CHAPTER 2

Discrete-time Signals and Systems

We begin with the concepts of signals and systems in discrete time. A number of important types of signals and their operations are introduced. Linear and shift-invariant systems are discussed mostly because they are easier to analyze and implement. The convolution and the difference equation representations are given special attention because of their importance in digital signal processing and in MATLAB. The emphasis in this chapter is on the representations and implementation of signals and systems using MATLAB.

2.1 DISCRETE-TIME SIGNALS

Signals are broadly classified into analog and discrete signals. An analog signal will be denoted by $x_a(t)$, in which the variable t can represent any physical quantity, but we will assume that it represents time in seconds. A discrete signal will be denoted by $x(n)$, in which the variable n is integer-valued and represents discrete instances in time. Therefore it is also called a discrete-time signal, which is a *number sequence* and will be denoted by one of the following notations:

$$x(n) = \{x(n)\} = \{\ldots, x(-1), \underset{\uparrow}{x(0)}, x(1), \ldots\}$$

where the *up-arrow* indicates the sample at $n = 0$.

In MATLAB we can represent a *finite-duration* sequence by a *row vector* of appropriate values. However, such a vector does not have any information about sample position n. Therefore a correct representation 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.

2.1.1 TYPES OF SEQUENCES

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

1. **Unit sample sequence**:

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

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**:

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

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 produces an attenuation (if <0) or amplification (if >0) 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**:

$$x(n) = A\cos(\omega_0 n + \theta_0), \forall n$$

where A is an amplitude and θ_0 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. 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 this 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 $\{x(n), \quad 0 \leq n \leq N-1\}$, 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.

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