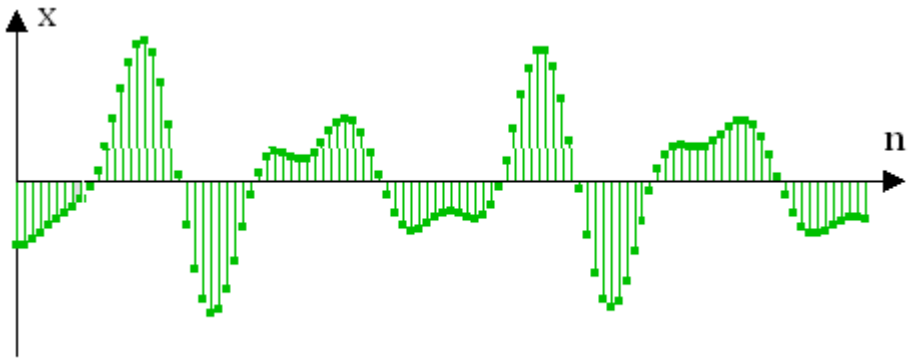

Digital Signal Processing

Student's Guide

by Tim Sharii, Ph.D.

Donetsk National University 2013

CONTENTS

PREFACE

This guide is intended for graduate students of the Department of Computer Technologies at DonNU. It contains both theory and helpful practical examples of Digital Signal Processing (DSP) techniques. DSP is very important and constantly developing part of modern information technologies and engineering. Many industrial companies are currently engaged in DSP research and development, since IT markets are tending to open up to applications for sound, image and video processing. Therefore, DSP today is a highly desirable skill needed by scientists and engineers. This guide covers the most fundamental topics related to DSP.

The guide includes six lab assignments containing theoretical and practical exercises. Students are required to prepare lab reports (in English).

Code examples are given as a MATLAB code, since MATLAB is a highly convenient tool for visualizing and programming signals. However, students may use any other programming language such as C++, C# or Java, in their labs.

➢ Goals of this Guide

The main purpose of this guide is to provide information about how to perform the most necessary operations on digital signals (such as filtering, spectral analysis, resizing, etc.). The guide does not focus primarily on the mathematical aspects of DSP. For a closer look at DSP you can refer to the list of recommended textbooks in the Bibliography section. Many illustrations and figures are taken from Alan Oppenheim's textbook [1] and "The Scientist and Engineer's Guide to Digital Signal Processing", copyright ©1997-1998 by Steven W. Smith. For more information visit the book's website at: www.DSPguide.com.

➢ Notations used in this Guide

Code examples have the title "LET'S CODE!". Exercises marked as (CODE) require coding. Other exercises should be done in writing.

➢ Structure of this Guide

Chapter 1 provides survey information about DSP.
Chapter 2 introduces two main concepts in DSP: the signal and the system.
Chapter 3 focuses on analysis of signals in frequency domain.
Chapter 4 reviews the task of signal filtering.
Chapter 5 discusses the issues regarding filter design and analysis as well as the Z-domain representation of digital filters.
Chapter 6 is devoted to the task of signal resampling.
Chapter 7 focuses on speech signal processing as one of the most needed and challenging applications of DSP.
Appendix A contains the list of useful MATLAB DSP functions.

CHAPTER 1 WHAT IS DSP?

Digital Signal Processing is a part of Computer Science dealing with various kinds of signals such as audio, video, speech, image, communication, geophysical, sonar, radar, medical, musical signals, etc. DSP involves the algorithms and the techniques for manipulating signals represented in a digital form, as well as the rich mathematical background for these techniques. Nowadays, with the advent of multimedia technologies, DSP has become even more significant engineering area in IT industry.

The history of DSP began in the late 1950s. Computers were expensive at that time, and DSP was limited to only a few critical applications: radar & sonar; oil exploration; space exploration and medical imaging. Revolutionary changes were made due to a combination of two key factors: 1) theoretical factor – the Fast Fourier Transform algorithm, proposed in 1965 by J.W. Cooley and J. Tukey; 2) practical factor – the development of personal computers. In 1980s DSP exploded with new applications, including commercial applications. It found the use in such products as: multimedia devices, CD players, mobile telephones, and many others.

Currently, DSP is almost everywhere (Fig.1.1).

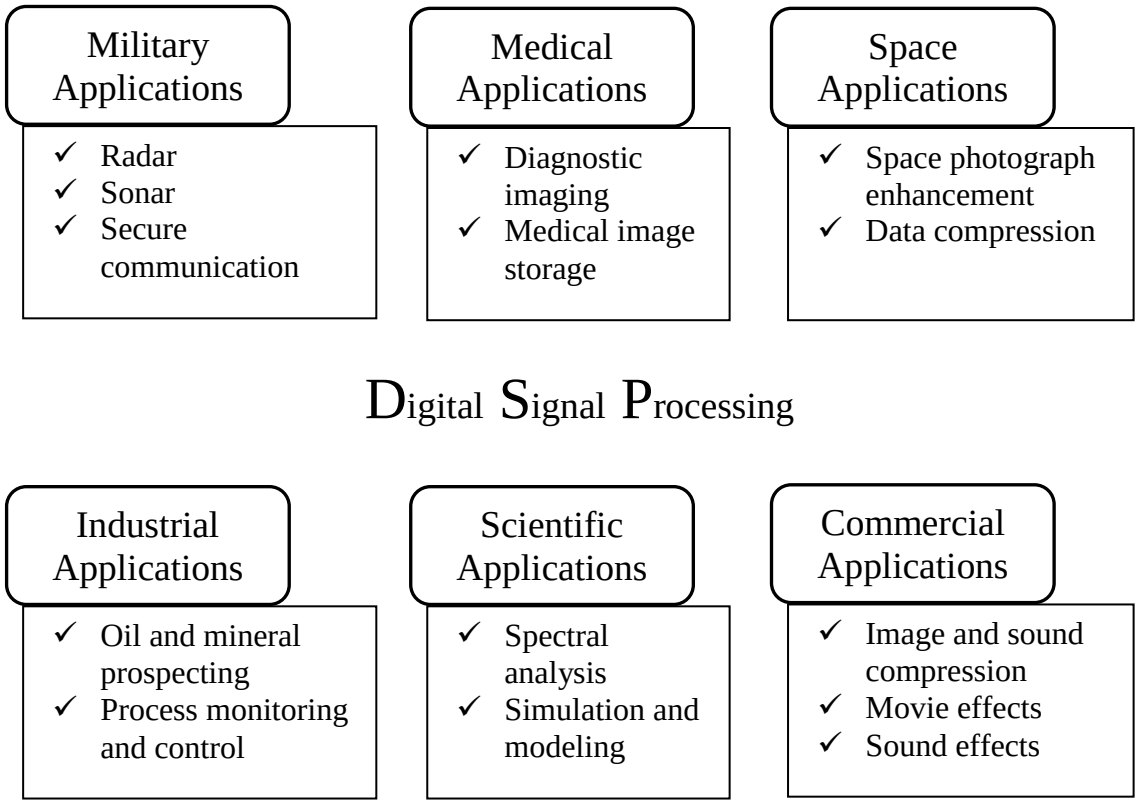

FIGURE 1.1. Applications of DSP

DSP is highly interdisciplinary engineering area. It borders and overlaps with such disciplines as analog and digital electronics, numerical analysis, control theory, information theory, decision theory, probability and statistics.

Besides math, DSP deals with a specific hardware designed especially for its purposes. Such devices are referred to as DSP microprocessors (synonyms: Digital Signal Processor, RISC (Reduced Instruction Set Computer)). At this moment there are four generations of DSP microprocessors. Among the most popular devices are the Texas Instruments TMS3206000, the Freescale MSC81xx, and the Analog Devices ADSP-21xx. DSP microprocessors are used in two ways: as slave modules under the control of conventional computer, or as an embedded processor, such as in a cellular telephone. Some models only handle fixed point numbers, while others can work with floating point. The internal architecture used to obtain the increased speed includes [3]:

- ✓ lots of very fast cache memory contained within the chip;
- ✓ separate buses for the program and data, allowing the two to be accessed simultaneously (called a Harvard Architecture);
- ✓ fast hardware for math calculations contained directly in the microprocessor;
- ✓ a pipeline design.

DSP programming requires good coding skills and experience. Real-time applications are usually written in C/C++ with the use of SIMD instructions (MMX/SSE2/SSE3) and specific technologies (e.g. CUDA). Regardless of programming language and technology used, engineer should know and be able to apply several optimization tips for increasing an execution speed:

- Using Integers instead of Floating Point variables whenever possible.
- Replacing transcendental functions (sin, cos, log) with series of additions, subtractions and multiplications. The well-known technique is based on McLaurin power series.
- Using Look-Up Tables (LUT) with precalculated values of functions (for instance, trigonometric tables).

In conclusion, Digital Signal Processing is complicated and constantly developing engineering area. Basically, all DSP techniques are designed to solve two key scientific and practical tasks (they will be discussed in the following chapters):

1) *Signal Modification* (processing one signal to obtain another signal). This kind of problem occurs when there's a need to enhance a signal or to separate signals;
2) *Signal Interpretation*. In this case the objective is to obtain the certain characterization of signal. For example, the objective of a speaker recognition system is to extract speaker-dependent features from speech signal and interpret them correctly.

CHAPTER 2 SIGNALS AND SYSTEMS

2.1. Classification of Signals

SIGNAL
is a function that conveys information about the behavior or attributes of some phenomenon.

From a physical point of view, signal is a physical quantity which varies with respect to time or space and conveys information from source to destination. From a mathematical point of view, signal is a custom function (Fig.2.1). For example, a sound wave can be expressed as the functional dependency *x(t)*, where *x* is the value of sound pressure changing with time *t*. Similarly, an image can be expressed as *f(x, y)*, where *f* is the brightness of a pixel with coordinates *x* and *y*.

There are several signal classification criteria:

- *The number of function arguments (number of independent variables)*:
  - **1-dimensional** signal (e.g. sound);
  - **2-dimensional** signal (e.g. image);
  - **3-dimensional** signal (e.g. video signal);
  - **multi-dimensional** signal.

  This guide focuses primarily on 1D-signals.
- *The limitation*:
  - **finite** signal. Its argument is finite. In other words, a finite signal has the starting point and the ending point;
  - **infinite** signal. Its argument is infinite;
  - **bounded** signal. At any time its value is less than some finite value (for instance, sinusoidal signal is bounded above by 1 and bounded below by -1);
  - **unbounded** signal (e.g. exponential signal).
- *The repetition*:
  - **periodic** signal. It consists of constantly repeating sequences;
  - **aperiodic** signal. It's not periodic.
- *The continuity*:
  - **continuous signal**. Its arguments and values are both continuous. Continuous signals are also often referred to as analog signals;
  - **discrete-time (sampled)** signal. Its *argument* is a discrete value;
  - **discrete** signal. Its *values* are discrete;
  - **digital (digitized)** signal. It is both sampled and discrete. Obviously, computers can store and process only digital signals.
- *The certainty of description*:
  - **deterministic** signal. It can be uniquely determined by a well-defined process such as mathematical expression, algorithm, or look-up table;

- **random** signal. It can be expressed as a random variable; it is characterized statistically (e.g. by probability density).

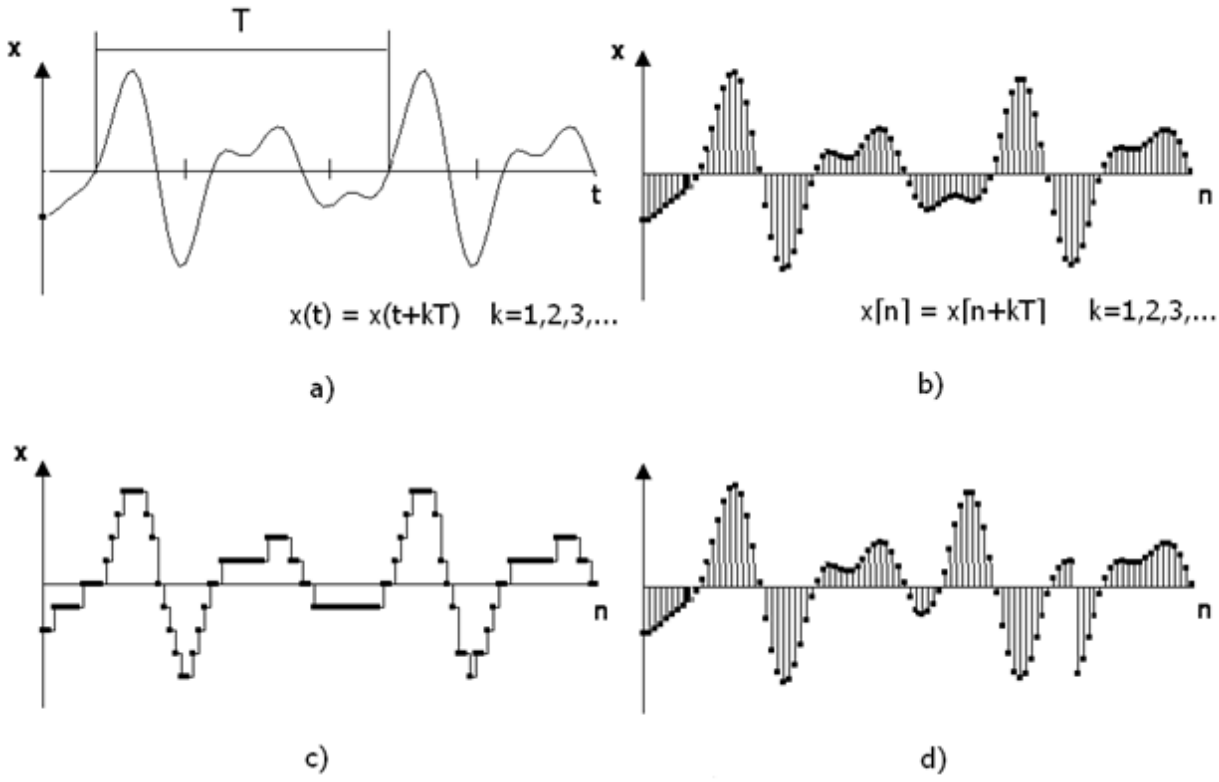

FIGURE 2.1. Signal types: a) continuous periodic; b) discrete-time periodic; c) digital periodic; d) discrete-time aperiodic

Let's consider a *sinusoidal* one-dimensional signal (*sine wave*, or *tone*):

$$x(t) = A\sin(\omega t + \varphi) = A\sin(2\pi f t + \varphi), \qquad (2.1)$$

where A is the *amplitude* of the signal, f is the *frequency* in cycles per second (measured in Hz), $\omega = 2\pi f$ is the *angular frequency* in radians per second, φ is the *phase* (measured in radians). Since $\cos(t) = \sin(t + \pi/2)$, the signal $y(t) = A\cos(\omega t + \varphi)$ is also a sinusoidal signal.

The signal *x(t)* (2.1) is continuous, bounded, infinite, deterministic (it is described by a strict formula) and periodic (the period is equal to 2π).

Sinusoidal signals play a very important role in DSP for two reasons:

1) many analog signals are inherently created from superimposed sinusoids (sound waves, for instance);
2) Fourier Transform (one of the basic transforms in DSP) decomposes signals into sine waves.

Our next step is to consider *discrete-time* sinusoidal signals. But before we proceed, we need to discuss what happens with analog signals in DSP systems.

2.2. Sampling and Quantization

Analog signal sampling and quantization is very important and complicated DSP issue. In this chapter we will consider only the very basics of these processes for a better understanding of fundamental DSP techniques.

As mentioned above, computers can process only digital signals. But most signals come from the real "analog" world and after computer processing should become analog again. Therefore, digital signal processing scheme must include two additional stages (Fig.2.2):

1) *sampling* of continuous signals;

2) *reconstruction* of continuous signals.

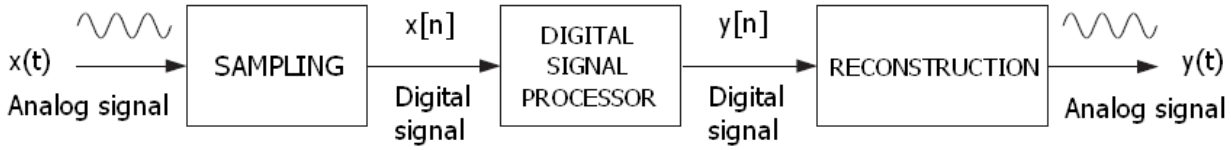

FIGURE 2.2. The structure of DSP system

Sampling is done in two steps:

1) *Sample-and-Hold (S/H)*

2) *Quantization, Analog-to-Digital Converter (ADC)*.

Basically, the first step is making a continuous signal discrete-time, and the second step is making the discrete-time signal digital (Fig.2.3).

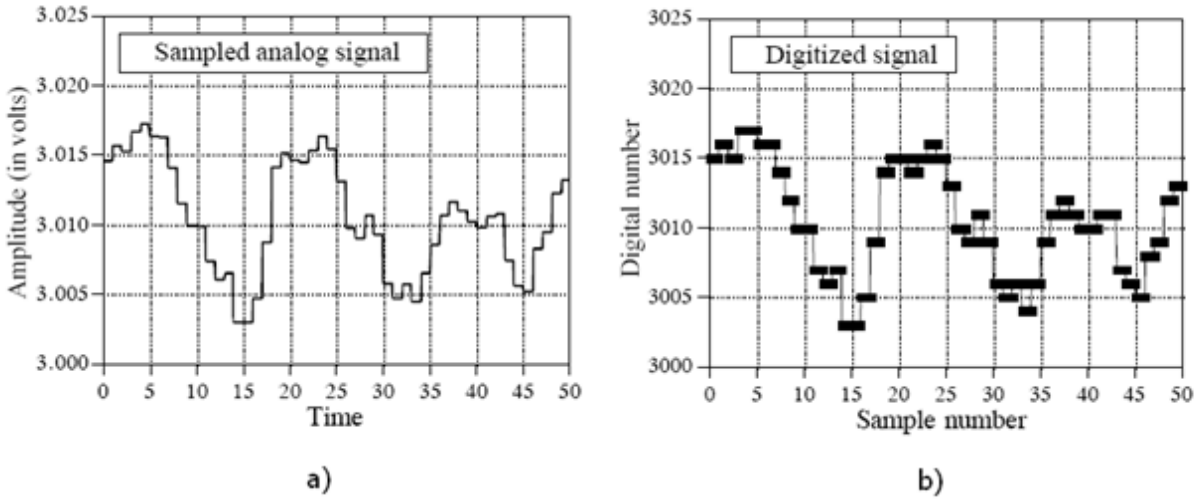

FIGURE 2.3. Analog signal sampling: a) Sample-and-Hold; b) Quantization

Let's consider the *S/H* process. At this stage the signal $x_a(t)$ (voltage level) is sampled at uniformly spaced instants of time. The sampling process converts an analog signal into a discrete-time signal $x(nT_s)$ with continuous amplitude value:

$$x[n] = x_a(nT_s) \tag{2.2}$$

Notice, it is a common practice to use parentheses to denote continuous signals (x(t)) and brackets to denote discrete-time signals (x[n]).

The time interval T_s between adjacent samples is known as the *sampling period*. The *sampling rate* is a reciprocal of sampling period and represents the number of samples per second:

$$f_s = \frac{1}{T_s} \tag{2.3}$$

However, an actual (non-ideal) sampling circuit cannot capture the value of $x_a(t)$ at a single time instant. Instead, the output of a sampling circuit is the average value of $x_a(t)$ over a short interval ΔT, where $\Delta T << T_s$:

$$x[n] = \frac{1}{2\Delta T} \int_{nT_s - \Delta T}^{nT_s + \Delta T} x_a(t)dt \tag{2.4}$$

Example 2.1.
Given an analog audio signal 400ms long sampled at sampling rate 16000 Hz. How many samples does the signal contain?

Solution:
Since f_s=16000Hz, and sampling rate is, by definition, the number of samples per second (1000ms), the number of samples, contained in signal 0.4 seconds long (400ms), is: N = 16000 * 0.4 = 6400.

Usually, very high sampling rates aren't desirable, since we would want to use computer memory effectively. But what will happen if we reduce the sampling rate? It's intuitively clear that we can't reduce the sampling rate without losing information about analog signal. So the question is: "What minimum sampling rate is sufficient to capture all necessary information of continuous signal in its sampled version?". The sampling theorem answers this question.

THE NYQUIST-SHANNON-KOTELNIKOV THEOREM.
Continuous signal can be properly sampled if and only if it does not contain frequency components above 1/2 of the sampling rate.

"Proper sampling" means that it's possible to exactly reconstruct continuous signal from samples of its digital representation. Formally, sampling theorem's conditions can be written as:

$$1)\ |f| \le f_{max} \tag{2.5}$$

$$2)\ f_s > 2\, f_{max} \tag{2.6}$$

The minimum sampling rate f_s is called the *Nyquist rate*, whereas one-half of the sampling rate $f_{max}=f_s/2$ is called the *Nyquist frequency*.

Now let's get back to the sinusoidal signal defined in Eq.2.1. According to Eq.2.2, a discrete-time sinusoidal signal is expressed as:

$$x[n] = x_a(nT_s) = A\sin(\omega nT_s + \varphi) = A\sin(2\pi f nT_s + \varphi) \tag{2.7}$$

Eq.2.7 can be also expressed as:

$$x[n] = A\sin(\Omega n + \varphi) = A\sin(\pi F n + \varphi), \tag{2.8}$$

where $\Omega = \omega T_s = \dfrac{2\pi f}{f_s}$ is the *digital angular frequency* (in radians per sample), and $F = \dfrac{\Omega}{\pi} = \dfrac{f}{(f_s/2)}$ is the *normalized digital frequency*.

LET'S CODE!
Generate 300 samples of sinusoidal signal with frequency 500 Hz and sampling rate 4000 Hz. Plot first 25 samples of the signal.

```
f = 500;
fs = 4000;
ang_freq = 2*pi*f/fs;    % digital angular frequency
n = 1:300;               % sample numbers
x = sin(ang_freq*n);     % sinusoidal signal
plot(n(1:25),x(1:25));   % plot samples
```

What will happen if the sampling rate does not satisfy sampling theorem's conditions? Let's illustrate this situation by example. Suppose we have two sinusoidal signals with frequencies f_1=6000 Hz and f_2=10000 Hz that are sampled at f_s=16000 Hz (which is too low, since f_s must be not less than 20000 Hz, according to the sampling theorem):

$$x_1[n] = \cos\left(2\pi \cdot 6000 \cdot n \cdot \frac{1}{16000}\right) = \cos\left(\frac{6\pi}{8} n\right) \tag{2.9}$$

$$x_2[n] = \cos\left(2\pi \cdot 10000 \cdot n \cdot \frac{1}{16000}\right) = \cos\left(\frac{10\pi}{8} n\right) \qquad (2.10)$$

Using reduction formulae, we can rewrite Eq.2.10 as:

$$x_2[n] = \cos\left(\frac{10\pi}{8} n\right) = \cos\left(2\pi n - \frac{6\pi}{8} n\right) = \cos\left(\frac{6\pi}{8} n\right) = x_1[n] \qquad (2.11)$$

Thus, it's impossible to distinguish the two signals apart from their sampled versions. This phenomenon, known as *aliasing*, occurs whenever ($f_2 \pm f_1$) is a multiple of the sampling rate. Figure 2.4 illustrates the phenomenon of aliasing.

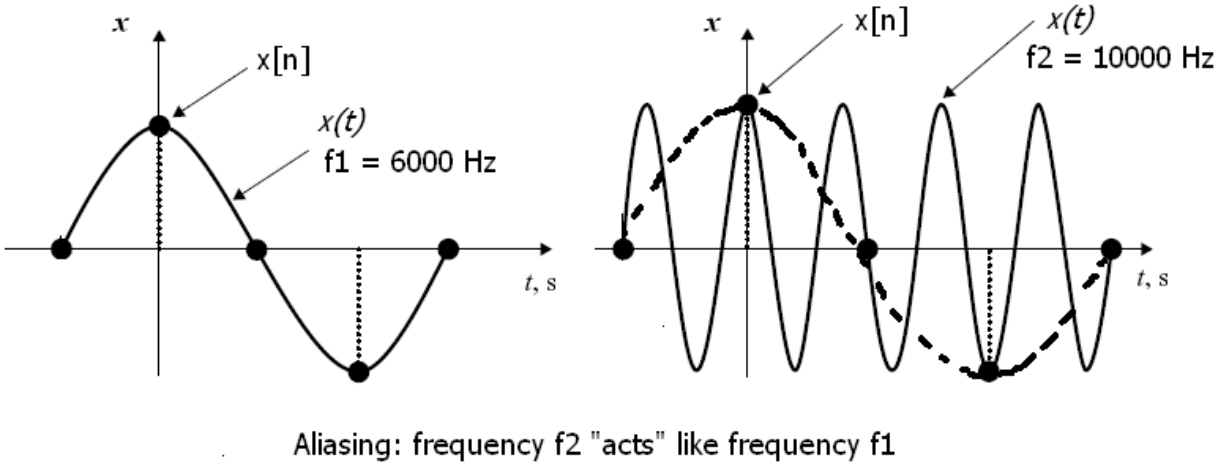

FIGURE 2.4. Demonstration of the aliasing phenomenon

The sampling theorem is important not only from a theoretical point of view. It also has a significant practical value. Most real-world analog signals are not *bandlimited* (i.e. they don't satisfy the condition 2.5). In order to sample a signal at a desired rate f_s and satisfy the conditions of the sampling theorem, the signal must be filtered by a low-pass analog filter, known as an *antialiasing filter*. The cutoff frequency of the filter, f_{max}, must be not greater than the Nyquist frequency. The output of the filter will then be bandlimited to maximum frequency f_{max} and may be sampled properly at the desired rate f_s.

Now, let's consider the *quantization* step of signal sampling (Fig.2.3b). At this step the amplitude of each discrete-time sample is quantized into one of the 2^B levels, where B is the number of bits that the ADC has to represent for each sample (*ADC rate*). For example, 8-bit ADC will provide only 2^8=256 possible values for quantized signal.

Any single sample in the digitized signal can have a maximum error of ±1/2 LSB (Least Significant Bit, the distance between adjacent quantization levels). In most cases, quantization results in nothing more than the addition of a specific amount of random noise to the signal. The additive noise is uniformly distributed

between ±1/2 LSB, has a mean of zero, and a standard deviation of 1/12 LSB (≈0.29 LSB). However, sometimes the quantization error stops being random (the output remains stuck on the same digital number for many samples in a row). In this case special techniques, such as *dithering*, are applied for improving signal digitization.

Thus, we have considered issues related to signal sampling and quantization. As we mentioned earlier, the final block of DSP systems is signal reconstruction. At this stage, the digital signal is passed through a *Digital-to-Analog Converter (DAC)* and a special low-pass filter set to the Nyquist frequency. This output filter is called a *reconstruction filter*. The detailed description of reconstruction process is beyond the scope of this guide.

2.3. Discrete-time Signals

In the previous section we considered only one class of signals – sinusoidal signals. There are several other important classes of signals we need to review.

Unit impulse (or *unit sample*, or *Kronecker's δ-function*) is defined as:

$$\delta[n] = \begin{cases} 1, & n = 0 \\ 0, & n \neq 0 \end{cases} \tag{2.12}$$

Unit step is given by:

$$u[n] = \begin{cases} 1, & n \geq 0 \\ 0, & n < 0 \end{cases} \tag{2.13}$$

Unit step is related to unit impulse by:

$$u[n] = \sum_{k=-\infty}^{n} \delta[k] \tag{2.14}$$

Exponential signals have the following general form:

$$x[n] = A\alpha^n \tag{2.15}$$

The unit impulse, the unit step, and two examples of exponential signals, are shown in Fig.2.5. Notice that all signals we have considered are deterministic. That is, each signal sample is strictly defined by a particular mathematical formula. However, in many cases, the real processes that generate signals are very complicated, and, therefore, the precise signal description is extremely difficult, undesirable or even impossible. In such cases, the random signals are analyzed.

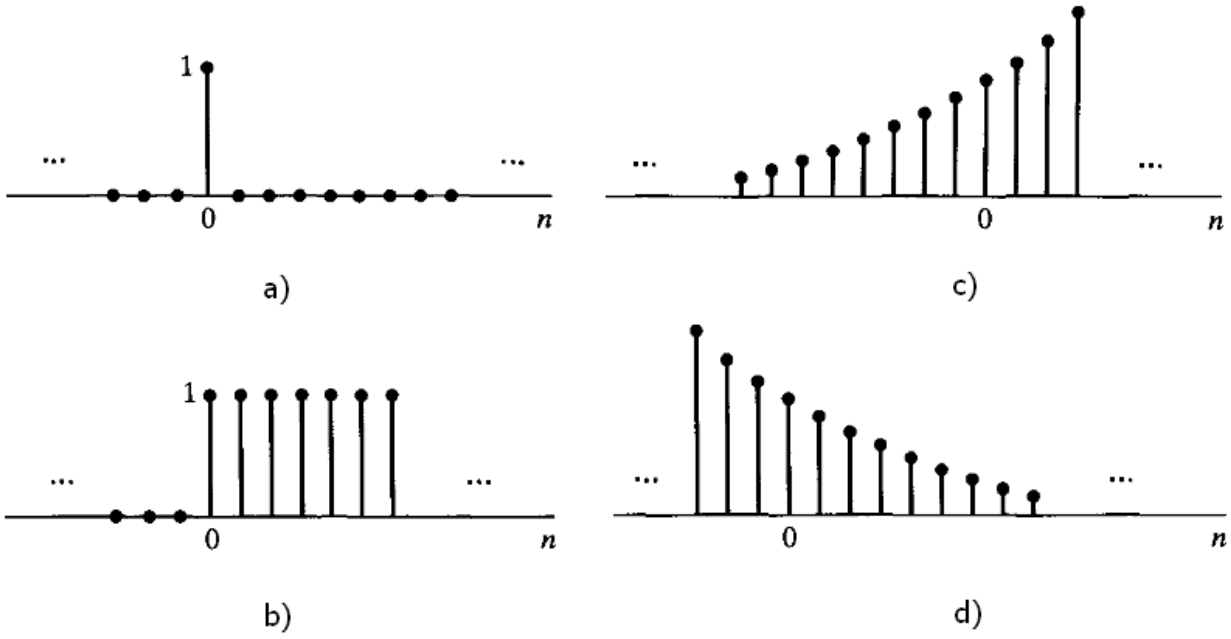

FIGURE 2.5. Basic discrete-time signals: a) unit impulse; b) unit step; c) exponential signal (α>1); d) exponential signal (0<α<1)

Random signals (such as speech, music, various kinds of noise, etc.) are modeled in terms of stochastic signals. In other words, each sample *x[n]* of a random signal is assumed to be an outcome of some underlying random variable $\mathbf{x_n}$. The entire signal is represented by a collection of such random variables, one for each sample time, $-\infty<n<+\infty$. We assume that a particular sequence of samples {x[n]} for $-\infty<n<+\infty$ has been generated by the random process that underlies the signal. Therefore, probability distributions must be specified, in order to describe the random process [1].

The noise is an obvious example of a random signal. For instance, if we need to describe Gaussian noise, we don't have to specify the particular values of samples. Instead, we generate a sample set with normal (Gaussian) distribution:

$$P(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu)^2/2\sigma^2}, \qquad (2.16)$$

where μ is the mean of distribution; σ is the standard deviation (square root of the variance) of distribution.

We should keep in mind that statistical parameters (like mean and variance) of the underlying process may differ from statistical parameters of the signal we actually observe and analyze.

The formula for *mean* is:

$$\mu = \frac{1}{N}\sum_{i=0}^{N-1} x_i \,, \tag{2.17}$$

and the formula for *variance* is:

$$\sigma^2 = \frac{1}{N}\sum_{i=0}^{N-1} (x_i - \mu)^2 \,, \tag{2.18}$$

where N is the number of analyzed samples; x_i is the i^{th} sample of signal.

Random signals can be either *stationary*, or *non-stationary*. *Stationarity in a strict sense (SSS)* means that *all* of the distribution functions of random process do not change over time. A weaker form of stationarity commonly used in DSP is known as *wide-sense stationarity (WSS)*. WSS random processes only require that *mean* and *covariance* do not vary with respect to time. Most of the real-world signals are non-stationary. The examples of stationary signals are stationary noises (e.g. white noise).

LET'S CODE!
Generate 100 samples of Gaussian noise and plot the result.

```
m = 3;                                  % mean
sigma = 10;                             % std. deviation
x=random('Normal',m,sigma,100,1); % Gaussian noise
plot(x);                                % plot samples
```

Signal-to-Noise ratio (SNR) is the measure that compares the level of a desired signal to the level of a background noise:

$$SNR = \frac{P_S}{P_N} = \frac{A_S^2}{A_N^2}, \tag{2.19}$$

where P_S (A_S) is the average power (amplitude) of desired signal; P_N (A_N) is the average power (amplitude) of background noise.

There is also an alternative definition of SNR (used mostly in image processing), according to which SNR is the ratio of mean to standard deviation of a signal:

$$SNR = \frac{\mu}{\sigma} \tag{2.20}$$

Finally, let's consider general *amplitude-time* parameters of signals. If the signal's independent variable is time or sample number, the signal is said to be represented in *time domain*. There are other domains, even more useful for signal analysis and processing, and they will be discussed in the following chapters. In time domain, we can evaluate the *energy* of signal and *zero-crossing rate*.

Energy of a signal segment is defined as:

$$E(k,N) = \sum_{i=k}^{k+N-1} |x_i|^2, \tag{2.21}$$

and zero-crossing rate of a signal segment is defined as:

$$ZCR(k,N) = \frac{1}{N} \sum_{i=k}^{k+N-1} \frac{|sign(x_{i+1}) - sign(x_i)|}{2}, \tag{2.22}$$

where k is the position of the first sample of signal segment; N is the number of samples in signal segment; x_i is the i^{th} sample of signal.

2.4. Discrete-time Systems

DISCRETE-TIME SYSTEM
is a transformation or operator that maps an input discrete-time signal into an output discrete-time signal.

A discrete-time system (Fig.2.6) performs prescribed operations on signals. Basic operations include addition / subtraction, multiplication, and delay. Operator $T\{.\}$ can represent any formula or algorithm for obtaining the output signal.

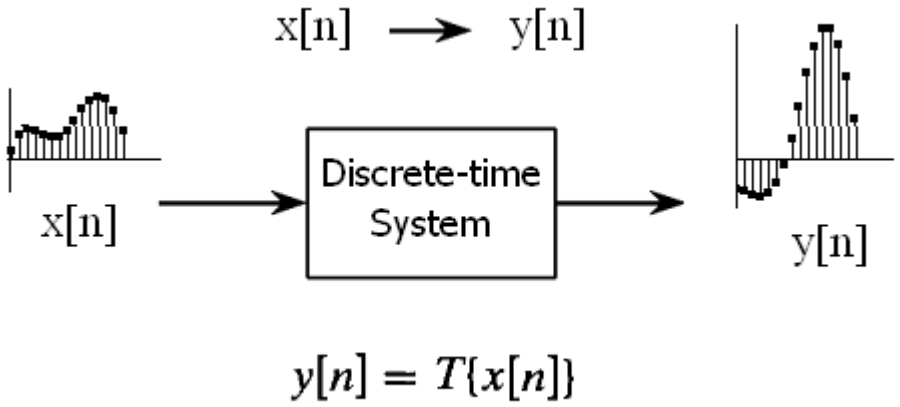

FIGURE 2.6. Representation of a discrete-time system

Most of the DSP systems are viewed as *linear systems*. Nonlinear systems are much more complicated and do not have such a rich theoretical background as linear systems do. Linear systems have two mathematical properties:

1) *Homogeneity*, or *scaling*, property:

$$\text{IF } x[n] \rightarrow y[n] \text{ THEN}$$
$$k x[n] \rightarrow k y[n] \text{ for any } k = const \tag{2.23}$$

2) *Additivity* property:

$$\text{IF } (x_1[n] \rightarrow y_1[n]) \text{ AND } (x_2[n] \rightarrow y_2[n]) \text{ THEN}$$
$$(x_1[n] + x_2[n]) \rightarrow (y_1[n] + y_2[n]) \tag{2.24}$$

A particularly important class of discrete-time systems includes *Linear Time-Invariant (LTI)* systems. In fact, the term "Linear system" often implies the term "LTI system" in DSP. An LTI system is a linear system with the following additional condition (property):

$$\text{IF } x[n] \rightarrow y[n] \text{ THEN}$$
$$x[n-d] \rightarrow y[n-d] \text{ for any } d = const \tag{2.25}$$

The Eq.2.25 represents the *time-invariance (or shift-invariance) property*. It means that a certain delay in the input signal causes the same delay in the output signal. In the rest of this guide we'll consider, mostly, LTI systems.

The combination of Eq.2.23 and Eq.2.24 leads to the *Superposition Principle*, one of the fundamental concepts of DSP:

$$\text{IF } (x_1[n] \rightarrow y_1[n]) \text{ AND } (x_2[n] \rightarrow y_2[n]) \text{ THEN}$$
$$(a x_1[n] + b x_2[n]) \rightarrow (a y_1[n] + b y_2[n]) \text{ for any } a, b = const \tag{2.26}$$

The superposition principle is very useful if a signal can be considered as a composition (addition) of simpler signals. According to this principle, instead of directly passing the input signal through the system, we can process its components separately, and then make up the output signal by combining their outputs.

One of the most important and visual examples of signal decompositions is an *impulse decomposition*. Impulse decomposition of a signal, along with Fourier decomposition (discussed in chapter 3), provides understanding of the most DSP techniques. The main idea behind the impulse decomposition is that any signal can be represented as a sum of scaled and delayed unit impulses:

$$x[n] = \sum_{k=-\infty}^{+\infty} x[k]\,\delta[n-k] \qquad (2.27)$$

The illustration of impulse decomposition is shown in Fig.2.7: a signal *x[n]*, containing *N* samples, is broken into *N* component signals, each containing *N* samples. The rest of additive components are zeros. Impulse decomposition allows signals to be examined one sample at a time.

There are also several minor signal decompositions occasionally used in DSP, such as *an even-odd decomposition*, *an interlaced decomposition*, and *a step decomposition*. See [3, Chapter 5] for details.

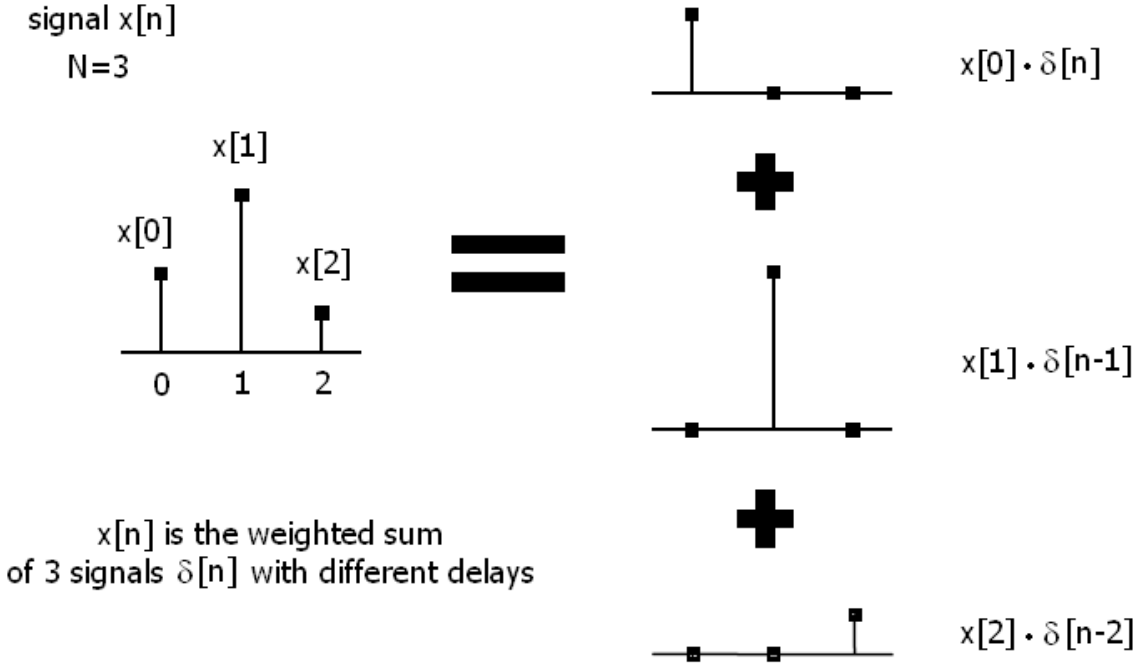

FIGURE 2.7. The example of an impulse decomposition

The Eq.2.23-2.26 can be viewed both as the properties and the requirements for linear systems. There are also two worth-mentioning properties of linear systems: *static linearity* and *sinusoidal fidelity*. They are not requirements: if a system is linear, then it has these properties; but if a system does have these properties, it isn't necessarily linear (in most cases, it is, though).

Static linearity property: if the input signal is constant, then the output signal is also constant:

$$c_1[\mathrm{n}] \rightarrow c_2[\mathrm{n}], \qquad c_1, c_2 = const \qquad (2.28)$$

Sinusoidal fidelity property: if the input signal is sinusoidal, then the output signal is also sinusoidal, with the same frequency:

$$A_1 \sin(\omega n + \varphi_1) \rightarrow A_2 \sin(\omega n + \varphi_2) \tag{2.29}$$

Notice, that amplitude and phase may change!

Example 2.2.
Given a system transforming input signal in the following way:

$5\sin(20\pi n + \pi/3) + 2\sin(5\pi n) \rightarrow \cos(5\pi n)$

Does the system have a sinusoidal fidelity property?

Solution:
Yes, it does. The component "2sin(5πn)" is transformed into "cos(5πn)", so only the phase and the amplitude are changing here, while the frequency is the same. The component "5sin(20πn+π/3)" doesn't have a corresponding output component. We may assume that the amplitude of this component is reducing to 0.

Example 2.3.
The *accumulator* system is defined by the following equation:

$$y[n] = \sum_{k=-\infty}^{n} x[k] \tag{2.30}$$

Prove that the accumulator system is LTI.

Solution:
First, we'll show that the system satisfies the requirements for linearity (i.e. superposition principle). Let's define input signals $x_1[n]$ and $x_2[n]$. Their outputs are $y_1[n] = \sum_{k=-\infty}^{n} x_1[k]$ and $y_2[n] = \sum_{k=-\infty}^{n} x_2[k]$, respectively.
Let's define a signal $x_3[n] = ax_1[n] + bx_2[n]$. We must show that the corresponding output is $y_3[n] = ay_1[n] + by_2[n]$:

$$y_3[n] = \sum_{k=-\infty}^{n} x_3[k] = \sum_{k=-\infty}^{n} (ax_1[k] + bx_2[k]) = a\sum_{k=-\infty}^{n} x_1[k] + b\sum_{k=-\infty}^{n} x_2[k] \tag{2.31}$$

Thus, $y_3[n] = ay_1[n] + by_2[n]$, and the system satisfies the superposition principle. Next, we'll show that the system is time-invariant. Let's see how the system will respond to a signal $x_d = x[n-n_0]$:

$$y_d[n] = \sum_{k=-\infty}^{n} x[k - n_0] \tag{2.32}$$

Substituting the change of variables $m=k-n_0$ into summation leads to:

$$y_d[n] = \sum_{m=-\infty}^{n-n0} x[m] = y[n - n_0] \tag{2.33}$$

That is, the system transforms an input signal $x[n-n_0]$ into an output signal $y[n-n_0]$. Therefore, it's an LTI system.

A discrete-time system is *causal*, if every output signal sample depends only on the preceding input signal samples. For example, the *forward difference system* defined by the relationship

$$y[n] = x[n+1] - x[n] \tag{2.34}$$

is not causal, since the n^{th} output sample depends on the future $(n+1)^{th}$ input sample. Contrarily, the *backward difference system* defined by relationship

$$y[n] = x[n] - x[n-1] \tag{2.35}$$

is causal, since each output sample depends only on the present and one past input samples.

A discrete-time system is *stable*, if and only if it transforms a bounded input signal x[n] into a bounded output signal y[n]:

$$| x[n] | \leq B_x < \infty, \quad \text{and} \quad | y[n] | \leq B_y < \infty, \quad \text{for all n} \tag{2.36}$$

2.5. Convolution

In LTI systems, *convolution* is used to describe the relationship between the input signal, the impulse response, and the output signal. Let's give the definition of an *impulse response* (Fig.2.8).

IMPULSE RESPONSE
is the output signal of a system when the input signal is a unit impulse.

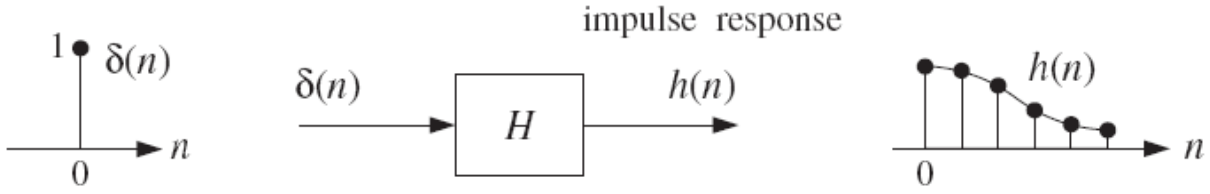

FIGURE 2.8. Representation of impulse response h[n]

The impulse response is unique characteristic of any LTI system. But what is the use of impulse response? In order to answer this question, we need to recall that any custom signal can be decomposed into a set of scaled and delayed unit impulses (Fig.2.7). Therefore, according to the superposition principle, if we know how the system responds to a single unit impulse, then we know how the system will respond to any signal! Indeed, keeping in mind Eq.2.27, and the definition of an impulse response, we can express the output signal as:

$$y[n] = \sum_{k=-\infty}^{+\infty} x[k]\, h[n-k] \tag{2.37}$$

The Eq.2.37 is called the *convolution sum*. The convolution is denoted with symbol "*": y[n]=x[n]*h[n]. Fig.2.9 demonstrates how we obtain the output signal.

Eq.2.37 reflects the idea of the so called *input-side algorithm* for calculating convolution. It analyzes how *each* sample in the input signal affects *many* samples in the output signal. The input-side algorithm is very useful for conceptual understanding of convolution. However, from a programming point of view, the most straightforward technique would be to loop through the *output* signal samples. This idea underlies an *output-side algorithm*, according to which *each* output signal sample is some combination of *many* values of the input signal and impulse response:

$$y[n] = x[n] * h[n] = \sum_{k=-\infty}^{+\infty} x[n-k]\, h[k] \tag{2.38}$$

The infinite limits of summation in Eq.2.37, Eq.2.38 have rather theoretical meaning: they emphasize a universality of corresponding techniques. In practice, we deal with finite signals, so only several samples used in calculations are non-zero. Assuming that the input signal contains *N* samples, and the impulse response consists of *M* samples, the previous formula can be rewritten as:

$$y[n] = x[n] * h[n] = \sum_{k=0}^{M-1} x[n-k]\, h[k] \tag{2.39}$$

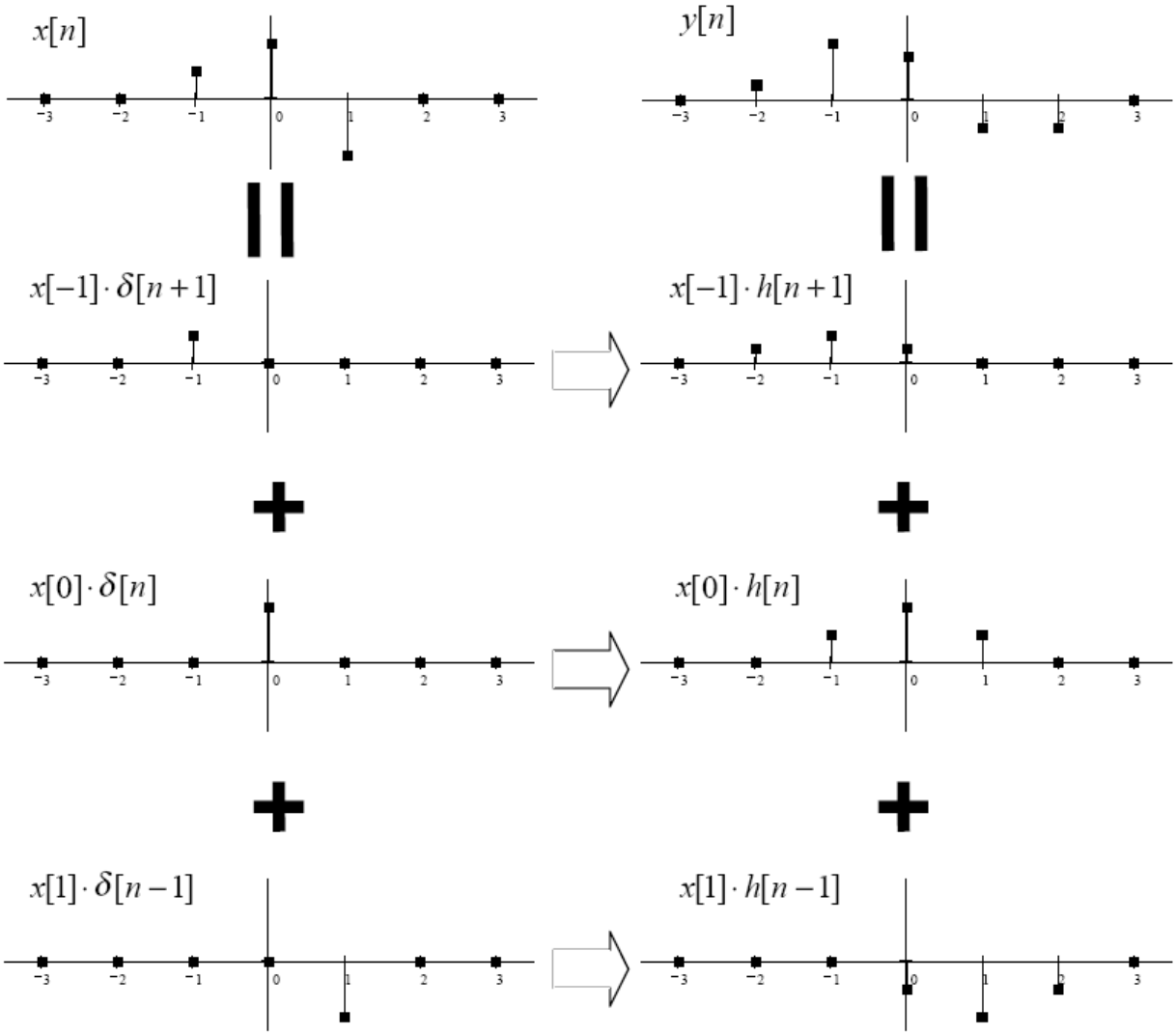

FIGURE 2.9. The convolution sum

The component *h[k]* is often referred to as the *convolution machine*. It slides along the input signal samples. The resulting signal will contain *N+M–1* samples. Formula (2.39) requires that *N*M* multiplications must be performed. Let's illustrate it.

LET'S CODE!
Convolve signal x = {1,5,3,2,6} with signal h = {2,3,1}. Plot the result.

```
x = [1 5 3 2 6];     % 5-point signal x
h = [2 3 1];         % 3-point signal h
y = conv(x, h)       % convolution
plot(y);             % plot resulting signal
```

MATLAB gives us the following result: y = [2 13 22 18 21 20 6].

Let's do the MATLAB work step-by-step, and calculate the output values.

During calculation of the first output sample, we'll notice that the output-side algorithm deals with non-existing input samples x[-2] and x[-1]:

$$y[0] = \mathbf{x[-2]}\, h[0] + \mathbf{x[-1]}\, h[1] + x[0]\, h[2] \qquad (2.40)$$

The same problem will occur during the calculation of the last output samples, in which non-existing samples x[5] and x[6] will be involved.

The common practice is simply to set these coefficients equal to zero. And so, our calculations will look like:

```
y[0] = x[0]h[0]                                        |          |
y[1] = x[1]h[0] + x[0]h[1]                             | M-1      |
y[2] = x[2]h[0] + x[1]h[1] + x[0]h[2]                  |          | N
y[3] = x[3]h[0] + x[2]h[1] + x[1]h[2]                  |          |
y[4] = x[4]h[0] + x[3]h[1] + x[2]h[2]                  |          |
y[5] =            x[4]h[1] + x[3]h[2]                  | M-1      |
y[6] =                       x[4]h[2]                  |          |
```

The output signal contains *5+3–1*=7 samples, and *5*3*=15 multiplications are performed. Notice that the first 2 samples (*M–1* samples) and the last 2 samples (*M–1* samples) in the output signal are based on less information about input signal than the samples between! In most cases, the beginning and ending samples don't represent any useful information for a signal analysis.

Example 2.4.
Convolve signal x = {1,5,3,2,6} with signal h = {2,3,1}.

Solution:

```
y[0] = 1·2                          = 2
y[1] = 5·2 + 1·3                    = 13
y[2] = 3·2 + 5·3 + 1·1              = 22
y[3] = 2·2 + 3·3 + 5·1              = 18
y[4] = 6·2 + 2·3 + 3·1              = 21
y[5] =       6·3 + 2·1              = 20
y[6] =             6·1              = 6
```

In real-time DSP applications, the required *N*M* multiplications can take a lot of time. DSP microprocessors are optimized for convolution-like operations, but if you write a program for a conventional CPU, you must take into account the above-mentioned fact. If signals are long enough, FFT convolution is commonly used, instead of direct implementation of formula (2.39). FFT convolution is reviewed in the next chapters.

Now, let's take a look at the examples of common impulse responses:

1) $h[n] = \delta[n]$. It's the simplest impulse response you can imagine. In this case, the LTI system simply passes signals without a change (the output signal is exactly the same as the input signal):

$$y[n] = x[n] * \delta[n] = x[n] \tag{2.41}$$

2) $h[n] = \delta[n-d]$. In this case, the LTI system shifts the input signal (making signal *delay*, if $d > 0$, or *advance*, if $d < 0$):

$$y[n] = x[n] * \delta[n-d] = x[n-d] \tag{2.42}$$

3) $h[n] = k\delta[n]$. The LTI systems characterized by this impulse response are called *amplifiers* (if $k>1$), or *attenuators* (if $0<k<1$):

$$y[n] = x[n] * k\delta[n] = kx[n] \tag{2.43}$$

Mathematical properties of convolution are particularly important:

1) *Commutative* property:

$$x[n] * h[n] = h[n] * x[n] \tag{2.44}$$

2) *Associative* property:

$$(x[n] * h_1[n]) * h_2[n] = x[n] * (h_1[n] * h_2[n]) \tag{2.45}$$

3) *Distributive property*:

$$x[n] * h_1[n] + x[n] * h_2[n] = x[n] * (h_1[n] + h_2[n]) \tag{2.46}$$

The properties (2.44)-(2.46) indicate that it's possible to rearrange and recombine the LTI systems (Fig.2.10).

2.6. Cross-correlation

Cross-correlation is a measure of similarity of two signals. It is commonly used for searching a long-signal for a shorter, known feature. The cross-correlation is similar in nature to the convolution of two signals. It is defined as:

$$y[n] = \sum_{k=-\infty}^{+\infty} x[n+k]\, g[k] \tag{2.47}$$

As we can see, the difference between convolution and cross-correlation is in their convolution machines: $h[k] = g[-k]$. The formula (2.39) essentially slides the signal $x[n]$ along the x-axis, calculating the sum of $x[n+k]*g[k]$ at each

position *n*. When the signals match, the value of *y[n]* is maximized. This is because when peaks are aligned, they make a large contribution to the sum. Similarly, when troughs align, they also make a large contribution to the sum because the product of two negative numbers is positive. And so, cross-correlation can be used to find how much the signal *x[n]* must be shifted along the x-axis to make it identical to the signal *g[n]*.

Cross-correlation finds application in pattern recognition, cryptanalysis, and neurophysiology.

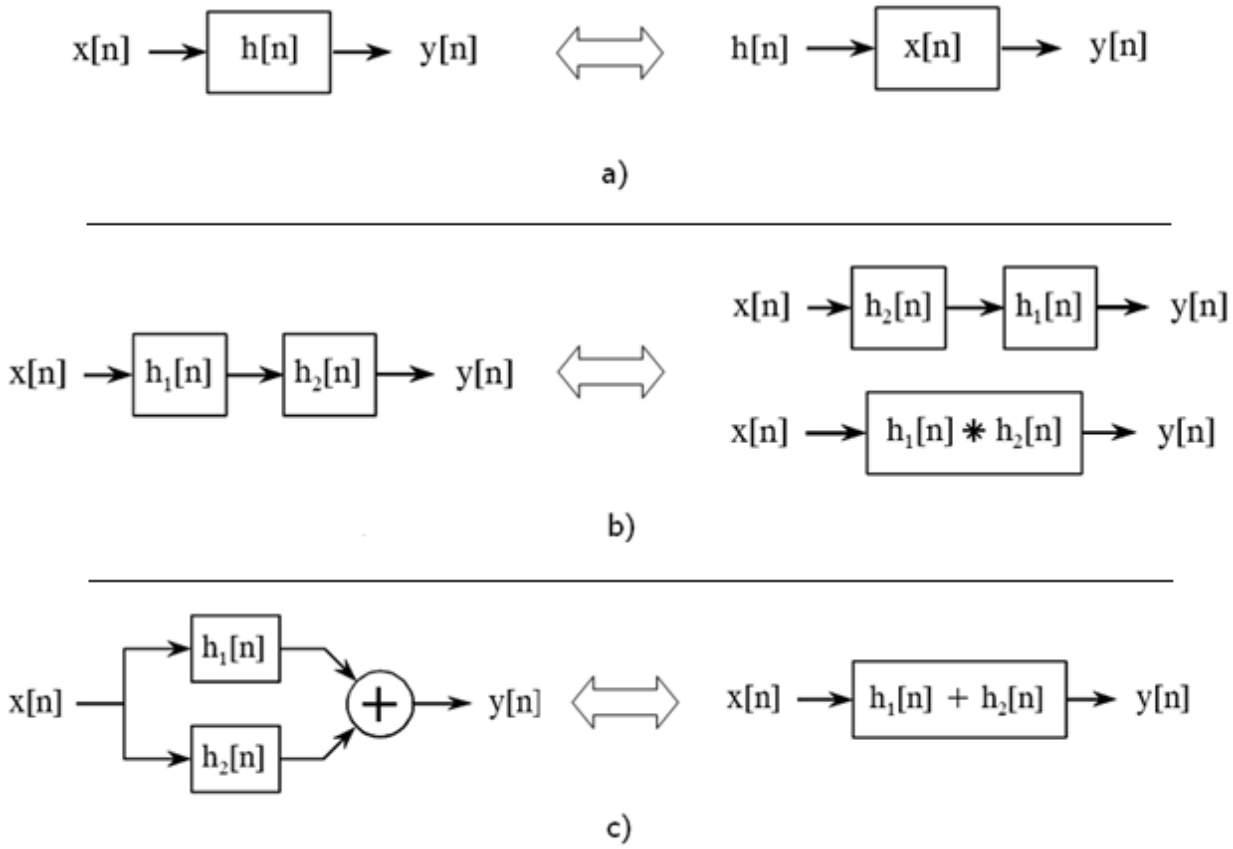

FIGURE 2.10. Illustration of convolution properties:
a) commutative property; b) associative property; c) distributive property

2.7. Nonlinear Systems

In the previous sections we have considered the most basic issues regarding linear systems. In such systems signals can be added, multiplied by constant, and delayed. However, there are also important cases of *nonlinear* DSP systems, in which signals are *multiplied*, *convolved*, etc. For example, an amplitude modulation of signals is, basically, a multiplication of signal by the carrier wave. The problem is that an analysis of nonlinear systems is very complicated task. The most preferable way of dealing with non-linear systems is to apply a *linearizing transform*, such as homomorphic transform (discussed in chapter 7).

Example 2.5.
Given a system transforming input signal in the following way:

$$\mathbf{x \rightarrow 2x + \sin(x)}$$

Prove that the system is nonlinear.

Solution:
In order to prove that the system is nonlinear, we just need to show at least one example of the input-output pair that violates the superposition principle. Let's consider input signal $x[n]=\pi/2$. The system's response is:

$$y[n] = 2x[n] + \sin(x[n]) = \pi + 1$$

According to homogeneity property, if the system is linear then $2x \rightarrow 2y$. However, in our case the output corresponding to input $x'[n]=2x[n]$ is:

$$y'[n] = 2*2x[n] + \sin(2x[n]) = 2\pi \neq 2\pi + 2 = 2y[n]$$

Therefore, the system is nonlinear.

Examples of linear systems are: digital filters and amplifiers; signal effects (reverberation, resonance, blurring, etc.); wave propagation.

Examples of nonlinear systems are: amplitude modulation; systems for peak detection; clipping and crossover distortion; systems with a threshold.

LAB1	SIGNAL PROCESSING IN TIME DOMAIN
	points: 15 hours: 4

Objectives

1. To understand the concepts of signal sampling.
2. To examine the properties of convolution.

Exercise 1.1 [1 point]

Answer the following questions:

a. What is signal? What is system?
b. Give the classification of signals.
c. What does the Nyquist-Shannon-Kotelnikov sampling theorem state? How is it used in practice?
d. Define LTI system. What are the main properties of LTI systems?
e. What is system's impulse response?
f. Describe the convolution operation.
g. What is cross-correlation of signals?

Exercise 1.2 [4 points]

Given an analog audio signal 100ms long with frequencies up to 7000Hz. Answer the following questions:

a. What is the minimum required sampling frequency that allows an exact reconstruction of the signal from its samples?
b. What will happen if the sampling frequency is 5000 Hz?
c. What will happen if the sampling frequency is 15000 Hz?
d. How many samples are needed to store the corresponding digital signal if the sampling frequency is 15000 Hz?

Exercise 1.3 (CODE) [5 points]

In this task you'll programmatically generate digital audio signals. Write code to do the following:

1. Generate signal *s1* consisting of three sinusoids with frequencies *f1*, *f2* and *f3*, respectively. The signal has to be 600 milliseconds long, and it should be sampled with sampling frequency *fs*. The values of the frequencies are given in table 1.1.
2. Generate noise signal *sn* 600 milliseconds long.
3. Superimpose signals *s1* and *sn* (get the "noisy" version of *s1*). The resulting signal will be *s2*.
4. Generate signal *s3* by shifting *s1* by 300 samples.
5. Generate signal *s4*. It must be identical to signal *s1* except that it has to be sampled at frequency fs_4=11025 Hz.
6. Plot the first 50 samples of each signal.

7. Save each signal to WAVE file (set number of channels equal to 1).
8. Load each signal from corresponding WAVE file. Listen to signal *s1* and *s4*. Explain the difference.

Exercise 1.4 (CODE) [3 points]

Write code that computes the convolution and cross-correlation of signals and plots the results. Use signals from Example 2.4 and signal pairs {*s1*, *s2*} and {*s1*, *s3*} as the input into your program. Explain the results. Compare your results with the results of MATLAB functions *conv()* and *xcorr()*.

Exercise 1.5 (CODE) [2 points]

Write code that processes a segment of an input signal and evaluates the following characteristics: 1) energy, 2) zero-cross rate, 3) mean, and 4) variance. The program should allow setting the positions of the first and the last samples of a signal segment to process.

TABLE 1.1. Table of frequencies

№	fs (Hz)	f1 (Hz)	f2 (Hz)	f3 (Hz)
1	44100	900	1400	7100
2	44100	2000	3000	7000
3	44100	1500	4500	8100
4	44100	1200	2500	6300
5	44100	1000	3500	7200
6	22050	900	1400	7100
7	22050	2000	3000	7000
8	22050	1500	4500	8100
9	22050	1200	2500	6300
10	22050	1000	3500	7200
11	16000	900	1400	6100
12	16000	2000	3000	6800
13	16000	1500	4500	6100
14	16000	1200	2500	6300
15	16000	1000	3500	6200

CHAPTER 3 FOURIER ANALYSIS OF SIGNALS

3.1. Signals and Systems in Frequency Domain

In the previous chapter we have introduced the discrete-time signals and systems. We focused on time-domain representation, since it was the most obvious way to look at signals. Any signal is a certain function, so we just analyze given values of the function. Moreover, in most cases, it is the signal's time-domain representation that is stored in computer memory.

There is, however, another way to look at signals and systems. It's called the *frequency domain*, and it's connected with the fact that signals can be represented in terms of exponentials or sinusoids, and LTI systems handle such kind of signals in a unique way. This fact has tremendous importance in DSP. Roughly speaking, the frequency-domain representation of a signal shows what frequency components and phase components are present in signal.

Let's begin with considering what happens to exponentials after passing them through an LTI system. The formula for an exponential is:

$$x[n] = e^{j\omega n}, \qquad -\infty < n < +\infty \tag{3.1}$$

Notice that the exponential in Eq.3.1 is in complex form. So far we have only considered real numbers. From this moment we'll deal, mostly, with complex numbers. According to Eq.2.39, the output of an LTI system is:

$$y[n] = \sum_{k=-\infty}^{+\infty} x[n-k]\,h[k] = \sum_{k=-\infty}^{+\infty} e^{j\omega(n-k)}\,h[k] = e^{j\omega n} \sum_{k=-\infty}^{+\infty} e^{-j\omega k}\,h[k] \tag{3.2}$$

The far right sum in Eq.3.2 is constant with respect to an argument of the exponential. Let's denote it as:

$$H(e^{j\omega}) = \sum_{k=-\infty}^{+\infty} e^{-j\omega k}\,h[k] \tag{3.3}$$

Therefore, the Eq.3.2 can be rewritten as:

$$y[n] = H(e^{j\omega})\,e^{j\omega n} = H(e^{j\omega})\,x[n] \tag{3.4}$$

Consequently, $e^{j\omega n}$ is the eigenfunction of the system, and $H(e^{j\omega})$ is the corresponding eigenvalue. The latter is called *frequency response*.

FREQUENCY RESPONSE
characterizes how an LTI system responds to a signal in the frequency domain, i.e. how an LTI system modifies the frequency components of an input signal.

For example, the frequency response of a system that suppresses all signal frequencies higher than cut-off frequency (i.e. *low-pass filter*) is shown in Fig.3.1. Low-pass filters will be discussed in the following chapters.

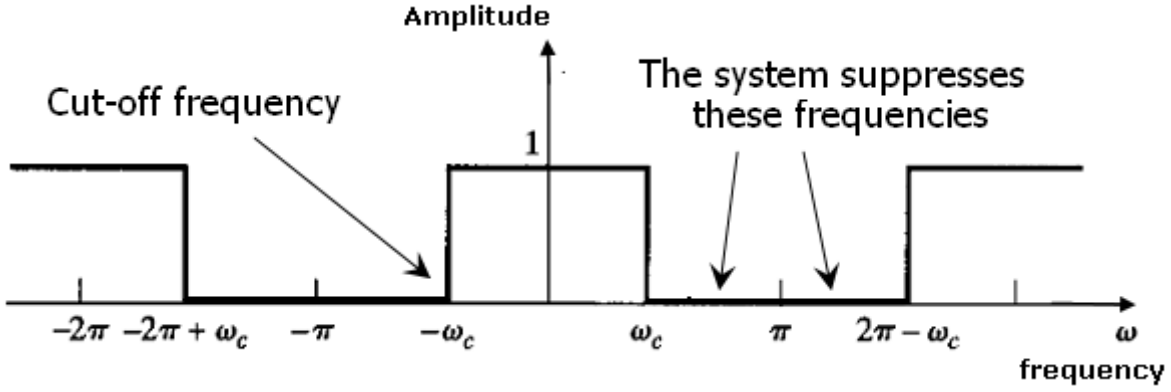

FIGURE 3.1. The example of frequency response

Note the frequency response shown in Fig.3.1 is 2π-periodic. Indeed:

$$H(e^{j(\omega+2\pi)}) = \sum_{k=-\infty}^{+\infty} e^{-jk(\omega+2\pi)}\, h[k] = \sum_{k=-\infty}^{+\infty} e^{-jk\omega} e^{-j2\pi k}\, h[k] = \sum_{k=-\infty}^{+\infty} e^{-jk\omega}\, h[k] = H(e^{j\omega}) \tag{3.5}$$

Therefore, we can specify $H(e^{j\omega})$ only over some 2π-interval. The interval [-π; π] is commonly used. With respect to this interval, the "low frequencies" are those that are close to 0, while the "high frequencies" are those that are close to ±π.

Remember that we're talking about the *complex* frequency response that can be expressed in rectangular form (in terms of real and imaginary parts) as

$$H(e^{j\omega}) = H_{RE}(e^{j\omega}) + jH_{IM}(e^{j\omega}), \tag{3.6}$$

or in polar form as

$$H(e^{j\omega}) = |H(e^{j\omega})|\, e^{\angle H(e^{j\omega})}, \tag{3.7}$$

where $|H(e^{j\omega})|$ is the magnitude, and $\angle H(e^{j\omega})$ is the phase.

And last but not least fact about complex exponentials is that they are related to sinusoids due to Euler's formula:

$$\begin{cases} e^{j\omega} = \cos\omega + j\sin\omega \\ e^{-j\omega} = \cos\omega - j\sin\omega \end{cases} \qquad \begin{cases} \cos\omega = (e^{j\omega} + e^{-j\omega})/2 \\ \sin\omega = (e^{j\omega} - e^{-j\omega})/2j \end{cases} \tag{3.8}$$

Now, recall the property of sinusoidal fidelity (Eq.2.29). It reflects the same idea as the Eq.3.4: signal frequencies ω do not change, whereas signal amplitude and phase may change, and the information about how exactly they change is contained in $H(e^{j\omega})$. Rewriting sine and cosine functions as complex exponentials allows to simplify many formulae. In the following sections we'll sometimes refer to sinusoids and their frequencies, while using exponentials in mathematical formulae, so don't be confused! Just keep in mind the Eq.3.8.

Example 3.1.
Find the frequency response of the ideal delay system defined as:
y[n]=x[n-d]

Solution:
Let's see how the system responds to an exponential:

$$y[n] = x[n-d] = e^{j\omega(n-d)} = e^{-j\omega d}e^{j\omega n} \tag{3.9}$$

Comparing Eq.3.9 with Eq.3.4, we obtain:

$$H(e^{j\omega}) = e^{-j\omega d} \tag{3.10}$$

We've found the solution. Let's analyze it. From Eq.3.7, the magnitude and the phase are:

$$|H(e^{j\omega})| = 1 \tag{3.11}$$

$$\angle H(e^{j\omega}) = -\omega d \tag{3.12}$$

Hence, the system doesn't change the amplitude of frequency components, but it shifts all phase components in proportion to the delay parameter *d*.

3.2. The Family of Fourier Transforms

Is there a way we can convert any given continuous signal between the time and frequency domains? The answer is positive, and the mathematical tool for such conversion is *Fourier Transform* named after a French mathematician and physicist, Jean Baptiste Joseph, Baron de Fourier (1768-1830).

FOURIER TRANSFORM (FT)
is a mathematical technique for the signal analysis and synthesis based on decomposition of the signal into a set of sine wave components with different frequencies.

From a mathematical point of view, FT is a pair of mathematical operators. The first operator decomposes signal into a set of different sinusoids. This set forms the *frequency spectrum* of signal. The operator performs the signal analysis, and is sometimes referred to as the *Forward Fourier transform*. The second operator called the *Inverse Fourier transform* synthesizes signal from its frequency components.

Fig.3.2 illustrates the spectrums of a pure sine wave and a random signal. In the first case the signal power is concentrated in one frequency (the frequency of the sine wave), while in the second case it is spread in the frequency-domain.

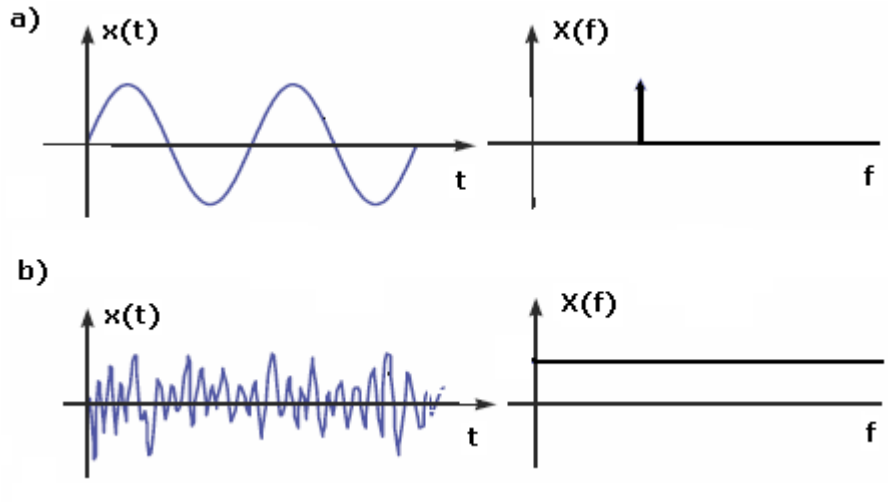

FIGURE 3.2. The examples of frequency spectrums X(f): a) case of a sinusoidal signal x(t); b) case of a random signal x(t) (white noise)

The FT plays a central role in many DSP techniques. It's used in filtering, music processing, signal coding, signal synthesis, feature extraction for speech and image recognition, spectral analysis in physics, to name a few.

There is a whole family of Fourier transforms, since the four basic types of signals can be encountered (see Fig.3.3).

The motivation for Fourier analysis comes from the study of **Fourier series**, the first member of FT family. In the study of Fourier series, *periodic* functions with period T_0 are written as the sum of sinusoids. The corresponding formula in complex exponential form is

$$x(t) = \sum_{k=-\infty}^{+\infty} c_k e^{jk\omega_0 t}, \qquad (3.13)$$

where $\omega_0 = \frac{2\pi}{T_0}$. The Eq.3.13 is the formula for signal *synthesis*. It is complemented with the formula for *analysis*:

$$c_k = \frac{1}{T_0}\int_{T_0} x(t)e^{-jk\omega_0 t}dt \qquad (3.14)$$

FOURIER SERIES	**FOURIER TRANSFORM**
Continuous signal Discrete spectrum	Continuous signal Continuous spectrum
periodic signal	*aperiodic signal*
periodic signal	*aperiodic signal*
Discrete signal Discrete spectrum	Discrete signal Continuous spectrum
DISCRETE FOURIER TRANSFORM (DFT)	**DISCRETE-TIME FOURIER TRANSFORM (DTFT)**

FIGURE 3.3. The Family of Fourier Transforms

Essentially, the synthesis formula represents an inverse FT, and the analysis formula represents a forward FT. Wave components whose frequencies are multiples of ω_0 (i.e. ω_0, $2\omega_0$, $3\omega_0$, etc.) are called *harmonics*. Each complex coefficient c_k is a component of discrete spectrum, and holds the information about the amplitude and phase of the k^{th} harmonic. The distance between adjacent samples of the spectrum is equal to ω_0. In general, the Fourier series expansion of a periodic signal has infinite number of harmonics. Fig.3.4 illustrates the nature of Fourier series.

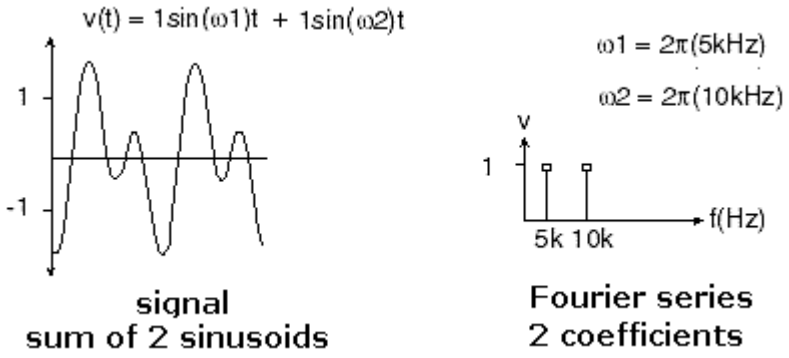

FIGURE 3.4. Fourier Series

The Fourier transform is an extension of the Fourier series that results when the period of analyzed signal is infinitely long. Most real-world signals aren't periodic, so the following technique is applied. We keep treating signal as periodic, but we assume that T_0 goes to infinity. With T_0 tending to infinity, ω_0 tends to 0, so the distance between adjacent samples in the spectrum becomes infinitesimally small (see Fig.3.5a). In other words, the spectrum becomes continuous, and the formulae for synthesis and analysis change (we give them here without derivation):

$$\begin{cases} x(t) = \dfrac{1}{2\pi}\displaystyle\int_{-\infty}^{+\infty} X(\omega)e^{j\omega t}d\omega \\ X(\omega) = \displaystyle\int_{-\infty}^{+\infty} x(t)e^{-j\omega t}dt \end{cases} \tag{3.15}$$

Fig.3.5b illustrates the nature of continuous Fourier transform.

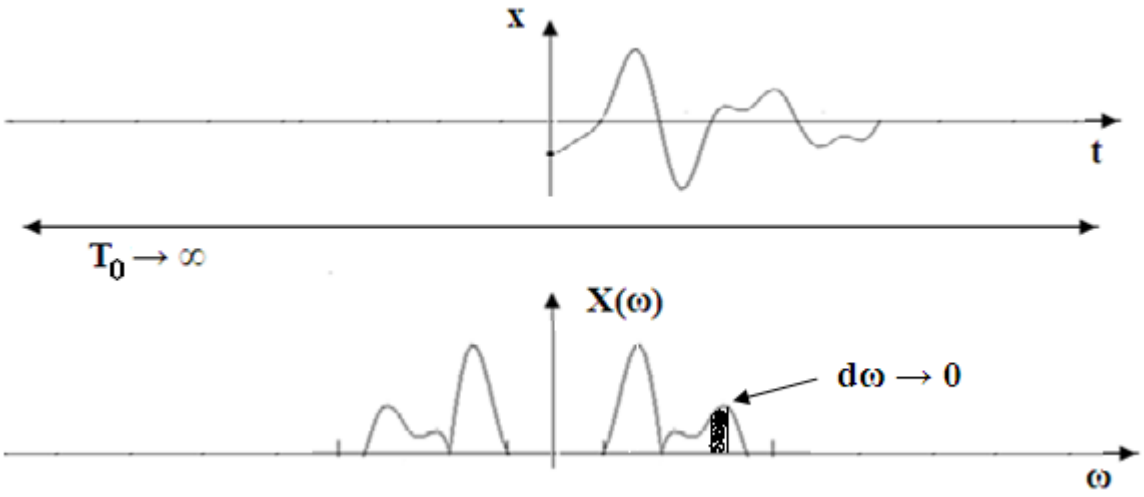

FIGURE 3.5. Continuous-time Fourier Transform: a) aperiodic signal x(t) becomes ∞-periodic; b) Fourier transform becomes continuous

The first two members of FT family deal with continuous signals. We have reviewed them because they provide understanding of general principles of the Fourier transform. However, they don't suit DSP purposes. In order to adapt the continuous FT for discrete-time signals, **Discrete-time Fourier transform (DTFT)** was developed. Since DTFT deals with the discrete-time signal $x[n]=x(nT_s)$, instead of the continuous signal $x(t)$, the formulae (3.15) are rewritten as (again, we don't show here how they are derived):

$$\begin{cases} x[n] = \dfrac{1}{2\pi}\displaystyle\int_{2\pi} X(\Omega)e^{j\Omega n}d\Omega \\ X(\Omega) = \displaystyle\sum_{n=-\infty}^{+\infty} x[n]e^{-j\Omega n} \end{cases}, \qquad (3.16)$$

where $\Omega = \omega T_s = \dfrac{2\pi f}{f_s}$ is the digital frequency (see Section 2.2). Note that the integral in analysis formula is replaced with the discrete sum because the continuous parameter *t* (time) is replaced with the discrete parameter *n* (a sample number). As we can see, since the digital frequency is still a continuous variable, the formula for synthesis is almost the same as the formula (3.15), except one major difference: the interval of integration is 2π. This is because, like the frequency response, and unlike the continuous FT-spectrum, the DTFT-spectrum is periodic with period 2π:

$$X(\Omega + 2\pi k) = \sum_{k=-\infty}^{+\infty} x[n]e^{-j\Omega n + 2\pi knj} = \sum_{k=-\infty}^{+\infty} x[n]e^{-jn\Omega} = X(\Omega) \qquad (3.17)$$

Hence, any frequency can be mapped onto the range [0;2π] or [-π;π], namely, the *Nyquist interval*. Usually, instead of "*X(Ω)*", the notation similar to the notation of a frequency response is used: "$X(e^{j\omega})$". This notation reflects the same idea: each value of the spectrum determines the relative amount of each complex sinusoidal component. It may be not quite obvious, so let's take a look how the frequency spectrum can be represented in a complex plane (Fig.3.6). Basically, the frequencies of sinusoidal components are expressed in terms of the phase angles ω varying from 0 to 2π.

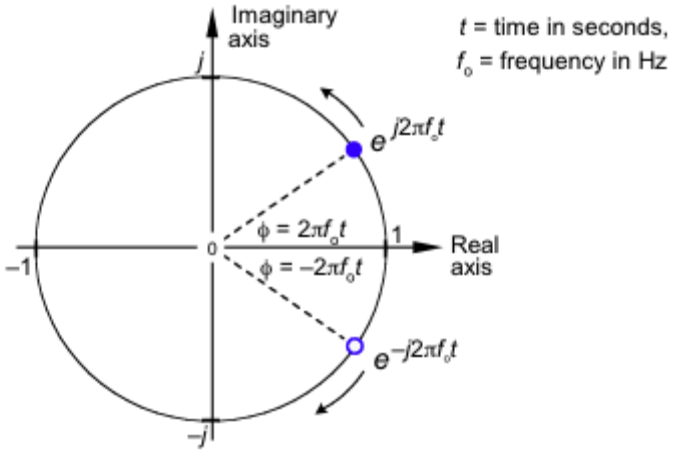

FIGURE 3.6. Complex frequency plane

The magnitude $|X(e^{j\omega})|$ of DTFT is called the *Magnitude spectrum*, and the phase $\angle X(e^{j\omega})$ of DTFT is called *the Phase spectrum*. The *Power spectrum* is defined as $|X(e^{j\omega})|^2$. $X(e^{j\omega})$ is sometimes referred to as the *Fourier image* of signal.

Fig.3.7 illustrates the nature of DTFT. Pay attention to the title of x-axis. There are three common ways to label x-axis of the frequency domain (Fig.3.8): 1) as the frequency in Hz; 2) as the fraction of the sampling rate (normalized digital frequency in cycles per sample); 3) as the digital frequency in radians per sample.

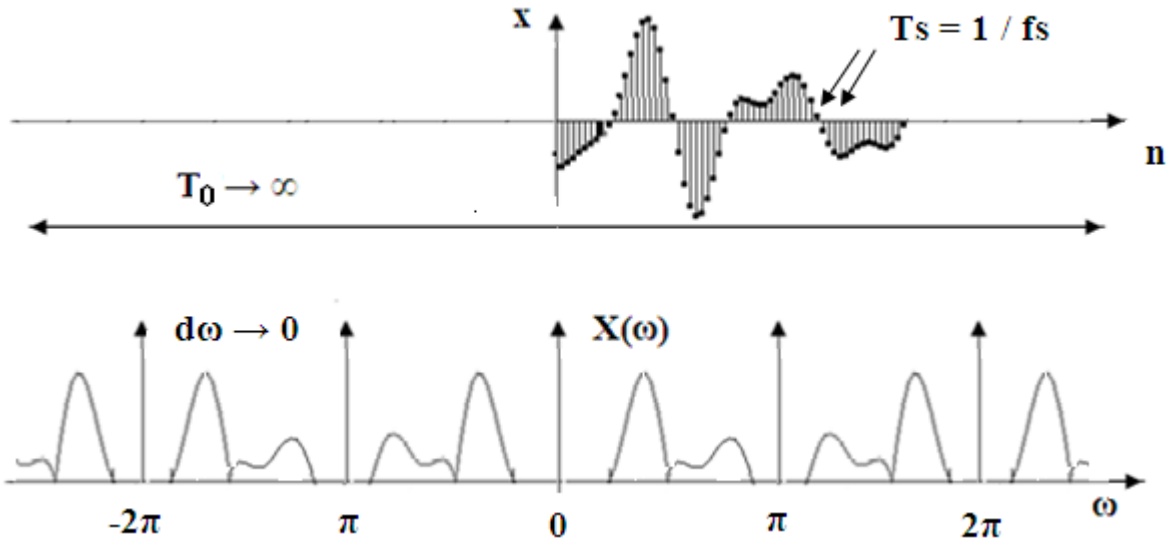

FIGURE 3.7. Discrete-time Fourier transform

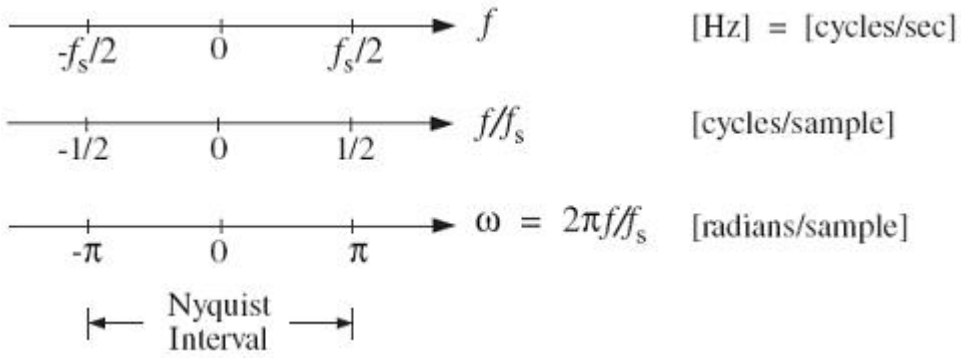

FIGURE 3.8. Frequency units commonly used in DSP

DTFT is what we should use for theoretical analysis of discrete-time signals. Nevertheless, it's still impossible to program DTFT because the spectrum is continuous. The only programmable representative of FT family is the sampled version of DTFT called the **Discrete Fourier transform (DFT)**. The DFT can be derived from the DTFT in two steps: 1) signal truncating; 2) frequency sampling. Firstly, the discrete-time signal is truncated: we select only N signal samples for

computations. We do this only to get rid of the infinite sum in the analysis formula (3.16). Remember that any representative of FT family *treats* the signal as infinite and periodic because the signal is decomposed into a set of infinite and periodic sinusoids. Therefore, DFT assumes that N samples are contained in exactly one period of infinite, periodic (and, in fact, non-existing) signal. This signal is shown in Fig.3.9.

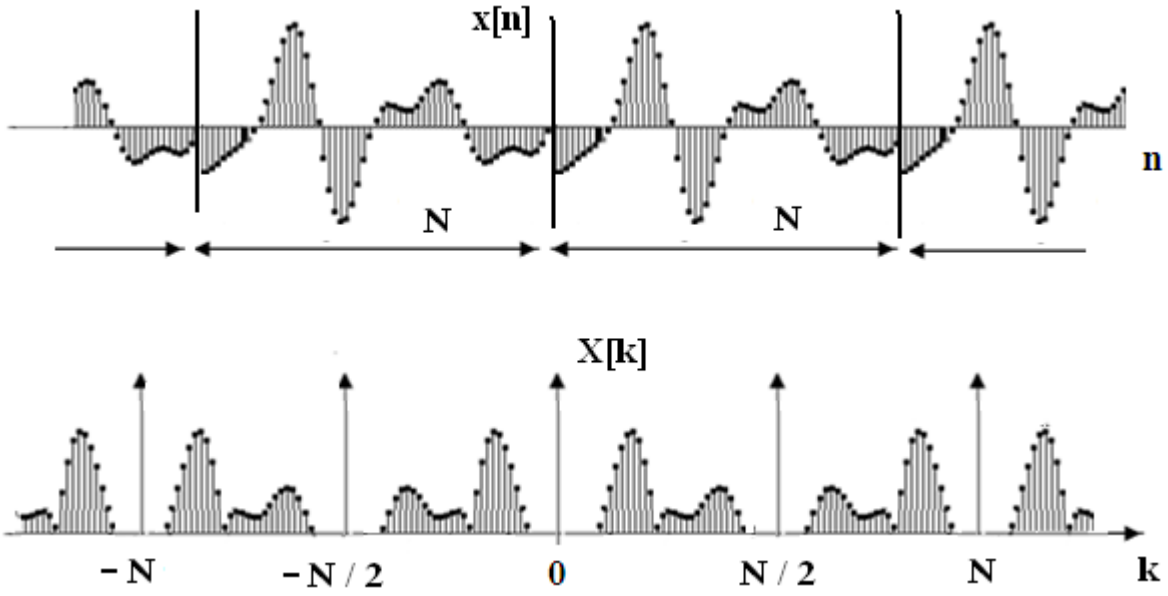

FIGURE 3.9. Discrete Fourier transform

Secondly, we sample uniformly the frequency domain at N points. The resulting frequency spectrum will contain N components:

$$\omega_k = \frac{2\pi k}{N}, \qquad k = 0,..,N-1 \tag{3.18}$$

The analysis formula now looks like:

$$X(e^{j\omega})\Big|_{\omega=\frac{2\pi k}{N}} = \sum_{k=-\infty}^{+\infty} x[n]e^{-j\omega n}\Big|_{\omega=\frac{2\pi k}{N}} = \sum_{k=0}^{N-1} x[n]e^{-jnk\frac{2\pi}{N}} = X[k] \tag{3.19}$$

The final formulae for *N-point* complex DFT and inverse DFT are:

$$\begin{cases} x[n] = \dfrac{1}{N}\displaystyle\sum_{k=0}^{N-1} X[k]e^{jnk\frac{2\pi}{N}} \\ X[k] = \displaystyle\sum_{n=0}^{N-1} x[n]e^{-jnk\frac{2\pi}{N}} \end{cases} \tag{3.20}$$

The *frequency resolution* of the N-point DFT is measured in Hz and is defined as:

$$\Delta = \frac{f_s}{N}, \tag{3.21}$$

where f_s is the sampling frequency. The frequency resolution is essential for interpretation of the DFT-spectrum. The k^{th} element in the DFT-coefficients array stands for frequency f_k whose value is:

$$f_k = k\Delta = \frac{kf_s}{N}, \qquad k = 0,..,N-1 \tag{3.22}$$

The Nyquist frequency ($f_s/2$) corresponds to the frequency index $k = N/2$. The magnitude $|X(k)|$ is an even function of k, therefore only a part of the spectrum for $0 \le k \le N/2$ (i.e. $0 \le \omega_k \le \pi$) is usually displayed.

Example 3.2.
Given a signal sampled at frequency 8000Hz. What is the frequency resolution of 256-point DFT of the signal? What frequency (in Hz) does the 50^{th} spectral sample represent?

Solution:
The frequency resolution is: $\Delta = 8000 / 256 = 31.25$ (Hz).
Therefore, the 50^{th} spectral sample represents the frequency:

$$f_{50} = 50 * 31.25 = 1562.5 \text{ (Hz)}.$$

Note that DFT is only an approximation of DTFT leading to specific effects that should be accounted for. We'll discuss these effects in the Section 3.4. Just keep in mind: during the analysis of signals in the frequency domain, *the DTFT is what we want to get, whereas the DFT is what we can compute*.

Thus, we've considered all members of the FT family. Notice the regularity in mutual nature of time and frequency domains. A periodic signal can be analyzed using a discrete frequency domain (the case of Fourier series and DFT). Dually, a discrete-time signal gives rise to a periodic frequency spectrum (the case of DTFT and DFT). Consequently, if a time signal is both discrete and periodic, a frequency spectrum is both periodic and discrete (DFT).

As we have already mentioned, the DFT is the only way computers can perform the Fourier transform. However, the DFT was not used in engineering for a long time because of its computational expensiveness. An algorithm based on a straightforward implementation of the formulae (3.20) has the time complexity of $O(N^2)$ and is too slow for real-time applications. A true revolution was made in 1965, with the inventing of *Fast Fourier transform algorithm (FFT)* by J.W.Cooley and J.Tukey. Currently, there are many different FFT algorithms

involving a wide range of mathematics: *Prime-factor FFT* algorithm, *Bruun's FFT* algorithm, *Rader's FFT* algorithm, and *Bluestein's FFT* algorithm. By far the most commonly used FFT is the Cooley–Tukey algorithm. This is radix-2 algorithm that has the time complexity of $O(N*\log_2 N)$. The term "radix-2" means that the algorithm performs N-point FFT for N a power of 2.

LET'S CODE!
generate 256 samples of a sine wave with frequency 1000Hz, A=2, and sampling rate of 8000 Hz. Plot the magnitude and phase spectrums.

```
N = 256; n = 0:N-1;
fs = 8000; f = 1000;
xn = 2*sin(2*pi*f/fs*n);     % generate sine wave
spec = fft(xn, N);           % Complex FFT
mag = 20*log10(abs(spec));   % magnitude spectrum
phase = angle(spec);         % phase spectrum
subplot(2, 1, 1);             % plot magnitude spectrum
plot(mag);
subplot(2, 1, 2);            % plot phase spectrum
plot(phase);
```

In the above code the commonly used *logarithmic scale* of *decibel (dB)* units is introduced. The decibel level is evaluated by the formula:

$$L_{dB} = 10\log\frac{P}{P_0} = 20\log\frac{A}{A_0}, \tag{3.23}$$

where P and A are the power and amplitude of signal, respectively; P_0 and A_0 are the reference power and reference amplitude of signal, respectively. The values of A_0 and P_0 are usually chosen in terms of particular practical purposes. Since decibels are a way of expressing the ratio between two signals, they are ideal for describing the gain of a system, i.e. the ratio between the output and the input signal. However, engineers also use decibels to specify the amplitude of a single signal, by referencing it to some standard. For example, the term: dBV means that the signal is being referenced to a 1 volt rms signal. Likewise, dBm indicates a reference signal producing 1 mW into a 600 ohms load (about 0.78 volts rms) [3]. In the previous code example we have just set A_0=1.

3.3. Fourier Transform Properties

The most important mathematical properties of the DTFT are given below. The operation of taking the Fourier transform is denoted as "$\overset{F}{\longleftrightarrow}$".

1) *Linearity*:

$$z[n] = ax[n] + by[n] \quad \xleftrightarrow{F} \quad Z(e^{j\omega}) = aX(e^{j\omega}) + bY(e^{j\omega}) \qquad (3.24)$$

2) *Time Shifting*:

$$y[n] = x[n-d] \quad \xleftrightarrow{F} \quad Y(e^{j\omega}) = e^{-j\omega d} X(e^{j\omega}) \qquad (3.25)$$

3) *Frequency Shifting*:

$$y[n] = e^{j\theta n} x[n] \quad \xleftrightarrow{F} \quad Y(e^{j\omega}) = X(e^{j(\omega-\theta)}) \qquad (3.26)$$

4) *The Convolution Theorem*:

$$z[n] = x[n] * y[n] \quad \xleftrightarrow{F} \quad Z(e^{j\omega}) = X(e^{j\omega})Y(e^{j\omega}) \qquad (3.27)$$

5) *The Modulation or Windowing Theorem*:

$$y[n] = x[n]w[n] \quad \xleftrightarrow{F} \quad Y(e^{j\omega}) = \frac{1}{2\pi}\int_{2\pi} X(e^{j\theta})\, W(e^{j(\omega-\theta)}) d\theta \qquad (3.28)$$

Eq.3.28 is a *periodic convolution*, i.e. a convolution of two periodic functions with the limits of integration extending over only one period. Notice: the Modulation theorem is the dual version of the Convolution theorem. These two theorems have tremendous importance in Fourier analysis of signals. The Convolution theorem essentially means that a convolution of signals in time domain can be replaced with a multiplication of their Fourier transforms, while the Modulation theorem implies that a multiplication of signals in time domain is equivalent to a convolution of their Fourier transforms.

6) *Parseval's theorem*:

$$E = \sum_{n=-\infty}^{+\infty} |\, x[n]\,|^2 = \frac{1}{2\pi}\int_{2\pi} |\, X(e^{j\omega})\,|^2 \, d\omega \qquad (3.29)$$

Parseval's theorem states that the time and frequency domains have the same energy *E*.

7) *Symmetry* of DTFT means that the real part of DTFT is an even function, and the imaginary part of DTFT is an odd function (if *x[n]* is real; if *x[n]* is complex, the corresponding formulae are slightly different):

$$X_{RE}(e^{j\omega}) = X_{RE}(e^{-j\omega}) \tag{3.30}$$

$$X_{IM}(e^{j\omega}) = -X_{IM}(e^{-j\omega}) \tag{3.31}$$

Hence, the magnitude DTFT-spectrum is symmetric as well, so we can consider only one half of the magnitude spectrum (frequencies from 0 to π, or from 0 Hz to $f_s/2$ Hz). The other half (corresponding to *negative frequencies*) repeats the first half. It's easy to see that the Eq.3.30 and Eq.3.31 agree well with the sampling theorem.

Now, let's recall the process of signal sampling covered in the Section 2.2, and analyze it using the Fourier transform. In other words, let's consider how the *sampling* process affects the continuous signal in *frequency domain*.

An ideal sampler measures the continuous input signal $x_c(t)$ at the sampling instants $t=nT_s$. The *continuous* output signal $x_s(t)$ of the sampler can be considered as the signal $x_c(t)$ modulated with the periodic *impulse train* $\delta(t-nT_s)$:

$$x_s(t) = \sum_{n=-\infty}^{+\infty} x_c(nT_s)\,\delta(t-nT_s) \tag{3.32}$$

Notice that we *do not* transform a continuous signal into a discrete-time signal. What we do is representing the signal $x_c(t)$ in terms of another *continuous* signal $x_s(t)$. The signal $x_s(t)$ is zero everywhere, except at integer multiples T_s, and therefore, it looks like a discrete-time signal in some sense.

The Fourier transform of a periodic impulse train is also periodic impulse train:

$$S(\Omega) = \frac{2\pi}{T}\sum_{k=-\infty}^{+\infty}\delta(\Omega - k\Omega) \tag{3.33}$$

Since, $x_s(t)$ is a multiplication of the signal $x_c(t)$ by $\delta(t-nT_s)$, according to the Eq.3.28, the spectrum of signal $x_s(t)$ is the convolution of spectra:

$$X_s(\Omega) = \frac{1}{2\pi}X_c(\Omega) * S(\Omega) = \frac{1}{T}\sum_{k=-\infty}^{+\infty}X_c(\Omega - k\Omega_s) \tag{3.34}$$

The Fig.3.10 depicts the frequency domain representation of sampling.

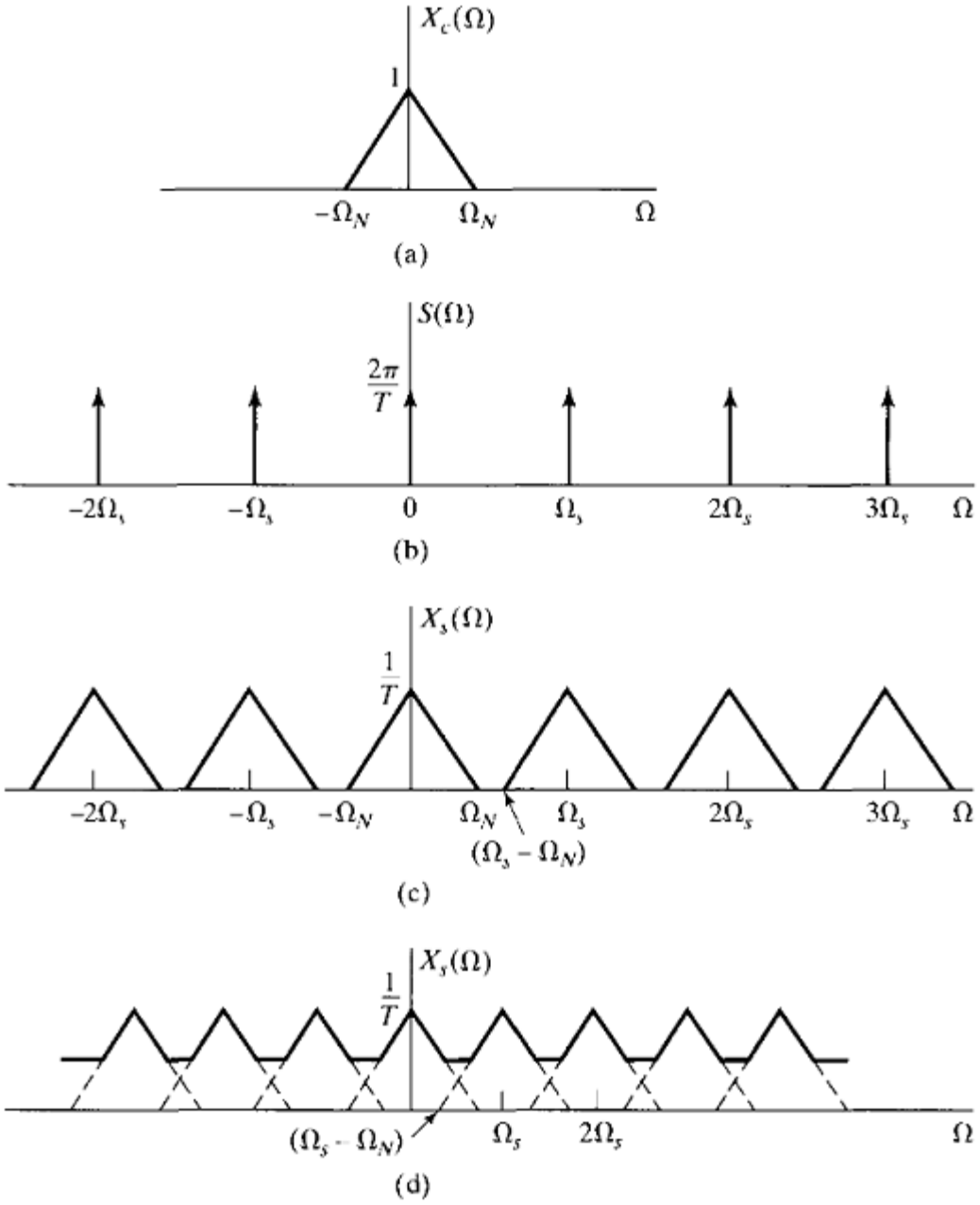

FIGURE 3.10. Effect in the frequency domain of sampling in the time domain: a) spectrum of the original signal; b) spectrum of the impulse train; c) spectrum of the sampled signal (no aliasing); d) aliased spectrum

As we can see in Fig.3.10, when

$$\Omega_s > 2\Omega_{\max} \tag{3.35}$$

the replicas of $X_c(\Omega)$ don't overlap in the spectrum $X_s(\Omega)$. Therefore, $x_c(t)$ can be recovered from $x_s(t)$ after the low-pass filtering (i.e. suppressing all frequencies higher than Ω_{max}). If the inequality of Eq.3.35 doesn't hold, the replicas of $X_c(\Omega)$ overlap and are added together, so it's impossible to recover $x_c(t)$.

Thus, we've come up again to the *aliasing* issue. The inequalities (2.6) and (3.35) are identical. We have demonstrated that the frequency domain provides another useful way to understand the aliasing problem.

And the last important fact about Fourier transform to be discussed in this section is that it connects together the two main characteristics of LTI systems: the impulse response and the frequency response.

FREQUENCY RESPONSE
of an LTI system is the FOURIER TRANSFORM of its IMPULSE RESPONSE.

Example 3.3.
Show that frequency response is FT of impulse response.
Solution:
Substituting the Eq.3.3 into the DTFT analysis formula (3.16), we get:

$$h^*[n] = \frac{1}{2\pi}\int_{2\pi} H(e^{j\omega})e^{j\omega n}d\omega = \frac{1}{2\pi}\int_{2\pi}\sum_{k=-\infty}^{+\infty} h[k]e^{-j\omega k}\, e^{j\omega n}d\omega \qquad (3.36)$$

Interchanging the summation with the integration gives:

$$h^*[n] = \frac{1}{2\pi}\sum_{k=-\infty}^{+\infty} h[k]\int_{2\pi} e^{j\omega(n-k)}d\omega = h[n] \qquad (3.37)$$

since $\int_{2\pi} e^{j\omega(n-k)}d\omega = \begin{cases} 2\pi, & n=k \\ 0, & n\neq k \end{cases}$.

Thus:

$$h[n] \quad \overset{F}{\longleftrightarrow} \quad H(e^{j\omega}) \qquad (3.38)$$

3.4. Spectral Analysis of Signals

One of the major DFT applications is the analysis of the frequency content of relatively long or even indefinitely long signals, also referred to as the *Fourier analysis*, or *spectral analysis* of signals. Fourier analysis is most widely used for non-stationary signals such as music, speech, radar signals, etc. Nevertheless, the spectral analysis can also be applied to stationary signals. In this case it's often called the *periodogram analysis*. The issues regarding the periodogram analysis are beyond the scope of this guide.

The spectral analysis is based on the following cornerstones:

1) *block decomposition*;
2) *windowing*;
3) *short-time Fourier transform (STFT)*.

Suppose we need to analyze the frequency content of some sound signal coming from a microphone. We can't perform the DFT of the entire signal for at least two reasons: 1) the signal is incomplete, i.e. only its current part is available for analysis; 2) even assuming the signal is complete, the resulting spectrum will be noisy, since it will contain all signal frequency components, including those that capture the slowly varying changes of signal.

The alternative and the right way to analyze the frequency content of a non-stationary signal is to decompose it into short blocks and process these blocks one after another. In general case, blocks may overlap (Fig.3.11).

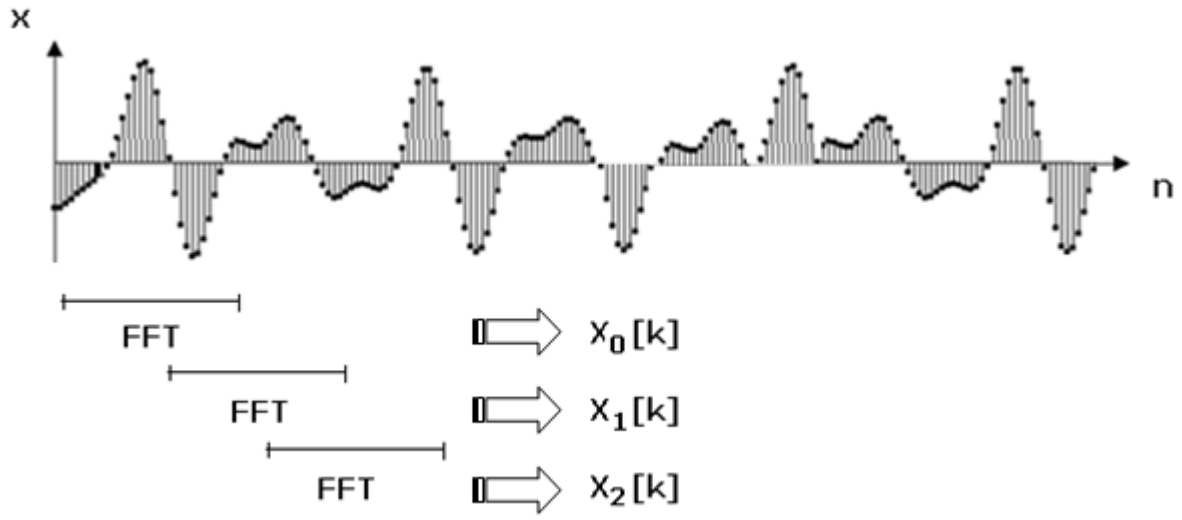

FIGURE 3.11. Block decomposition of signal

Extracting each block from the signal is equivalent to multiplication of the signal by the *rectangular window function*, or *rectangular pulse*:

$$w[n] = \begin{cases} 1, & n \le N \\ 0, & n > N \end{cases}, \tag{3.39}$$

where N is the block length. This is called the *windowing*. According to the Eq.3.28, in the frequency domain it leads to the periodic convolution:

$$y[n] = x[n]w[n] \quad \overset{F}{\longleftrightarrow} \quad Y(e^{j\omega}) = \frac{1}{2\pi}\int_{2\pi} X(e^{j\theta})\, W(e^{j(\omega-\theta)})d\theta, \tag{3.40}$$

where $W(e^{j\omega})$ is the Fourier image of the rectangular window.

Example 3.4.
Find the Fourier transform of the rectangular pulse.

Solution:
Substituting the Eq.3.3 into the DTFT analysis formula (3.16), we get:

$$W(e^{j\omega}) = \sum_{n=0}^{N-1} e^{-j\omega n} = \frac{1-e^{-j\omega(N-1)}}{1-e^{-j\omega}} \tag{3.41}$$

Since

$$\sin(\omega/2) = \frac{e^{j\omega/2} - e^{-j\omega/2}}{2j} = \frac{1-e^{-j\omega}}{2j\, e^{j\omega/2}} \tag{3.42}$$

Eq.3.41 can be rewritten as:

$$W(e^{j\omega}) = e^{-j\omega(N-1)/2} \frac{\sin(\omega N/2)}{\sin(\omega/2)} \approx Ne^{-j\omega(N-1)/2} sinc(fN) \tag{3.43}$$

where *sinc(x)=sin(πx)/ πx* is the so called *Sinc* function. The sinc function and the rectangular pulse are one of the examples of *Fourier transform pairs*.

Thus, the resulting DTFT-spectrum $Y(e^{j\omega})$ will be distorted. The spectrum $X(e^{j\omega})$ will be convolved with the spectrum of the rectangular pulse, and the two types of windowing effects will take place: the *spectral leakage* and the *reduced resolution*. The resolution is influenced primarily by the width of the main lobe of $W(e^{j\omega})$, while the degree of leakage depends on the relative amplitude of the main lobe and the side lobes of $W(e^{j\omega})$.

So far we've analyzed the effects of signal block decomposition in context of the DTFT because it's the DTFT that shows us how the frequency spectrum really looks. However, actually we compute the *DFT*. The DFT of the finite-length signal *y[n]* corresponds to equally spaced samples of the DTFT-spectrum:

$$Y[k] = Y(e^{j\omega})\big|_{\omega=\frac{2\pi k}{N}} \tag{3.44}$$

Hence, we shouldn't be misled by the results of this spectral sampling. Let's demonstrate the effects of windowing and spectral sampling by the following example (taken from [1]). Let's consider a signal containing two cosine waves:

$$x[n] = \cos(\frac{2\pi}{16}n) + 0.75\cos(\frac{2\pi}{8}n) \qquad (3.45)$$

Assume we take 64-point DFT (that is, we apply the rectangular window of length N=64). The resulting DFT-spectrum is shown in Fig.3.12b. As we can see, the DFT-spectrum *Y[k]* has two strong spectral lines at the frequencies of the two sinusoidal components in the signal and no content at the rest of the frequencies. However, the actual frequency content of signal is represented by the DTFT-spectrum shown in Fig.3.12c. The reason for such clean appearance of DFT is that the frequencies of the cosine components coincide exactly with two frequencies of 64-point DFT: the frequency $2\pi/16$ corresponds exactly to the DFT sample k=4, and the frequency $2\pi/8$ to the DFT sample k=8. Another way to explain this is to note that, since the signal frequencies are all multiples of $2\pi/64$, the signal is periodic with period N=64. Therefore, 64-point rectangular window selects exactly one period of the signal.

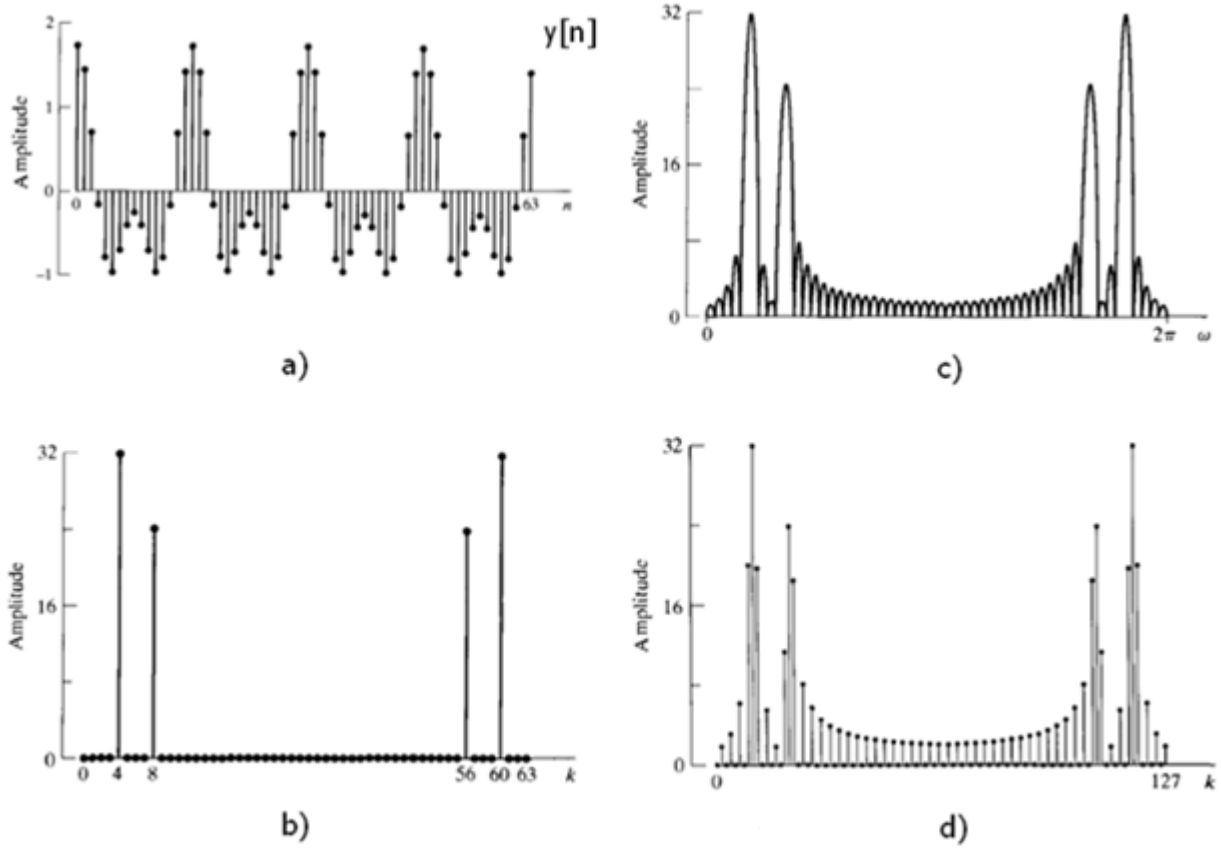

FIGURE 3.12. Windowing effects: a) original signal; b) 64-point DFT-spectrum; c) DTFT-spectrum; d) 128-point DFT-spectrum

If we extend *y[n]* by zero-padding to obtain 128-point sequence, there will be more samples in the DFT-spectrum, and the presence of significant content at other frequencies will become apparent (Fig.3.12d). We have used a common

technique in Fourier analysis: *zero-padding*. It can be easily shown that adding any number of zeros to the end of the signal doesn't affect the resulting DFT-spectrum. It just adds new samples into spectrum, making the frequency sampling finer.

Can we get rid of the windowing effects? No, we can't. But we can reduce them. The solution would be to replace the rectangular window with another specific function called the *window function*. But before we consider various window functions let's find out what's "wrong" with the rectangular pulse.

The rectangular window doesn't change any block of signal, so if the DFT size doesn't coincide with the period of the block (it happens in almost all cases), there will be discontinuities at the "transitions" between adjacent periods. How the Fourier transform will deal with discontinuous signals? From the Fourier theory it is known that the sufficient condition for the existence and uniform convergence of the Fourier transform representation of signal is the absolute summability:

$$\sum_{n=-\infty}^{+\infty} |\, x[n]\,| < \infty \tag{3.46}$$

Discontinuous signals are *not absolutely* summable. However, if the number of discontinuities in one period is finite (like in the case with rectangular window), they are *square* summable (the convergence in *mean-square sense*):

$$\lim_{m\to\infty} \int_{-\pi}^{\pi} |X(e^{j\omega}) - X_M(e^{j\omega})|^2 \, d\omega = 0 \tag{3.47}$$

$$X_M(e^{j\omega}) = \sum_{n=-M}^{M} x[n] e^{-j\omega n} \tag{3.48}$$

The Eq.3.47 means that error $|\, X(e^{j\omega}) - X_M(e^{j\omega})\,|$ may not approach zero at each frequency as M→∞, but the total "energy" in the error does. The Eq.3.48 reflects the idea of signal reconstruction from *M* sinusoidal components. The Fig.3.13 illustrates this for various values of *M*.

Note that there are always oscillations at the discontinuity points. As *M* increases the oscillations become more frequent and the width of the overshoot decreases. However, the size of the ripples does not decrease (their amplitude is equal to approximately 9% of the total amplitude). This phenomenon is known as the *Gibbs effect (Gibbs phenomenon)*. Although the functions $X(e^{j\omega})$ and $X_M(e^{j\omega})$ differ only at discontinuity points, and the energy of difference tends to zero, the Gibbs effect has important implications in the filter design.

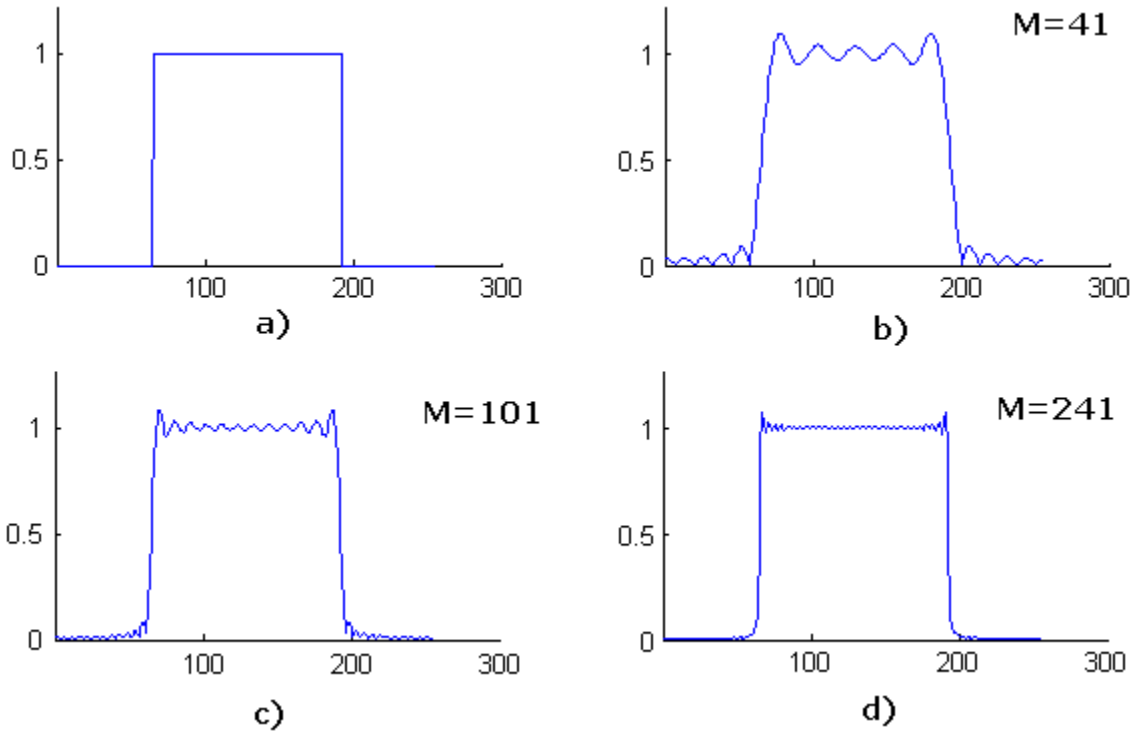

FIGURE 3.13. The illustration of Gibbs effect: a) initial signal; b) signal reconstructed from M=41 FT samples; c) M=101; d) M=241

In order to overcome the discontinuity problems, the numerous *window functions* are applied. These functions differ from the rectangular window because their values are close to zero at each end of the window. Thus, the periodically repeated signal multiplied by the window function will not contain discontinuities between adjacent periods, and its DTFT-spectrum will contain lower side lobes. Nevertheless, the rectangular pulse has one advantage over other window functions: it has the narrowest width of its main lobe for a given length (but it has the largest side lobes of all commonly used windows).

The most widely used window functions are given in Table 3.1.

The concepts related to spectral analysis reviewed in this section had underlain the special transform called the *Short-time Fourier transform (STFT)*, or *Time-dependent Fourier transform*.

SHORT-TIME FOURIER TRANSFORM (STFT)
is a Fourier-related transform used to determine the frequency and phase content of local sections of a signal as it changes over time.

Mathematically, the STFT can be expressed as:

$$X(m,\omega) = \sum_{n=-\infty}^{+\infty} x[n+m]w[m]\,e^{-j\omega m} \qquad (3.49)$$

TABLE 3.1. Window Functions

Name	Formula for *w[n], n = 0..M–1*	Width of main lobe	Peak side-lobe amplitude (relative)
Rectangular	$w[n]=1$	4π / (M+1)	-13 dB
Bartlett	$w[n]=\begin{cases} 2n/M, & 0 \le n \le M/2 \\ 2-2n/M, & M/2 < n < M \end{cases}$	8π / M	-25 dB
Hamming	$w[n]=0.54-0.46\cos(\frac{2\pi}{M}n)$	8π / M	-31 dB
Hanning	$w[n]=0.5-0.5\cos(\frac{2\pi}{M}n)$	8π / M	-41 dB
Blackman	$w[n]=0.42-0.5\cos(\frac{2\pi}{M}n)+0.08\cos(\frac{4\pi}{M}n)$	12π / M	-57 dB

The graph of a 3-dimensional power spectrum $|X(m,\omega)|^2$ of the STFT is called the *spectrogram* of the signal. The spectrogram uses the x-axis to represent time and the y-axis to represent frequency. The gray level (or color) at point (m, k) is proportional to $|X(m,\omega)|^2$. The MATLAB Signal Processing Toolbox provides a function *specgram()* to compute spectrogram.

LET'S CODE!
Plot the spectrogram of the signal contained in file "d:\1.wav".

```
[x,fs]=wavread('d:\1.wav');
N = 256;
Olap = N / 4;

% Plot spectrogram
specgram(x, N, fs, blackman(N), Olap );
```

In conclusion, the primary purpose of the window in the STFT is to limit the extent of the signal to be transformed so that the spectral characteristics are stationary over the duration of the window. The more rapidly signal characteristics change, the shorter the window should be. However, as the window becomes shorter, frequency resolution decreases. On the other hand, as the window length decreases, the ability to resolve changes in time increases. Consequently, the choice of window length is trade-off between frequency resolution and time resolution [1].

3.5. Fast Convolution

The Convolution theorem (Eq.3.27) states that the convolution of two signals in time domain can be replaced with the multiplication of their spectra in frequency domain. This is the basic idea behind the so called *Fast convolution, or FFT convolution* technique.

Fast convolution is quite easy to understand and to implement. However, we must be careful with this technique. The Eq.3.27 isn't fraught with danger, but the problem is that we essentially use the DFT for computations, not DTFT. Moreover, the certain size of the DFT must be set. We should remember that an N-point DFT views the time domain as being an infinitely long periodic signal, with N samples per period, and the result of DFT also contains N samples. According to the Section 2.5, the convolution of an N-point signal with an M-point signal results in an $N+M-1$ point output signal. Therefore, we must take care that the size of the DFT is not less than the size of the convolution. For example, if we want to convolve a 256-point signal with a 100-point signal, it would be wrong to apply a 256-point DFT to both signals because the convolution contains 355 samples! Hence, the size of DFT should be at least N=355. In most cases the radix-2 FFT algorithm is used, so the nearest power of 2 must be taken: i.e. N=512. In order to form a 512-point input signal, we can simply apply the zero-padding technique discussed in the previous section.

What will happen if we don't follow the above-mentioned conditions? The resulting signal will be a distorted version of the correct signal because the so called *Circular convolution* will take place. If we try to evaluate N_1-point convolution using N_2-point DFT ($N_1>N_2$), then the last ($N_2–N_1$) samples of the resulting *actual* convolution will be overlapped and added to the first ($N_2–N_1$) samples of the DFT-spectrum. This process is illustrated in Fig.3.14.

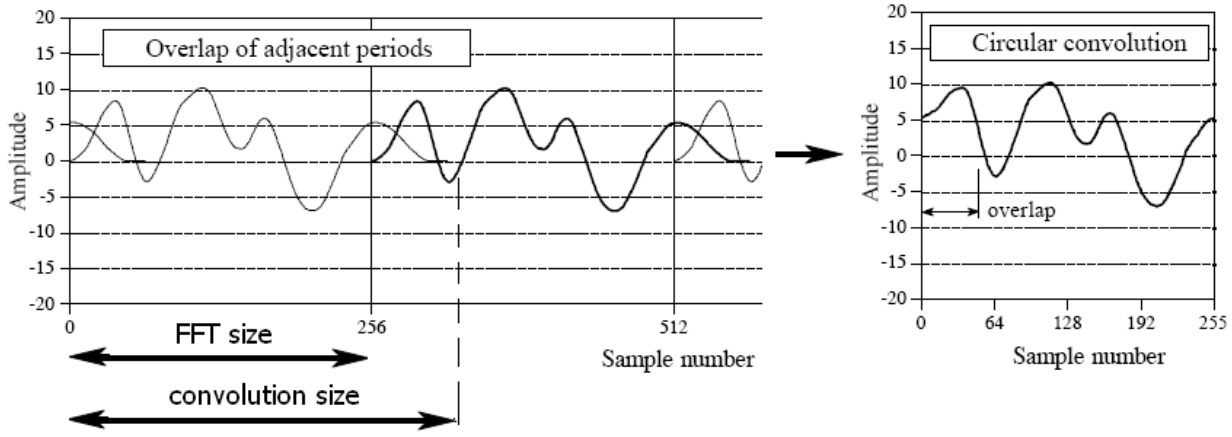

FIGURE 3.14. The illustration of a circular convolution

A block diagram of the FFT convolution is shown in Fig.3.15. Pay attention to the multiplication block. Don't forget that we're dealing with complex numbers!

A computation of convolution via the frequency domain can be useful for at least two reasons:

1) Execution speed in most practical cases is much higher (as have been mentioned in the Section 2.5).
2) It's possible to carry out signal *deconvolution*. Deconvolution is a process used to reverse the effects of convolution. For example, consider an LTI system that convolves the input signal *x[n]* with the impulse response *h[n]*: *y[n]=x[n]*h[n]*. Suppose we have information about the impulse response *h[n]* of the system, and we observe the output signal *y[n]*. Our task is to obtain the input signal *x[n]*. It's practically impossible to solve this task in time domain. However, in frequency domain we can simply apply the complex division $X(e^{j\omega}) = Y(e^{j\omega}) / H(e^{j\omega})$. After performing an inverse DFT, we'll obtain the required signal *x[n]*.

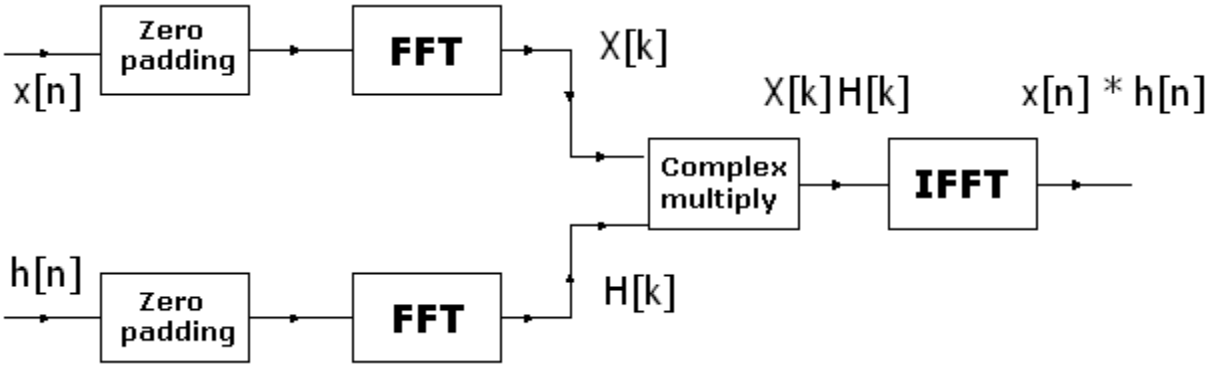

FIGURE 3.15. The FFT convolution

LAB2	SIGNAL PROCESSING IN FREQUENCY DOMAIN points: 15 hours: 4

Objectives

1. To understand the concepts of the DTFT and DFT.
2. To use FFT for spectral analysis and convolution of signals.
3. To demonstrate the Gibbs effect and windowing effects.

Exercise 2.1 [1 point]

Answer the following questions:

a. What is the matter of Fourier transform?
b. What are the Complex Discrete Fourier transform (DFT) and Inverse Complex Discrete Fourier transform (IDFT)?
c. What are magnitude spectrum and phase spectrum of a signal?
d. What does the Convolution theorem state?
e. How is FFT Convolution carried out?
f. What is circular convolution? How to avoid it?
g. What is deconvolution?
h. What is frequency response of a system?
i. How is sinc function related to a rectangular pulse?
j. What is the purpose of window functions? Give the equations of Hanning window, Blackman window, Hamming window, Kaiser window.

Exercise 2.2 [2 points]

Given a signal sampled at frequency 16000 Hz. The FFT is performed over N = 2048 samples.

a. Determine the length of analyzed block in milliseconds.
b. Determine the frequency resolution.

Exercise 2.3 (CODE) [3 points]

In this exercise you will perform simple spectral analysis of signals. Write code to do the following:

1. Open WAVE files containing signals *s1*, *s2*, *s3*, *s4*, *sn* from exercise 1.3. You will also be given speech file LAB2_SPEECH.wav containing signal *s5*. Compute and plot magnitude and phase spectrums of each signal (use FFT size $N = fs$ so that you'll analyze 1 sec of a signal). What can you say about the magnitude spectrum of signal *s4*? Is frequency *f3* present there? Explain your results.
2. Perform the Inverse FFT. Plot your results. Compare obtained signal with source signal.
3. Plot the spectrogram of each signal.

Exercise 2.4 (CODE) [3 points]

In this task you will investigate various window functions. Write code to do the following:

1. Compute and plot the rectangular, Hamming, Hanning, Blackman, and Kaiser window functions of length N = 93 on a single figure.
2. Compute and plot magnitude spectrum on a dB scale of each window. Use FFT size NF=512. What can you say about rectangular window?

Exercise 2.5 (CODE) [3 points]

In this task you will investigate the effect of windowing. Use 1024 samples for FFT. Write code to do the following:

1. Compute the DFT of the signal $x(n) = cos(\pi n/4)$. Use sampling frequency fs = 4096 Hz. Plot the magnitude and phase spectrum of $x(n)$.
2. Truncate $x(n)$ using Hamming window of size 93. Plot its magnitude and phase spectrum. Obtain the 512-point signal $z[n]$ by zero-padding $x[n]$. Plot its magnitude and phase spectrum.
3. Repeat steps 1-2 for signals *s1*, *s2*, *s3*, and *sn*.

Exercise 2.6 (CODE) [3 points]

Write code that carries out FFT convolution of two signals. Plot the result. Compare the result with your solution of exercise 1.4.

CHAPTER 4 SIGNAL FILTERING

4.1. Basics of Digital Filtering

(FREQUENCY-SELECTIVE) DIGITAL FILTER
is a discrete-time system whose main purpose is to modify certain frequencies relative to others.

Digital filters are used for two general purposes [3]:

1) *separation of signals* that have been combined. Signal separation is needed when a signal has been contaminated with interference, noise, or other signals. For example, imagine a device for measuring the electrical activity of a baby's heart while still in the womb. The raw signal will likely be corrupted by the breathing and heartbeat of the mother. A filter might be used to separate these signals so that they can be individually analyzed;

2) *restoration of signals* that have been distorted in some way. For example, an audio recording made with poor equipment may be filtered to better represent the sound as it actually occurred. Another example is the de-blurring of an image acquired with an improperly focused lens, or a shaky camera.

Analog filters can be used for the same tasks; however, digital filters can achieve far superior accuracy and precision. The entire process of digital filtering includes: 1) Analog-to-Digital conversion; 2) the filtering of the digital signal; 3) Digital-to-Analog conversion of the filtered digital signal.

The main characteristics of digital filters are:

1) Impulse response; 2) Frequency response; 3) Step response.

We're already familiar with the former two characteristics of discrete-time systems, while the *step response* has not been previously discussed.

STEP RESPONSE
is an output signal of a system when the input signal is a unit step.

The step response is very useful in time domain analysis because it matches the way humans view the information contained in signals, representing a division between the dissimilar regions of a signal. A combination of the step functions divides the signal information into regions of similar characteristics. The step response, in turn, is important because it describes how the dividing lines are being modified by the filter.

Since the unit step is the sum of unit impulses (Eq.2.14), the step response is the sum of the impulse response values. This provides two ways to find the step response: 1) feed a unit step into the filter and store the output result; 2) calculate the running sum of the impulse response.

The most important parameters of the step response are *duration* (or the *rise-time*), and the *overshoot* (Fig.4.1). If the duration is long (the rise-time is slow), the

events in a signal are distinguished poorly. The overshoot changes the amplitude of signal samples, and, therefore, in most cases it's not desirable. Hence, the *short* step responses *without* overshoot are the best.

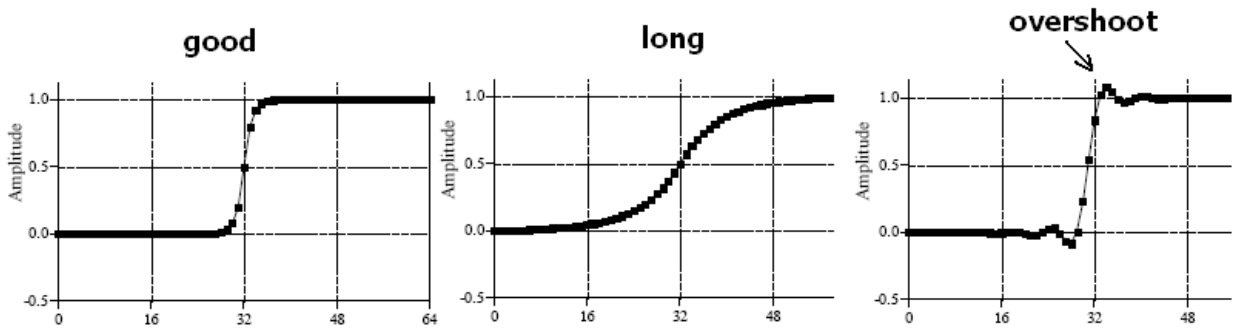

FIGURE 4.1. Step response

The relation between the main filter characteristics is shown in Fig.4.2

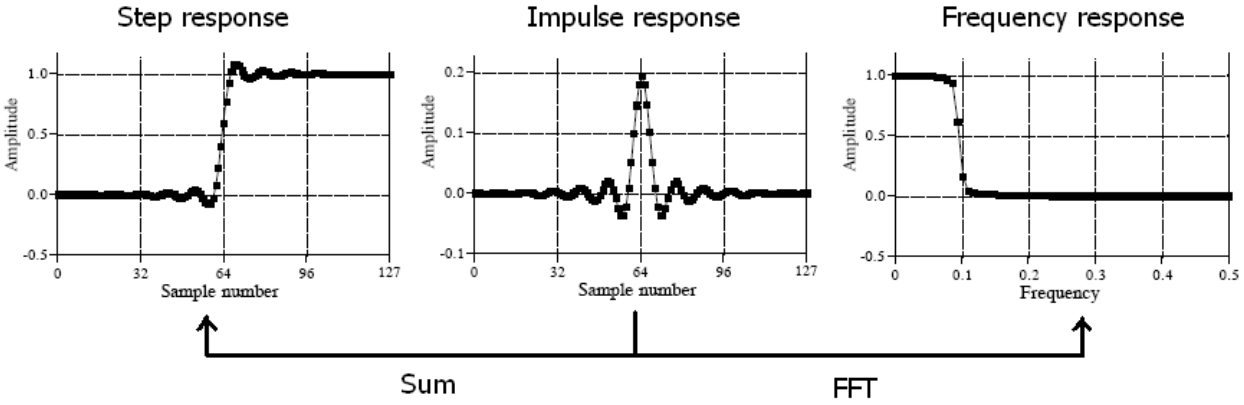

FIGURE 4.2. Step response, impulse response and frequency response

The impulse response of a digital filter is also referred to as the *filter kernel*.

FILTER KERNEL
is the impulse response of a digital filter.

Since we're talking about the impulse response, we should mention an important nuance regarding causality of filters. Remember that the output of a causal LTI system depends only on the present and previous samples of the input (See Section 2.4). If we implement a real-time DSP system, we can operate only with the data we already have, and therefore, the causal filters are the only option. On the other hand, if the full amount of data is available (e.g., fully loaded image), we may use non-causal filters for signal processing. In terms of impulse response, a digital system is called the causal system if and only if:

$$h[n]=0,\ n<0 \qquad (4.1)$$

Probably, the most intuitively understandable characteristic of frequency-selective digital filters is a frequency response, since it is the frequency response that describes filter behavior in the frequency domain (i.e., what frequencies of an input signal should be changed and how). Ideal filters have a perfect rectangular shape. In practice, however, it's impossible to achieve such shape. Figure 4.3 illustrates the difference between the actual low-pass filter and ideal low-pass filter, and introduces the parameters of actual filter's frequency response.

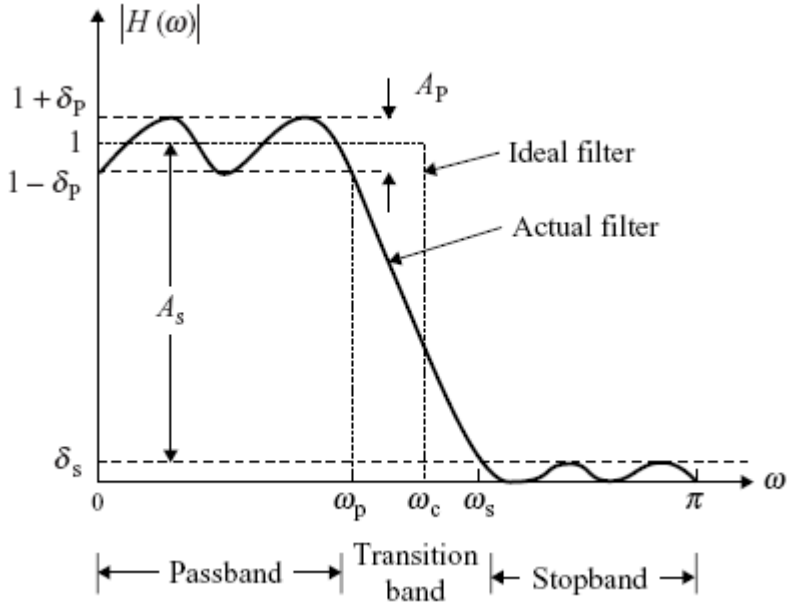

FIGURE 4.3. Frequency response of a common low-pass filter

The *pass-band* refers to those frequencies that are passed, while the *stop-band* contains those frequencies that are blocked. The *transition band* is between. A fast *roll-off* means that the transition band is very narrow. The division between the pass-band and transition band is called the *cutoff frequency*. The *pass-band ripple* δ_p (measured in dB) is the allowed variation in magnitude response in the pass-band. The *stop-band ripple* (or *stop-band attenuation*) δ_s describes the minimum attenuation for signal components above the *normalized stop-band cutoff frequency* ω_s.

The digital filters can be classified according to the following criteria:

1) The shape of frequency response:
   - **Low-pass filters**
   - **High-pass filters**
   - **Band-pass filters**
   - **Band-reject filters**
   - **Custom**
2) The type of impulse response:
   - **FIR (Finite Impulse Response) filters**
   - **IIR (Infinite Impulse Response) filters**

3) The shape of phase response:

- **Zero phase filters**
- **Linear phase filters**
- **Nonlinear phase filters**

The FIR and IIR filters will be reviewed in the next section.

The frequency responses of the low-pass, high-pass, band-pass, and band-reject filters are shown in Fig.4.4. The names of these filters stand for themselves. Usually, only examples of the low-pass filters are considered, since the filters of other types can be combined from the low-pass filters (see Section 5.4).

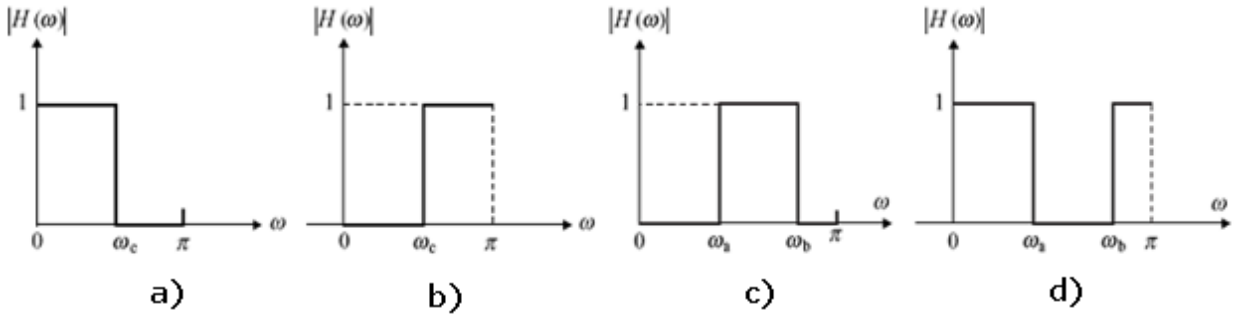

FIGURE 4.4. Filters: a) low-pass; b) high-pass; c) band-pass; d) band-reject

The examples of *zero phase*, *linear phase* and *nonlinear phase* are given in Fig.4.5.

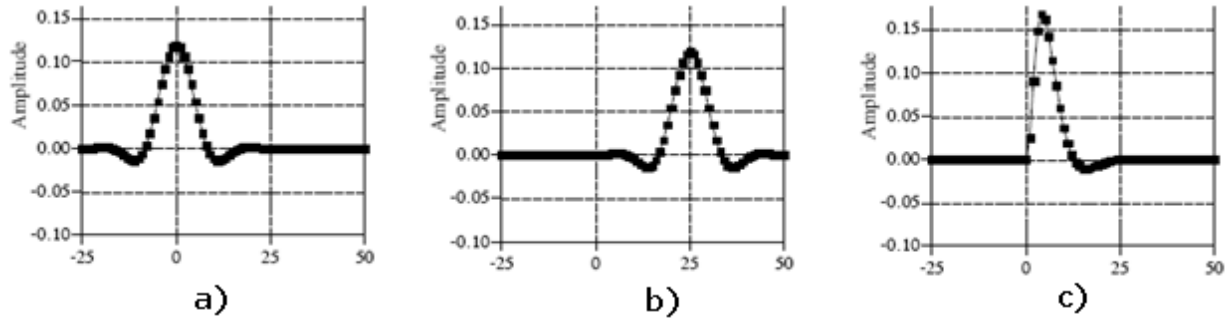

FIGURE 4.5. Filter kernels: a) zero phase; b) linear phase; c) nonlinear phase

The impulse response of zero phase filter is symmetrical around sample zero *h[0]*. The phase response of such filter is entirely zero. The symmetry of filter kernel is very useful property, however, a zero phase filter contains the negative numbered samples, i.e., it is non-causal. Contrarily, a linear phase filter is causal. Its filter kernel is also symmetrical, but the center of symmetry lies in the positive middle sample (the length of the kernel must be an odd number). The phase response of such filter is a straight line. Since the shift in the impulse response simply produces an identical shift in the output signal, the linear phase filter is

equivalent to the zero phase filter for most purposes. Lastly, the nonlinear phase filters are characterized by an asymmetric impulse response.

The linear phase filters are often desirable because nonlinear phase can have a major effect on the shape of a signal even when the frequency response magnitude is constant. For example, the result of an image smoothing may appear asymmetric, if the nonlinear phase filter is applied.

4.2. Digital Filters in Time Domain

Just as continuous-time systems are described by the differential equations, discrete-time systems are described by the *difference equations*:

$$y[n] = \sum_{k=0}^{N} a_k x[n-k] - \sum_{m=1}^{M} b_m y[n-m] \tag{4.2}$$

The Eq.4.1 consists of two parts: the *recursive part* $\sum_{m=1}^{M} b_m y[n-m]$, and the *non-recursive part* $\sum_{k=0}^{N} a_k x[n-k]$. If the recursive part with feedback coefficients is present in filter's description, then the filter is called the *Infinite Impulse Response (IIR)* filter. Otherwise, the filter is called the *Finite Impulse Response (FIR)* filter.

Where does the name "Infinite Impulse Response filter" come from? In order to answer this question, we'll consider example 4.1.

Example 4.1.
Find the impulse response of the following filter (assuming it's causal, and all coefficients are nonzero):

$$y[n] = a_0 x[n] - \sum_{m=1}^{M} b_m y[n-m] \tag{4.3}$$

Solution:
Let's feed a unit impulse to the filter ($x[n] = \delta[n]$) and see what the output result will be:
$h[0] = a_0x[0] – b_1h[-1] = a_0$
$h[1] = a_0x[1] – (b_1h[0] + b_0h[-1]) = −b_1h[0]$
$h[2] = a_0x[2] – (b_2h[1] + b_1h[0] + b_0h[-1]) = −(b_2h[1] + b_1h[0])$
...

In general, $h[n] = a_0\delta[n] - \sum_{m=1}^{M} b_m h[n-m]$, 0<n<∞. Consequently, this filter has *infinite* impulse response h[n]. Furthermore, any filter that requires the feedback coefficients is IIR, since each sample of its impulse response is nonzero, and computed from the previous samples of its impulse response.

The *Moving Average (Running Average)* filter is a simple example of *FIR* filter. This filter is often applied for signal smoothing because it reduces random noise while retaining the sharpest step response. *L-point* moving average filter is described by the following difference equation:

$$y[n] = \frac{1}{L}\sum_{k=0}^{L-1} x[n-k] \tag{4.4}$$

Example 4.2.
Find the impulse response of the moving average filter.

Solution:
Let's feed a unit impulse to the filter (x[n] = δ[n]) and see what the output result will be:

$$h[n] = \frac{1}{L}\sum_{k=0}^{L-1}\delta[n-k] = \begin{cases} \frac{1}{L} & , n = 0,1,2,..,L-1 \\ 0 & , n \geq L \end{cases} \tag{4.5}$$

Since, *h[n]<>0* only for *n<L*, *h[n]* is, indeed, finite.
If we take a closer look at the algorithm of moving average filter, we'll see that most of the samples of moving window at each step of computations are the same. Only the "oldest" sample is replaced with the new one. The same output signal as in Eq.4.4, can be obtained by the following recursive formula:

$$y[n] = y[n-1] + \frac{1}{L}(x[n] - x[n-L]) \tag{4.6}$$

The formula (4.6) is much faster and looks like a description of an IIR filter, but don't be confused! Although the moving average filter *can be* implemented recursively, it is characterized by the finite impulse response!

The relatives of moving average filter are *Gaussian* and *Blackman* filters. They perform slightly better in the frequency domain because they have better stop-band attenuation. Nevertheless, none of them work well as *frequency-selective filters* (we need to use other types of filters for this purpose; they will be discussed in the next chapter). The impulse response of Blackman filter is the Blackman window function given in Table 3.1. The filter kernel of Gaussian filter is pure Gaussian.

Although the filter design is the subject of the next chapter, we'll consider in this section some equations that can already be used for simple IIR filters [3] (without mathematical derivation). The simplest IIR filter is the filter that has only one recursive coefficient. It's called the *single pole* filter (Section 5.2 of the guide explains what is "pole"):

$$y[n] = \sum_{k=0}^{N} a_k x[n-k] - b_1 y[n-1] \tag{4.7}$$

Single pole *low-pass filter* has only two coefficients calculated by the following formulae:

$$a_0 = 1 - e^{-\omega_c} = 1 - e^{-2\pi f_c / f_s} \tag{4.8}$$

$$b_1 = -e^{-\omega_c} = -e^{-2\pi f_c / f_s}, \tag{4.9}$$

where f_c is the cut-off frequency (in Hz), f_s is the sampling frequency (in Hz), and ω_c is the digital cut-off frequency (in radians per sample). Single pole *high-pass filter* has three coefficients calculated by the formulae:

$$a_0 = (1 + e^{-\omega_c}) / 2 \tag{4.10}$$

$$a_1 = -(1 + e^{-\omega_c}) / 2 \tag{4.11}$$

$$b_1 = -e^{-\omega_c} \tag{4.12}$$

Example 4.3.
We would like to implement fast and simple high-pass filter with the cut-off frequency f_c=1000 Hz. We expect signals to be sampled at sampling frequency 16000 Hz. What difference equation should we use to program the filter?

Solution:
According to Eq.4.10-4.12, the coefficients are:
a_0 = (1+exp(-2·3.1415·1000/16000)) / 2 = 0.8376;
a_1= –(1+exp(-2·3.1415·1000/16000)) / 2 = –0.8376;
b_1= exp(-2·3.1415·1000/16000) = –0.6752.

Therefore, the difference equation is:

y[n]=0.8376x[n] – 0.8376x[n-1] + 0.6752y[n–1].

LET'S CODE!
Filter signal x[n] using the filter designed in Example 4.3.

```
a = [0.8376 -0.8376];
b = [1 -0.6752];
y = filter(a, b, x);
```

Single pole low-pass filters are used for smoothing and high frequency noise suppression. They are a great choice for these tasks because they are very fast. Unfortunately, like moving average filter, these filters have little ability to separate one band of frequencies from another. In other words, they perform well in the time domain, and poorly in the frequency domain. The frequency response can be improved slightly by cascading several stages. For example, a four stage low-pass filter is already comparable to the Gaussian filter. Its coefficients are calculated by formulae:

$$a_0 = (1 - e^{-14.445 f_c / f_s})^4 \tag{4.13}$$

$$b_1 = -4e^{-14.445 f_c / f_s} \tag{4.14}$$

$$b_2 = 6e^{-14.445*2 f_c / f_s} \tag{4.15}$$

$$b_3 = -4e^{-14.445*3 f_c / f_s} \tag{4.16}$$

$$b_4 = e^{-14.445*4 f_c / f_s} \tag{4.17}$$

However, despite better stop-band attenuation, the transition band of the four stage low-pass filter's magnitude response is too wide to use this filter for a frequency band separation.

Finally, we consider here difference equations for band-pass and band-reject IIR filters used to isolate some narrow band of frequencies in a signal. The coefficients of the band-pass filter are:

$$a_0 = 1 - K \tag{4.18}$$

$$a_1 = 2(K - R)\cos(\omega_c) \tag{4.19}$$

$$a_2 = R^2 - K \tag{4.20}$$

$$b_1 = -2R\cos(\omega_c) \tag{4.21}$$

$$b_2 = R^2 \tag{4.22}$$

where parameters K and R are computed by following formulae:

$$K = \frac{1 - 2R\cos(\omega_c) + R^2}{2 - 2\cos(\omega_c)} \tag{4.23}$$

$$R = 1 - 3B_W / f_s \tag{4.24}$$

B_W is the desired width of the frequency band (in Hz). The coefficients of the band-reject filter are computed by the formulae:

$$a_0 = K \tag{4.25}$$
$$a_1 = -2K\cos(\omega_c) \tag{4.26}$$
$$a_2 = K \tag{4.27}$$
$$b_1 = -2R\cos(\omega_c) \tag{4.28}$$
$$b_2 = R^2 \tag{4.29}$$

4.3. Block Convolution

Basically, there are three ways to carry out digital filtering:

1) Using the difference equations to obtain the output signal (time domain);
2) Convolution of the signal with the filter kernel (time domain);
3) Fast convolution of the signal with the filter kernel (frequency domain).

All of the above-mentioned techniques can be applied for FIR filters. The techniques based on the convolution are practically inapplicable for IIR filters, since we need a finite impulse response for computer implementation. It's possible to use the truncated version of the infinite filter kernel, though. But in this case the results of filtering may be not satisfying.

The technique based on using difference equations is straightforward: simply use the Eq.4.2 to calculate each value of the output signal. The other techniques are based on the so called *block convolution* in which the input signal is segmented into sections of length L. Each section is convolved with the finite-length P-point filter kernel and the filtered sections are fitted together in an appropriate specific way.

The two procedures have been developed for digital filtering via the block convolution: 1) *the Overlap-Add* method, and 2) the *Overlap-Save* method. The former method implies that each L-point section of the signal is padded with $P–1$ zeros. Since the beginning of each input section is separated from its neighbors by L samples and each filtered section has length $(L+P–1)$, the nonzero points in the filtered sections will overlap by $(P–1)$ samples, and these overlap samples must be added in carrying out the total convolution. This is how the Overlap-Add method works (Fig.4.6).

According to the Overlap-Save method, the signal is divided into sections of length *L (L>P)* so that each input section overlaps the preceding section by *(P–1)* points. Since the circular convolution takes place, we simply discard the first *(P–1)* samples of each filtered section and replace them with *(P–1)* samples saved from the previous block. The Overlap-Add and Overlap-Save methods are mathematically proven to produce exactly the same output signal as direct convolution.

Note that only the block convolution with very short filter kernels can be efficiently implemented by means of a standard convolution. For longer filter kernels, the FFT convolution introduced in the Section 3.5 is much more efficient. For example, if we need to apply the filter with the kernel of length P=351, we may choose the length of each section L=162 so that the FFT size is a power of 2: $L+P-1$=351+162–1=512. Or we may choose the length of each section L=674 so that the FFT size is 1024, etc. With FFT convolution, the filter kernel can be made as long as we want, with very little penalty in execution time.

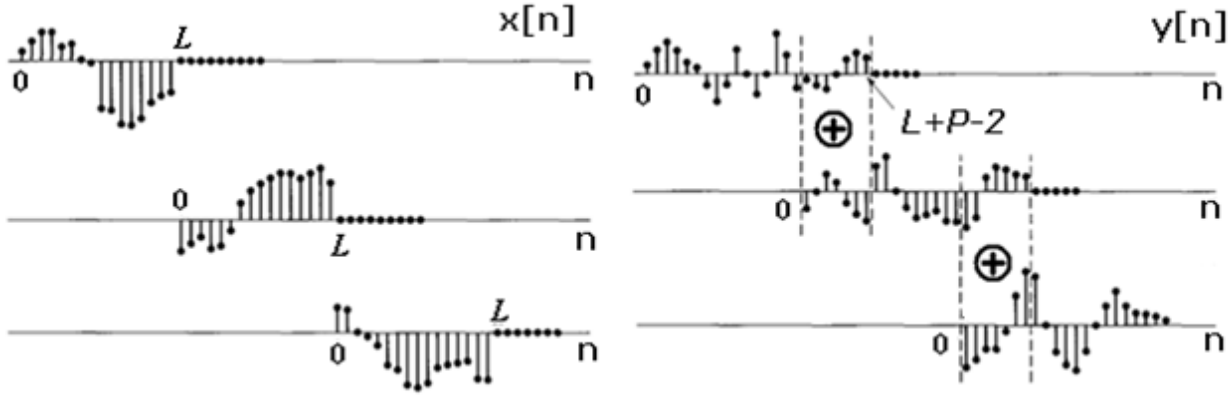

FIGURE 4.6. Overlap-Add method

The Overlap-Save method is slightly different (Fig.4.7).

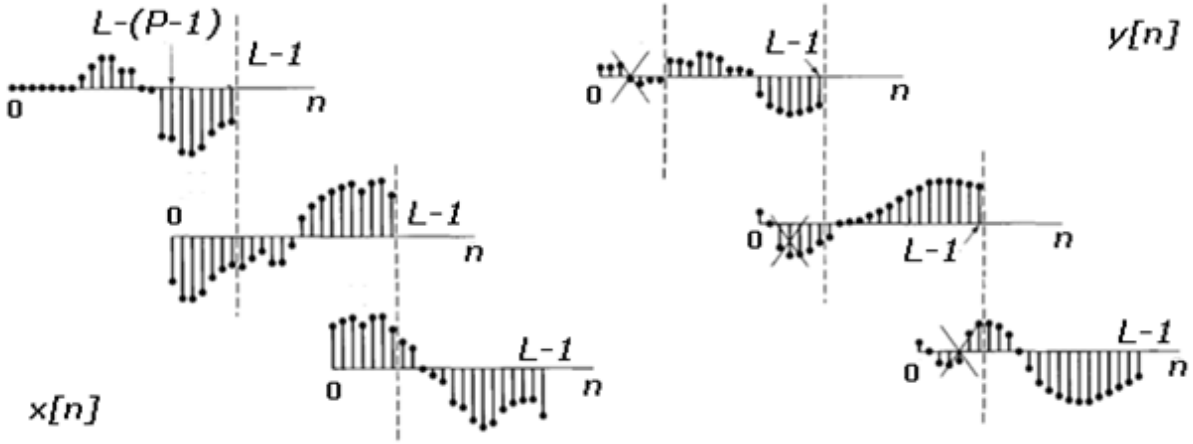

FIGURE 4.7. Overlap-Save method

The following question arises: “What filter kernel to use?”. The answer to this question will be given in the next chapter. In the meantime, you can use the MATLAB FDA (Filter Design and Analysis) toolbox for these purposes. Just set the desired frequency response and type of a filter in the corresponding window and press the button “Design Filter”. MATLAB will generate the filter kernel (the corresponding set of non-recursive coefficients for the filter’s difference equation). Another way to design filter is to use MATLAB functions: *fir1(), fir2(), remez()*. Also, the *fftfilt()* function is available for signal filtering via Overlap-Add method. See Appendix A for details.

LET’S CODE!
Design a bandstop FIR filter with edge frequencies of 1200 Hz and 2400 Hz, and filter the noisy signal x[n].

```
Wn = [1200 2400] / 8000;     % Edge frequencies
b = fir1(128, Wn, 'stop');   % Design band-stop filter

% Overlap / Add filtering using FFT convolution
y = fftfilt(b, x);

N = 512;                      % Spectrogram
specgram(y, N, fs, blackman(N), N/4 );
```

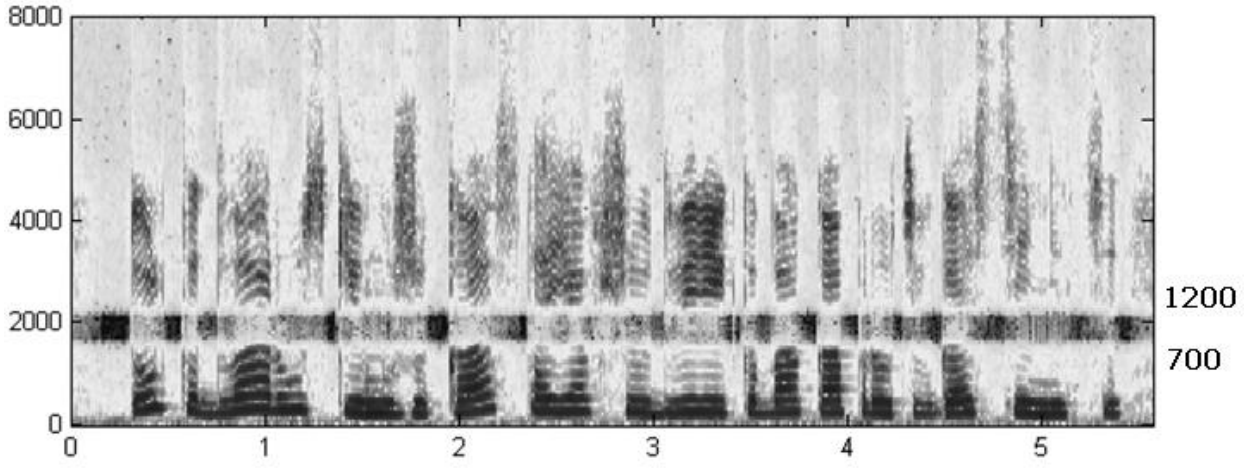

FIGURE 4.8. Spectrogram example of the filtered speech signal

LAB3	SIGNAL FILTERING
	points: 15 hours: 4

Objectives

1. To know the different types of digital filters.
2. To implement FIR and IIR filters and demonstrate the effect of filtering on example signals.

Exercise 3.1 [1 point]

Answer the following questions:

a. What is the step response of an LTI system?
b. What is filter kernel?
c. Give classification of filters and brief description of each class.
d. Describe the Overlap-Add method.

Exercise 3.2 [3 points]

Given three filters described by the following difference equations:

1) y[n]=0.41x[n] + 0.8y[n-1] - 0.24y[n-2] + 0.032y[n-3] - 0.002y[n-4]
2) y[n]=0.93x[n] – 0.93x[n-1] + 0.86y[n-1]
3) y[n]=0.32x[n] + 0.68y[n-1]

Answer the following questions (for each filter):

a. Is it FIR or IIR filter? Is it high-pass or low-pass filter?
b. Determine filter's cutoff frequency.

Exercise 3.3 (CODE) [3 points]

Write code that filters an input signal using MATLAB functions *filter(), fir1(), fir2(), remez()*. Your filter should be low-pass with cutoff frequency *fc=f1*. Save filtered signal to WAVE file. Plot the spectrograms of signal before and after filtering.

Exercise 3.4 (CODE) [4 points]

Open MATLAB Filter Design & Analysis Tool. Design low-pass FIR filter with cutoff frequency *fc=f1*. Write code that applies the designed filter using the Overlap-Add method. Write code that applies the designed filter using the Overlap-Save method. Save filtered signals to WAVE files. Plot the spectrograms of each signal before and after filtering.

Exercise 3.5 (CODE) [4 points]

Open MATLAB Filter Design & Analysis Tool. Design low-pass IIR filter with cutoff frequency *fc=f1*. Write code that directly implements in time-domain filter you've designed. Save filtered signals to WAVE files. Plot the spectrograms of each signal before and after filtering.

CHAPTER 5 FILTER DESIGN AND ANALYSIS

5.1. Z-Transform

The process of digital filter design lies, essentially, in determining of the system function of an LTI system whose characteristics satisfy a certain set of requirements. Some of the typical requirements are listed below:

- the filter should have a specific frequency response;
- the filter should have a specific phase response;
- the filter should have a specific impulse response;
- the filter should be causal;
- the filter should be stable.

We'll focus, mostly, on the former type of requirements. Roughly speaking, the system function is a function that describes the system's behavior. We've already considered the difference equations as one way to describe the system's behavior. Another way to look at LTI systems is connected with the so called Z-transform and transfer functions of a system in Z-domain.

Z-TRANSFORM
can be considered as a bridge between the difference equations and the frequency response of an LTI system. It plays the same role for discrete-time systems as the Laplace transform does for continuous time systems: it allows to replace the difference equations with the algebraic equations that are much easier to solve.

The Z-transform is defined as the power series:

$$X(z) = Z\{x[n]\} = \sum_{n=-\infty}^{+\infty} x[n]z^{-n} \tag{5.1}$$

The Eq.5.1 is often referred to as the *bilateral z-tranform* (or *two-sided z-transform*). In case of causal systems it's replaced with the *one-sided z-transform*:

$$X(z) = Z\{x[n]\} = \sum_{n=0}^{+\infty} x[n]z^{-n} \tag{5.2}$$

Don't forget: variable *z* is complex! If we substitute variable *z* in polar form into the Eq.5.1, we'll get:

$$X(re^{j\omega}) = \sum_{n=-\infty}^{+\infty} x[n](re^{j\omega})^{-n} = \sum_{n=-\infty}^{+\infty} (x[n]r^{-n})e^{-j\omega n} \tag{5.3}$$

It's easy to see that z-transform is closely related to Discrete-time Fourier transform: when r=1 (i.e., |z|=1), the z-transform reduces to the DTFT. This is one of the main reasons why the notation $e^{j\omega}$ is used for DTFT.

When DSP engineers use complex plane to analyze z-transform, they refer to it as the *complex z-plane*. In the z-plane, the contour corresponding to |z|=1 is called the *unit circle*. The z-transform evaluated on the unit circle corresponds to the Fourier transform (Fig.5.1).

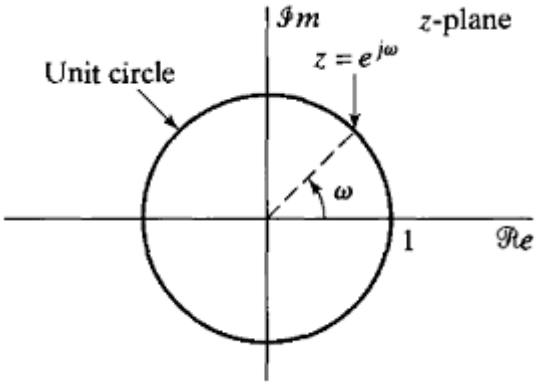

FIGURE 5.1. The complex z-plane and the unit circle

Since the z-transform involves an infinite power series, it exists only for those values of z where the power series defined in Eq.5.1 converges. The region on the complex z-plane in which the power series converges is called the *Region of Convergence (ROC)*. More formally:

$$ROC = \left\{ z \in C \mid \sum_{n=-\infty}^{+\infty} x[n] z^{-n} < \infty \right\}, \tag{5.4}$$

where C denotes the complex z-plane.

The ROC is an important concept in many respects: it allows the unique inversion of the z-transform and provides convenient characterizations of the causality and stability properties of a discrete-time system.

Note, since the signal is multiplied by z^{-n}, it is possible for z-transform to converge even if the corresponding Fourier transform does not converge. For example, the Fourier transform of the unit step *u[n]* doesn't converge absolutely. However, its z-transform converges if |z|>1.

The ROC consists of all z such that the inequality (5.4) holds. This means that if the ROC contains value $z=z_0$, then it also contains all values of z on the circle $|z|=|z_0|$. In other words, any ROC contains a set of circles in z-plane.

Example 5.1.
Find z-transform of the signal *x[n] = k*, where *k* = const, *n*=0,1,2,…

Solution:

$$X(z) = \sum_{n=0}^{+\infty} kz^{-n} \tag{5.5}$$

For $|z|>1$ the series (5.5) converges:

$$X(z) = \sum_{n=0}^{+\infty} kz^{-n} = k \cdot \frac{1}{1-z^{-1}} = \frac{kz}{z-1} \tag{5.6}$$

For $|z|\leq 1$ the series (5.5) diverges. Therefore, ROC consists of all z: $|z|>1$. Note, we've considered the signal $x[n]=k$ for $n>0$. The same equation is often rewritten as: $x[n] = ku[n]$, where $u[n]$ is a unit step.

Instead of computing each time z-transform of a given signal, we can use the table 5.1 with most common *z-transform pairs*.

The properties of the z-transform are extremely important for the analysis of discrete-time LTI systems. These properties are quite expectedly similar to the corresponding properties of Fourier transform. However, in case of the z-transform it is important to mention not only the relationship between the time-domain and z-domain but also the properties of the ROC:

1) *Linearity*:

$$ax[n] + by[n] \quad \overset{Z}{\longleftrightarrow} \quad aX(z) + bY(z) \tag{5.7}$$

ROC is the intersection of ROCs of *X(z)* and *Y(z)*

2) *Time Shifting*:

$$x[n-k] \quad \overset{Z}{\longleftrightarrow} \quad z^{-k}X(z) \tag{5.8}$$

ROC is the same (except for possible addition/deletion of $z=0$ or $z=\infty$)

3) *Frequency Shifting*:

$$z_0 x[n] \quad \overset{Z}{\longleftrightarrow} \quad X(z/z_0) \tag{5.9}$$

ROC is scaled by $|z_0|$

4) *Convolution*:

$$x[n] * y[n] \quad \overset{Z}{\longleftrightarrow} \quad X(z)Y(z) \tag{5.10}$$

ROC is the intersection of ROCs of *X(z)* and *Y(z)*

TABLE 5.1. Common z-transform pairs

Signal	Z-transform	ROC
$\delta[n]$	1	All z
$ku[n]$	$\frac{k}{1-z^{-1}}$	$\|z\|>1$
$-ku[-n-1]$	$\frac{k}{1-z^{-1}}$	$\|z\|<1$
$\delta[n-k]$	z^{-k}	All z, except $z=0$ (if $k>0$) and $z=\infty$ (if $k<0$)
$k^n u[n]$	$\frac{1}{1-kz^{-1}}$	$\|z\|>\|k\|$
$-k^n u[-n-1]$	$\frac{1}{1-kz^{-1}}$	$\|z\|<\|k\|$
$nk^n u[n]$	$\frac{kz^{-1}}{(1-kz^{-1})^2}$	$\|z\|>\|k\|$
$[\cos(\omega n)]\, u[n]$	$\frac{1-[\cos\omega]z^{-1}}{1-[2\cos\omega]z^{-1}+z^{-2}}$	$\|z\|>1$
$[\sin(\omega n)]\, u[n]$	$\frac{[\sin\omega]z^{-1}}{1-[2\cos\omega]z^{-1}+z^{-2}}$	$\|z\|>1$
$r^n\,[\cos(\omega n)]\, u[n]$	$\frac{1-r[\cos\omega]z^{-1}}{1-[2r\cos\omega]z^{-1}+r^2z^{-2}}$	$\|z\|>r$
$r^n\,[\sin(\omega n)]\, u[n]$	$\frac{r[\sin\omega]z^{-1}}{1-[2r\cos\omega]z^{-1}+r^2z^{-2}}$	$\|z\|>r$

The inverse z-transform is defined as

$$x[n] = Z^{-1}\{X(z)\} = \frac{1}{2\pi j}\oint_C X(z)z^{n-1}dz\,, \tag{5.11}$$

where *C* denotes the closed contour of *X(z)* taken in a counterclockwise direction. Formal analytic solution of the Eq.5.11 is based on the Cauchy integral theorem and can be quite difficult. Fortunately, several informal methods are available for finding the inverse z-transform:

1) *inspection method*;
2) *partial fraction expansion*;
3) *power series expansion*.

We'll review here the former two methods. The idea of the *Inspection method* lies simply in recognizing familiar transform pair (i.e., "by inspection").

Example 5.2.
Find the signal whose z-transform is:

$$X(z) = \frac{2}{1-0.8z^{-1}} + 5z^{-3} \tag{5.12}$$

Solution:
By inspection, we can recognize in Eq.5.12 three z-transform pairs:

$$2(0.8)^n u[n] \overset{Z}{\longleftrightarrow} \frac{2}{1-0.8z^{-1}} \qquad \text{for } |z|>|0.8| \tag{5.13}$$

$$-2(0.8)^n u[n-1] \overset{Z}{\longleftrightarrow} \frac{2}{1-0.8z^{-1}} \qquad \text{for } |z|<|0.8| \tag{5.14}$$

$$5\delta[n-3] \overset{Z}{\longleftrightarrow} 5z^{-3} \tag{5.15}$$

Hence, the inverse z-transform is either

$x[n] = 2(0.8)^n u[n] + 5\delta[n-3]$ for $|z|>|0.8|$

or

$x[n] = -2(0.8)^n u[-n-1] + 5\delta[n-3]$ for $|z|<|0.8|$

The *Partial Fraction expansion method* eventually comes to the Inspection method. The basic idea of the method is to represent a ratio of two polynomials in *z* or z^{-1} as the sum of simpler individual terms of the type (5.13). If the degree of the numerator *N(z)* is not greater than the degree of the denominator *D(z)*, the partial fraction expansion is expressed as:

$$X(z) = \frac{N(z)}{D(z)} = A_0 + \frac{A_1}{1-p_1 z^{-1}} + \frac{A_2}{1-p_2 z^{-1}} + \ldots + \frac{A_N}{1-p_N z^{-1}} \tag{5.16}$$

The formula for coefficients A_i is:

$$A_i = [(1 - p_i z^{-1}) X(z)]_{z=p_i} = \left[\frac{N(z)}{\prod_{i \neq j} 1 - p_j z^{-1}} \right]_{z=p_i} \tag{5.17}$$

If the degree of *N(z)* is strictly less than the degree of *D(z)*, then A_0=0. If the numerator and denominator have the same degree, we need to compute additionally the coefficient A_0:

$$A_0 = [X(z)]_{z=0} \tag{5.18}$$

Note we need to express *X(z)* in terms of *z*, not z^{-1}, since we substitute z=0 into the Eq.5.18.

Finally, if the degree of *N(z)* is greater than the degree of *D(z)*, we need to divide polynomials and express *X(z)* as:

$$X(z) = Q(z) + \frac{R(z)}{D(z)} \tag{5.19}$$

If the polynomial *D(z)* has multiple roots, the formulae for *X(z)* become more complicated. In most general case:

$$X(z) = Q(z) + \sum_{k=1, k \neq i}^{N} \frac{A_i}{1 - p_i z^{-1}} + \sum_{r=1}^{R} \frac{B_r}{(1 - p_i z^{-1})^r} \tag{5.20}$$

where the coefficients B_r are calculated by formula:

$$B_r = \frac{1}{(R-r)!(-p_i)^{R-r}} \left\{ \frac{d^{R-r}}{dz^{R-r}} [(1 - p_i z)^R X(z^{-1})] \right\}_{z=p_i^{-1}} \tag{5.21}$$

We'll consider only the examples of polynomials with single roots.

Example 5.3.
Find the signal whose z-transform is:

$$X(z) = \frac{8 - 2.2z^{-1} - 3.2z^{-2} - 0.8z^{-3}}{1 - 1.6z^{-1} - 0.8z^{-2}} \tag{5.22}$$

Solution:

Firstly, the degree of the numerator is greater than the degree of the denominator. Therefore, we need to divide polynomials:

$$\begin{array}{l|l} -0.8z^{-3}-3.2z^{-2}-2.2z^{-1}+8 & -0.8z^{-2}-1.6z^{-1}+1 \\ \underline{-} \; -0.8z^{-3}-1.6z^{-2}+z^{-1} & \hline z^{-1}+2 \\ \quad -1.6z^{-2}-3.2z^{-1}+8 & \\ \quad \underline{-} \; -1.6z^{-2}-3.2z^{-1}+2 & \\ \hline \qquad\qquad 6 & \end{array}$$

Hence:

$$X(z)=z^{-1}+2+\frac{6}{1-1.6z^{-1}-0.8z^{-2}}$$

After factoring the denominator we get:

$$\frac{6}{1-1.6z^{-1}-0.8z^{-2}}=\frac{6}{(1-2z^{-1})(1+0.4z^{-1})}=\frac{A}{1-2z^{-1}}+\frac{B}{1+0.4z^{-1}}$$

Using the Eq.5.17 for computing A and B, we obtain:

$$A=\left[\frac{6}{1+0.4z^{-1}}\right]_{z=2}=\frac{6}{1.2}=5$$
$$B=\left[\frac{6}{1-2z^{-1}}\right]_{z=-0.4}=\frac{6}{6}=1$$

Hence:

$$X(z)=z^{-1}+2+\frac{5}{1-2z^{-1}}+\frac{1}{1+0.4z^{-1}}$$

And now we can apply the Inspection method. According to the table 5.1, three different inverse z-transforms are possible:

$x[n] = \delta[n-1] + 2\delta[n] + 5(2)^n u[n] + (-0.4)^n u[n]$ for $|z|>|2|$

$x[n] = \delta[n-1] + 2\delta[n] - 5(2)^n u[-n-1] + (-0.4)^n u[n]$ for $|-0.4|<|z|<|2|$

$x[n] = \delta[n-1] + 2\delta[n] - 5(2)^n u[-n-1] - (-0.4)^n u[-n-1]$ for $|z|<|-0.4|$, $z\neq 0$

Note the circles determined by z=2 and z=–0.4 divide the z-plane into three non-overlapping regions. Each of these regions can be chosen as the

ROC of the z-transform *X(z)*. Therefore, three different signals correspond to *X(z)*, depending on the choice of the ROC.

5.2. Digital Filters in Z-Domain

How exactly can we apply the z-transform in digital filter analysis and design? Why is the z-transform so useful and convenient? Recall the difference equation of a digital filter (Eq.4.2). Taking the z-transform of both sides (and keeping in mind the Eq.5.8), we have

$$Y(z) = X(z)\sum_{k=0}^{N} a_k z^{-k} - Y(z)\sum_{m=1}^{M} b_m z^{-m} \tag{5.23}$$

And now it's time to introduce one of the major concepts in filter design and analysis. The *Transfer Function* of an LTI system is defined as:

$$H(z) = \frac{Y(z)}{X(z)} \tag{5.24}$$

TRANSFER FUNCTION
is the ratio of the z-transform of the output of a system to the z-transform of the input of a system. According to the convolution property of z-transform, the transfer function is the z-transform of the impulse response of a system.

From Eq.5.23 we get:

$$H(z) = \frac{\sum_{k=0}^{N} a_k z^{-k}}{1 + \sum_{m=1}^{M} b_m z^{-m}} \tag{5.25}$$

Given a transfer function *H(z)* we can obtain the following characteristics (descriptions) of a filter:

- the impulse response *h[n]*;
- the difference equation relating the output *y[n]* to the input *x[n]*;
- the frequency response *H(ω)*.

Conversely, given any of the above-mentioned characteristics, we can obtain the transfer function *H(z)*.

In practice, a typical usage of these descriptions is to start by specifying a set of desired frequency response specifications, that is, the desired shape of *H(ω)*. Then, through a filter design method, obtain a transfer function *H(z)* that satisfies

the given specifications. From *H(z)* we can then derive the corresponding difference equation and the processing algorithm that tells us how to operate the designed filter in real time. We can also find the impulse response of a filter by taking the inverse z-transform of its transfer function. Lastly, we can find the analytic form of the frequency response of a filter by replacing *z* by $e^{j\omega}$ into *H(z)*.

Example 5.4.
Given a filter with the transfer function:

$$H(z) = \frac{1 + 1.5z^{-1} + 1.5z^{-2}}{1 + 0.25z^{-1} - 0.375z^{-2}} \tag{5.26}$$

A. What difference equation should we use to program the filter?
B. Find the impulse response of the filter.
C. Find the frequency response of the filter.

Solution A:
Rewriting the Eq.5.26 in form of the Eq.5.23, and taking the inverse z-transform of *Y(z)* and *X(z)*, we obtain:

y[n] = x[n] + **1.5**x[n-1] + **1.5**x[n-2] – **0.25**y[n-1] + **0.375**y[n-2]
(numerator) (denominator)

Hence, the processing algorithm of the filter is based on the Eq.5.16.

Solution B:
The impulse response of the filter can be obtained by taking the inverse z-transform of *H(z)*. In order to find $h[n] = Z^{-1}\{H(z)\}$, we can use the partial fraction expansion method. The degree of the numerator is equal to the degree of the denominator. Therefore:

$$H(z) = \frac{1 + 1.5z^{-1} + 1.5z^{-2}}{(1 - 0.5z^{-1})(1 + 0.75z^{-1})} = A + \frac{B}{1 - 0.5z^{-1}} + \frac{C}{1 + 0.75z^{-1}}$$

where

$$A = \left[\frac{1 + 1.5z^{-1} + 1.5z^{-2}}{(1 - 0.5z^{-1})(1 + 0.75z^{-1})} \right]_{z=0} = \left[\frac{z^2 + 1.5z + 1.5}{(z - 0.5)(z + 0.75)} \right]_{z=0} = -4$$

$$B = \left[\frac{1 + 1.5z^{-1} + 1.5z^{-2}}{1 + 0.75z^{-1}} \right]_{z=0.5} = \frac{10}{2.5} = 4$$

$$C = \left[\frac{1+1.5z^{-1}+1.5z^{-2}}{1-0.5z^{-1}}\right]_{z=-0.75} = \left[\frac{z^2+1.5z+1.5}{z-0.5}\right]_{z=-0.75} = 1$$

Hence:

$$H(z) = -4 + \frac{4}{1-0.5z^{-1}} + \frac{1}{1+0.75z^{-1}}$$

According to the table 5.1, three different inverse z-transforms are possible:

$h_1[n] = -4\delta[n] + 4(0.5)^n u[n] + (-0.75)^n u[n]$ for $|z|>|-0.75|$

$h_2[n] = -4\delta[n] + 4(0.5)^n u[-n-1] - (-0.75)^n u[n]$ for $0.5<|z|<0.75$

$h_3[n] = -4\delta[n] - 4(0.5)^n u[-n-1] - (-0.75)^n u[-n-1]$ for $|z|<0.5$

As can be seen in the above equations, only the former impulse response is causal. Moreover, only the former impulse response is stable. In the end of this section we'll find out why.

Solution C:
The frequency response of the filter can be obtained by simply replacing *z* by $e^{j\omega}$ into *H(z)*:

$$H(e^{j\omega}) = \frac{1+1.5e^{-j\omega}+1.5e^{-2j\omega}}{1+0.25e^{-j\omega}-0.375e^{-2j\omega}}$$

Note the ROC of a transfer function must contain the unit circle (to guarantee that the corresponding DTFT exists).

Thus, we can express the difference equation and the processing algorithm of a filter from the transfer function. But what does the filter do? How exactly does the filter work in the frequency domain? Can we find a basic shape of the filter's frequency response from its transfer function? The answer is positive and lies in the so called *pole/zero pattern of a system*.

If we factor the numerator and denominator polynomials of *H(z)*, Eq.5.25 will be rewritten as the following rational function:

$$H(z) = \frac{a_0 (1 - z_1 z^{-1})(1 - z_2 z^{-1}) \ldots (1 - z_N z^{-1})}{b_0 (1 - p_1 z^{-1})(1 - p_2 z^{-1}) \ldots (1 - p_M z^{-1})} = \frac{a_0}{b_0} \frac{\prod_{k=1}^{N} (1 - z_k z^{-1})}{\prod_{m=1}^{M} (1 - z_m z^{-1})} \qquad (5.27)$$

The roots of the numerator polynomial are called the *zeros* of the transfer function *H(z)* since they are the values of *z* for which *H(z)*=0. Thus, *H(z)* given in Eq.5.27 has *N* zeros at $z = z_1, z_2, \ldots, z_N$. The roots of the denominator polynomial are called the *poles* since they are the values of *z* such that *H(z)*=∞. There are *M* poles at $z=p_1, p_2, \ldots, p_M$.

Note, *H(z)* is often represented as a ratio of polynomials in *z* (not z^{-1}):

$$H(z) = \frac{G(z - z_1)(z - z_2) \ldots (z - z_N)}{(z - p_1)(z - p_2) \ldots (z - p_M)} = G \frac{\prod_{k=1}^{N} (z - z_k)}{\prod_{m=1}^{M} (z - z_m)} \qquad (5.28)$$

A pole-zero diagram provides an insight into the properties of an LTI system. The shape of filter frequency response is related to the relative geometric locations of the poles and zeros on the *z*-plane.

Fig.5.2 illustrates a simple z-transform having a single pole at $z = p_1$ and a single zero at $z = z_1$. The magnitude spectrum $|X(\omega)|$ is the ratio of the distance of the point $e^{j\omega}$ to the zero z_1, namely, $|e^{j\omega} - z_1|$ divided by the distance of $e^{j\omega}$ to the pole p_1, namely, $|e^{j\omega} - p_1|$. As $e^{j\omega}$ moves around the unit circle, these distances will vary. As $e^{j\omega}$ passes near the pole, the denominator distance will become small causing the value of $|X(\omega)|$ to increase. If ω_1 is the phase angle of the pole p_1, then the point of closest approach to p_1 will occur at $\omega = \omega_1$ causing a peak in $|X(\omega)|$ there. The closer the pole is to the unit circle, the smaller the denominator distance will become at $\omega = \omega_1$, and the sharper the peak of $|X(\omega)|$.

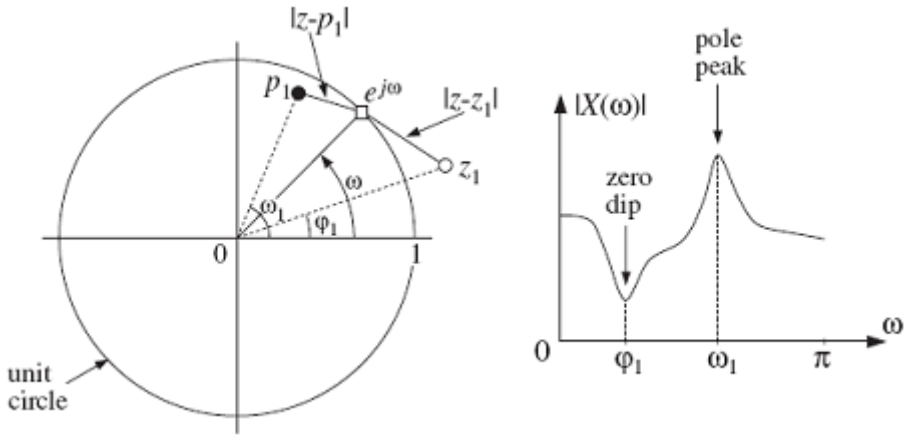

FIGURE 5.2. Relation between the spectral shape and zero/pole pattern

Similarly, as $e^{j\omega}$ passes near the zero z_1, the numerator distance will become small, causing $|X(\omega)|$ to decrease. At the zero's phase angle, say $\omega = \varphi_1$, this distance will be smallest, causing a dip in $|X(\omega)|$ there. The closer the zero to the unit circle the sharper the dip. The zero z_1 can also lie on the unit circle, in which case $|X(\omega)|$ will vanish exactly at $\omega = \varphi_1$.

In summary, we can draw a rough sketch of the spectrum $|X(\omega)|$ by letting $e^{j\omega}$ trace the unit circle and draw peaks as $e^{j\omega}$ passes near poles, and dips as it passes near zeros. By proper location of the zeros and poles of $X(z)$, we can design any desired shape for $X(z)$.

LET'S CODE!
Plot pole-zero diagram and frequency response of the filter given in example 5.4.

```
b=[1 1.5 1.5];
a=[1 0.25 -0.375];
figure(1);
zplane(b,a);
figure(2);
freqz(b,a);
```

Lastly, we'll mention how the poles and zeros of a transfer function in z-domain characterize the *causality* and *stability* of an LTI system. Causal systems are characterized by a transfer function which ROC is outside the maximum pole circle:

$$|z| > \max_i \{|p_i|\}$$

A necessary and sufficient condition for the *stability* of an LTI system is that the ROC of its transfer function contains the unit circle. Clearly, stability is not necessarily compatible with causality.

AN LTI SYSTEM IS BOTH STABLE AND CAUSAL
if all its poles lie strictly inside the unit circle in the z-plane.

Fig.5.3 demonstrates the examples of causal and stable filers in the *z*-plane.

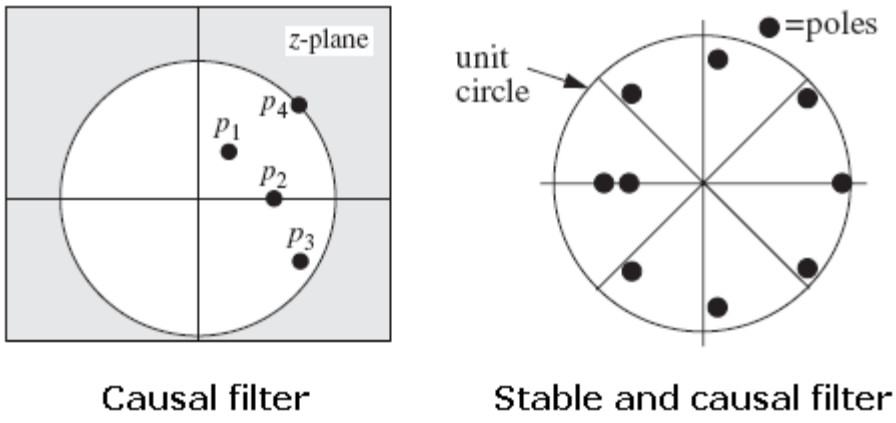

FIGURE 5.3. Causal and stable filters in the z-plane

Example 5.5.
Given a filter with the transfer function:

$$H(z)=\frac{0.1}{1-0.9z^{-1}} \tag{5.29}$$

What does the frequency response of the filter look like? Find the causal and stable impulse response of the filter.

Solution:
The transfer function has only one pole p=0.9 located at the beginning of the frequency circle (ω=0). Therefore, the filter is low-pass. A rough sketch of the frequency response is shown in Fig.5.4.

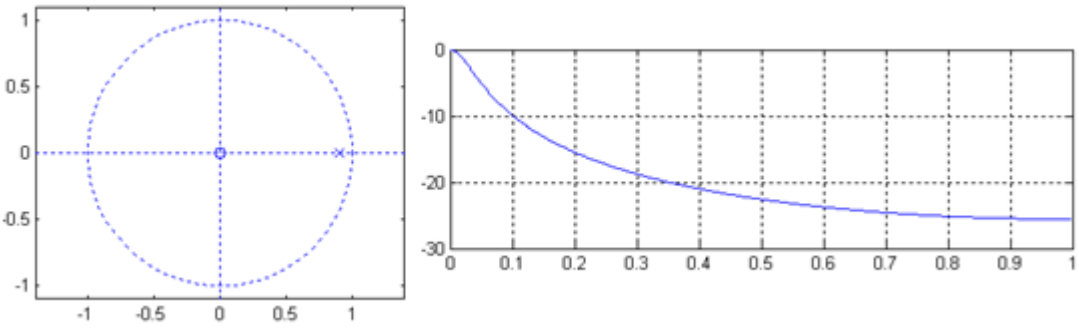

FIGURE 5.4. The frequency response of a single pole filter

Recall the single pole filter reviewed in the Section 4.2. It's exactly the kind of filter we're considering here. In terms of Eq.4.8 and Eq.4.9: a_0=0.1, b_1=0.9. At this moment it should be already clear why the filter is called the *single pole* filter. Furthermore, we have made sure that the filter is, indeed, low-pass.

The pole is quite close to the unit circle. Consequently, the peak of the frequency response should be quite sharp. Actually, we can calculate and compare the frequency response values at the edges of the Nyquist interval:

$$H(\omega)\big|_{\omega=0} = H(z)\big|_{z=1} = \frac{0.1}{1-0.9} = 1$$

$$H(\omega)\big|_{\omega=\pi} = H(z)\big|_{z=-1} = \frac{0.1}{1+0.9} \approx 0.05$$

According to the table 5.1, two inverse z-transforms of Eq.5.29 are possible:

$h_1[n] = 0.1(0.9)^n u[n]$ for $|z|>0.9$

$h_2[n] = -0.1(0.9)^n u[-n-1]$ for $|z|<0.9$

The latter impulse response is not stable, since the ROC of the corresponding transfer function does not contain the unit circle. Conversely, the former impulse response is both stable and causal.

Example 5.6.
Find the transfer function of a *moving average filter*. What does the frequency response of the filter look like?

Solution:
According to the definition of the z-transform and the formula for the impulse response of a moving average filter (Eq.4.5), the transfer function of the filter is:

$$H(z) = \frac{1}{L} + \frac{1}{L}z^{-1} + \frac{1}{L}z^{-2} + \ldots + \frac{1}{L}z^{-L} = \frac{1}{L}\cdot\frac{1-z^{-L}}{1-z^{-1}} \qquad (5.30)$$

The transfer function *H(z)* has *L* zeros. These zeros are the roots of the equation: $z^{-L}=1$, i.e.:

$$z_k = e^{j2\pi k/L}, \quad k = 0,1,2,..L-1 \qquad (5.31)$$

Since the zeros lie exactly on the unit circle, the frequency response reduces to zero at the corresponding frequencies. However, note there is also a single pole $p=1$ in the Eq.5.30, and it coincides with one of the zeros. This situation is known as *zero/pole cancellation*. In case of the moving average filter the pole $p=1$ prevails over the zero $z_0=1$. It's easy

to see that *H(z)* has the maximum value at *z*=1 (i.e., at the frequency ω=0): *H(1) = 1*.

The frequency response of the filter is shown in Fig.5.5.

Now recall the recursive formula Eq.4.6. If we rewrite the right part in the Eq.5.30 in form of the Eq.5.23, and then take the inverse z-transform, we'll get the same formula. Therefore, z-domain analysis provides another way to derive the Eq.4.6.

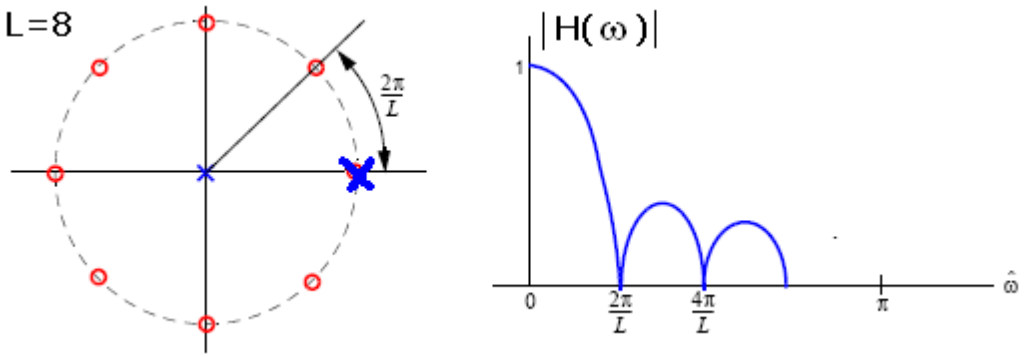

FIGURE 5.5. The frequency response of an L-point moving average filter (L=8)

5.3. Filter Design by Windowing

So far we have discussed how the frequency response of a filter is connected with the transfer function. At this moment we know how a general shape of the frequency response can be reflected in the transfer function of a filter. But what if we want to design a custom filter with lots of sharp peaks and dips, or a filter having the frequency response of a very specific structure?

Currently, many methods have been developed for digital filter design. The most efficient methods involve a lot of complex math, and we'll not discuss them in this guide. Note the methods for IIR filter design differ from the methods for FIR filter design. In this section we'll consider, perhaps, the simplest method for FIR filter design: *the Window method*.

The basic idea behind the Window method is very simple: specify the desired frequency response explicitly, in an array of numbers; take the inverse discrete Fourier transform to obtain the filter kernel; truncate and weigh the filter kernel; take the DFT and see if the resulting spectrum satisfies your needs (i.e., is the spectrum a sufficient approximation of the desired frequency response?). If it doesn't, then change the parameters of truncation and windowing, and repeat the procedure. Let's review the process of filter design by windowing step-by-step:

1. Specify the frequency response of the desired filter in array of length 2^N. Don't forget that frequency response is symmetric and complex. Thus, we need to define the magnitude array and phase array, or the array of real and imaginary parts (in rectangular form). The former way is much more obvious. Moreover, in most DSP applications the phase response is not

important (however, zero or linear phase is important and desirable). So, we can set all elements of the phase array to zero.

2. Obtain the filter impulse response by taking the inverse 2^N-point DFT of the ideal frequency response. The filter kernel will be infinitely long since in most cases the desired frequency response has discontinuities at the band edges, and the filter kernel has to reproduce them (recall the Section 3.4). The simplest (and optimal in terms of a mean-square error) solution is to truncate the filter kernel to $M+1$ points, symmetrically chosen around the zero sample. All samples outside these $M+1$ points are set to zero. Hence, the sample numbers run from $-M/2$ to $M/2$. The resulting filter is zero phase, by definition. This is a very "good" property, but the following programming problem arises: the filter kernel needs to be represented with negative indexes. The solution is to shift the entire kernel to the right so that it runs from 0 to M. This shift will lead to nothing more than the shift of the output signal by the same amount. Thus, the filter will change from zero phase to linear phase.
3. In principle, the filter kernel obtained at the previous step is ready to use. However, if we take its DFT we'll see that the resulting spectrum has an excessive ripple in the pass-band and poor attenuation in the stop-band. This happens due to the Gibbs effect and windowing effects mentioned in the Section 3.4, since the truncation of a filter kernel is equivalent to its multiplication by the rectangular window. To suppress the spectral side lobes one of the window functions given in the table 3.1 must be applied. This will increase the width of the transition region between the pass and stop bands, but will lower the side lobe levels outside the pass band and eliminate the ripple at the edges of the pass-band.
4. Since the modified filter kernel is only an approximation of the ideal filter kernel, it will not have an ideal frequency response. To find the frequency response of the filter, we can take the DFT. If the resulting spectrum is far from the desired frequency response, the width of the window must be increased or another window function chosen at the step 3. The filter design process is shown in Fig.5.6.

5.4. High-pass, Band-pass and Band-reject Filters

Finally, we'll consider how filters of various types can be designed if we have designed previously a low-pass filter with desired cutoff frequency.

There are two methods for the *low-pass to high-pass* conversion: *spectral inversion* and *spectral reversal*. Both are equally useful [3, Chapter 14].

The first method for low-pass to high-pass conversion is *spectral inversion*. According to this method, two things must be done to change the low-pass filter kernel into a high-pass filter kernel:

1) change the sign of each sample in the filter kernel;
2) add 1 to the middle sample (the sample at the center of symmetry).

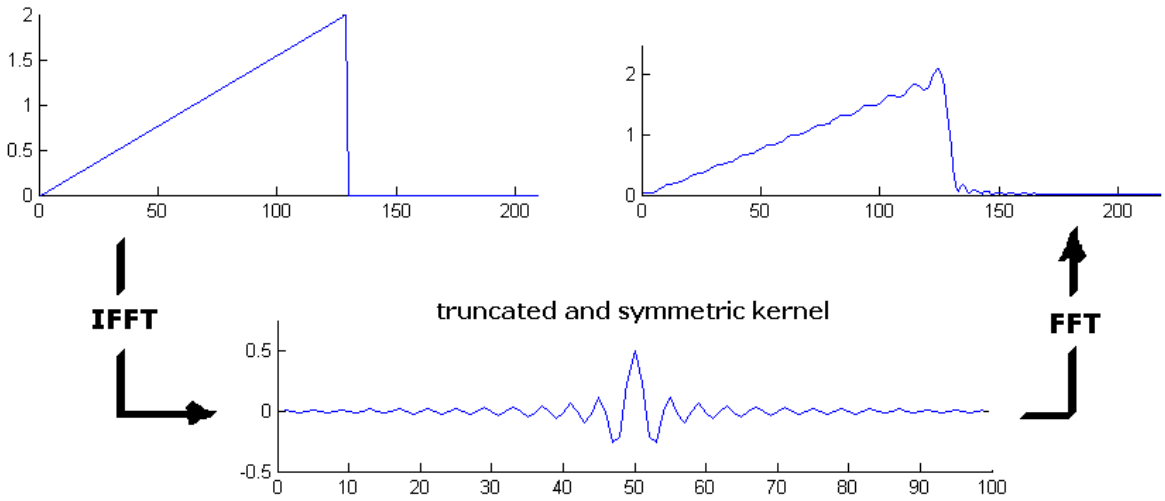

FIGURE 5.6. Filter design by windowing

The Fig.5.7 illustrates how this technique works. Spectral inversion flips the frequency response top-for-bottom, changing the pass-bands into stop-bands, and the stop-bands into pass-bands. For this technique to work, the low-frequency components exiting the low-pass filter must have the same phase as the low-frequency components exiting the all-pass system. Otherwise a complete subtraction cannot take place. This places two restrictions on the method:

(1) the original filter kernel must have *left-right symmetry* (i.e., a zero or linear phase);

(2) the impulse must be added at the *center* of symmetry.

The second method for low-pass to high-pass conversion is *spectral reversal*. The high-pass filter kernel is formed by changing the sign of every other sample in the low-pass filter kernel. This flips the frequency domain left-for-right: 0 becomes 0.5 and 0.5. *Changing the sign of every other sample is equivalent to multiplying the filter kernel by a sinusoid with a frequency of 0.5. This has the effect of shifting the frequency domain by 0.5*.

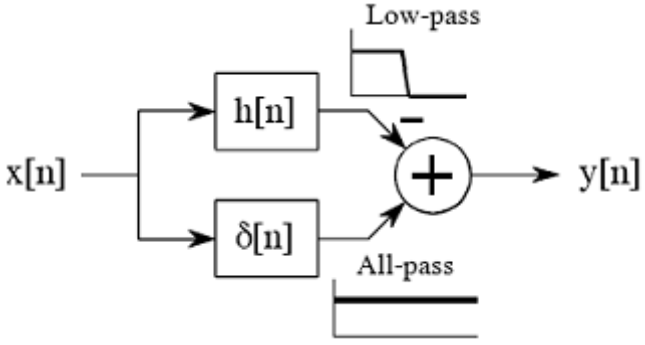

FIGURE 5.7. Spectral inversion: low-pass to high-pass conversion

Lastly, Fig.5.8 shows how low-pass and high-pass filter kernels can be combined to form band-pass and band-reject filters.

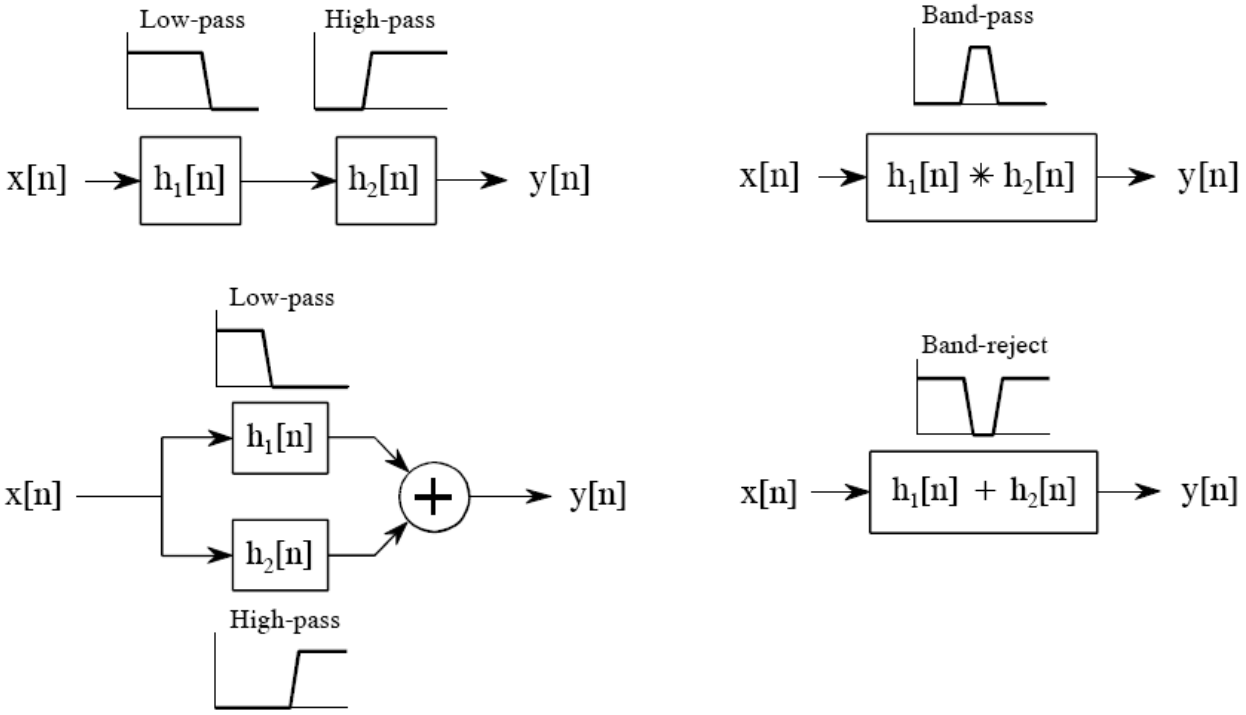

FIGURE 5.8. Band-pass and Band-reject filters

Figure 5.8 shows that adding the filter kernels produces a band-reject filter, while convolving the filter kernels produces a band-pass filter. These are based on the way cascaded and parallel systems are be combined. Multiple combination of these techniques can also be used.

LAB4	FILTER DESIGN AND ANALYSIS
	points: 17 hours: 4

Objectives

1. To understand the concepts of the *z*-domain.
2. To understand the concepts of digital filter design and analysis.

Exercise 4.1 [1 point]

Answer the following questions:

a. What is the transfer function of a filter?
b. How is the transfer function of a filter related to its difference equation?
c. What are the zeros and poles of a transfer function?
d. Describe FIR filter design process using windowing.
e. How to obtain the kernel of a high-pass filter from the kernel of a corresponding low-pass filter?

Exercise 4.2 [5 points]

Given five filters described by the following difference equations:

1) $y[n] = 0.36x[n] + 0.22x[n-1] – 0.85x[n-2]$
2) $y[n] = 0.76x[n] + 0.32x[n-1] + 0.15y[n-1]$
3) $y[n] = x[n] – x[n-5]$
4) $y[n] = 0.8x[n] – 0.2y[n-1] – 0.3y[n-2] + 0.8y[n-3]$
5) $y[n] = x[n] – x[n-2] + 0.9y[n-1] – 0.6y[n-2]$

Answer the following questions (for each filter):

a. Is it IIR or FIR filter?
b. Determine filter's transfer function.
c. Determine zeros and poles of filter's transfer function.
d. Draw a rough sketch of filter's frequency response.

Exercise 4.3 (CODE) [3 points]

Write your own code to design low-pass FIR-filter with custom order and cutoff frequency using Window method. Plot impulse response and step response of the designed filter.

Exercise 4.4 (CODE) [2 points]

Write code to plot:

a) frequency response
b) poles and zeros of transfer function

of each of the filters given in exercises 3.2 and 4.2. Compare figures with your analytic solutions of exercises 3.2 and 4.2.

Exercise 4.5 (CODE) [2 points]

Use filter kernel computed in exercise 3.4. Write code that computes new filter kernel to make filter high-pass with the same cutoff frequency. Implement your high-pass filter. Plot the spectrograms of each signal before and after filtering.

Exercise 4.6 (CODE) [4 points]

Create low-pass, band-pass and band-reject filters with the coefficients defined in Eq.4.13-Eq.4.29 and with the following parameters: f_c=1200 Hz; B_W=500 Hz. What does the transfer function of each filter look like? Write code to plot the filter frequency response and its zero-pole diagram. Apply each filter to the signals *s1*, *s2*, and *s3*. Plot the spectrograms of each signal before and after filtering.

CHAPTER 6 SAMPLING RATE CONVERSION

6.1. Decimation and Interpolation

In many DSP applications sampling frequency changes are necessary. The examples are: image resizing (or scaling), conversion between different sound formats, to name a few. The process of converting a digital signal to a different sampling rate is called *sampling rate conversion*.

Suppose we have a sampled version *x[n]* of the analog signal $x_c(t)$ with sampling period *T* (i.e., $x[n]=x_c(nT)$). The sampling rate conversion (Fig.6.1) implies obtaining a new discrete-time representation of the underlying analog signal $x_c(t)$:

$$x_1[n] = x_c(nT_1), \quad T_1 \neq T \tag{6.1}$$

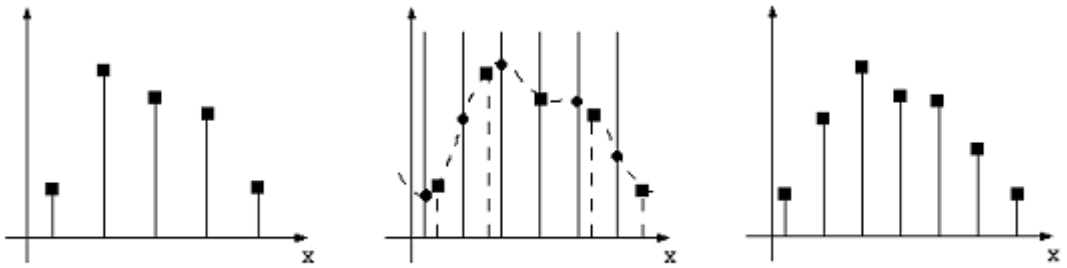

FIGURE 6.1. Sampling rate conversion

The intuitive approach for doing the sampling rate conversion is to reconstruct a continuous signal from a discrete-time signal, and then sample it at the new rate. However, in this case we'll lose some information, since DAC and ADC reconstruct and sample signals not ideally. Fortunately, there are useful and efficient methods of sampling rate conversion right in the discrete-time domain.

We'll begin with considering the *sampling rate reduction by an integer factor (D)*. This process is also referred to as the *downsampling*. It is defined as:

$$x_d[n] = x[nD] = x_c(nDT) \tag{6.2}$$

Since *D* is integer, it's easy to see that signal $x_d[n]$ is identical to the signal that would be obtained from $x_c(t)$ by sampling with period *DT*. However, the frequency content of $x_d[n]$ needs to be clarified. It can be shown that the Fourier transform of $x_d[n]$ is expressed in terms of the Fourier transform of *x[n]* as:

$$X_d(\Omega) = \frac{1}{D}\sum_{k=0}^{D-1} X\left(\frac{\Omega}{D} - \frac{2\pi k}{D}\right) \tag{6.3}$$

The Eq.6.3 is similar to Eq.3.34. Just like the Eq.3.34 expresses the relation between the spectra of the analog and sampled signals, the Eq.6.3 expresses the relation between the spectra of the sampled signal *x[n]* and its "further sampled" version $x_d[n]$. the Eq.6.3 means that the DTFT of $x_d[n]$ contains *D* copies of the DTFT of *x[n]*, frequency scaled by *D* and shifted by multiples of 2π (See Fig.6.2).

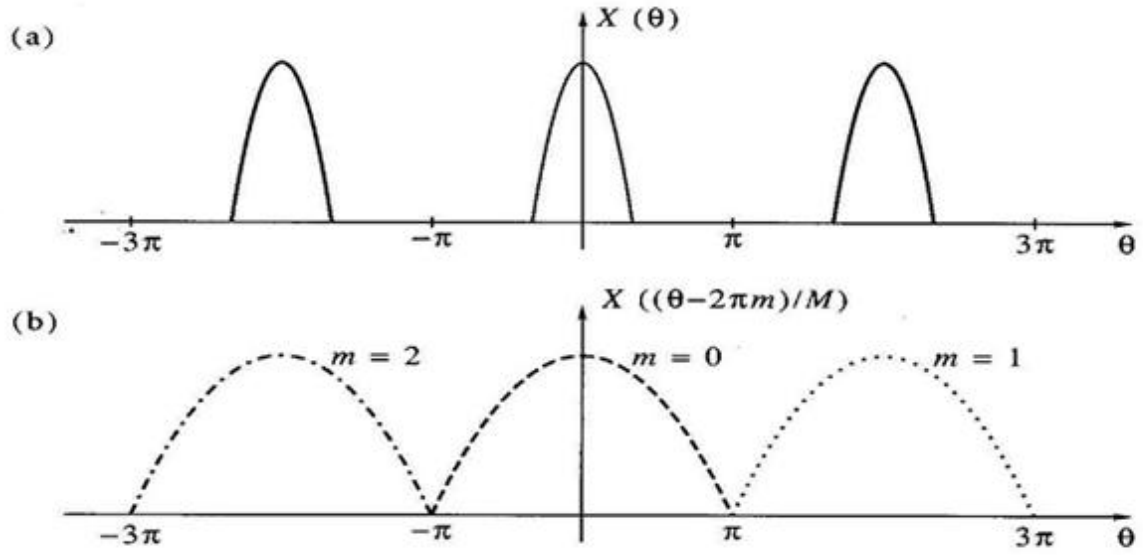

FIGURE 6.2. Spectra: a) of signal *x[n]*; b) of downsampled signal $x_d[n]$

Therefore, to avoid aliasing in the spectrum of $x_d[n]$ we need to guarantee that the spectrum of *x[n]* is bandlimited with the maximum frequency

$$\omega_{\max} \le \pi / D \tag{6.4}$$

In other words, for a proper downsampling we need first to apply an anti-aliasing filter with cutoff frequency f_s/D. A general system consisting of low-pass pre-filter and downsampler is called the *decimator* (Fig.6.3).

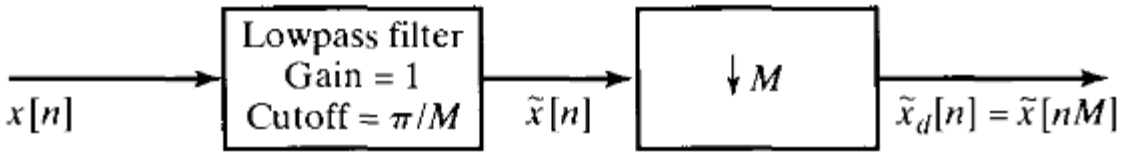

FIGURE 6.3. Decimator system

Now let's consider how the sampling rate is *increased by an integer factor (U)*. This process is also referred to as the *upsampling*. It is defined as:

$$x_u[n] = \begin{cases} x[n/U] = x_c(nT/U), & n = 0, \pm U, \pm 2U, \ldots \\ 0, & otherwise \end{cases} \qquad (6.5)$$

In other words, we copy all samples of *x[n]* into $x_d[n]$ and insert *U*–1 zero samples between each two adjacent samples of *x[n]*. The Eq.6.5 can be rewritten in a more compact form:

$$x_u[n] = \sum_{k=-\infty}^{+\infty} x[k]\delta[n - kU] \qquad (6.6)$$

In the frequency domain this is expressed in terms of the Fourier transform $X(e^{j\omega})$ of *x[n]* as:

$$X_u(e^{j\omega}) = \sum_{n=-\infty}^{+\infty} \left(\sum_{k=-\infty}^{+\infty} x[n]\delta[n - kU] \right) e^{-j\omega n} = \sum_{k=-\infty}^{+\infty} x[k] e^{-j\omega U k} = X(e^{j\omega U}) \qquad (6.7)$$

The effect reflected in Eq.6.7 is shown in Fig.6.4.

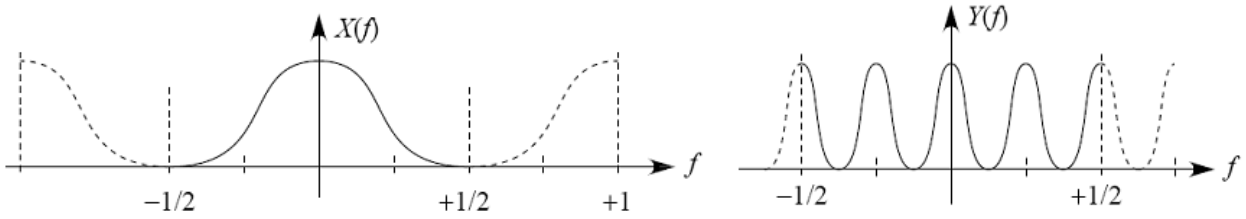

FIGURE 6.4. Spectra: a) of signal *x[n]*; b) of upsampled signal $x_u[n]$

Therefore, $X_u(e^{j\omega})$ can be obtained from the spectrum of $x_u[n]$ by removing all replicas of $X_c(e^{j\omega})$ except at integer multiples of 2π. Besides, inserting zeros into *x[n]* leads to signal enlargement (*U* times), so we need to "correct" the energy of spectrum (multiply it by *U*). That is, the low-pass filter with a gain of *U* and cutoff frequency f_s/U must be applied.

The system consisting of the upsampler and low-pass post-filter is called an *interpolator* (Fig.6.5), since it fills in the missing (zero) samples. Sometimes this kind of interpolation is referred to as sinc-interpolation to distinguish it from the well-known linear or cubic interpolation. In other words, the ideal low-pass filtering is one of techniques for a numerical interpolation.

Finally, we'll consider how to *change the sampling rate of a signal by an arbitrary non-integer factor*. The idea of this conversion is very simple: first interpolate by *U*, then decimate by *D*. Note that combining the interpolator and decimator systems (Fig.6.5 and Fig.6.3) leads to a cascade connection of two low-

pass filters that differ only by cutoff frequency. Hence, we can replace two filters with one filter: the one with the lesser cutoff frequency (Fig.6.6).

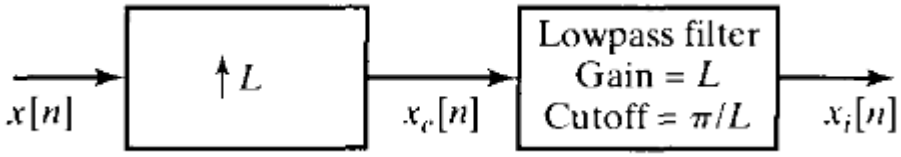

FIGURE 6.5. Interpolator system

If the factors *U* and *D* have a common divisor, it makes sense to divide them by their greatest common divisor before resampling. For example, if we have a signal sampled at rate 8000 Hz, and we want to resample it at 6000Hz, there's no need to interpolate by 6000 and then decimate by 8000. We'll get the same result if we interpolate by 3 and decimate by 4 in this case (3/4 = 6000/8000).

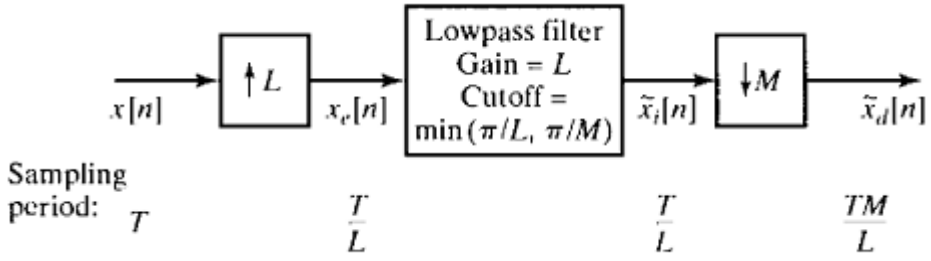

FIGURE 6.6. General resampling scheme

LET'S CODE!
Given a signal sampled at rate 8000 Hz consisting of two sinusoids with frequencies 500 Hz and 900 Hz. Resample the signal at rate 6000 Hz.

```
n = 1:100;
fs = 8000;
x = sin(2*pi*500/fs*n) + sin(2*pi*900/fs*n);
subplot(3,1,1); plot(x);
y = interp(x,3);
subplot(3,1,2); plot(y, 'g');
z = decimate(y,4,'fir');
subplot(3,1,3); plot(z,'r');
```

6.2. Polyphase Filtering

Clearly, if the values of factors *U* and *D* are large, the direct implementation of interpolation and decimation requires a lot of computations. But if we look at the upsampled signal, we'll see that most of its samples are zero. Therefore, we can

skip most of the multiplications during resampling. The question is: "How exactly?". *Polyphase filtering* is a special technique developed for efficient resampling. We'll give here only a brief review of polyphase filters.

The polyphase filtering is part of the so called *multirate signal processing*. Multirate techniques are used in sampling rate converters, ADCs and DACs, filter banks for analysis and processing of signals.

Polyphase filtering is based on two *multirate identities* also known as the *Noble identities*:

$$(\downarrow D)H(z) = H(z^D)(\downarrow D) \tag{6.8}$$

$$H(z)(\uparrow U) = (\uparrow U)H(z^U), \tag{6.9}$$

where (↓D) denotes downsampling by a factor *D*, and (↑U) denotes upsampling by a factor *U*. The multirate identities are very important, since they state that we can interchange the order of decimation and filtering (as well as the order of interpolation and filtering). All we need to do is change the structure of corresponding low-pass filter. It's intuitively clear that downsampling by a large factor in the first stage followed by the filtering of a shorter signal is much more efficient than downsampling after a "long" filtering. Similarly, the filtering of a shorter signal followed by upsampling is more efficient than the filtering of a long upsampled signal.

Polyphase decomposition is the decomposition of a signal into set of *M* signals, each consisting of every *M*th value of successively delayed versions of the signal. In terms of a filter kernel, a polyphase filter is a set of *M* sub-filters:

$$e_k[n] = h_k[nM] = h[nM + k], \qquad k=0,1,2,..,M-1 \tag{6.8}$$

It can be shown that in z-domain the transfer function of a filter is the sum of polyphase components:

$$H(z) = \sum_{k=0}^{M-1} E_k(z^M) z^{-k} \tag{6.9}$$

Thus, the filter *H(z)* can be decomposed into *M parallel* sub-filters $E_k(z^M)$. These sub-filters differ only by their phase. That's why the whole filter is called *poly*phase. Using Noble identities we can build polyphase filters for decimation (Fig.6.7a) and interpolation (Fig.6.7b). The main advantage of polyphase filtering shown in Fig.6.7 is that ***the actual filtering takes place at the original sampling rate***.

The number of polyphase filters is equal to the corresponding factor of decimation or interpolation. If the length of the low-pass filter kernel is not a multiple of resampling factor, the sub-filter kernels need to be zero-padded.

LET'S CODE!
Perform decimation of signal x[n] by factor 4 and then reconstruct x[n] (interpolate the result by factor 4).

```
% --- design low-pass filter kernel h here
% --- zero-pad the kernel

% ------ Polyphase decomposition of filter kernel h
h1 = h(1:4:end);
h2 = h(2:4:end);
h3 = h(3:4:end);
h4 = h(4:4:end);

x1 = [x(1:4:end) 0];
x2 = [0 x(4:4:end)];
x3 = [0 x(3:4:end)];
x4 = [0 x(2:4:end)];

y1 = filter(h1,1,x1);
y2 = filter(h2,1,x2);
y3 = filter(h3,1,x3);
y4 = filter(h4,1,x4);

y1 = [y1 0];
y = y1 + y2 + y3 + y4;     % decimated signal

L = 4;
u1 = L * filter(h1,1,y);
u2 = L * filter(h2,1,y);
u3 = L * filter(h3,1,y);
u4 = L * filter(h4,1,y);

U = [u1; u2; u3; u4];
res = U(:);                % resampled signal
```

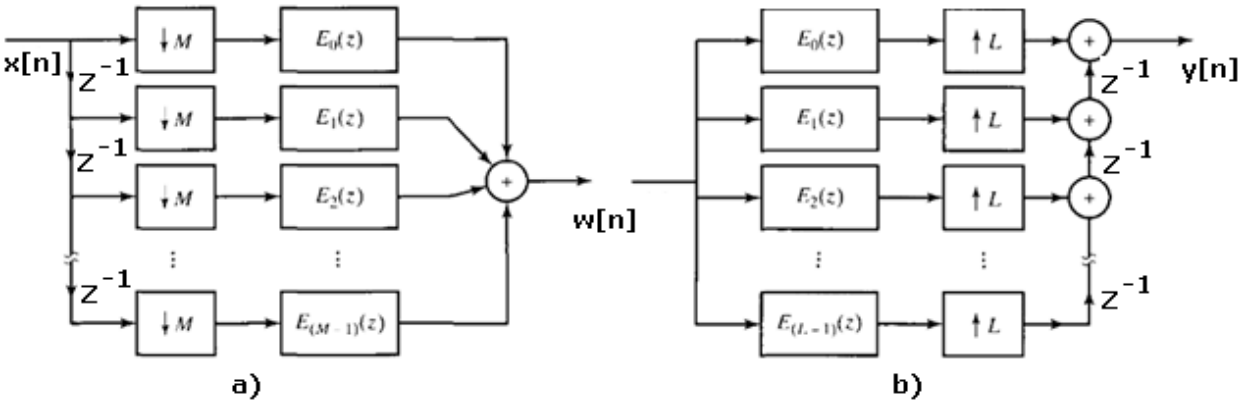

FIGURE 6.7. Polyphase structure: a) of decimation; b) of interpolation

LAB5	SAMPLING RATE CONVERSION points: 8 hours: 2

Objectives

1. To understand the concepts of interpolation and decimation.
2. To understand the structure of polyphase filters.

Exercise 5.1 [1 point]

Answer the following questions:

a. What is interpolation?
b. What is decimation?
c. Why is filtering needed after the interpolation?
d. Why is filtering needed before the decimation?
e. What is polyphase filtering?

Exercise 5.2 (CODE) [2 points]

Write code to do the following:

1. Open WAVE files containing signals *s1*, *s2*, *s3*, *s4*, *sn* from exercise 1.3.
2. Resample each signal at new rate f_{Ns}=48kHz using MATLAB functions *interp()* and *decimate()*.
3. Resample each signal at new rate f_{Ns}=48kHz using MATLAB function *resample()*.
4. Plot spectrogram of each signal after resampling.

Exercise 5.3 (CODE) [2 points]

Write code to do the following:

1. Open WAVE files containing signals *s1*, *s2*, *s3*, *s4*, *sn* from exercise 1.3.
2. Resample each signal at new rate f_{Ns}=48kHz using MATLAB function *upfirdn()*. How does this function work?
3. Plot spectrogram of each signal after resampling.

Exercise 5.4 (CODE) [3 points]

Write code to do the following:

1. Open WAVE files containing signals *s1*, *s2*, *s3*, *s4*, *sn* from exercise 1.3.
2. Resample each signal at new rate f_{Ns}=48kHz using polyphase filters.
3. Plot frequency response of each polyphase filter.

CHAPTER 7 SPEECH SIGNAL PROCESSING

7.1. Basics of Speech Signal Processing

The Speech Signal Processing is, currently, one of the most needed and challenging applications of DSP. The broad area of speech processing can be broken down into several individual areas according to both applications and technology. These include:

1. Speech coding.
2. Text-To-Speech (TTS) conversion.
3. Automatic Speech Recognition (ASR).
4. Speaker verification.

We'll consider in this guide only the very basics of speech signal processing: 1) what speech signals look like; 2) how they can be represented in simplified form. Let's take a look at the common spectrogram of a speech signal (Fig.7.1).

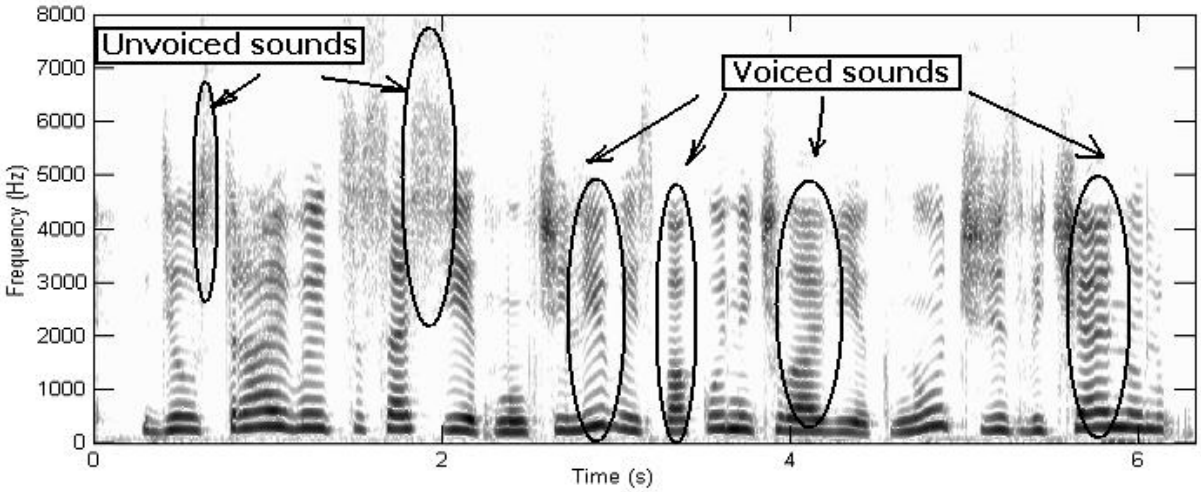

FIGURE 7.1. The spectrogram of speech signal

Most human speech sounds can be classified as either *voiced* or *fricative (unvoiced)*. Voiced sounds occur when air is forced from the lungs, through the vocal cords, and out of the mouth and/or nose. Vowels are an example of voiced sounds. In speech spectrograms such sounds can be easily seen, since their spectra have periodic nature (they contain multiple harmonics of one frequency called *pitch*) (Fig.7.1). In comparison, fricative sounds originate as random noise, not from vibration of the vocal cords. This occurs when the air flow is nearly blocked by the tongue, lips, and/or teeth, resulting in air turbulence near the constriction. Fricative sounds (in English) include: [s], [f], [sh], [z], [v], and [th].

Many speech processing techniques attempt to represent the variety of speech sounds as the set of parameters of some unified speech model. We'll consider two examples of such techniques in the following sections.

7.2. Linear Prediction

The LPC method is based on the speech production model including excitation input, gain, and vocal tract filter (Fig.7.2).

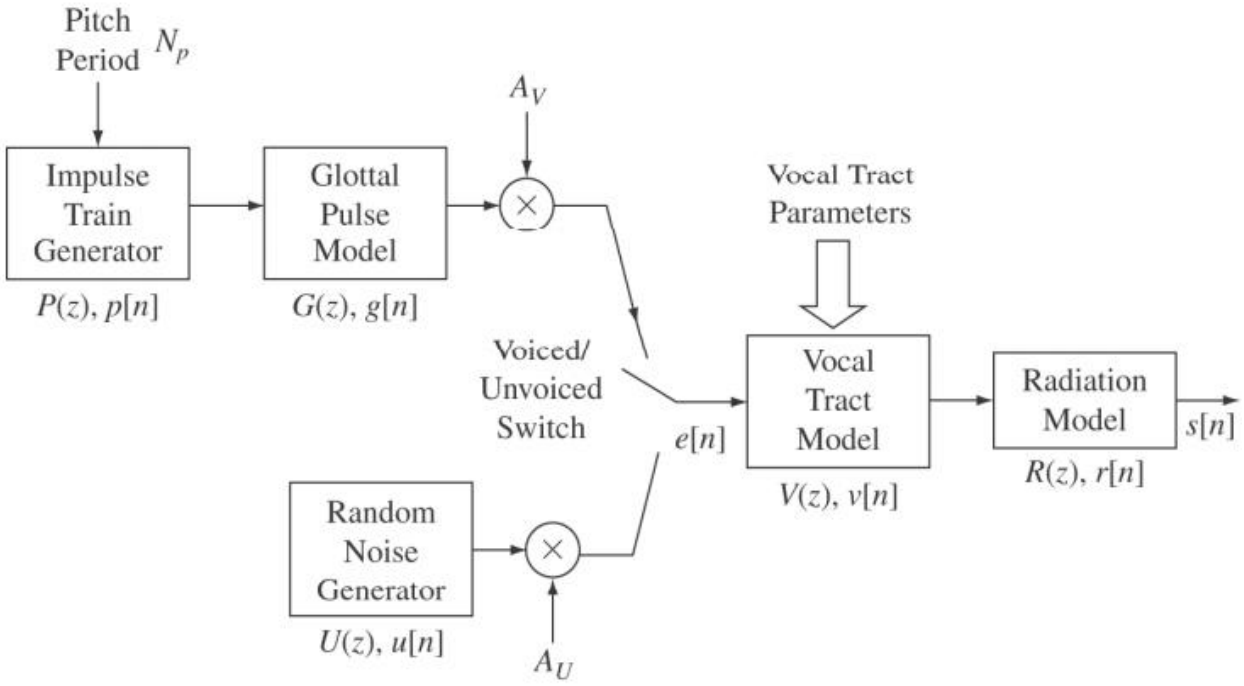

FIGURE 7.2. Speech production model

It is necessary to determine a given segment or frame (usually in the range of 5–30 ms) in voiced or unvoiced speech. Segmentation is formed by multiplying the speech signal by a Hamming window. The successive windows are overlapped. For a voiced speech, the pitch period is estimated and used to generate the periodic excitation input. For an unvoiced speech, a random noise will be used as the excitation input. The vocal tract is modeled as an all-pole digital filter.

The time-varying short-term synthesis filter *G/A(z)* represents the model of human vocal tract; it is updated frame by frame. The vocal tract transfer function is expressed as

$$V(z) = \frac{G}{A(z)} = \frac{G}{1 - \sum_{k=1}^{p} a_k z^{-k}} \tag{7.1}$$

where a_k is the k^{th} short-term LPC coefficient, G is the gain of the filter and p is the filter order.

The overall model is referred to as *Linear Prediction Coefficients*, since the basic assumption behind LPC is the correlation between the *n*-th sample and the *P* previous samples of the speech signal. Namely, the *n*-th signal sample is represented as a linear combination of the previous *P* samples, plus a residual representing the prediction error:

$$x[n] = -a_1 x[n-1] - a_2 x[n-2] - \ldots - a_P x[n-P] + e[n] \qquad (7.2)$$

In z-domain the Eq.7.2 reduces to:

$$E(z) = A(z)X(z) \qquad (7.3)$$

where *A(z)* is the polynomial with coefficients a_1,..., a_P. In the case of voice signal analysis, the filter *1/A(z)* is called the all-pole formant filter because, if the proper order *P* is chosen, its magnitude frequency response follows the envelope of the signal spectrum, with its broad resonances (peaks) called *formants*. The filter *A(z)* is called the inverse formant filter because it extracts from the voice signal a residual resembling the vocal tract excitation. *A(z)* is also called a whitening filter because it produces a residual having a flat spectrum.

The most popular method to calculate the gain and LPC coefficients is the autocorrelation method. Due to the characteristics of speech sounds, windows are applied to calculate the autocorrelation coefficients as follows:

$$R_n(j) = \sum_{m=0}^{N-1-j} s_n(m) s_n(m+j) \qquad (7.4)$$

where *N* is the frame size, *n* is the frame index, and *m* is the sample index in the frame. We need to solve the following matrix equation to derive the prediction filter coefficients a_i:

$$\begin{bmatrix} R_n(0) & R_n(1) & \vdots & R_n(p-1) \\ R_n(1) & R_n(0) & \vdots & R_n(p-2) \\ \cdots & \cdots & \cdots & \cdots \\ R_n(p-1) & R_n(p-2) & \vdots & R_n(0) \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \cdots \\ a_p \end{bmatrix} = \begin{bmatrix} R_n(0) \\ R_n(1) \\ \cdots \\ R_n(p) \end{bmatrix} \qquad (7.5)$$

The left-hand side matrix is symmetric, all the elements on its main diagonal are equal, and the elements on any other diagonal parallel to the main diagonal are also equal. This square matrix is Toeplitz. Several efficient recursive algorithms have been derived for solving Eq.7.5. The most widely used algorithm is the Levinson–Durbin recursion summarized as follows:

$$E_n^{(0)} = R_n^{(0)} \qquad (7.6)$$

$$k_i = \frac{R_n(i) - \sum_{j=1}^{i-1} a_j^{(i-1)} R_n(|i-j|)}{E_n^{(i-1)}} \tag{7.7}$$

$$a_i^{(i)} = k_i \tag{7.8}$$

$$a_j^{(i)} = a_j^{(i-1)} - k_i a_{i-j}^{(i-1)}, \quad 1 \le j \le i-1 \tag{7.9}$$

$$E_n(i) = (1-k_i^2) E_n^{(i-1)} \tag{7.10}$$

After solving these equations recursively for i = 1, 2, ... , p, the parameters a_i are given by:

$$a_j = a_j^{(p)}, \quad 1 \le j \le p \tag{7.11}$$

Thus, we've done the LPC-analysis, that is we've found the coefficients of the filter (7.1). If we compute frequency response of the synthesis filter we'll notice that LPC-spectrum reflects the basic shape of the corresponding DFT-spectrum (Fig.7.3). In general, LPC-synthesis is the process of excitation signal filtering using the filter (7.1). We can find formants by computing the roots of the denominator polynomial in Eq.7.1.

LET'S CODE!

Load a speech signal from some WAVE file. Choose any frame corresponding to a voiced sound. Find signal LPC-coefficients and formant frequencies in the frame. Draw the DFT-spectrum and LPC-spectrum on the same plot.

```
startPos = 5000;
y = x(startPos:startPos+1024)
% -------------------- compute LPC coefficients
[a, G] = lpc(y,15);
% -------------------- find formant peaks
r = roots(a)
% -------------------- plot LPC-spectrum
G = sqrt(G);
z = 10*log10(abs(freqz(G,a)));
plot(z,'r*'); hold on;

% --------------- plot normalized DFT-spectrum
plot( 10*log10(1/sqrt(1024)*abs(fft(y))) );
```

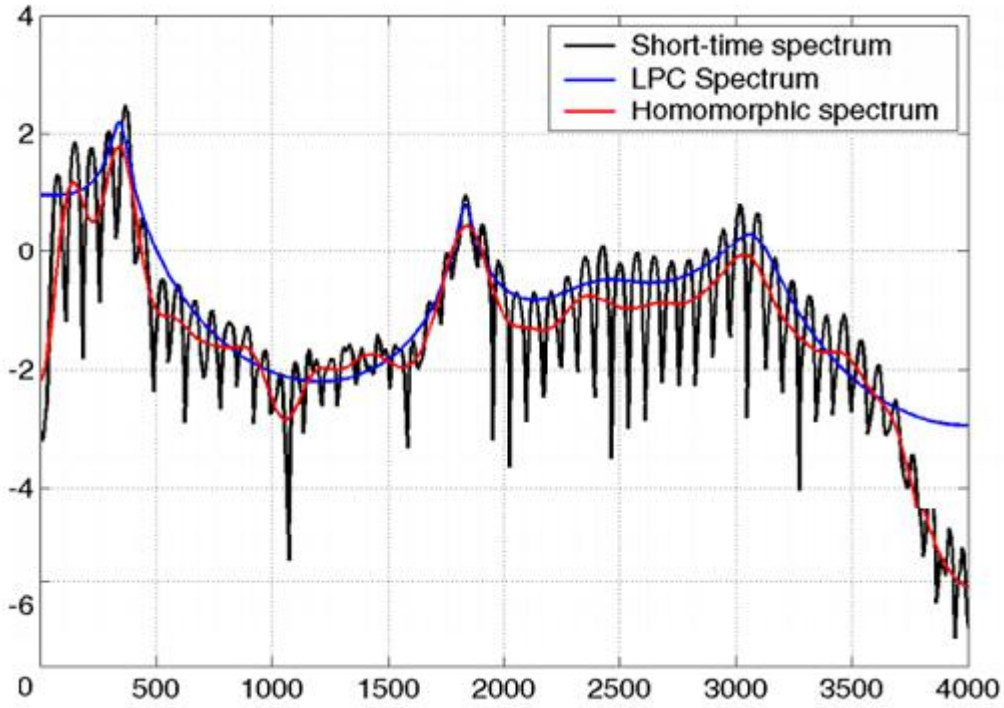

FIGURE 7.3. LPC and homomorphic representations of DFT-spectrum

7.3. Homomorphic Processing of Speech

Another important and widely used technique for speech signal processing is called *homomorphic signal processing* [3]. This term literally means: the same structure. Homomorphic techniques attempt to separate signals combined in a nonlinear way by making the problem become linear. That is, the problem is converted to the same structure as a linear system.

In case of speech signals, we can assume that any speech signal is the result of *convolution* of the excitation signal with the impulse response of the vocal tract.

Fig.7.4 shows a homomorphic system for separating signals that have been convolved. An application where this has proven useful (except the speech signal analysis) is in removing echoes from audio signals. That is, the audio signal is convolved with an impulse response consisting of a delta function plus a shifted and scaled delta function. The homomorphic transform for convolution is composed of two stages, the Fourier transform, changing the convolution into a multiplication (according to the Convolution theorem), followed by the logarithm, turning the multiplication into an addition (log(xy) = log(x) + log(y)). As before, the signals are then separated by linear filtering, and the homomorphic transform undone.

An interesting twist in Fig.7.4 is that the linear filtering is dealing with frequency domain signals in the same way that time domain signals are usually processed. In other words, the time and frequency domains have been swapped

from their normal use. For example, if FFT convolution were used to carry out the linear filtering stage, the "spectra" being multiplied would be in the time domain. This role reversal has given birth to a strange jargon. For instance, *cepstrum* (a rearrangment of *spectrum*) is the Fourier transform of the logarithm of the Fourier transform. Likewise, there are *long-pass* and *short-pass* filters, rather than low-pass and high-pass filters. Some authors even use the terms *pahse*, *Quefrency Alanysis* and *liftering*.

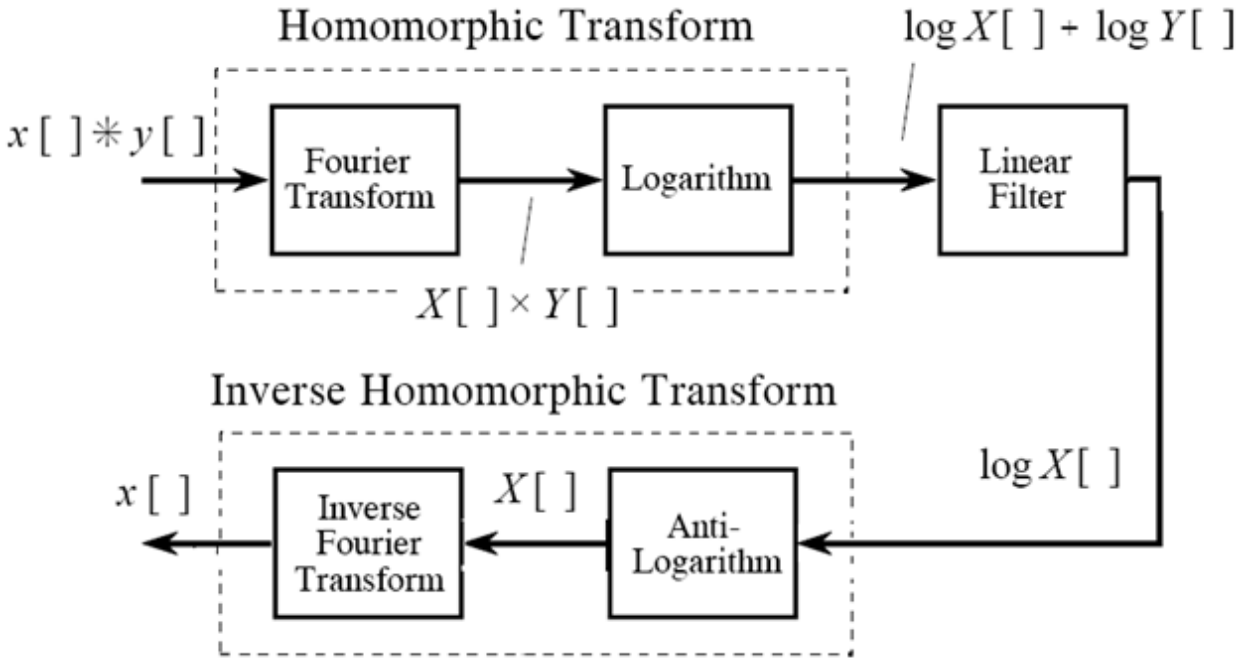

FIGURE 7.4. Homomorphic processing

LET'S CODE!
Compute and plot cepstrum of signal x[n].

```
ceps = real(ifft(log(abs(fft(x)))));
```

What does the signal cepstrum show? After we replace the multiplication with addition, the slowly varying signal (low frequencies) will be "contained" in the beginning of cepstrum, and the quickly varying signal (high frequencies) will be "contained" in higher coefficients of the cepstrum. Thus, we can eliminate the high frequency part of cepstrum (set higher coefficients to zero), and then take the inverse homomorphic transform (apply function e(x) (anti-logarithm) and FFT()). This is how the *homomorphic filtering* works.

Cepstrum can also be useful in *pitch estimation* task. Cepstrum of harmonic signals contains strict peak near the position of pitch (or fundamental frequency). In a similar manner, a peak in a complex cepstrum of signal indicates that signal contains echo (its own delayed version).

There are also two MATLAB functions *rceps()* and *cceps()* for cepstrum computation. They differ by the way they work with logarithm function. Formally, cepstrum is expressed as:

$$\hat{x}[n] = \frac{1}{2\pi}\int_{-\pi}^{\pi} \log[X(e^{j\omega})]e^{j\omega n}d\omega \tag{7.12}$$

Note, in general, logarithm in Eq.7.12 is *complex*:

$$\log[X(e^{j\omega})] = \log|X(e^{j\omega})| + j\arg[X(e^{j\omega})] \tag{7.13}$$

Such cepstrum is called *complex cepstrum* (MATLAB function cceps()). We can replace complex logarithm with its real part. In this case we'll obtain the *real cepstrum* (MATLAB function rceps()). We won't discuss the details concerning complex and real cepstrums in the guide.

After taking the inverse z-transform we get the following formula reflecting the general properties of the cepstrum (in particular, we can see that cepstrum is a decaying function):

$$\hat{x}[n] = \begin{cases} \log|A|, & n = 0 \\ \sum_{k=1}^{N}\frac{c_k^n}{n} - \sum_{k=1}^{M}\frac{a_k^n}{n}, & n > 0 \\ \sum_{k=1}^{M}\frac{b_k^n}{n}, & n < 0 \end{cases} \tag{7.14}$$

LAB6	SPEECH SIGNAL PROCESSING
	points: 10 hours: 2

Objectives

1. To perform the cepstral analysis of signals.
2. To implement the linear prediction method for analysis and synthesis of speech signals.

Exercise 6.1 [1 point]

Answer the following questions:

a. Describe the Linear Prediction (LP) method for speech signal processing. How are LP coefficients used to synthesize speech signal?
b. Give the homomorphic processing schemes for multiplication and convolution of signals.
c. How is signal cepstrum computed?
d. How is cepstrum used to obtain smoothed spectrum?
e. What is the difference between MATLAB functions *rceps()* and *cceps()*?

Exercise 6.2 (CODE) [3 points]

Write code that loads speech signal from file SPEECH.wav (you'll find it among other electronic sources of this course) and plots its spectrogram. The size of frames should be 512 samples. Write code for finding a pitch in each frame using cepstral coefficients. Plot pitch dynamics. Plot signal cepstrum computed in one of the frames corresponding to a voiced sound (choose any sound you want).

Exercise 6.3 (CODE) [2 points]

Write code that does the spectral smoothing via homomorphic filtering:

a. Load speech signal from file SPEECH.wav.
b. Compute real cepstrum of any voiced region of the signal.
c. Apply a low-pass filter in the cepstral domain (set all cepstral samples to zero, except the first N samples).
d. Perform inverse homomorphic transform of the resulting signal.

Exercise 6.4 (CODE) [4 points]

In this exercise you'll analyze and synthesize speech signal using LPC method. Write code to do the following:

a. Load signal from file SPEECH.wav, decompose it into frames 512 samples long and compute LP coefficients in each frame. Find formant frequencies (they are the poles of LP-filter transfer function *A(z)*). Plot dynamics of the 1^{st}, the 2^{nd} and the 3^{rd} formant frequencies. Plot LPC spectrum of the signal in any frame that corresponds to a voiced sound.
b. Synthesize new speech signal using computed LP coefficients and inverse filter *1/A(z)*. Use white noise as the source signal (excitation signal). Save the resulting signal into WAVE file.

LIST OF LAB ASSIGNMENTS

BIBLIOGRAPHY

1. Oppenheim, A. Discrete-time signal processing / Alan V. Oppenheim, Ronald W. Schafer. – Prentice Hall. – 1989. – 870p.
2. Orfanidis, S. Introduction to signal processing. – Prentice Hall. – 2010. – 783p.
3. The Scientist and Engineer's Guide to Digital Signal Processing [Electronic Resource] / S.W. Smith. – URL: http://www.dspguide.com. – 2011.
4. Kuo, Sen M. Real-time digital signal processing : implementations, applications and experiments with the TMS320C55X / S.M Kuo, B.H. Lee, W.Tian. – 2nd Edition. – John Wiley & Sons. – 2006. – 646p.
5. Lyons R.G. Understanding Digital Signal Processing. – 3rd Edition. – Prentice Hall. – 2011. – 984p.
6. Ifeachor, E., Jarvis B. Digital Signal Processing: A Practical Approach. – 2nd Edition. – Prentice Hall. – 2001. – 960p.
7. Blanchet, G., Charbit M. Digital signal and image processing using MATLAB. – ISTE Ltd. – 2006. – 763p.
8. V.K.Madisetti, D.B.Williams. Digital Signal Processing Handbook: CRCnetBASE 1999. – Taylor & Francis. – 1999. – 1760p.
9. Crochiere, R.E., Rabiner, L.R. Multirate Digital Signal Processing. – Prentice Hall. – 1983. – 412p.
10. Huang, X. Spoken Language Processing: A guide to theory, algorithm, and system development / X.Huang, A.Acero, H.Hon. – Prentice Hall. – 2001. – 980p.

APPENDIX A. LIST OF MATLAB DSP FUNCTIONS

blackman	Returns Blackman window in a column vector
cceps	Returns the complex cepstrum of real signal
conv	Convolution and polynomial multiplication
decimate	Resample data at a lower rate after low-pass filtering
filter	Filters input signal using difference equation coefficients
fir1	FIR filter design using the window method
fir2	FIR arbitrary shape filter design using the frequency sampling method
fft	Fast Fourier Transform algorithm
fftfilt	Overlap-add method of FIR filtering using FFT
freqz	Returns digital filter frequency response
hamming	Returns Hamming window in a column vector
hanning	Returns Hanning window in a column vector
ifft	Inverse discrete Fourier transform (Inverse FFT)
impulse	Returns the impulse response of an LTI system
interp	Resample data at a higher rate using lowpass interpolation
intfilt	Interpolation (and Decimation) FIR Filter Design
lp2bp	Low-pass to band-pass analog filter transformation
lp2hp	Low-pass to high-pass analog filter transformation
lpc	Returns Linear Predictor Coefficients
rceps	Returns the real cepstrum of a signal
remez	Parks-McClellan optimal equiripple FIR filter design
resample	Change the sampling rate of a signal
specgram	Calculate spectrogram from signal
step	Returns the step response of an LTI system
tf2zp	Transfer function to zero-pole conversion
upfirdn	Upsample, apply a specified FIR filter, and downsample a signal
window	Window function gateway
xcorr	Cross-correlation function estimates
zp2tf	Zero-pole to transfer function conversion
zplane	Z-plane zero-pole plot

Note. The given list is not a full list of MATLAB DSP functions. It includes only the functions used in the guide. For a detailed information about any command type in MATLAB command window: "help <command_name>" (for example, "help blackman").