems are widely used in DSP.

Linear time-invariant (LTI) system A linear system in which an input-output pair, $x(n)$ and $y(n)$, is invariant to a shift n in time is called a linear time-invariant system. For an LTI system the $L[\cdot]$ and the shifting operators are reversible as shown below.

$$x(n) \longrightarrow \boxed{L[\cdot]} \longrightarrow y(n) \longrightarrow \boxed{\text{Shift by } k} \longrightarrow y(n-k)$$

$$x(n) \longrightarrow \boxed{\text{Shift by } k} \longrightarrow x(n-k) \longrightarrow \boxed{L[\cdot]} \longrightarrow y(n-k)$$

We will denote an LTI system by the operator $LTI[\cdot]$. Let $x(n)$ and $y(n)$ be the input-output pair of an LTI system. Then the time-varying function $h(n,k)$ becomes a time-invariant function $h(n-k)$, and the output from (2.9) is given by

$$y(n) = LTI[x(n)] = \sum_{k=-\infty}^{\infty} x(k)h(n-k) \tag{2.10}$$

The impulse response of an LTI system is given by $h(n)$. The mathematical operation in (2.10) is called a *linear convolution sum* and is denoted by

$$y(n) \triangleq x(n) * h(n) \tag{2.11}$$

Hence an LTI system is completely characterized in the time domain by the impulse response $h(n)$ as shown below.

$$x(n) \longrightarrow \boxed{h(n)} \longrightarrow y(n) = x(n) * h(n)$$

We will explore several properties of the convolution in Problem 2.12.

Stability This is a very important concept in linear system theory. The primary reason for considering stability is to avoid building harmful systems or to avoid burnout or saturation in the system operation. A system is said to be *bounded-input bounded-output (BIBO) stable* if every bounded input produces a bounded output.

$$|x(n)| < \infty \Rightarrow |y(n)| < \infty, \forall x, y$$

An LTI system is BIBO stable if and only if its impulse response is *absolutely summable.*

$$\text{BIBO Stability} \Longleftrightarrow \sum_{-\infty}^{\infty} |h(n)| < \infty \tag{2.12}$$

Causality This important concept is necessary to make sure that systems can be built. A system is said to be causal if the output at index n_0 depends only on the input up to and including the index n_0; that is, the output does not depend on the future values of the input. An LTI system is causal if and only if the impulse response

$$h(n) = 0, \quad n < 0 \tag{2.13}$$

Such a sequence is termed a *causal sequence.* In signal processing, unless otherwise stated, we will always assume that the system is causal.

CONVOLUTION

We introduced the convolution operation (2.11) to describe the response of an LTI system. In DSP it is an important operation and has many other uses that we will see throughout this book. Convolution can be evaluated in many different ways. If the sequences are mathematical functions (of finite or infinite duration), then we can analytically evaluate (2.11) for all n to obtain a functional form of $y(n)$.

□ **EXAMPLE 2.5** Let the rectangular pulse $x(n) = u(n) - u(n-10)$ of Example 2.4 be an input to an LTI system with impulse response

$$h(n) = (0.9)^n u(n)$$

Determine the output $y(n)$.

Solution The input $x(n)$ and the impulse response $h(n)$ are shown in Figure 2.5. From (2.11)

$$y(n) = \sum_{k=0}^{9} (1)\,(0.9)^{(n-k)}\, u(n-k) = (0.9)^n \sum_{k=0}^{9} (0.9)^{-k}\, u(n-k) \tag{2.14}$$

The sum in 2.14 is almost a geometric series sum except that the term $u(n-k)$ takes different values depending on n and k. There are three different conditions under which $u(n-k)$ can be evaluated.

CASE i $n < 0$: Then $u(n-k) = 0, \quad 0 \le k \le 9$. Hence from (2.14)

$$y(n) = 0 \tag{2.15}$$

In this case the nonzero values of $x(n)$ and $h(n)$ *do not overlap.*

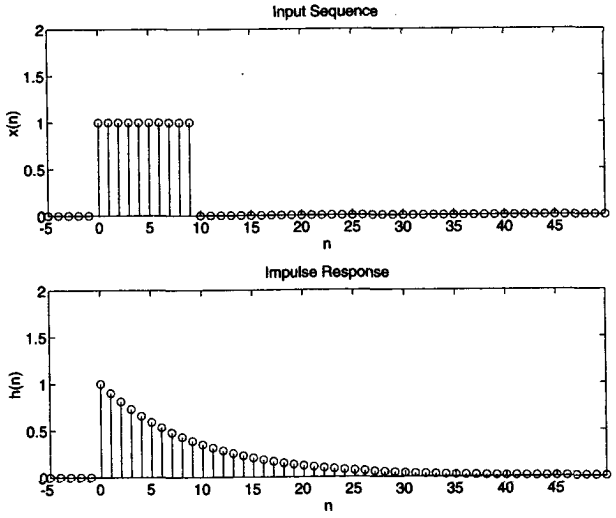

FIGURE 2.5 *The input sequence and the impulse response in Example 2.5*

CASE ii $0 \le n < 9$: Then $u(n-k) = 1, \quad 0 \le k \le n$. Hence from (2.14)

$$y(n) = (0.9)^n \sum_{k=0}^{n} (0.9)^{-k} = (0.9)^n \sum_{k=0}^{n} \left[(0.9)^{-1}\right]^k \tag{2.16}$$

$$= (0.9)^n \frac{1-(0.9)^{-(n+1)}}{1-(0.9)^{-1}} = 10\left[1-(0.9)^{n+1}\right], \quad 0 \le n < 9$$

In this case the impulse response $h(n)$ *partially overlaps* the input $x(n)$.

CASE iii $n \ge 9$: Then $u(n-k) = 1$, $0 \le k \le 9$ and from (2.14)

$$y(n) = (0.9)^n \sum_{k=0}^{9} (0.9)^{-k} \tag{2.17}$$

$$= (0.9)^n \frac{1-(0.9)^{-10}}{1-(0.9)^{-1}} = 10\,(0.9)^{n-9}\left[1-(0.9)^{10}\right], \quad n \ge 9$$

In this last case $h(n)$ *completely overlaps* $x(n)$.

The complete response is given by (2.15), (2.16), and (2.17). It is shown in Figure 2.6 which depicts the distortion of the input pulse. □

The above example can also be done using a method called graphical convolution, in which (2.11) is given a graphical interpretation. In this method $h(n-k)$ is interpreted as a *folded-and-shifted* version of $h(k)$. The output $y(n)$ is obtained as a sample sum under the overlap of $x(k)$ and $h(n-k)$. We use an example to illustrate this.

EXAMPLE 2.6 Given the following two sequences

$$x(n) = \left[3, \underset{\uparrow}{11}, 7, 0, -1, 4, 2\right], \quad -3 \le n \le 3; \qquad h(n) = \left[2, \underset{\uparrow}{3}, 0, -5, 2, 1\right], \quad -1 \le n \le 4$$

determine the convolution $y(n) = x(n) * h(n)$.

Solution In Figure 2.7 we show four plots. The top-left plot shows $x(k)$ and $h(k)$, the original sequences. The top-right plot shows $x(k)$ and $h(-k)$, the folded version

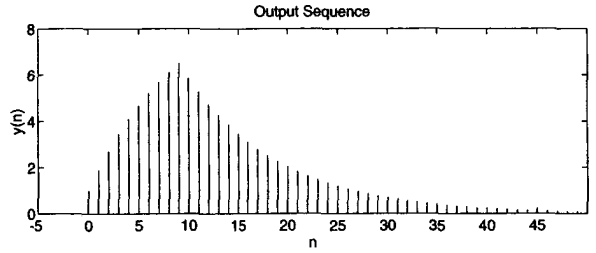

FIGURE 2.6 *The output sequence in Example 2.5*

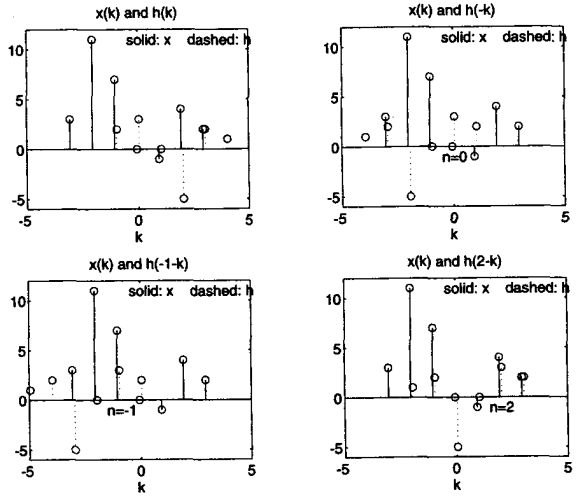

FIGURE 2.7 *Graphical convolution in Example 2.6*

of $h(k)$. The bottom-left plot shows $x(k)$ and $h(-1-k)$, the folded-and-shifted-by--1 version of $h(k)$. Then

$$\sum_k x(k)h(-1-k) = 3\times(-5) + 11\times 0 + 7\times 3 + 0\times 2 = 6 = y(-1)$$

The bottom-right plot shows $x(k)$ and $h(2-k)$, the folded-and-shifted-by-2 version of $h(k)$, which gives

$$\sum_k x(k)h(2-k) = 11\times 1 + 7\times 2 + 0\times(-5) + (-1)\times 0 + 4\times 3 + 2\times 2 = 41 = y(2)$$

Thus we have obtained two values of $y(n)$. Similar graphical calculations can be done for other remaining values of $y(n)$. Note that the beginning point (first nonzero sample) of $y(n)$ is given by $n = -3 + (-1) = -4$, while the end point (the last nonzero sample) is given by $n = 3 + 4 = 7$. The complete output is given by

$$y(n) = \left\{6, 31, 47, 6, \underset{\uparrow}{-51}, -5, 41, 18, -22, -3, 8, 2\right\}$$

Students are strongly encouraged to verify the above result. Note that the resulting sequence $y(n)$ has a *longer length* than both the $x(n)$ and $h(n)$ sequences. □

MATLAB IMPLEMENTATION If arbitrary sequences are of infinite duration, then MATLAB cannot be used *directly* to compute the convolution. MATLAB does provide a built-in function called conv that computes the convolution between two finite-

duration sequences. The conv function assumes that the two sequences begin at $n = 0$ and is invoked by

```
>> y = conv(x,h);
```

For example, to do the convolution in Example 2.5, we could use

```
>> x = [3, 11, 7, 0, -1, 4, 2];
>> h = [2, 3, 0, -5, 2, 1];
>> y = conv(x,h)
y =
  6    31    47    6    -51    -5    41    18    -22    -3    8    2
```

to obtain the correct $y(n)$ values. However, the conv function neither provides nor accepts any timing information if the sequences have arbitrary support. What is needed is a beginning point and an end point of $y(n)$. Given finite duration $x(n)$ and $h(n)$, it is easy to determine these points. Let

$$\{x(n);\ n_{xb} \le n \le n_{xe}\} \quad \text{and} \quad \{h(n);\ n_{hb} \le n \le n_{he}\}$$

be two finite-duration sequences. Then referring to Example 2.6 we observe that the beginning and end points of $y(n)$ are

$$n_{yb} = n_{xb} + n_{hb} \quad \text{and} \quad n_{ye} = n_{xe} + n_{he}$$

respectively. A simple extension of the conv function, called conv_m, which performs the convolution of arbitrary support sequences can now be designed.

```
function [y,ny] = conv_m(x,nx,h,nh)
% Modified convolution routine for signal processing
% -------------------------------------------------
% [y,ny] = conv_m(x,nx,h,nh)
%  [y,ny] = convolution result
%  [x,nx] = first signal
%  [h,nh] = second signal
%
nyb = nx(1)+nh(1); nye = nx(length(x)) + nh(length(h));
ny = [nyb:nye];
y = conv(x,h);
```

□ **EXAMPLE 2.7** Perform the convolution in Example 2.6 using the conv_m function.

Solution

MATLAB Script

```
>> x = [3, 11, 7, 0, -1, 4, 2]; nx = [-3:3];
>> h = [2, 3, 0, -5, 2, 1]; ny = [-1:4];
```



```
>> [y,ny] = conv_m(x,nx,h,nh)
y =
  6    31    47    6    -51    -5    41    18    -22    -3    8    2
ny =
 -4    -3    -2    -1    0    1    2    3    4    5    6    7
```

Hence

$$y(n) = \left\{6, 31, 47, 6, \underset{\uparrow}{-51}, -5, 41, 18, -22, -3, 8, 2\right\}$$

as in Example 2.6. □

An alternate method in MATLAB can be used to perform the convolution. This method uses a matrix-vector multiplication approach, which we will explore in Problem 2.13.

SEQUENCE CORRELATIONS REVISITED

If we compare the convolution operation (2.11) with that of the crosscorrelation of two sequences defined in (2.6), we observe a close resemblance. The crosscorrelation $r_{yx}(\ell)$ can be put in the form

$$r_{yx}(\ell) = y(\ell) * x(-\ell)$$

with the autocorrelation $r_{xx}(\ell)$ in the form

$$r_{xx}(\ell) = x(\ell) * x(-\ell)$$

Therefore these correlations can be computed using the conv function if sequences are of finite duration.

□ **EXAMPLE 2.8** In this example we will demonstrate one application of the crosscorrelation sequence. Let

$$x(n) = \left[3, 11, 7, \underset{\uparrow}{0}, -1, 4, 2\right]$$

be a prototype sequence, and let $y(n)$ be its noise-corrupted-and-shifted version

$$y(n) = x(n-2) + w(n)$$

where $w(n)$ is Gaussian sequence with mean 0 and variance 1. Compute the crosscorrelation between $y(n)$ and $x(n)$.

Solution From the construction of $y(n)$ it follows that $y(n)$ is "similar" to $x(n-2)$ and hence their crosscorrelation would show the strongest similarity at $\ell = 2$. To test this out using MATLAB, let us compute the crosscorrelation using two different noise sequences.

```
% noise sequence 1
>> x = [3, 11, 7, 0, -1, 4, 2]; nx=[-3:3]; % given signal x(n)
>> [y,ny] = sigshift(x,nx,2);              % obtain x(n-2)
>> w = randn(1,length(y)); nw = ny;        % generate w(n)
>> [y,ny] = sigadd(y,ny,w,nw);             % obtain y(n) = x(n-2) + w(n)
>> [x,nx] = sigfold(x,nx);                 % obtain x(-n)
>> [rxy,nrxy] = conv_m(y,ny,x,nx);         % crosscorrelation
>> subplot(1,1,1), subplot(2,1,1);stem(nrxy,rxy)
>> axis([-5,10,-50,250]);xlabel('lag variable l')
>> ylabel('rxy');title('Crosscorrelation: noise sequence 1')
%
% noise sequence 2
>> x = [3, 11, 7, 0, -1, 4, 2]; nx=[-3:3]; % given signal x(n)
>> [y,ny] = sigshift(x,nx,2);              % obtain x(n-2)
>> w = randn(1,length(y)); nw = ny;        % generate w(n)
>> [y,ny] = sigadd(y,ny,w,nw);             % obtain y(n) = x(n-2) + w(n)
>> [x,nx] = sigfold(x,nx);                 % obtain x(-n)
>> [rxy,nrxy] = conv_m(y,ny,x,nx);         % crosscorrelation
>> subplot(2,1,2);stem(nrxy,rxy)
>> axis([-5,10,-50,250]);xlabel('lag variable l')
>> ylabel('rxy');title('Crosscorrelation: noise sequence 2')
```

From Figure 2.8 we observe that the crosscorrelation indeed peaks at $\ell = 2$, which implies that $y(n)$ is similar to $x(n)$ shifted by 2. This approach can be

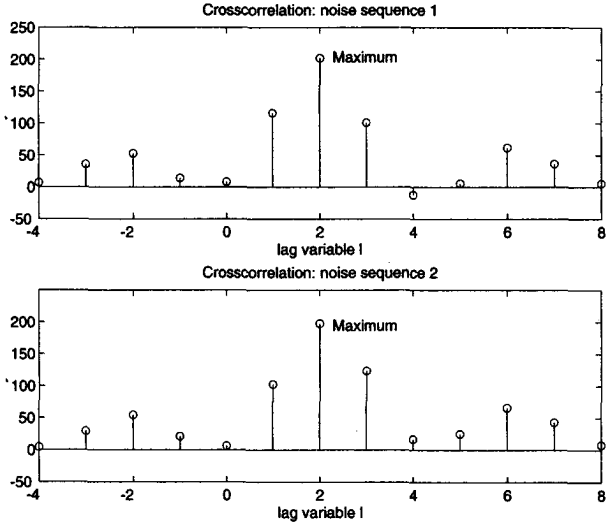

FIGURE 2.8 *Crosscorrelation sequence with two different noise realizations*

used in applications like radar signal processing in identifying and localizing targets. □

It should be noted that the signal-processing toolbox in MATLAB also provides a function called `xcorr` for sequence correlation computations. In its simplest form

```
>> xcorr(x,y)
```

computes the crosscorrelation between vectors `x` and `y`, while

```
>> xcorr(x)
```

computes the autocorrelation of vector `x`. This function is not available in the Student Edition of MATLAB. It generates results that are identical to the one obtained from the proper use of the `conv_m` function. However, the `xcorr` function cannot provide the timing (or lag) information (as done by the `conv_m` function), which then must be obtained by some other means. Therefore we will emphasize the use of the `conv_m` function.

DIFFERENCE EQUATIONS

An LTI discrete system can also be described by a linear constant coefficient difference equation of the form

$$\sum_{k=0}^{N} a_k y(n-k) = \sum_{m=0}^{M} b_m x(n-m), \quad \forall n \tag{2.18}$$

If $a_N \neq 0$, then the difference equation is of order N. This equation describes a recursive approach for computing the current output, given the input values and previously computed output values. In practice this equation is computed forward in time, from $n = -\infty$ to $n = \infty$. Therefore another form of this equation is

$$y(n) = \sum_{m=0}^{M} b_m x(n-m) - \sum_{k=1}^{N} a_k y(n-k) \tag{2.19}$$

A solution to this equation can be obtained in the form

$$y(n) = y_H(n) + y_P(n)$$

The *homogeneous part* of the solution, $y_H(n)$, is given by

$$y_H(n) = \sum_{k=1}^{N} c_k z_k^n$$

where $z_k, k = 1, \ldots, N$ are N roots (also called *natural frequencies*) of the characteristic equation

$$\sum_{0}^{N} a_k z^k = 0$$

This characteristic equation is important in determining the stability of systems. If the roots z_k satisfy the condition

$$|z_k| < 1, \; k = 1, \ldots, N \tag{2.20}$$

then a causal system described by (2.19) is stable. The *particular part* of the solution, $y_P(n)$, is determined from the right-hand side of (2.18). In Chapter 4 we will discuss the analytical approach of solving difference equations using the z-transform.

MATLAB IMPLEMENTATION

A routine called `filter` is available to solve difference equations numerically, given the input and the difference equation coefficients. In its simplest form this routine is invoked by

```
y = filter(b,a,x)
```

where

```
b = [b0, b1, ..., bM]; a = [a0, a1, ..., aN];
```

are the coefficient arrays from the equation given in (2.18), and `x` is the input sequence array. The output `y` has the same length as input `x`. One must ensure that the coefficient `a0` not be zero. We illustrate the use of this routine in the following example.

□ **EXAMPLE 2.9** Given the following difference equation

$$y(n) - y(n-1) + 0.9y(n-2) = x(n); \quad \forall n$$

a. Calculate and plot the impulse response $h(n)$ at $n = -20, \ldots, 100$.
b. Calculate and plot the unit step response $s(n)$ at $n = -20, \ldots, 100$.
c. Is the system specified by $h(n)$ stable?

Solution From the given difference equation the coefficient arrays are

```
b = [1]; a=[1, -1, 0.9];
```


a. MATLAB Script

```
>> b = [1]; a = [1, -1, 0.9];
>> x = impseq(0,-20,120); n = [-20:120];
>> h = filter(b,a,x);
>> subplot(2,1,1); stem(n,h);
>> title('Impulse Response'); xlabel('n'); ylabel('h(n)')
```

The plot of the impulse response is shown in Figure 2.9.

b. MATLAB Script

```
>> x = stepseq(0,-20,120);
>> s = filter(b,a,x);
>> subplot(2,1,2); stem(n,s)
>> title('Step Response'); xlabel('n'); ylabel('s(n)')
```

The plot of the unit step response is shown in Figure 2.9.

c. To determine the stability of the system, we have to determine $h(n)$ for all n. Although we have not described a method to solve the difference equation, we can use the plot of the impulse response to observe that $h(n)$ is practically zero for $n > 120$. Hence the sum $\sum |h(n)|$ can be determined from MATLAB using

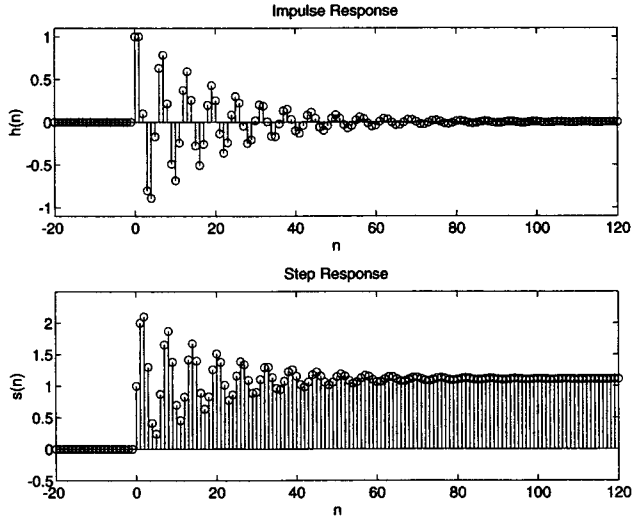

FIGURE 2.9 *Impulse response and step response plots in Example 2.9*

```
>> sum(abs(h))
ans = 14.8785
```

which implies that the system is stable. An alternate approach is to use the stability condition (2.20) using MATLAB's `roots` function.

```
>>z = roots(a);
>>magz = abs(z)
magz =  0.9487
        0.9487
```

Since the magnitudes of both roots are less than one, the system is stable. □

In the previous section we noted that if one or both sequences in the convolution are of infinite length, then the `conv` function cannot be used. If one of the sequences is of infinite length, then it is possible to use MATLAB for numerical evaluation of the convolution. This is done using the `filter` function as we will see in the following example.

□ **EXAMPLE 2.10** Let us consider the convolution given in Example 2.5. The input sequence is of finite duration

$$x(n) = u(n) - u(n-10)$$

while the impulse response is of infinite duration

$$h(n) = (0.9)^n u(n)$$

Determine $y(n) = x(n) * h(n)$.

Solution If the LTI system, given by the impulse response $h(n)$, can be described by a difference equation, then $y(n)$ can be obtained from the `filter` function. From the $h(n)$ expression

$$(0.9)\,h(n-1) = (0.9)\,(0.9)^{n-1} u(n-1) = (0.9)^n u(n-1)$$

or

$$\begin{aligned} h(n) - (0.9)\,h(n-1) &= (0.9)^n u(n) - (0.9)^n u(n-1) \\ &= (0.9)^n\,[u(n) - u(n-1)] = (0.9)^n\,\delta(n) \\ &= \delta(n) \end{aligned}$$

The last step follows from the fact that $\delta(n)$ is nonzero only at $n = 0$. By definition $h(n)$ is the output of an LTI system when the input is $\delta(n)$. Hence substituting $x(n)$ for $\delta(n)$ and $y(n)$ for $h(n)$, the difference equation is

$$y(n) - 0.9y(n-1) = x(n)$$

Now MATLAB's `filter` function can be used to compute the convolution indirectly.

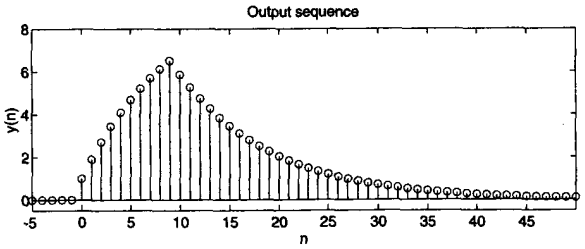

FIGURE 2.10 *Output sequence in Example 2.10*

```
>> b = [1]; a = [1,-0.9];
>> n = -5:50; x = stepseq(0,-5,50) - stepseq(10,-5,50);
>> y = filter(b,a,x);
>> subplot(1,1,1);
>> subplot(2,1,2); stem(n,y); title('Output sequence')
>> xlabel('n'); ylabel('y(n)'); axis([-5,50,-0.5,8])
```

The plot of the output is shown in Figure 2.10, which is exactly the same as that in Figure 2.6. □

In Example 2.10 the impulse response was a one-sided exponential sequence for which we could determine a difference equation representation. This means that not all infinite-length impulse responses can be converted into difference equations. The above analysis, however, can be extended to a linear combination of one-sided exponential sequences, which results in higher-order difference equations. We will discuss this topic of conversion from one representation to another one in Chapter 4.

ZERO-INPUT AND ZERO-STATE RESPONSES

In digital signal processing the difference equation is generally solved forward in time from $n = 0$. Therefore initial conditions on $x(n)$ and $y(n)$ are necessary to determine the output for $n \geq 0$. The difference equation is then given by

$$y(n) = \sum_{m=0}^{M} b_m x(n-m) - \sum_{k=1}^{N} a_k y(n-k);\ n \geq 0 \qquad \textbf{(2.21)}$$

subject to the initial conditions:

$$\{y(n);\ -N \leq n \leq -1\} \quad \text{and} \quad \{x(n);\ -M \leq n \leq -1\}$$

A solution to (2.21) can be obtained in the form

$$y(n) = y_{ZI}(n) + y_{ZS}(n)$$