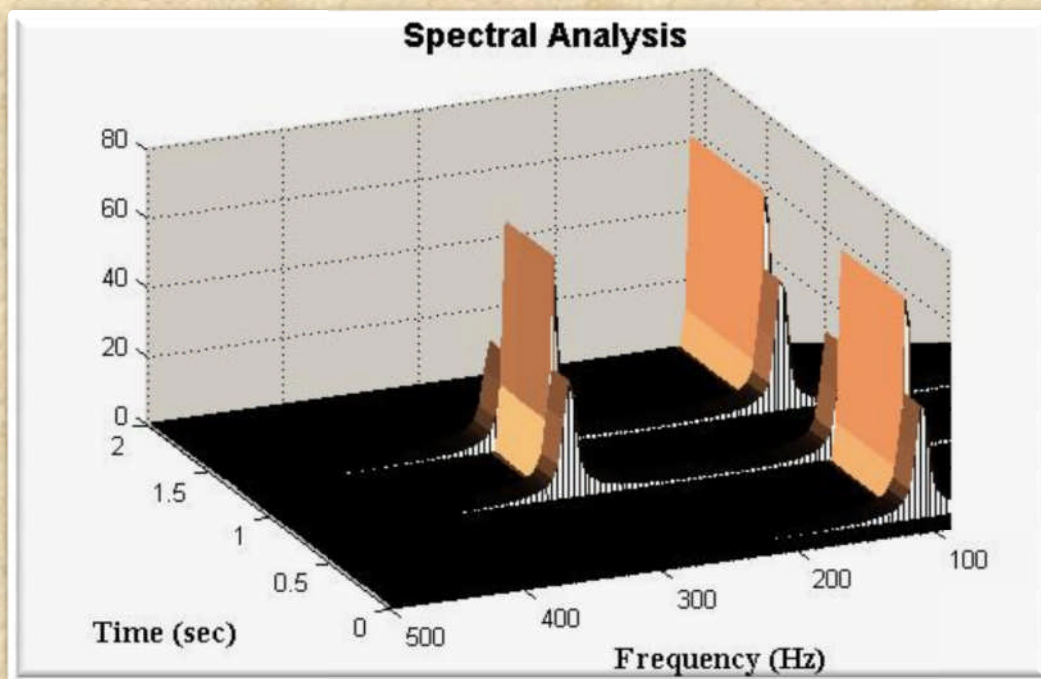


Introduction to Digital Signal Processing Theory and Applications Using MATLAB[®]



Kathleen A. K. Ossman

Introduction to Digital Signal Processing

Theory and Applications Using MATLAB[®]

Kathleen A.K. Ossman, Ph.D.

kathy.ossman@uc.edu

Department of Engineering Education

University of Cincinnati

Cincinnati, OH 45221

© 2017 Kathleen A. K. Ossman

The manuscript: "Introduction to Digital Signal Processing Theory and Applications using MATLAB[®]," is protected under the copyright laws of the United States and other countries in both electronic and printed form.

MATLAB[®] and Simulink[®] are registered trademarks of The MathWorks, Inc. and are used with permission. The MathWorks does not warrant the accuracy of the text or exercises in this book. This book's use or discussion of MATLAB[®] software or related products does not constitute endorsement or sponsorship by The MathWorks of a particular pedagogical approach or particular use of the MATLAB[®] software.

PERMISSIBLE USES

1. You may download any or all of the book in electronic form, for your personal use.
2. You may create one printed copy of any or all of the book, for your personal use.

PROHIBITED USES (Unless approved in writing by the author)

1. Using the electronic files, or printed copies derived therefrom, for any commercial purpose.
2. Making a copy of any portion of the book, either in electronic or printed form (except as allowed above).
3. Posting the book on a website or network.
4. All other uses that are prohibited under copyright law.

Table of Contents

Preface	vi
1. Introduction	1
MATLAB® Exercise: Introduction to MATLAB	3
2. Signals and Signal Spectra	11
2.1 Analog Signals	11
2.2 Discrete Signals	12
2.3 Digital Signals	13
2.4 Examples of Signals	14
2.5 Spectrum of Periodic Signals	20
2.6 Spectrum of Non-periodic Signals	24
2.7 Signal Bandwidth	26
2.8 Spectrogram	28
2.9 Application: Synthesis of Musical Instruments	30
2.10 Application: Simple Audio Effects	41
Answers to Challenge Questions	45
Chapter 2 Problems	49
MATLAB® Exercise: Sinusoidal Signals and Sound Synthesis	51
3. Sampling and Reconstructing Signals	57
3.1 The Basics of Sampling	57
3.2 A/D Conversion: Terminology and Equations	58
3.3 Types of A/D Converters	60
3.4 Nyquist-Shannon Sampling Theorem	66
3.5 Aliasing	69
3.6 Design of Anti-aliasing Filters	74
3.7 Practical Considerations	77
3.8 Reconstruction	80
3.9 Practical Application for Undersampling	83
Answers to Challenge Questions	84
Chapter 3 Problems	87
MATLAB® Exercise: Sampling, Aliasing, and Reconstruction	89
4. Key Mathematical Concepts	95
4.1 Fixed-point Representation	95
4.2 Floating-point Representation	102
4.3 Fixed-point vs. Floating-point Processors	105
4.4 Discrete Systems	105
4.5 Z-Transforms	107
4.6 Convolution	113
4.7 Inverse Z-Transforms	117
4.8 Correlation	120
Answers to Challenge Questions	124
Chapter 4 Problems	126

5. Finite Impulse Response (FIR) Filters	128
5.1 Finite Impulse Response (FIR) Digital Filters	128
5.2 Designing Low-Pass FIR Filters	130
5.3 Designing High-Pass, Band-Pass, and Notch FIR Filters	141
5.4 Designing FIR Filters using FDAT	145
5.5 Practical Considerations for FIR Filters	156
5.6 Application: Special Types of FIR Filters	163
5.7 Multi-Rate Digital Signal Processing	175
Answers to Challenge Questions	185
Chapter 5 Problems	189
MATLAB® Exercise: FIR Filters	192
6. Infinite Impulse Response (IIR) Filters	202
6.1 Infinite Impulse Response (IIR) Digital Filters	202
6.2 IIR Filter Design	205
6.3 Designing IIR Filters using FDAT	207
6.4 Filter Realizations	215
6.5 Practical Considerations for IIR Filters	218
6.6 Examples of of IIR Filters	230
6.7 Application: Audio Effects Using IIR Filters	233
Answers to Challenge Questions	247
Chapter 6 Problems	250
MATLAB® Exercise: IIR Filters	252
7. Discrete Fourier Transforms	258
7.1 Discrete Fourier Transforms	258
7.2 Windowing the Data to Enhance Spectral Analysis	270
7.3 Fast Fourier Transforms	272
7.4 Power Spectral Density	283
7.5 Convolution and Correlation using Fast Fourier Transforms	286
7.6 Matched Filters	291
7.7 Algorithms for Focused Spectral Analysis	295
Answers to Challenge Questions	303
Chapter 7 Problems	305
MATLAB® Exercise: Fast Fourier Transforms	308
8. Real-Time Processing Algorithms	313
8.1 Difference Equations and Circular Addressing	313
8.2 Overlap-Add and Overlap-Save Algorithms for Convolution or Correlation	314
8.3 Real-time Spectral Analysis using Fast Fourier Transforms	318
8.4 Sliding Discrete Fourier Transform	329
8.5 Adaptive Filtering	342
Answers to Challenge Questions	347
Chapter 8 Problems	349
MATLAB® Exercise: Introduction to Adaptive FIR filters	351

9. Introduction to Image Processing	357
9.1 Terminology and Common Image Formats	357
9.2 Image Effects	359
9.3 De-noising Images	374
9.4 Image Compression: JPEG Format	377
Answers to Challenge Questions	405
Chapter 9 Problems	407
MATLAB® Exercise: Image Processing	409
10. Introduction to Radar Processing	414
11.1 Introduction	414
11.2 Performance Metrics	416
11.3 Matched Filters, Chirp Signals, and Pulse Compression	420
11.4 Determining Range	432
11.5 Pulse Doppler Processing to Estimate Target Velocity	455
Chapter 10 Problems	465
MATLAB® Exercise: Radar Range Processing	468
Appendix	471
Table A.1: Fourier Series	472
Table A.2: Fourier Transforms	473
Table A.3: Fourier Transform Properties	474
Table A.4: Z-Transform Properties	474
Table A.5: Z-Transforms	475
Bibliography	476
Index	479

Preface

This book provides an introduction to the field of digital signal processing (DSP) and includes several application areas in addition to basic theory.

Pedagogical Approach

- Thorough coverage of sampling, design of digital filters, and Discrete Fourier transforms is provided with practical considerations included throughout.
- Every DSP concept introduced in the text is followed with clear, carefully written numerical examples to illustrate the concept.
- Graphics are used to help students visualize concepts rather than becoming lost in the mathematics.
- MATLAB[®] is used extensively throughout the textbook both to help students comprehend digital signal processing concepts and to teach students to use MATLAB[®] as a design tool.
- Lab exercises are included at the end of each chapter to provide students with hands on practice with the material covered in the chapter. These exercises could be used as a lab manual for a lab course accompanying a DSP lecture course.
- Challenge questions are spread throughout each chapter to allow students to test their understanding of the material as they are reading the text. Answers are provided at the end of each chapter.
- The text includes many practical applications for DSP including image processing, music synthesis, and radar processing to keep students interested in the material.
- Problems are included at the end of each chapter.

Intended Audience

This book is written for electrical and computer engineering students and for engineering technology students. For engineering students, the text serves as a practical introduction to the subject with interesting applications. For engineering technology students, the text provides a basis for developing good applications and projects. Engineering professionals with little or no background in digital signal processing would find this text to be a useful introduction to DSP as well providing good instruction on how to use MATLAB[®] and Simulink[®].

About the Author

Kathleen Ossman earned her BSEE and MSEE from the Georgia Institute of Technology and a Ph.D. in Electrical Engineering from the University of Florida. She is an associate professor at the University of Cincinnati and has enjoyed teaching electrical engineering courses and labs for many, many years.

Chapter Synopses

Chapter 1: Introduction

This chapter introduces a digital signal processing system, list several practical applications, and provides a brief outline of the rest of the text. The lab exercise at the end of the chapter provides an introduction to using MATLAB® – inputting data, data formats, plotting signals, and writing functions.

Chapter 2: Signals and Signal Spectra

This chapter introduces signals, both analog and digital. Periodic and non-periodic signals are discussed. Signals are represented both in the time domain (very familiar to students) and in the frequency domain (not so familiar) through several illustrative examples. Fourier series, Fourier transform, definition of signal bandwidth, and spectrograms are included in the frequency domain discussion. Applications include using additive or FM synthesis to synthesize various musical instruments and an introduction to simple audio effects such as tremolo and ring modulation. The lab exercise at the end of the chapter stresses signal representation in both the time and frequency domain using MATLAB® and includes some interesting audio applications.

Chapter 3: Sampling and Reconstructing Signals

This chapter deals with an extremely important DSP concept – sampling analog signals. A/D converters are covered in detail including basic terminology, types of converters (successive approximation, flash, and sigma-delta), and the importance of matching the input signal to the A/D converter. The sampling theorem is stated and aliasing is discussed and illustrated through several examples. The design of analog anti-aliasing filters is covered using a really nice design tool available for free from Texas Instruments called FilterPro. The process of reconstruction, recreating an analog signal from digital samples using interpolation filters, is also covered. The lab exercise at the end of the chapter includes several exercises to illustrate the principles of A/D conversion, sampling frequency, aliasing, and reconstruction.

Chapter 4: Key Mathematical Concepts

This chapter covers fixed-point and floating-point representation of numbers and compares and contrasts fixed-point and floating-point processors for various applications. Discrete systems and Z-transforms are also covered in preparation for the chapters on digital filter design and analysis. Finally, convolution and correlation are covered in preparation for the chapters on digital filters, fast Fourier transforms, and radar processing.

Chapter 5: Finite Impulse Response (FIR) Filters

This chapter begins by defining an FIR filter then covers the design of FIR filters using the common techniques of windowing, optimal design, and custom design. The Filter Design and Analysis Tool (FDAT), a MATLAB® tool for designing digital filters, is introduced. This chapter includes multiple examples of many different types of FIR filters. Special types of FIR filters are introduced including feedforward comb filters, Gaussian filters for blurring images, and filters for creating chorus and flanging effects for audio signals. Multi-rate processing, the conversion from one sampling rate to another, is covered at the end of the chapter. The lab exercise at the end of the chapter provides students with practice using the FDAT and allows them to experiment with different designs using windowing and optimal design.

Chapter 6: Infinite Impulse Response (IIR) Filters

This chapter begins by defining an IIR filter then covers the design of IIR filters using the FDAT to illustrate concepts through several examples. The critically important concept of IIR filter realization using biquads is covered in detail. FIR and IIR filters are compared and contrasted for various practical applications. Several common IIR filters including echo filters, reverb filters, and graphic equalizers are discussed. The lab exercise at the end of the chapter provides students with additional practice in using the FDAT for IIR filter design and allows them to experiment with the effect of coefficient wordlength on filter stability and performance for a full and a biquad realization of filters.

Chapter 7: Discrete Fourier Transforms

This chapter introduces students to one of the more difficult and mathematically challenging DSP topics, discrete Fourier transforms (DFT). Several illustrative examples are included to stress the effect of sampling frequency and time duration on spectral resolution and accuracy. The Fast Fourier Transform (FFT), an efficient computational algorithm for computing a DFT, is explained. The Radix-2 decimation in time FFT algorithm is explained through equations, graphs, and the classic butterfly diagram. Flow graphs (unique to this text) are used to clearly show the operations involved in this FFT algorithm and the memory allocation. The end of the chapter includes some of the practical applications for FFTs: power spectral density, fast convolution and correlation, matched filters, and Zoom FFTs for focused analysis. The lab includes several exercises that are designed to illustrate the effects of sampling frequency and time duration on the accuracy and resolution of the resulting spectrum.

Chapter 8: Real-Time Processing Algorithms

This chapter covers some useful algorithms for real-time processing of signals including the sliding Discrete Fourier Transform, the Overlap Add and Overlap Save algorithms for real-time convolution or filtering, and an introduction to adaptive filters. The first lab exercise requires students to write their own m-file for an overlap-add processor. The second lab exercise introduces adaptive filter blocks in Simulink®.

Chapter 9: Introduction to Image Processing

This chapter provides an introduction to image processing and includes filters to create image effects including contrast enhancement, blurring, sharpening, edge detection and de-noising. Image compression using the jpeg format is illustrated through several examples which detail the use of discrete cosine transforms and Huffman coding. In the lab exercise, students explore image processing concepts and gain experience with 3-D data using MATLAB®.

Chapter 10: Introduction to Radar Processing

This chapter provides an introduction to radar processing. Chirp signals and other pulse signals are introduced. Designing and using matched filters to process returning pulse signals in order to determine range and velocity of targets is illustrated through examples and through the lab exercise at the end of the chapter.

Appendix

The appendix includes reference tables for Fourier series, Fourier Transforms, and Z-Transforms

Adopting this Text

For schools on the quarter system, Chapters 1-6 can be covered in the first quarter with the remainder of the text (Chapters 7-10) covered in the second quarter. For schools on the semester system, Chapters 1-7 and additional topics from one of the later chapters can be covered in a single semester (Chapters 8-10 are independent of one another). The remaining chapters could be covered in a second semester with some supplemental material (perhaps project work using DSP hardware) added.

Acknowledgements

I would like to thank all of my students over that last couple of decades, particularly the individuals who strongly encouraged me to write a textbook and the ones who provided feedback on the rough draft of the manuscript. Watching my students progress from freshmen year, often uncertain as to what engineering is all about, to senior year, prepared and confident to move on into the workplace, has been a very rich and rewarding experience for me.

Thanks also go to my husband, Bill, and my sons, Chris and Mike, who were very patient with the late evenings and long weekends spent writing. I'd also like to acknowledge my parents, Norm and Julie Kurtz, who encouraged me to study hard, take challenging classes, and discover what I was capable of. My colleagues at the College of Applied Science, University of Cincinnati, were very supportive of my efforts in writing this text. Three colleagues in particular, also dear friends, offered valuable advice and encouragement: Claudia Taylor, Brian Resnick, and Virginia Westheider.

CHAPTER 1 INTRODUCTION

Digital signal processing (DSP) technology has expanded at a rapid rate to include such diverse applications as CDs, DVDs, MP3 players, iPods, digital cameras, digital light processing (DLP) for high quality television pictures, cellular phones, high-speed modems, satellite systems, GPS receivers, medical imaging (X-rays, MRIs), voice recognition systems, radar processing, missile guidance systems, and even toys. A few decades ago, digital signal processing was taught only in graduate schools to electrical engineering majors. Today, it is a required course in most undergraduate programs in electrical and computer engineering and is often included in engineering technology programs and many electronic music programs as well.

Figure 1.1 is a basic block diagram for a digital signal processing system. An analog signal is converted to a digital signal, processed in some manner, and then converted back to analog if appropriate. A simple example of this would be a CD. Analog music is sampled at a rate of 44.1 kHz (44,100 samples/sec) and the digital data is stored on a CD. When the CD is played, the digital data stored on the CD is converted back to analog and played through the speakers. There is a lot of additional processing including data interleaving and error-correction coding that is used for CDs, but for now, this simple explanation will have to suffice.

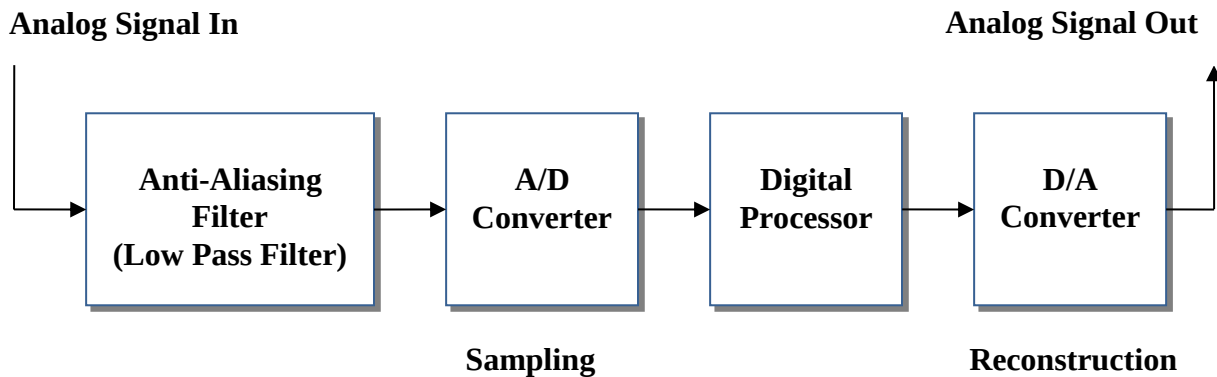


Figure 1.1: Block Diagram of a DSP System

This text introduces both the theory behind digital signal processing and some practical applications. Chapter 2 covers digital and analog signals and signal spectra. A lot of DSP processing occurs in the frequency domain so an understanding of signal spectra is critical. The sampling theorem and A/D and D/A conversion are covered in Chapter 3. Some of the key mathematical concepts including fixed-point and floating-point number systems, Z-Transforms, convolution, and correlation are introduced in Chapter 4. Chapters 5 and 6 cover digital filter design and many of the practical considerations associated with digital filter design and implementation. Multi-rate digital signal processing systems, including interpolation and decimation, are also covered in Chapter 5. Chapter 7 provides a detailed explanation of Discrete

Fourier Transforms – a mathematically challenging but extremely practical topic. Chapter 8 covers some useful algorithms for real-time processing of signals including the sliding Discrete Fourier Transform, the Overlap Add and Overlap Save algorithms for real-time convolution or filtering, and an introduction to adaptive filters. Although many applications are introduced in each of the earlier chapters, Chapters 9 and 10 focus on specific DSP applications. Chapter 9 is an introduction to image processing and includes blurring filters, sharpening filters, edge detectors, and other algorithms for altering an image. The text concludes with an introduction to radar processing in Chapter 10. Pulsed radar signals are processed to determine range to target and to estimate target velocity.

MATLAB[®] is used extensively throughout the text and is an excellent tool for learning about digital signal processing. Many of the examples in the text use MATLAB. The code used to generate the results and the figures is included. Students are encouraged to repeat these examples using MATLAB and try variations to develop an in-depth understanding of the topics. Most of the chapters also include an extensive MATLAB lab exercise designed to further explore concepts covered in the chapter. These lab exercises provide an opportunity to practice concepts and develop a deeper understanding of the material.

Each chapter also includes challenge questions to test students' understanding of the concepts and to provide a preview of topics covered later in the text. Students are encouraged to review the section concepts and try to think through these questions. Answers for all challenge questions are at the end of the chapter and should definitely be reviewed before moving on to the next section or topic.

The text website provides access to the MATLAB m-files, Simulink[®] mdl files, and other types of files required for some of the lab exercises and examples in the text. Links to some useful and informative DSP sites are also provided.

The lab exercise at the end of this chapter provides an introduction to using MATLAB. Students who are familiar with MATLAB may just want to skim the exercise for a quick review.

CHAPTER 1 LAB EXERCISE

Introduction to MATLAB

Objectives

1. Learn how to enter data into MATLAB and perform calculations.
2. Learn to plot functions in MATLAB.
3. Learn to write an m-functions for MATLAB.

Procedure

A. The MATLAB desktop

Launch MATLAB. A desktop similar to the one shown in Figure 1 should come up, but may differ in appearance depending on the version of MATLAB being used. The Command Window is where commands are typed and executed. The Workspace Window displays all variables defined in the current MATLAB session.

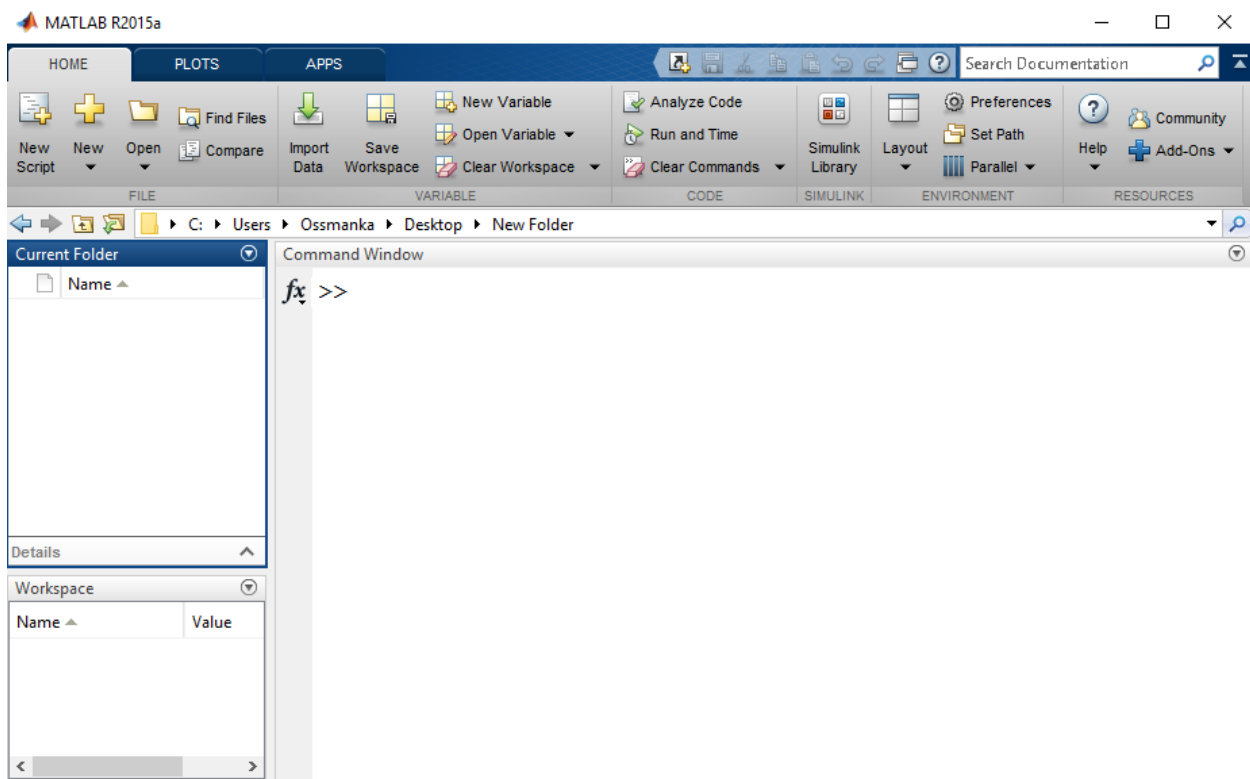


Figure 1: MATLAB Desktop

1. At the command prompt (`>>`) type the command: `a=5` then hit Enter. You should see `a = 5` appear in the Command Window. The variable, `a`, should also appear in the Workspace Window.

2. Next type the following command and hit Enter: `b=7;`
Placing a semicolon at the end of a command line suppresses the output! However, the variable `b` with a value of 7 has been created and shows up in the Workspace Window.
3. To view or display a variable, simply type the variable name and hit Enter. For example, typing `b` at the command prompt will cause `b = 7` to appear on the screen.
4. If you wish to repeat a previous command without re-typing it, there are two easy ways to accomplish this. In the Command Window, just hit the up-arrow as many times as necessary to travel back to the previous command. In the Command History Window, scroll back to the desired command and double click on it.

B. Variables, Vectors, and Matrices in MATLAB

Variable names can be defined using a combination of upper and lower case letters. If possible, variable names should reflect the nature of the variable being defined. Keep in mind:

- MATLAB is case sensitive (i.e., `A` is a different variable than `a`)
 - `i` and `j` are defined in MATLAB to be $\sqrt{-1}$ to accommodate complex numbers. Do not use these variables for anything else or you will overwrite $\sqrt{-1}$.
 - `pi` is defined in MATLAB to be the constant $\pi = 3.1416\dots$. Keep it reserved for this purpose.
1. Vectors and Matrices are entered into MATLAB using brackets `[ ]`. Entries are separated by spaces (or commas) while rows are separated using a semicolon. Type the following set of commands.

```
row_vector = [1 3 7 15 25]
column_vector = [1; 3; 5; 7]
matrix = [1 2 3; 4 5 6; 7 8 9]
```

Now hit Enter and record the output in the space below:

2. Complex numbers are entered using the pre-defined variables `i` or `j`. Type the following command: `d = 2 + 7i`

Result = _____

3. To get a list of currently defined variables, use the command **whos**. You should get a list of variables similar to the following showing everything you have already defined.

Name	Size	Bytes	Class
a	1x1	8	double array
b	1x1	8	double array
column_vector	4x1	32	double array
d	1x1	16	double array (complex)
matrix	3x3	72	double array
row_vector	1x5	40	double array

The **whos** command gives a complete listing of all variables currently defined and the size of those variables. The **who** command simply lists all variable names currently in use. This information is also available in the Workspace Window on the MATLAB desktop.

4. Vectors with equally spaced entries can easily be entered in MATLAB. These are useful as time or frequency variables. Type the following set of commands.

```
t1 = 0:0.1:1
t2 = linspace(0,5,10)
f = logspace(1,2,10)
```

Record the output in the space below.

Notes:

- The command ***linspace(t1,t2,N)*** will create a vector with N equally spaced points between t1 and t2.
- The command ***logspace(d1,d2,N2)*** will create a vector with N logarithmically spaced points between 10^{d1} and 10^{d2} .
- The command ***clear*** will clear all variables.
- The command ***clear t1,t2*** will simply clear variables t1 and t2.

C. Formatting Data

There are several options for formatting data in MATLAB.

1. Open the MATLAB command window and type *help format* at the prompt. Read about the different formats for displaying numbers.

Execute the following commands and write the output under each command:

`format short; number = 0.005152535455` Output = _____

`format long; number` Output = _____

`format short e; number` Output = _____

`format long e; number` Output = _____

2. Return MATLAB to the default (short) by typing *format* at the command prompt and hitting Enter.

D. MATLAB Operations and Functions

MATLAB utilizes many operations and functions and follows the math order of operations rules. Some of the operations and functions are listed below.

Operations	Functions	
+ Addition	<code>abs</code>	<i>returns magnitude of a number</i>
– Subtraction	<code>angle</code>	<i>returns angle of a number in radians</i>
* Multiplication	<code>cos</code>	<i>cosine assuming angle is in radians</i>
/ Divide	<code>sin</code>	<i>sine assuming angle is in radians</i>
^ Power	<code>exp</code>	<i>exponential function</i>
' Transpose	<code>sqrt</code>	<i>returns square root of a number</i>

Application: Solving Systems of Equations

Suppose we wish to solve the following set of linear equations:

$$2x + 3y - 7z = 10$$

$$3x + y + 4z = 12$$

$$x + 2y - z = -15$$

Execute the following commands (note that % is used for comments):

```
A = [2 3 -7; 3 1 4; 1 2 -1]; %Coefficients of the 3 equations
b = [10; 12; -15];           %Constants on right hand side
inv(A)*b
ans =    15.8437
      -17.6562
      -4.4688
```

The solution is: $x = 15.8437$, $y = -17.6562$, and $z = -4.4688$.

Exercise: Solve the following system of linear equations.

$$\begin{aligned}x + 2y - 5z &= -8 \\ 3x + y + 4z &= 9 \\ x + 2y - z &= 0\end{aligned}$$

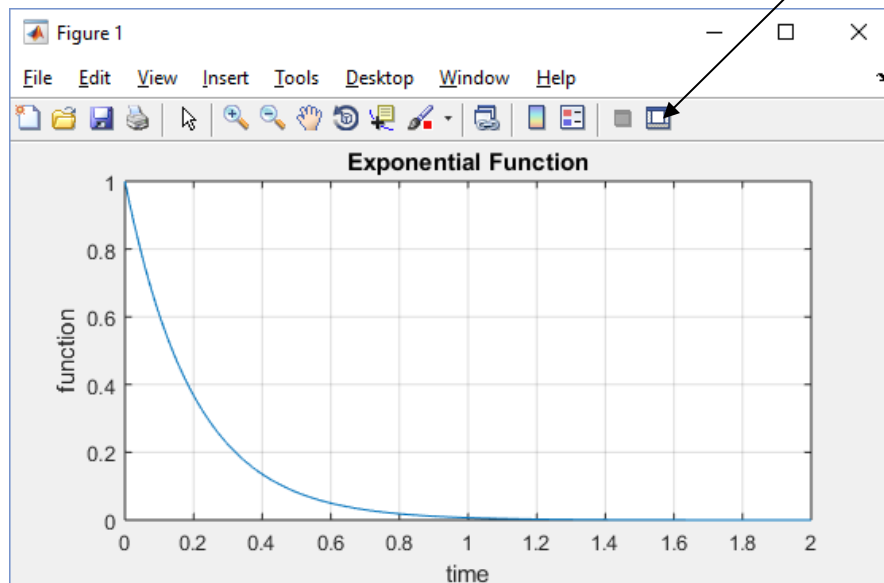
List MATLAB commands and solution below:

E. Plotting Signals

1. Plot the function $f(t) = e^{-5t}$ by executing the following set of commands:

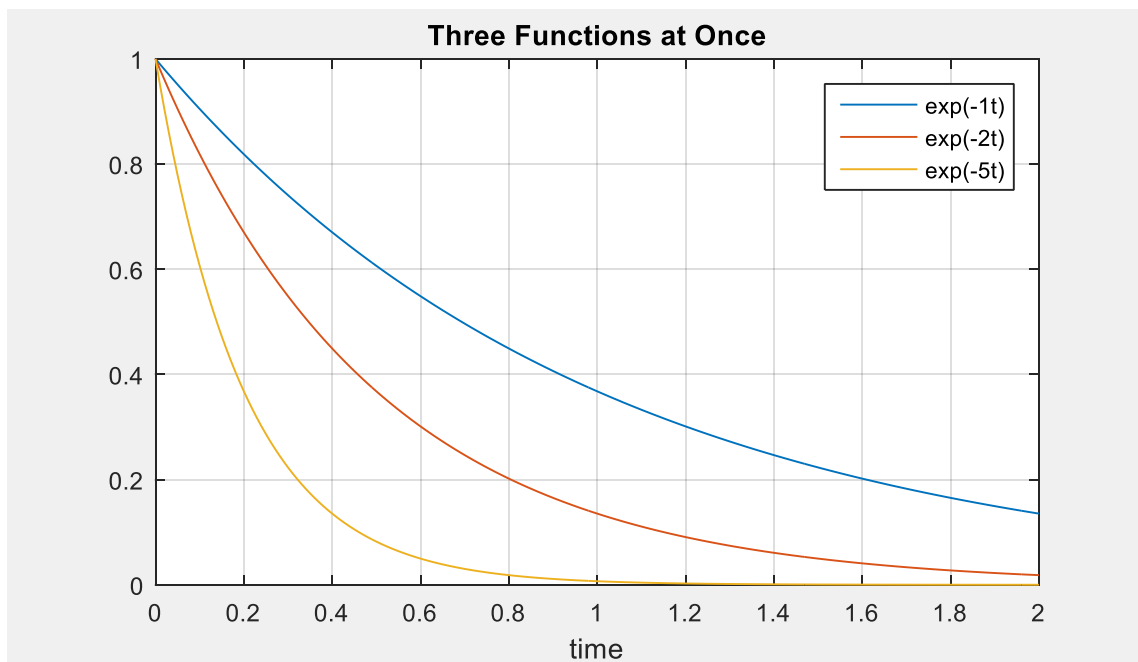
```
t = 0:0.01:2;
f=exp(-5*t);
plot(t,f); title('Exponential Function');
xlabel('time'); ylabel('function');grid
```

Plot Tools
Icon for
Editing Plots



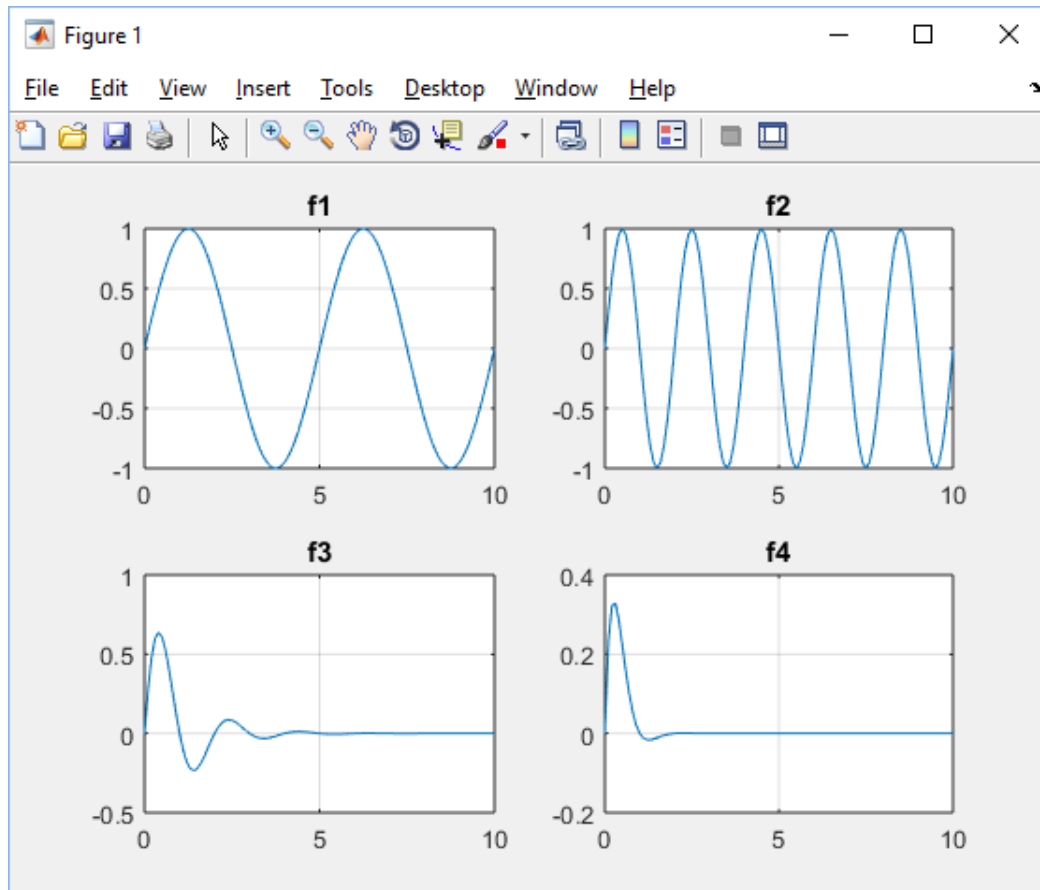
- Click on the plot tools icon at the top of the figure window. Experiment with changing color, line style, and other options within plot tools. Notice that title and axis labels can also be added using plot tools rather than as commands in the command window. When finished editing the plot, simply click on the close plot tools icon.
- Plot the functions $f_1(t) = e^{-1t}$, $f_2(t) = e^{-2t}$, and $f_3(t) = e^{-5t}$ on the same graph by executing the following commands:

```
t = 0:0.01:2;
f1 = exp(-1*t); f2 = exp(-2*t); f3 = exp(-5*t);
plot(t,f1,t,f2,t,f3);
title('Three Functions at Once'); xlabel('time'); grid;
legend('exp(-1t)', 'exp(-2t)', 'exp(-5t)')
```



- Execute the following set of commands to divide the figure into several partitions and put a plot in each partition using the subplot function:

```
t=0:0.1:10;
f1 = sin(2*pi*0.2*t);
f2 = sin(2*pi*0.5*t);
f3= exp(-1*t).*sin(2*pi*0.5*t);
f4=exp(-3*t).*sin(2*pi*0.5*t);
subplot(2,2,1); plot(t,f1); title('f1');grid
subplot(2,2,2); plot(t,f2); title('f2');grid
subplot(2,2,3); plot(t,f3); title('f3');grid
subplot(2,2,4); plot(t,f4); title('f4');grid
```

5. Figures created in MATLAB can easily be imported into other documents simply by clicking on Edit → Copy Figure in the Figure Window.
6. To find out about more of the plot features, type the command: *help plot* in the MATLAB command window.

F. Creating m-functions in MATLAB

1. In MATLAB, click on NEW, then choose Function. Type the MATLAB commands shown in Figure 2 into the editor window for your new function.

Comments:

- The first line is a function line that defines the outputs (variables that will be sent back to the MATLAB workspace) and the inputs (variables that must be defined in order to run the m-file). For this function, the output variables are *signal* and *t* and the input variables are *gain* and *power*.
- The next line(s) are comments documenting what the function does
- The next set of lines are the MATLAB commands needed to complete the function

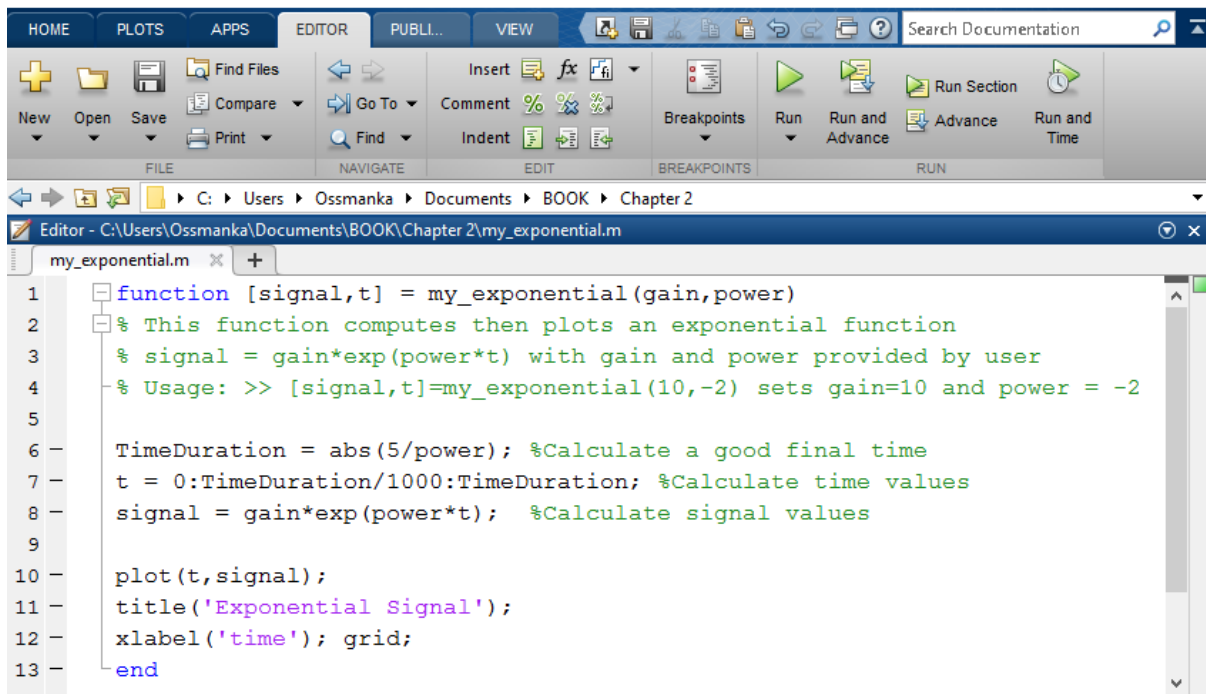


Figure 2: MATLAB FUNCTION

2. Save the file in your current directory folder as my_exponential. The file must be saved under the same name as the function. In order to run the function, it must be located in your current directory.
3. Close the edit session and in the MATLAB workspace type `help my_exponential` at the MATLAB command prompt. You should see all the comment lines prior to the first executable command. These are extremely helpful in reminding you what the function does and what input arguments need to be included.
4. At the MATLAB command prompt, type the command

```
[f,t] = my_exponential(2,-5);
```

You should see a plot of an exponentially decaying function. Also, the variables `f` and `t` should appear in your workspace window. `f` and `t` are vectors of the signal and time values created when the function was executed.

MATLAB functions make MATLAB a customizable program and are very useful for calculations that you need to do often. They are also very useful if you have a very long set of commands that you don't wish to execute one at a time in the command window. You can also create a **script** file instead of a **function** file. A **script** file has no input or output arguments and simply executes all of the commands listed in the file. A script file is executed by simply typing the name of the file in the command window or by selecting Run in the MATLAB editor window. As with m-function files, a script file will only run if the file is located in your current directory.

CHAPTER 2 SIGNALS AND SIGNAL SPECTRA

Knowledge of some basic types of signals and signal spectra is essential to understanding and applying digital signal processing. This chapter introduces analog and digital signals, explains what a signal spectrum is, defines signal bandwidth, and describes why these concepts are important. Applications include synthesis of musical instruments and simple audio effects.

2.1 ANALOG SIGNALS

Analog signals are signals that exist over a continuum of time and can take on a continuum of values. What does this mean? Looking at an example of an analog signal will help explain the definition.

Example 2.1: Graphing an Analog Signal using MATLAB®

Consider an analog sine wave signal defined as follows

$$y(t) = 5\sin(t) \quad 0 \leq t \leq 10$$

This signal could be plotted using a graphing calculator or a software program. The following set of commands in MATLAB will produce a nice plot of this signal shown in Figure 2.1:

```
t = 0:0.01:10;           %Produces t = [0 0.01 0.02 ... 9.99 10]
y = 5*sin(t);            %Evaluates function at each time in t
plot(t,y);               %Plots the sine function
title('5sin(t)');
xlabel('time (sec)');grid %Create graph labels
```

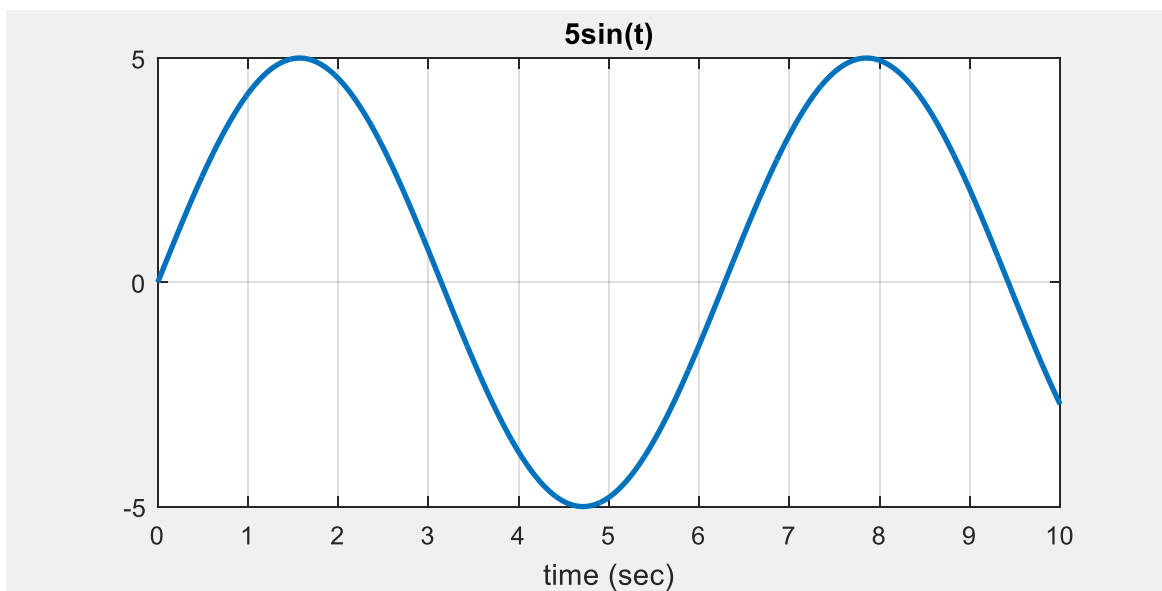


Figure 2.1: Analog Signal from Example 2.1

The signal, $y(t)$, as defined in the example exists over a continuum of time from 0 to 10 seconds. This means *any time* between 0 and 10 seconds can be selected, such as 2.54378 seconds, and the analog signal takes on a value at this point in time. The signal, $y(t)$, as defined in the example also takes on a continuum of values from -5 to $+5$. This means *any value* between -5 and $+5$ can be selected, such as 0.75637458, and there are points in time where the signal takes on that particular value.

Challenge Question 2.1

MATLAB was used to create a plot of the analog signal, $5\sin(t)$. The plot certainly looks like an analog signal. Is the signal y defined and stored in MATLAB an analog signal? Why or why not?

2.2 DISCRETE SIGNALS

Discrete Signals are signals that take on values only at evenly spaced instances of time.

Example 2.2: Graphing a Discrete Signal using MATLAB

Consider a discrete sine wave signal defined as follows

$$y(k) = 5\sin(0.2k) \quad k = 0, 1, 2, \dots, 50$$

This signal takes on values every 0.2 seconds starting at 0 seconds and ending at 10 seconds. The following set of commands in MATLAB produces the plot of this signal shown in Figure 2.2:

```
k = 0:1:50;           % k = [0 1 2 ... 50]
t = 0.2*k;            % t = [0 0.2 0.4 ... 10]
y = 5*sin(0.2*k);
plot(t,y,'b. ');
title('5sin(0.2k)'); xlabel('time (sec)'); grid
```

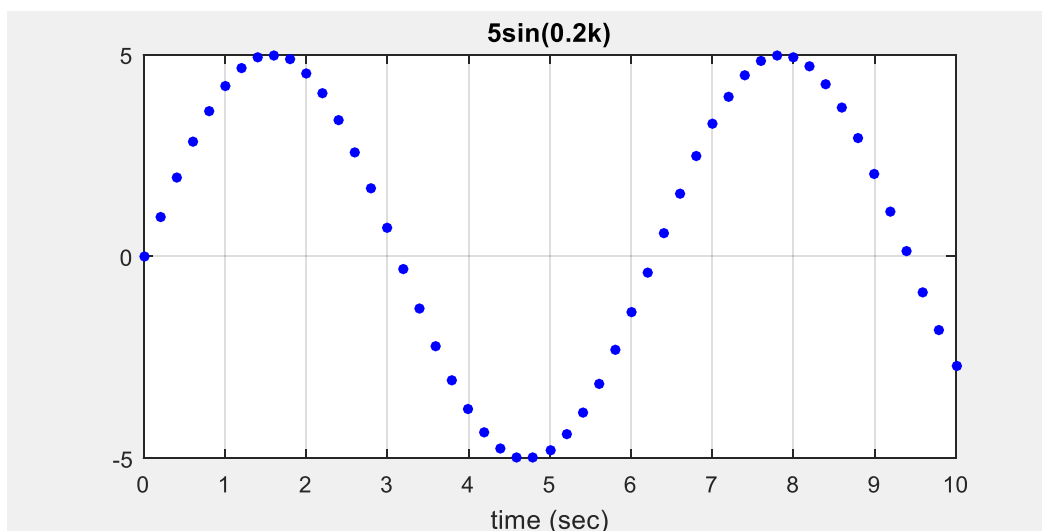


Figure 2.2: Discrete Signal from Example 2.2

The discrete signal in Example 2.2 is simply a sampled version of the analog signal in Example 2.1. Samples are taken every 0.2 seconds over a 10 second interval of time. The discrete signal matches the analog signal exactly at the sampling instants. The values of the analog signal for all of the times in between the sampling instants are lost. For example, the analog signal has a value at 0.1 seconds but the discrete signal does not.

Challenge Question 2.2

If the sampling interval was increased from 0.2 seconds to 2.0 seconds in Example 2.2, would the plot of the discrete signal look like the plot of the original analog signal?

2.3 DIGITAL SIGNALS

Digital signals are signals that can take on only a finite set of values (levels) over evenly spaced instances of time.

Example 2.3: Digital Signal vs. Discrete Signal

Consider the discrete sine signal from the previous example

$$y(k) = 5\sin(0.2k) \quad k = 0, 1, 2, \dots, 50$$

Suppose the values of $y(k)$ are rounded to the nearest tenth. This creates a digital signal that can take on only a finite set of values: $-5, -4.9, -4.8, \dots, 0, 0.1, 0.2, \dots, 4.9, 5$. Table 2.1 displays the values for both the discrete signal and the digital signal from 0 to 2 seconds.

Table 2.1: Sample Values for Sinusoidal Signal of Example 2.3

Time	Discrete Signal	Digital Signal
0	0	0.0
0.2	0.99335	1.0
0.4	1.94709	1.9
0.6	2.82321	2.8
0.8	3.58678	3.6
1	4.20735	4.2
1.2	4.6602	4.7
1.4	4.92725	4.9
1.6	4.99787	5.0
1.8	4.86924	4.9
2	4.54649	4.5

How does the digital signal compare to the original analog signal from Example 2.1? Information has definitely been lost. The digital signal only exists at the sampling instants of 0, 0.2, 0.4, $\dots$ 9.8, 10. The analog signal information between these sampling instants has been lost. In addition, the digital signal does not match the analog signal at the sampling instants.

The digital signal has been rounded to the nearest tenth resulting in rounding or quantization error. This loss of signal information will be discussed in greater detail in Chapter 3.

Challenge Question 2.3

How could the digital signal in Example 2.3 be changed or re-defined to reduce the level of quantization error?

2.4 EXAMPLES OF SIGNALS

Sine Waves

One very common analog signal is a sine wave. The sine wave is a periodic signal. Periodic signals are signals which repeat over a certain interval (period) of time. A general equation for a sine wave is

$$y(t) = A\sin(2\pi ft + \theta) = A\sin(\omega t + \theta) \quad (2.1)$$

A = amplitude (maximum value)

f = frequency (Hz) = number of cycles per second

ω = frequency (rad/sec)

θ = phase shift (rad)

$T = 1/f$ = period (sec) = time required for one complete cycle

Example 2.4: Plotting sinusoidal functions

Use MATLAB to plot four cycles of a 1250 Hz sine wave with amplitude equal to 3.5 and a phase shift of zero and four cycles of a 1250 Hz sine wave with amplitude equal to 4.5 and a phase shift of -45° .

Solution

First, write an equation for each of the sine waves:

$$\begin{aligned} y_1(t) &= 3.5\sin(2\pi(1250)t) \\ y_2(t) &= 4.5\sin(2\pi(1250)t - 45^\circ) \end{aligned}$$

How much time does four cycles of a 1250 Hz sine wave take? The period of a 1250 Hz signal is given by

$$T = 1/1250 = 0.0008 \text{ seconds}$$

Therefore, four cycles would require $4 \times 0.0008 = 0.0032$ seconds. The following set of commands in MATLAB will generate and plot the required signals:

```
t = 0:1e-5:4*0.0008;           % t = [0 0.00001 0.00002 ... 0.0032]
y1 = 3.5*sin(2*pi*1250*t);
y2 = 4.5*sin(2*pi*1250*t - 45*pi/180); % sin requires radians!
plot(t,y1,t,y2)
title('Two sine waves');
xlabel('time (sec)'); grid; legend('y1','y2')
```

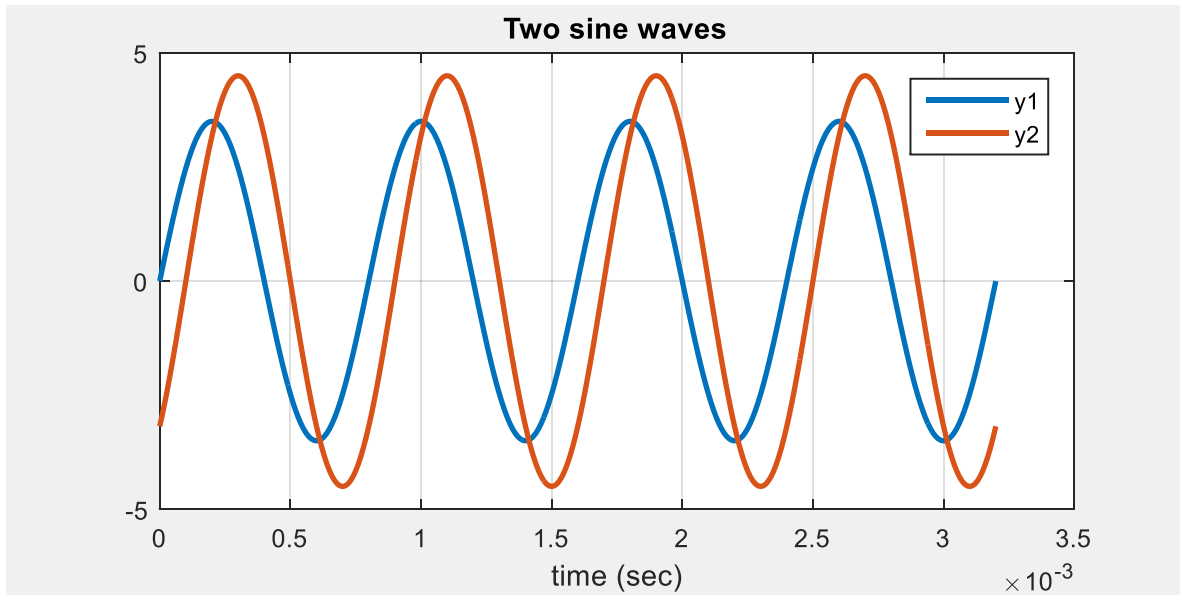


Figure 2.3: Signals from Example 2.4

Notice that the phase shift causes a time delay between the two signals.

Challenge Question 2.4

Look at the MATLAB code for Example 2.4. How many samples of the sine wave are being taken in each cycle?

Linear Combinations of Sinusoidal Signals

What happens when sine waves are added together? As shown in the next example, linear combinations of sine waves produce periodic signals.

Example 2.5: Sum of Sinusoidal Signals

- (a) Graph the signal: $\sin(2\pi t) + \sin(2\pi(4)t)$
- (b) Graph the signal $\sin(2\pi t) + \sin(2\pi(1.5)t)$

Solution

- (a) First, figure out the period of each sine wave in order to determine a good time range for the plot. The first sine wave has a frequency of 1 Hz and therefore has a period of 1 second. The second sine wave has a frequency of 4 Hz and therefore has a period of 1/4 second. So, a time range of two seconds would cover 2 cycles of the first sine wave and 8 cycles of the second sine wave. The following MATLAB commands will generate the plot in Figure 2.4:

```
t = 0:0.01:2;
y = sin(2*pi*t) + sin(2*pi*4*t);
plot(t,y);
title('sin(2*pi*t)+sin(2*pi*4*t)');
xlabel('time');grid
```

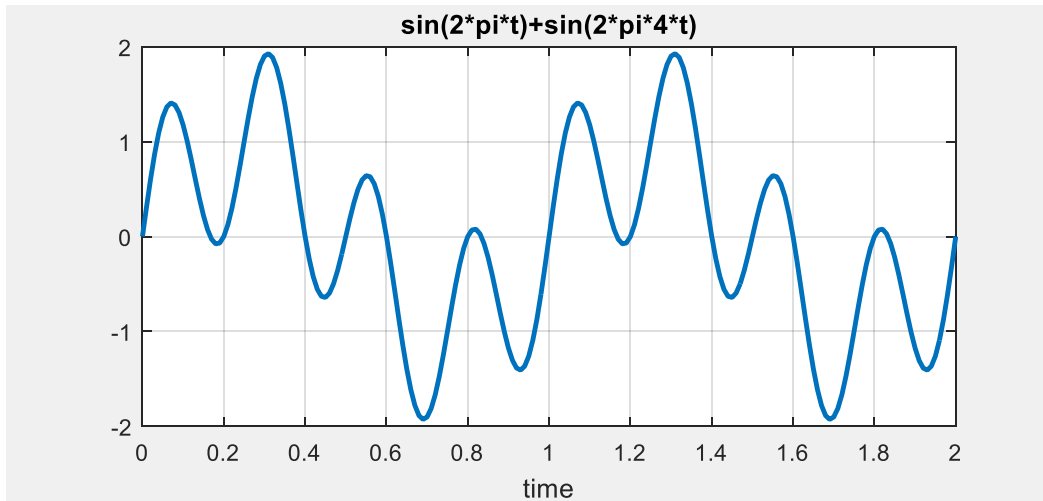


Figure 2.4: Sum of Two Sinusoids in Example 2.5a

The resulting signal no longer looks like a sine wave but it is periodic. The period is one second.

- (b) As determined in part (a), the first sine wave has a period of 1 second. The second sine wave has a frequency of 1.5 Hz and therefore has a period of $2/3$ seconds. So, a time range of 4 seconds should produce a decent graph. The following MATLAB commands produce the graph shown in Figure 2.5.

```
t = 0:0.01:4;
y = sin(2*pi*t) + sin(2*pi*1.5*t);
plot(t,y); title('sin(2*pi*t)+sin(2*pi*1.5*t)');
xlabel('time');grid
```

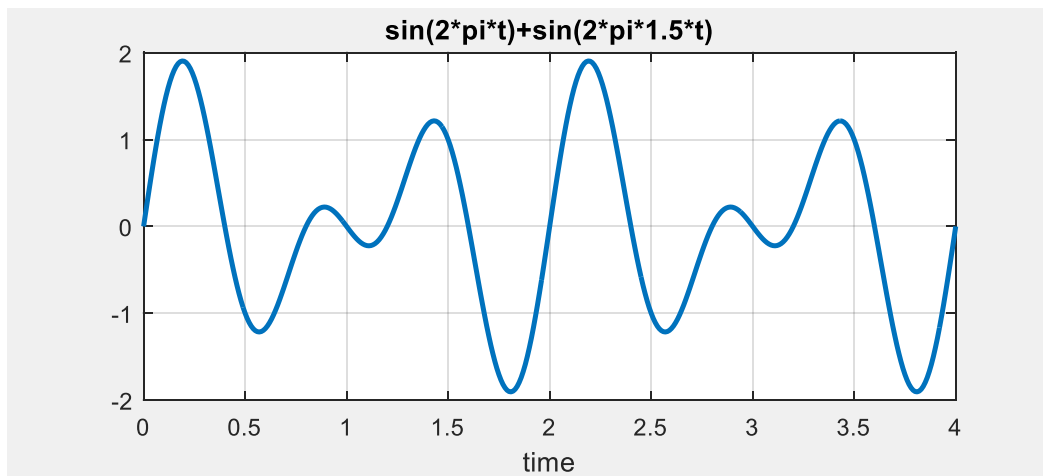


Figure 2.5: Sum of Two Sinusoids in Example 2.5b

Again, the resulting signal is clearly not a sine wave; however it is periodic with a period of two seconds.

Challenge Question 2.5

How is the period of the composite signal related to the periods of the two individual sine waves added together to produce the composite signal?

Products of Sinusoidal Signals

The product of two sine waves will also produce a periodic signal. The following formula (identity) from trigonometry proves it:

$$\sin(A)\sin(B) = 1/2[\cos(A-B) - \cos(A+B)] \quad (2.2)$$

The resultant signal does not have the original frequencies any longer – it has the sum and difference of the original frequencies. This will be discussed further in the section on signal spectra.

Example 2.6: Product of Sinusoidal Signals

Plot the signal $\sin(2\pi t) \sin(2\pi(4)t)$. The following MATLAB commands produce the graph shown in Figure 2.6.

```
t = 0:0.01:2;  
y = sin(2*pi*t).*sin(2*pi*4*t);  
plot(t,y); title('sin(2*pi*t)*sin(2*pi*4*t)');  
xlabel('time');grid
```

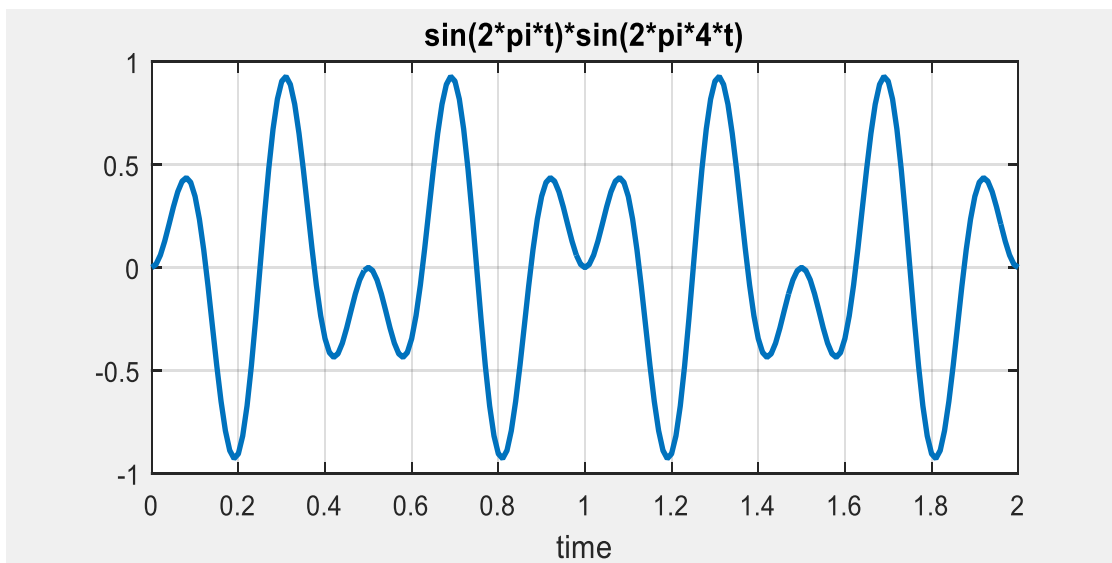


Figure 2.6: Product of Two Sinusoids in Example 2.6

Other Periodic Signals

Periodic signals are signals which repeat over a certain interval (period) of time. Sine waves and combinations of sine waves are periodic signals. Figure 2.7 shows other common periodic waveforms including square waves, saw-tooth waves, and triangular waves.

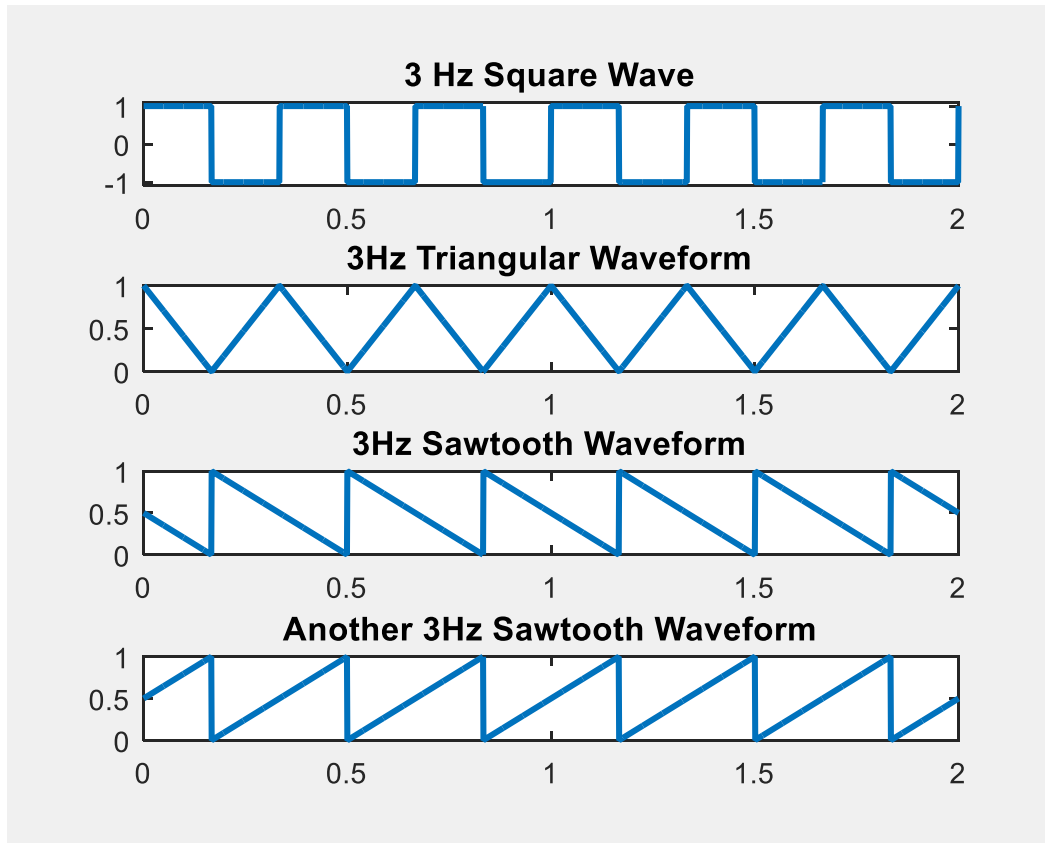


Figure 2.7: Common Periodic Waveforms

MATLAB code for Figure 2.7:

```
t=0:0.001:2;
y_square = square(2*pi*(3)*t); % 3Hz square wave
subplot(4,1,1); plot(t,y_square); title('3 Hz Square Wave');

D = 0:1/3:2; %Define where each new pulse will start in pulstran
y_triangular = pulstran(t,D,'tripuls',1/3,0); % 3Hz triangular
subplot(4,1,2); plot(t,y_triangular);
title('3Hz Triangular Waveform')
y_sawtooth = pulstran(t,D,'tripuls',1/3,-1); % 3Hz sawtooth wave
subplot(4,1,3); plot(t,y_sawtooth);
title('3Hz Sawtooth Waveform')
y_sawtooth2 = pulstran(t,D,'tripuls',1/3,1); % 3Hz sawtooth wave
subplot(4,1,4); plot(t,y_sawtooth2);
title('Another 3Hz Sawtooth Waveform')
```

Challenge Question 2.6

How would you modify the MATLAB code above to create a 10 Hz square wave that ranged from 0 to 5 instead of -1 to +1?

Non-periodic Signals

Non-periodic signals (also known as aperiodic) do not have repeating cycles. One example of a non-periodic signal is the sinc function which is defined as follows:

$$\text{sinc}(t) = \frac{\sin(\pi t)}{\pi t} \quad (\text{Note: } \text{sinc}(0) = 1) \quad (2.3)$$

The sinc function is very useful in many digital signal processing applications and will be used quite frequently in this text.

Several common non-periodic signals including the sinc function are shown in Figure 2.8 along with the accompanying MATLAB code used to create the figure.

```
t = linspace(-5,5); % Linearly spaced values between -5 and 5
r = rectpuls(t,1); % Rectangular pulse with width = 1
subplot(3,1,1); stairs(t,r); title('rect(t)')
tri = tripuls(t,2); % Triangular pulse with width = 2
subplot(3,1,2); plot(t,tri); title('tri(t)');
s = sinc(t); % Sinc function
subplot(3,1,3); plot(t,s); title('sinc(t)'); xlabel('time')
```

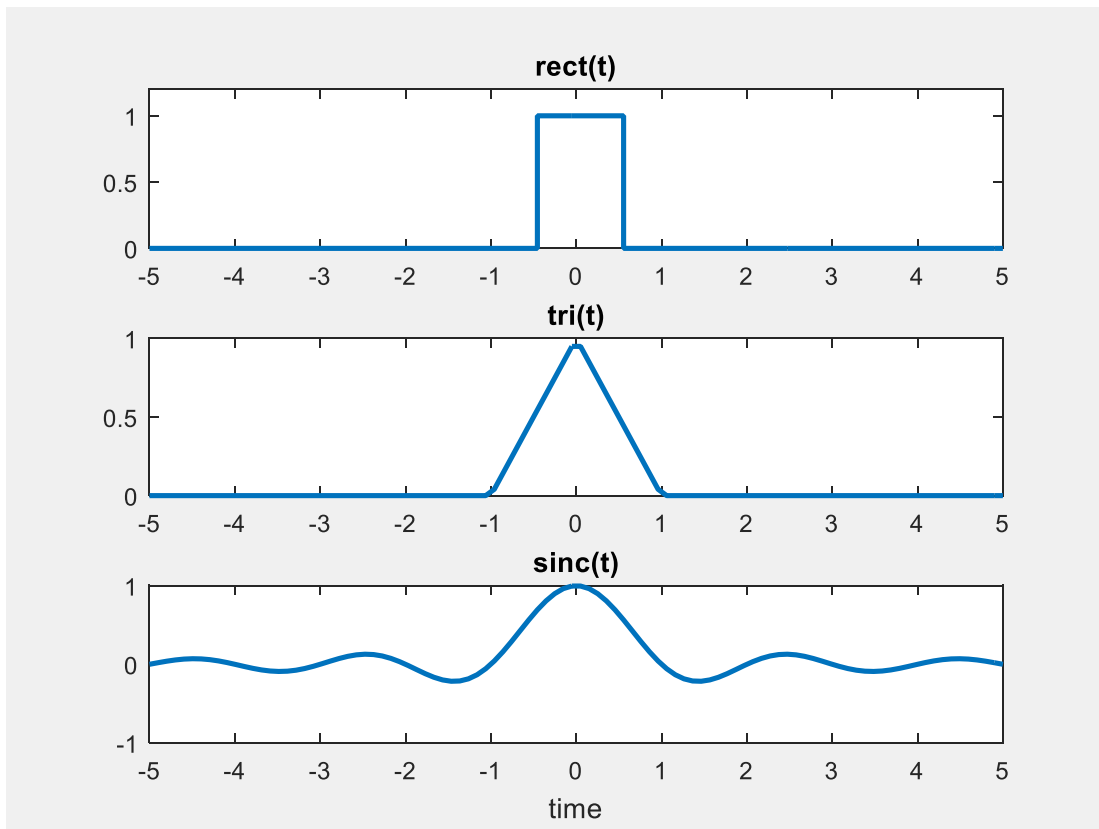


Figure 2.8: Common Non-periodic Waveforms

2.5 SPECTRUM OF PERIODIC SIGNALS

Signals can be represented in the time domain (as in the previous section) or in the frequency domain. The spectrum of a signal describes the frequency content of the signal. In this section, the spectrum of sine waves, combinations of sine waves, and other periodic signals will be introduced.

Sine Waves

A pure sine wave has one frequency component located at the frequency of the sine wave. For example, a 2 kHz sine wave has a frequency component at 2 kHz while a 10 kHz sine wave has a frequency component at 10 kHz as indicated in Figure 2.9.

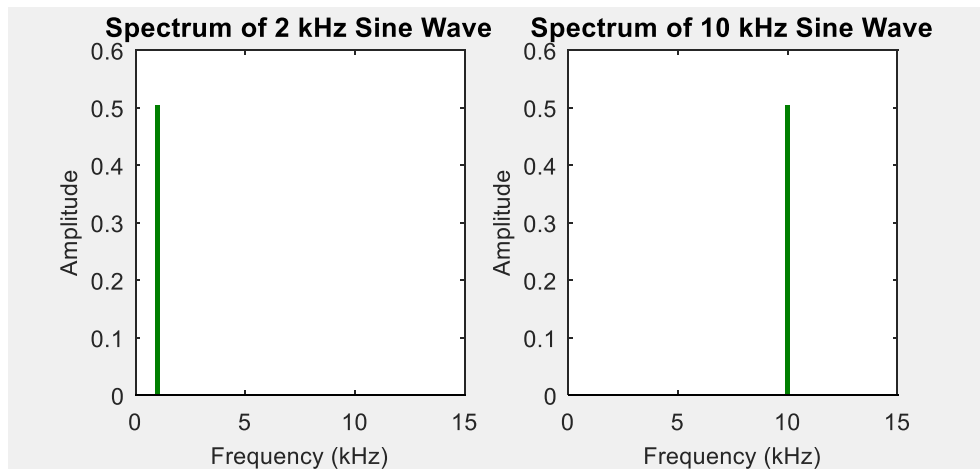


Figure 2.9: Spectra of Sine Waves

Linear Combinations of Sine Waves

When sine waves are linearly combined, the spectrum of the composite signal has frequency components at each of the individual sine wave frequencies. For example, the spectrum of the signal $10\sin(2\pi(1)t) + 5\sin(2\pi(1.5)t) + 3\sin(2\pi(4)t)$ is shown in Figure 2.10.

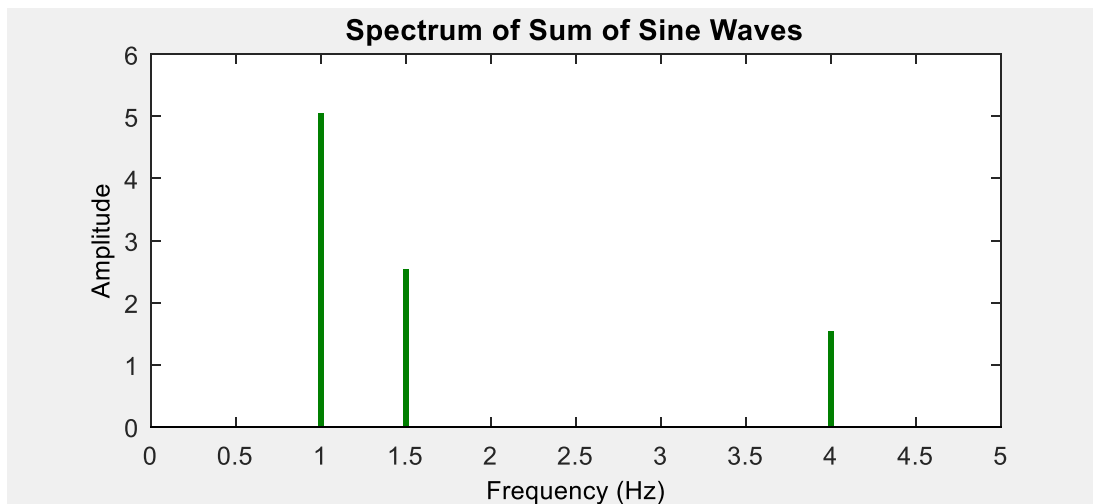


Figure 2.10: Linear Combination of Sine Waves

Products of Sine Waves

The spectrum of the product of two sine waves has frequency components at the sum and at the difference of the two individual frequencies. This can be seen by looking at the trig identity given in Section 2.4 for the product of two sine waves. For example, the spectrum of the signal $\sin(2\pi(1)t) \cdot \sin(2\pi(4)t)$ is shown in Figure 2.11.

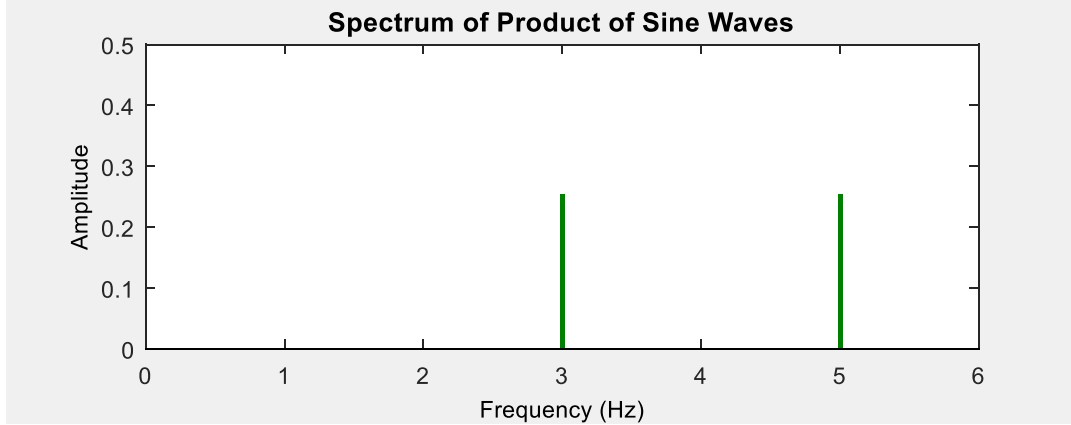


Figure 2.11: Spectrum of the Product of Two Sine Waves

Other Periodic Functions

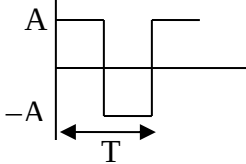
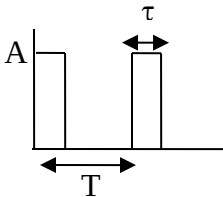
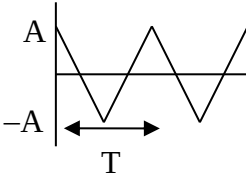
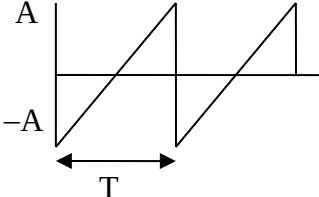
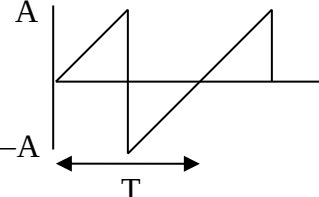
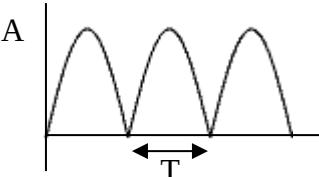
All periodic signals can be perfectly described as the sum of sine and cosine waves using the Fourier series. Suppose $f_p(t)$ is a periodic signal with period, T , and a fundamental frequency, $f_0 = 1/T$. The Fourier series for the periodic signal $f_p(t)$ is given by:

$$f_p(t) = a_0 + \sum_{n=1}^{\infty} a_n \cos(2\pi(nf_0)t) + \sum_{n=1}^{\infty} b_n \sin(2\pi(nf_0)t)$$
$$a_0 = \frac{1}{T} \int_0^T f_p(t) dt$$
$$a_n = \frac{2}{T} \int_0^T f_p(t) \cos(2\pi(nf_0)t) dt \quad b_n = \frac{2}{T} \int_0^T f_p(t) \sin(2\pi(nf_0)t) dt$$

(2.4)

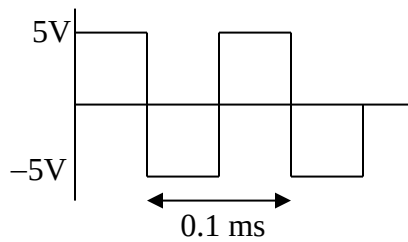
As indicated in equation 2.4, the periodic signal can be expressed exactly as a sum of cosine and sine waves with frequencies that are integer multiples (harmonics) of the fundamental frequency of the waveform. The constant term, a_0 , is simply the d.c. (average) value of the waveform. The constants a_n and b_n are the amplitudes of the sine and cosine waves at the fundamental frequency and the harmonic frequencies. These constants reflect how well the periodic signal matches, or correlates to, the corresponding cosine and sine waves. Table 2.2 shows the Fourier series expressions for several common periodic functions.

Table 2.2: Fourier Series Expressions

PERIODIC WAVEFORM	FOURIER SERIES EXPRESSION
	$\frac{4A}{\pi} \left[\sin(2\pi f_0 t) + \frac{1}{3} \sin(2\pi(3f_0)t) + \frac{1}{5} \sin(2\pi(5f_0)t) + \dots \right]$
	$Ad + \frac{A}{\pi} \{ [\sin(2\pi d) \cos(2\pi f_0 t) + 2 \sin^2(\pi d) \sin(2\pi f_0 t)] \\ + \frac{1}{2} [\sin(2\pi(2d)) \cos(2\pi(2f_0)t) + 2 \sin^2(\pi(2d)) \sin(2\pi(2f_0)t)] \\ + \frac{1}{3} [\sin(2\pi(3d)) \cos(2\pi(3f_0)t) + 2 \sin^2(\pi(3d)) \sin(2\pi(3f_0)t)] \\ + \dots \} \quad \text{where } d = \tau/T$
	$\frac{8A}{\pi^2} \left[\cos(2\pi f_0 t) + \frac{1}{3^2} \cos(2\pi(3f_0)t) + \frac{1}{5^2} \cos(2\pi(5f_0)t) + \dots \right]$
	$-\frac{2A}{\pi} \left[\sin(2\pi f_0 t) + \frac{1}{2} \sin(2\pi(2f_0)t) + \frac{1}{3} \sin(2\pi(3f_0)t) + \dots \right]$
	$\frac{2A}{\pi} \left[\sin(2\pi f_0 t) - \frac{1}{2} \sin(2\pi(2f_0)t) + \frac{1}{3} \sin(2\pi(3f_0)t) - \dots \right]$
	$\frac{2A}{\pi} - \frac{4A}{\pi} \left[\frac{1}{4 \cdot 1^2 - 1} \cos(2\pi(f_0)t) + \frac{1}{4 \cdot 2^2 - 1} \cos(2\pi(2f_0)t) \right. \\ \left. + \frac{1}{4 \cdot 3^2 - 1} \cos(2\pi(3f_0)t) + \dots \right]$

Example 2.7: Fourier Series

Consider the square wave shown below.



(a) Graph the first term of the Fourier Series.

Comparing the graph to the square wave in Table 2.2, it follows that $A = 5$ and $T = 0.1\text{ms}$. So the frequency $f_0 = 1/T = 10\text{ kHz}$. The first term of the Fourier Series is: $(20/\pi) \sin(2\pi(10,000)t)$. This can easily be graphed in MATLAB. The result is shown in Figure 2.12. Obviously this is a sine wave and is not a good description of the square wave.

```
t = 0:0.1e-3/100:3*0.1e-3; %100 samples/cycle with 3 cycles
y1 = (20/pi)*sin(2*pi*1e4*t);
subplot(3,1,1);plot(t,y1);
title('One Term of Fourier Series');grid
```

(b) Graph the first three terms of the Fourier Series.

The amplitude, A , and the frequency, f_0 , are exactly the same as calculated in part (a). The first three terms of the Fourier series would be:

$$(20/\pi) [\sin(2\pi(10,000)t) + (1/3)\sin(2\pi*3*(10,000)t) + (1/5)\sin(2\pi*5*(10,000)t)]$$

This function can be graphed in MATLAB by adding the following commands. The result is shown in Figure 2.12. This graph looks a bit more like the square wave we are trying to represent.

```
y2 = (20/pi)*(1/3*sin(2*pi*3*1e4*t));
y3 = (20/pi)*(1/5*sin(2*pi*5*1e4*t));
subplot(3,1,2);plot(t,y1+y2+y3);
title('First Three Terms of Fourier Series');grid
```

(c) Graph the first seven terms of the Fourier Series.

The first seven terms of the Fourier Series are:

$$(20/\pi) [\sin(2\pi(10,000)t) + (1/3)\sin(2\pi*3*(10,000)t) + (1/5)\sin(2\pi*5*(10,000)t) + \\ (1/7)\sin(2\pi*7*(10,000)t) + (1/9)\sin(2\pi*9*(10,000)t) + (1/11)\sin(2\pi*11*(10,000)t) + \\ (1/13)\sin(2\pi*13*(10,000)t)]$$

This function can be graphed in MATLAB by adding the following set of commands. The result is shown in Figure 2.12. Including more terms from the Fourier series gives a better match to the original square wave.

```
y4 = (20/pi)*(1/7*sin(2*pi*7*1e4*t));
y5 = (20/pi)*(1/9*sin(2*pi*9*1e4*t));
y6 = (20/pi)*(1/11*sin(2*pi*11*1e4*t));
y7 = (20/pi)*(1/13*sin(2*pi*13*1e4*t));
subplot(3,1,3);plot(t,y1+y2+y3+y4+y5+y6+y7);
title('First Seven Terms of Fourier Series');grid
```

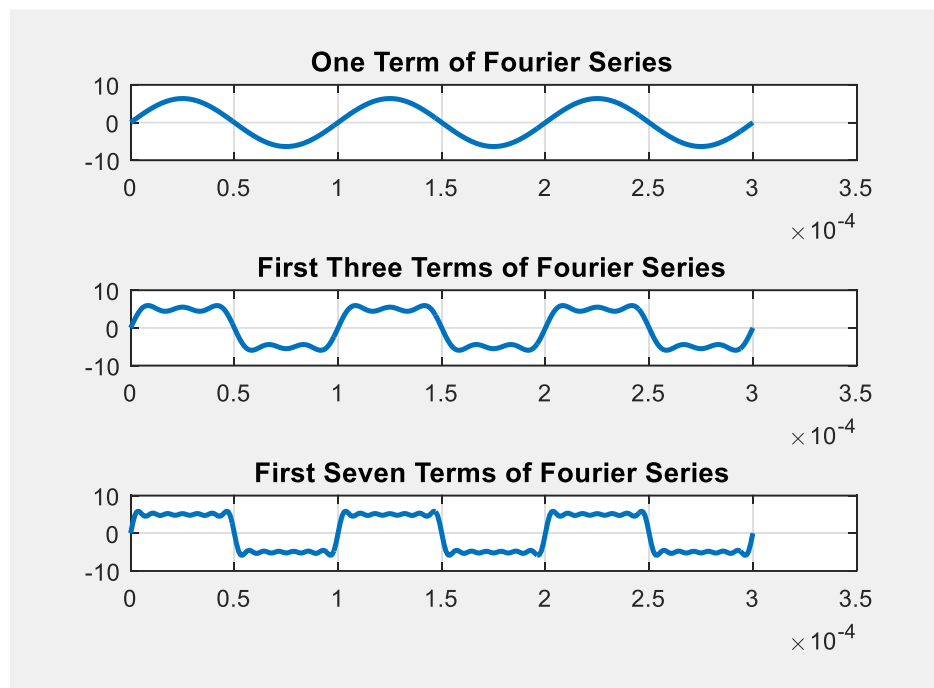


Figure 2.12: Fourier Series for a Square Wave

Challenge Question 2.7

What does the spectrum of the square wave in Example 2.7 look like?

2.6 SPECTRUM OF NON-PERIODIC SIGNALS

The spectrum of periodic signals can be determined using the Fourier series. As discussed in the previous section, periodic signals have frequency components at the fundamental frequency and at harmonics (integer multiples) of the fundamental frequency. Fourier series does not work for non-periodic signals. Non-periodic signals are made up of a continuum of sinusoidal frequencies with varying amplitude and phase. The Fourier Transform can be used to analyze the spectrum of non-periodic analog signals. The Fourier Transform is defined in equation 2.5:

$$F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-j\omega t} dt = \int_{-\infty}^{\infty} f(t)[\cos(\omega t) - j\sin(\omega t)]dt \quad (2.5)$$

Example 2.8: Fourier Transform

Consider the non-periodic, decaying ($a > 0$) exponential time signal:

$$f(t) = e^{-at} \text{ for } t \geq 0 \text{ and } f(t) = 0 \text{ for } t < 0.$$

Find the Fourier Transform of $f(t)$.

$$F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-j\omega t} dt = \int_0^{\infty} e^{-at} \cdot e^{-j\omega t} dt = \int_0^{\infty} e^{-(a+j\omega)t} dt = \frac{-1}{a+j\omega} e^{-(a+j\omega)t} \bigg|_{t=0}^{t=\infty} = \frac{1}{a+j\omega}$$

$$F(f) = \frac{1}{a + j2\pi f}$$

So what does this mean?

Go back for a moment to the 10 kHz square wave in Example 2.7. The 10 kHz square wave has frequency components at a discrete set of frequencies: 10 kHz, 30 kHz, 50 kHz, 70 kHz, etc. (i.e., the fundamental frequency and odd harmonics of the fundamental frequency). The non-periodic, exponential signal has frequency components over a whole continuum of frequencies. If we assume $a = 5$, then we have:

$$\begin{aligned} F(0.1 \text{ Hz}) &= 1/(5 + j(2\pi \cdot 0.1)) = 0.1984 \angle -7.2^\circ \\ F(1 \text{ Hz}) &= 1/(5 + j(2\pi \cdot 1)) = 0.1245 \angle -51.5^\circ \\ F(10 \text{ Hz}) &= 1/(5 + j(2\pi \cdot 10)) = 0.0159 \angle -81.5^\circ \\ F(100 \text{ Hz}) &= 1/(5 + j(2\pi \cdot 100)) = 0.0016 \angle -89.5^\circ \end{aligned}$$

Notice that the magnitude is decreasing with increasing frequency.

Example 2.9: Plotting Signal and Spectrum

Plot the time signal in Example 2.8 (assuming $a = 5$) and the magnitude of the frequency response. This can be done easily using the following MATLAB statements:

```
t = 0:0.001:1;
f = exp(-5*t);
subplot(2,1,1); plot(t,f);
title('Time signal exp(-5*t)');grid; xlabel('time(sec)')

freq=0:0.01:50;
F = 1./(5 + j*2*pi*freq); % Use ./for division by vector
Mag=abs(F);
subplot(2,1,2); plot(freq,Mag);
title('Magnitude of Frequency Response');grid;
xlabel('frequency (Hz)')
```

The time signal and signal spectrum are shown in Figure 2.13.

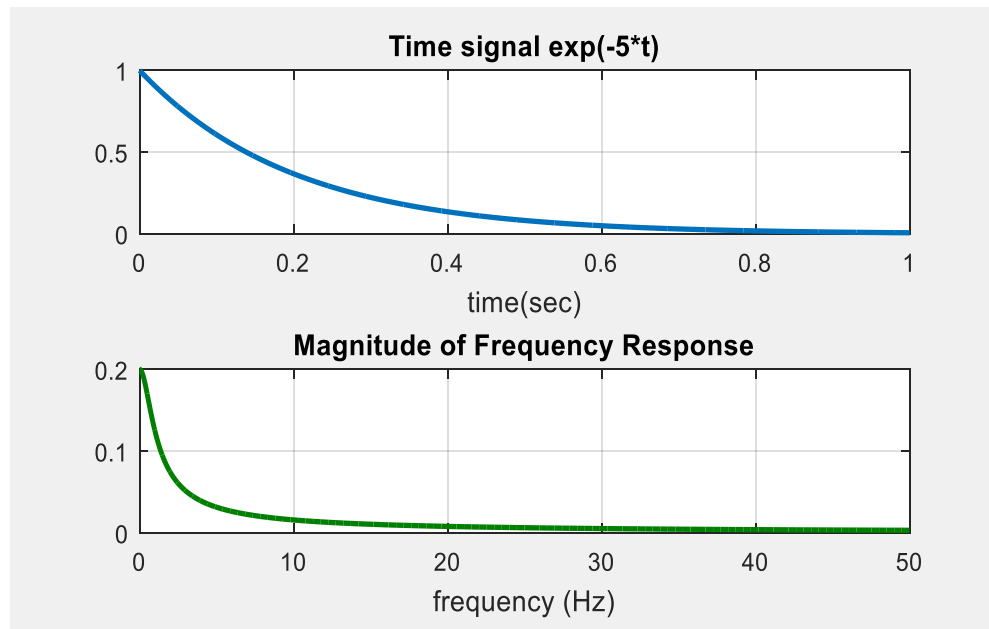


Figure 2.13: Time Signal and Signal Spectrum for Example 2.9

The main point to understand in this section is not how to compute the Fourier Transform of a signal by working through the integral definition. The key point is that a signal spectrum is simply the frequency content of a signal and that non-periodic analog signals have a continuum of frequency content. The Fourier Transform is simply a method for determining the spectrum (frequency content) of an analog signal. It is not at all difficult to come up with example signals for which it is impossible to perform the integration by hand as was done in Example 2.8. Fourier transforms of signals are typically found using a Fourier Transform table rather than the integral definition. A table of Fourier transforms is included in the Appendix.

In DSP applications, a fast Fourier Transform (FFT) is used to determine the spectrum of sampled analog or digital signals. Fast Fourier Transforms do not require integration but instead use easily programmed multiply and accumulate (MAC) routines. FFTs will be introduced in a Chapter 7.

2.7 SIGNAL BANDWIDTH

The bandwidth of a signal is the amount of space that a signal occupies in the frequency domain. Figure 2.14 shows two different signal spectrums for which bandwidth will be determined. Signal A is a baseband or low-pass signal. It has frequency components from 0 kHz to 10 kHz and therefore occupies 10 kHz worth of space in the frequency domain. So, the bandwidth of Signal A is 10 kHz. Signal B is a bandpass signal and has frequency components from 690 kHz to 710 kHz. It occupies 20 kHz worth of space in the frequency domain and therefore has a bandwidth of 20 kHz.

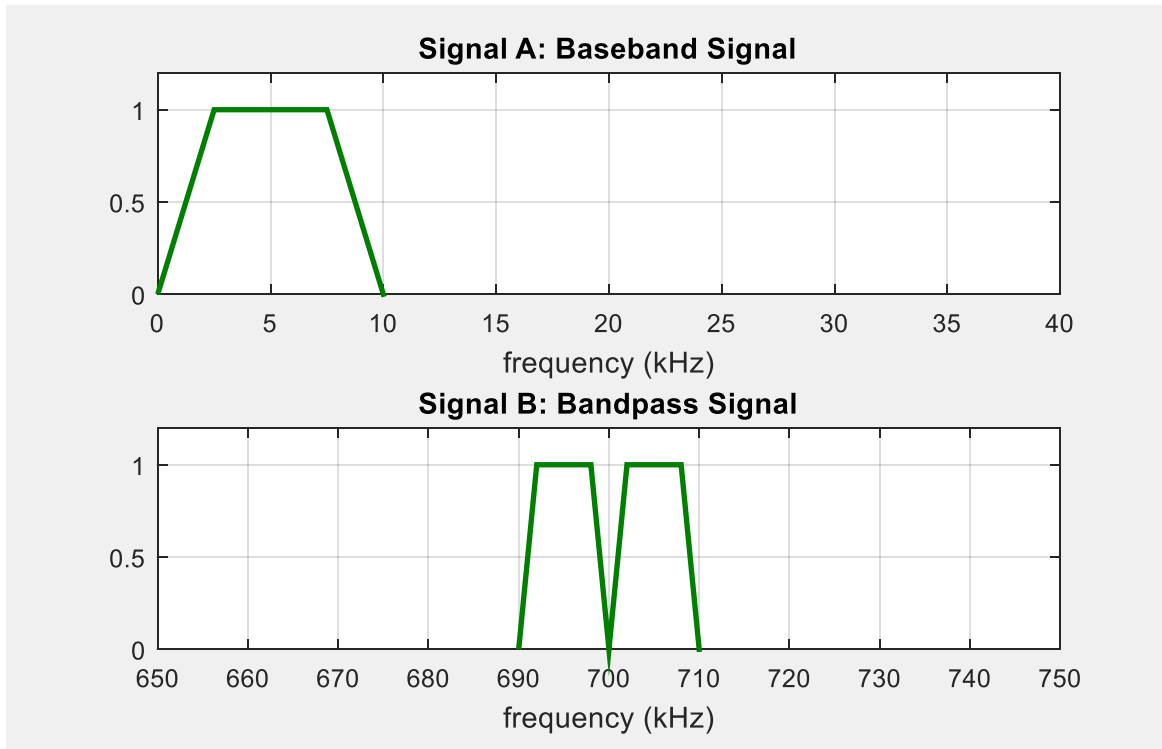


Figure 2.14: Illustration of Signal Bandwidth

Take a look at the signal spectrum in Figure 2.13. What would the bandwidth of this signal be? The problem with this signal is that although the magnitude is certainly getting small as frequency increases, it appears to stretch on forever indicating that the signal is not bandlimited (infinite bandwidth). In cases such as these, a more practical definition of bandwidth is needed.

Practical bandwidth is a measure of the range of frequency that adequately captures the original signal. That is, if we cut out the rest of the spectrum, the signal would not be significantly altered (still works OK for whatever the application is). There are several different ways to define a practical bandwidth – two of the most common will be discussed here.

Definition: 3dB Bandwidth

A measure of the range of frequency in which the magnitude response stays within 3dB (70.7%) of the peak magnitude.

Definition: Signal Power Bandwidth

A measure of the range of frequency that captures some percentage (typically 95% - 98%) of the total signal power.

The 3dB bandwidth definition works very well for filters (discussed later) and for signals with a spectrum that attenuates (drops off dramatically) around the 3dB frequency. The signal power bandwidth is much better for signals with spectrums that gradually attenuate.

Example 2.10: 3dB Bandwidth

Using Figure 2.13, approximate the 3dB bandwidth of the signal.

Solution

The peak magnitude of the frequency response for the signal in Figure 2.13 is 0.2. If we multiply this peak magnitude by 70.7%, we get 0.1414. From the figure, the frequency at which the magnitude equals 0.1414 is approximately 1 Hz. So, the 3dB bandwidth is approximately 1 Hz.

Challenge Question 2.8

What is the 3dB bandwidth of the 10 kHz square wave from Example 2.7? Does the 3dB bandwidth seem like a good practical bandwidth for this square wave? Why or why not? (Hint: take a look at the spectrum of this square wave in Challenge Question 2.7).

2.8 SPECTROGRAM

Audio, music and video signals change over time and therefore the spectrum (frequency content) also changes with time. A spectrogram is a plot of the frequency components and the relative strength of the frequency components (magnitude) over time. In order to display the three dimensions (frequency, time, and magnitude) on a 2-D plot, frequency is on the x-axis, time is on the y-axis and relative strength of the frequency components is shown using color (or grayscale). If the colormap is set to jet, then the color Red indicates a very strong frequency component and the color Blue indicates a very weak frequency component.

Example 2.11: Spectrogram of Signals

Plot the spectrogram for the signal $y = 10\sin(2\pi \cdot 1000 \cdot t) + 0.1\sin(2\pi \cdot 3000 \cdot t)$ over a 0.5 second time interval. Assume a sampling rate of 20 kHz.

Solution

The following MATLAB commands will plot the spectrogram of the signal, y:

```
Fs = 20000;  
t = 0:1/Fs:0.5;  
y = 10*sin(2*pi*1000*t) + 0.1*sin(2*pi*3000*t);  
spectrogram(y,256,240,256,Fs);  
colormap('jet');  
title('Spectrogram for Sum of 2 Sinwaves');
```

The resulting spectrogram is shown in Figure 2.15. The spectrogram for the signal y is not particularly interesting because signal y has a frequency content that does not vary with time. The frequency components of y are at 1 kHz and 3 kHz over the entire time interval. The amplitude of the 1 kHz signal is 100 times higher (40 dB) than the amplitude of the 3 kHz signal. So, we see a vertical red line at 1 kHz and a vertical yellow/green line (-40 dB) at 3 kHz over the entire time interval.

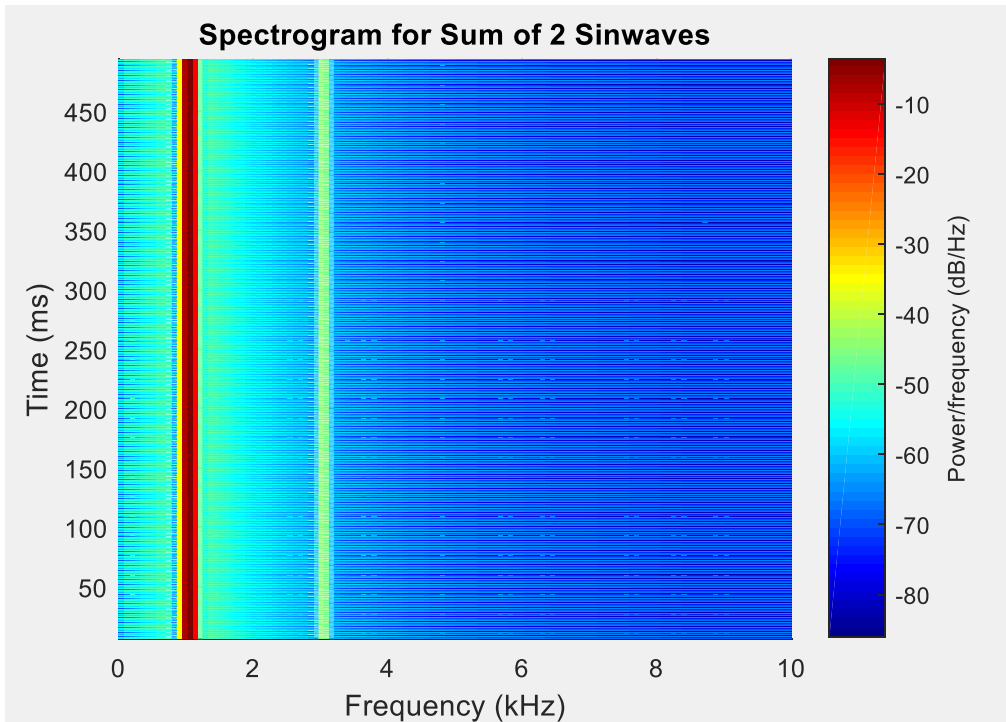


Figure 2.15: Spectrogram of the Sum of Two Sine Waves

A far more interesting spectrogram is shown in Figure 2.16. The left side of the figure plots a three second recording of an orca whale. The right side of the figure shows the spectrogram for the sound clip. Looking at the spectrogram, we can see how the frequency components change over time and vary in intensity. For example, between 0.75 and 0.9 seconds, the amplitude of the signal is very low and the spectrogram reflects this with only low-power frequency content. At about 1.6 seconds, there is a very short, rather high-pitched burst of sound which is reflected in the spectrogram. When time hits two seconds, it appears that the frequency content remains relatively constant over time, similar to Figure 2.15 but with many more frequencies in the mix. The MATLAB code used to generate this figure are:

```
[y, Fs] = audioread('orca.wav');
subplot(1,2,2);
spectrogram(y,512,256,512,Fs);
colormap('jet');
title('Orca Whale Spectrogram');

t = 0:1/Fs:1/Fs*(length(y)-1);
subplot(1,2,1); plot(y,t);
title('Orca Whale Sound');
xlabel('Sound Level'); ylabel('Time (secs)')
```

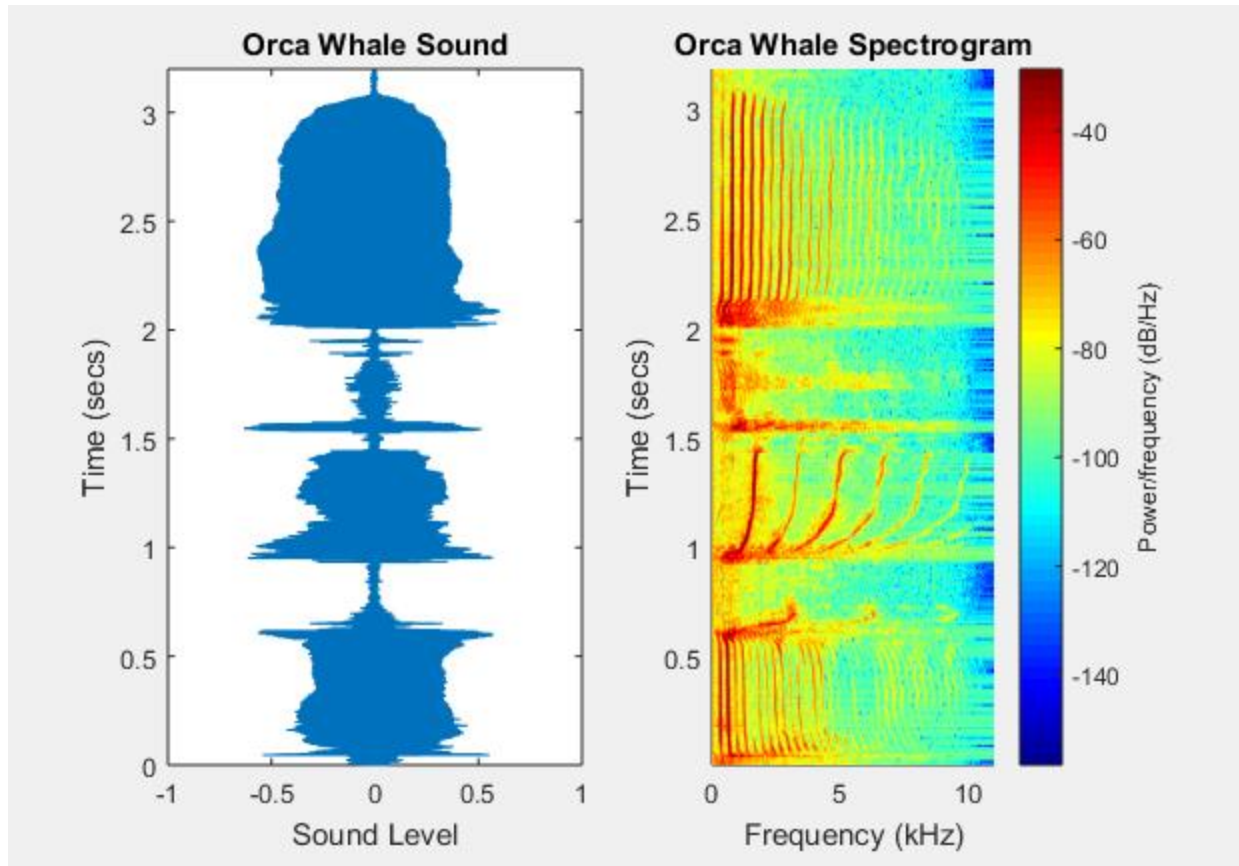


Figure 2.16: Spectrogram of an Orca wav file

You could use similar code to experiment with your own sound clips. Just be sure to pick short sound clips or the code will take quite a bit of time to run. To find out more about importing audio files, type `help audioread` or `doc audioread` at the command prompt in MATLAB.

There are multiple applications for spectrograms including speech recognition, music, radar/sonar, analysis of animal sounds, and many others. An interesting application in electronic music is to use the process in reverse to hide images in music. Starting with an image, an inverse Fast Fourier Transform will produce a sound clip corresponding to the image.

2.9 APPLICATION: SYNTHESIS OF MUSICAL INSTRUMENTS

Oscillators

Musical instruments can be synthesized using combinations of unit generators (ugens). A very simple unit generator is an oscillator that produces a periodic waveform at some specified frequency. An amplitude envelope can also be specified to control how the sound is phased in and phased out to avoid the undesirable clicking noises that occur with sudden step changes in amplitude. Combining oscillators with a variety of frequencies and amplitude envelopes creates different types of musical sound.

Example 2.12: Listening to Simple Periodic Waveforms

Open the simple_oscillator.mdl shown in Figure 2.17 from the text website or simply build the model yourself in Simulink®.

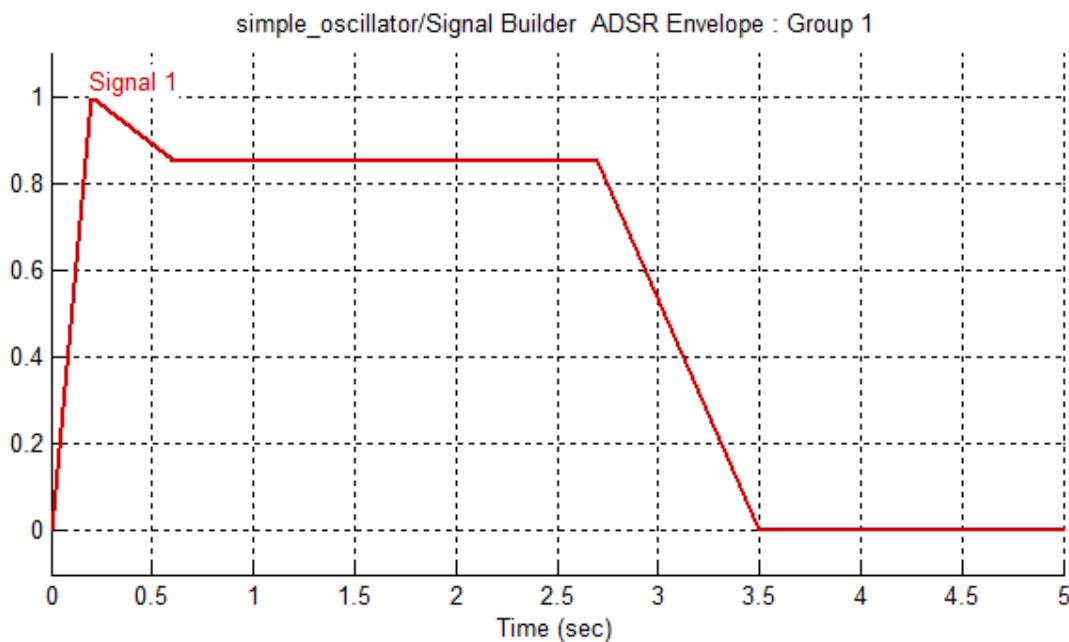
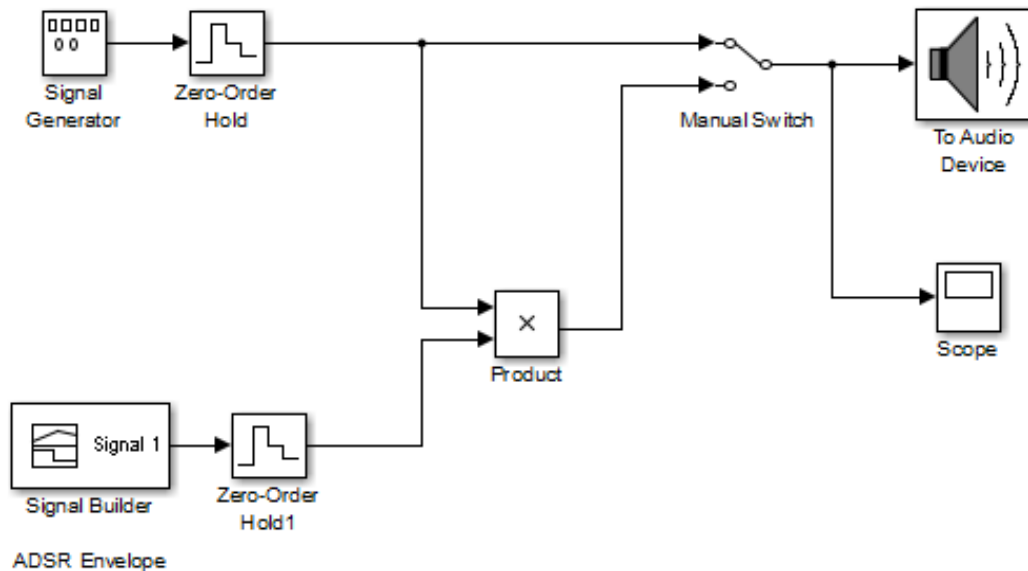


Figure 2.17: Simple_Oscillator

The signal generator should be set for a 440 Hz sine wave with amplitude equal to one. Both of the zero order holds (ZOH) should be set for a sample time of $1/44100$. The Audio Device (speaker) block should inherit the input sample rate and input type, and the queue duration should be set to 1 second. The Signal Builder block should be set up to produce the ADSR

(Attack, Decay, Sustain, and Release) amplitude envelope shown in Figure 2.17. The Stop Time should be set to 5 seconds. When running this model (and all of the audio models in this chapter), use headphones or decent external PC speakers. Most built-in PC speakers will not produce quality results over the frequency range required.

- (a) With the settings described above, open the scope block and run the simulation. You should hear a simple 440 Hz tone (corresponds to the A key above middle C on a piano). Notice also the audible click when the tone begins and again when the tone ends. Switch the manual switch so the amplitude of the sine wave is modified by the ADSR envelope function. Now, because the sound is gradually phased in and phased out by the ADSR envelope, the clicking sounds caused by sudden step changes in amplitude disappear.
- (b) Try different frequencies for the sine wave. What frequencies can you hear? How does the perceived loudness of the tone vary with frequency?
- (c) If the original frequency of 440 Hz is modified by a factor of two, the note will still sound like an “A” but it will be shifted by an octave for each factor of 2. Changing the frequency to 880 Hz raises the original tone by an octave, while a change to 110 Hz lowers the original tone by two octaves. Try this.
- (d) The tones generated don’t really sound like any musical instrument. Switch the signal generator block to a sawtooth wave and listen to the result. What does this waveform sound like? How about a square wave?

Challenge Question 2.9

A 440 Hz sine wave sounds like a pure tone. What makes the 440 Hz sawtooth and square waves sound different than the 440 Hz sine wave?

Example 2.12 introduces some simple yet important facts about sound synthesis and human hearing:

- Pitch is how humans perceive frequency. The audible frequency range is approximately 20 – 20,000 Hz, but human hearing is most sensitive in the range of 200 – 2000 Hz. Human response to changes in frequency is logarithmic (ratios) rather than linear (additive). In other words, the step change between the set of frequencies, 400, 440, 484, 532.4, and 585.64 would be perceived to be exactly the same because the ratio between adjacent frequencies is a constant 1:1.1.
- The perceived loudness of a tone depends not only on the obvious variable, amplitude, but also depends on frequency. Very low and very high frequencies sound softer than mid-range frequencies of the same amplitude. The human threshold for hearing a tone is frequency dependent; the lower and higher frequencies in the audible range have higher thresholds.
- Sudden step changes in amplitude cause clicks. Unless clicking is a desired effect in a music composition, amplitude envelopes should be used to phase sound in and out gradually.
- As addressed in Challenge Question 2.9, the sawtooth and square waves add harmonics to the fundamental frequency. These harmonics make the tone sound richer and much more realistic. The sawtooth waveform sounds very much like a clarinet.

Table 2.3 shows the frequencies associated with the keys on a piano based on the twelve-tone equal-tempered system. The equal-tempered system is based on the assumption that all twelve tones in an octave are equally spaced logarithmically and the spacing between octaves is a factor of 2.

Table 2.3: Frequencies of Piano Keys: Equal Tempered System

OCTAVE	0	1	2	3	4	5	6	7	8
C		32.703	65.406	130.813	261.626	523.251	1046.502	2093.005	4186.009
C#		34.648	69.296	138.591	277.183	554.365	1108.731	2217.461	
D		36.708	73.416	146.832	293.665	587.330	1174.659	2349.318	
D#		38.891	77.782	155.563	311.127	622.254	1244.508	2489.016	
E		41.203	82.407	164.814	329.628	659.255	1318.510	2637.020	
F		43.654	87.307	174.614	349.228	698.456	1396.913	2793.826	
F#		46.249	92.499	184.997	369.994	739.989	1479.978	2959.955	
G		48.999	97.999	195.998	391.995	783.991	1567.982	3135.963	
G#		51.913	103.826	207.652	415.305	830.609	1661.219	3322.438	
A	27.500	55.000	110.000	220.000	440.000	880.000	1760.000	3520.000	
A#	29.135	58.270	116.541	233.082	466.164	932.328	1864.655	3729.310	
B	30.868	61.735	123.471	246.942	493.883	987.767	1975.533	3951.066	

The frequency of any key can be calculated using the key number shown in Figure 2.18 and the formula:

$$f = 440 * 2^{(\text{keynum} - 49)/12} \quad (2.6)$$

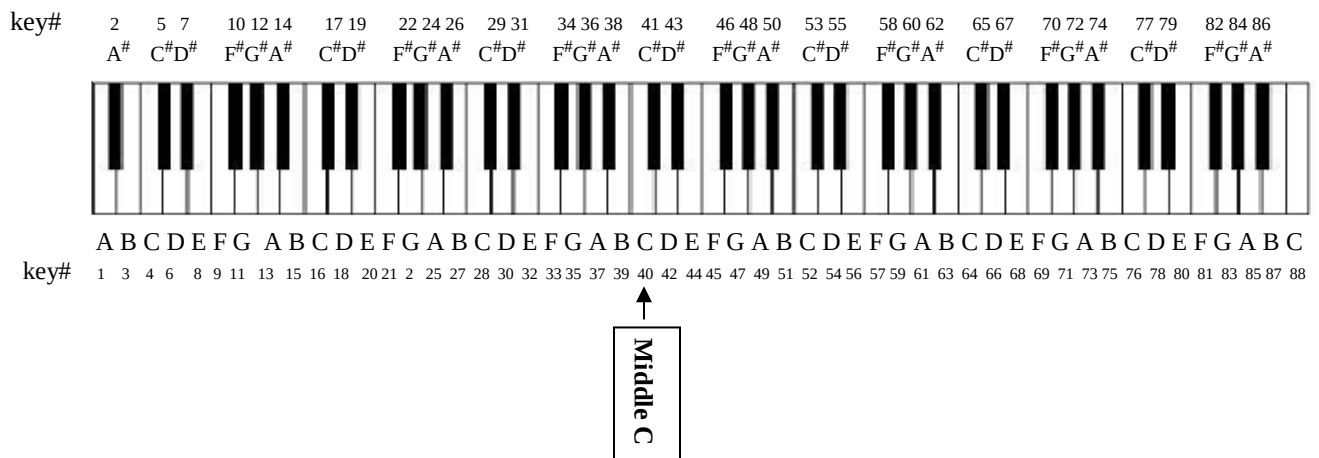


Figure 2.18: Piano Keyboard

Example 2.13: Mixing Two Tones

This example is designed to investigate the effects of mixing two tones together. Open the `two_tone.mdl` shown in Figure 2.19 from the text website or build it yourself in Simulink. Initially, the top Signal Generator should produce a 440 Hz sinusoid with amplitude 0.5, and the bottom Signal Generator should produce a 261 Hz sinusoid with amplitude 0.5. The Signal Builder should generate the same ADSR envelope shown in Figure 2.17. The Audio Device (speaker) inherits both sampling rate and data type from the input, and has a queue duration equal to one second. The simulation should be set to run for 5 seconds.

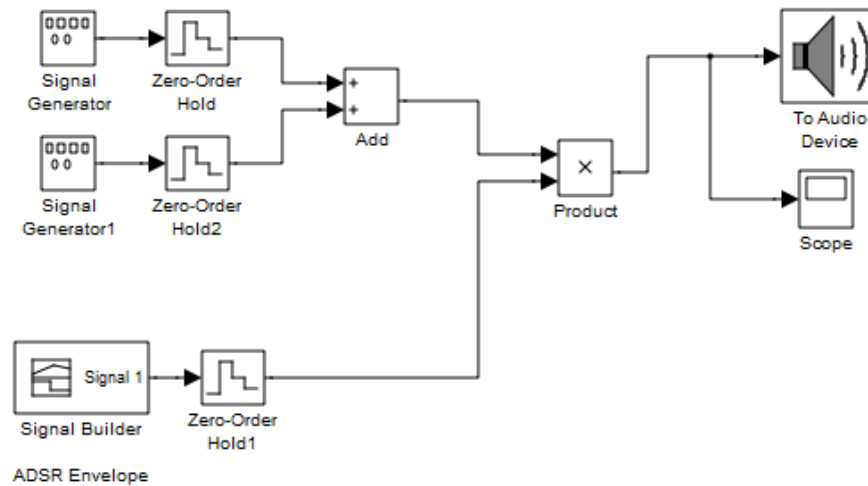


Figure 2.19: Two Tone Audio Signal

- Run the simulation. Two distinct tones should be audible – one corresponding to middle C on a piano (261 Hz) and the other corresponding to the “A” key above middle C (440 Hz).
- Now, change the frequency of the tone on the lower Signal Generator to 436 Hz and listen to the result. This time, only a single tone at 438 Hz should be audible (the average of the two frequencies) but it will beat at 4 Hz (the difference between the two frequencies). The scope plot of the audio signal should look like Figure 2.20 with a definite 4 Hz beat.
- Experiment with the spacing between the two tone frequencies. The beating effect should be audible when the two frequencies are within 10-15 Hz of one another.

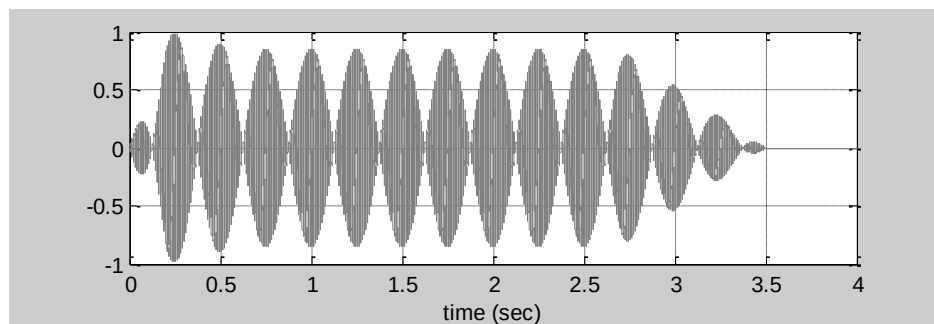


Figure 2.20: Beating Effect of Two Tones Spaced Apart by 4 Hz

Additive Synthesis

Additive Synthesis is a method for creating musical instruments by adding several oscillators with a variety of frequencies and a variety of amplitude envelopes together. Figure 2.21 shows a Simulink model which uses additive synthesis to create a bell instrument.

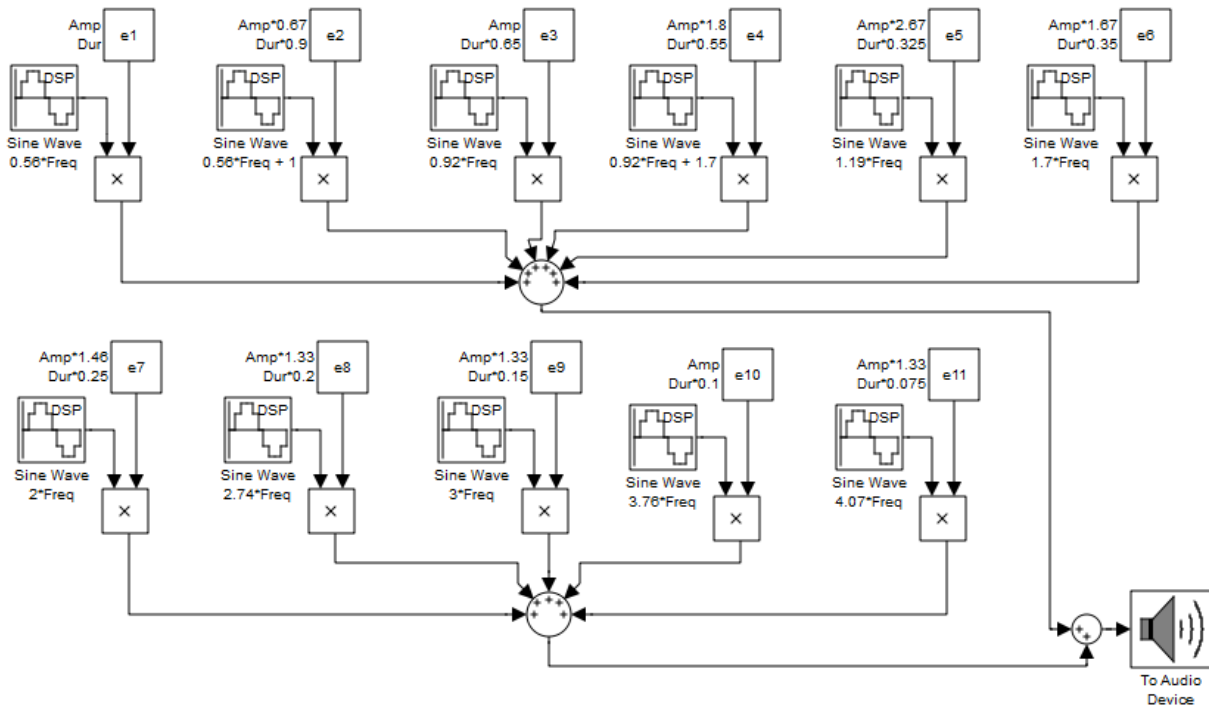


Figure 2.21: Simulink Model for Risset's Bell Instrument

The bell instrument is based on the model developed by Jean-Claude Risset in his “*Introductory Catalogue of Computer Synthesized Sounds.*” Risset basically began with the ratios 1:1.2:1.5:0.5:2 (fundamental, minor 3rd, perfect 5th, octave below, octave above), skewed them somewhat for an inharmonic effect, and then added higher frequencies. In addition, frequencies exactly 1 Hz above the two lowest frequencies were added to cause the beating effect examined in Example 2.13. The amplitude envelope for each of the oscillators is a decaying exponential which produces the strong attack followed by a rapid decay associated with a bell. A sample exponential envelope is shown in Figure 2.22 for an amplitude equal to 0.05 and a duration of 8 seconds. The relationship between the amplitudes and durations for each of the oscillators along with the frequencies are shown in Figure 2.21.

Example 2.14: Additive Synthesis – Risset's Bell Instrument

The Simulink model used to simulate Risset's bell instrument is shown in Figure 2.21. To run the model, first load the bell_env.m file from the text website into the current directory. This function will create the amplitude envelopes for each of the oscillators. Figure 2.22 shows the envelope, e1. In the MATLAB command window, run the following commands:

```
[e1,e2,e3,e4,e5,e6,e7,e8,e9,e10,e11] = bell_env(0.05,8);
Freq = 1000;
```

Open and run the Simulink model, bell.mdl. Vary the frequency (Freq) and listen to the effect.

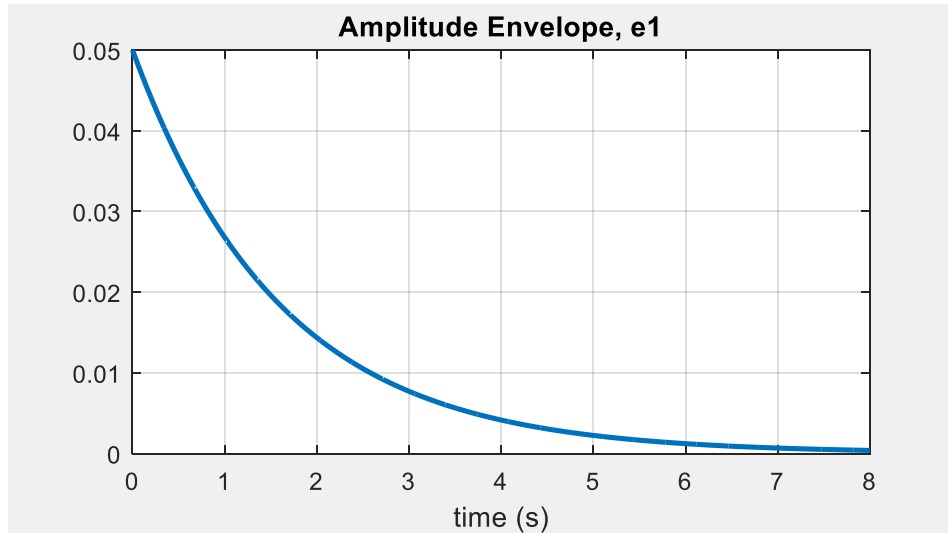


Figure 2.22: Amplitude Envelope for Bell Oscillators

FM Synthesis

Frequency Modulation (FM) synthesis is an example of distortion or non-linear synthesis for creating realistic musical instruments. Most electrical engineering and technology students are familiar with FM radio, where an audio signal is transmitted by using the audio signal to modulate (or vary) the frequency of a carrier signal. For FM radio, the carrier frequency is not within the audible frequency range but is instead much higher (88-108 MHz for FM radio). When the carrier frequency is moved down into the audible range, it is possible to synthesize musical instruments and create other neat sound effects. John Chowning introduced FM synthesis for music composition in the early 1970s. The basic equation for an FM signal modulated by a sinusoid is given by:

$$f_{FM}(t) = a(t) \sin(2\pi f_c t + i(t) \sin(2\pi f_m t)) \quad (2.7)$$

$a(t)$ = amplitude envelope for FM signal
 f_c = carrier frequency of FM signal
 $i(t)$ = amplitude envelope for modulating signal
 f_m = modulating frequency

Assuming for the moment that $i(t)$ is some constant, the FM signal defined in equation 2.7 has an instantaneous frequency of

$$f_c + I \cdot f_m \cdot \cos(2\pi f_m t) \quad (2.8)$$

Equation 2.8 indicates that the instantaneous frequency of the FM signal varies around the carrier frequency, f_c . The peak frequency deviation is the product of the modulation index, I , and the modulating frequency, while the rate of variation depends on the modulating frequency, f_m . The spectrum for an FM signal, assuming a sinusoidal modulating waveform with a constant modulation index, I , is shown in Figure 2.23

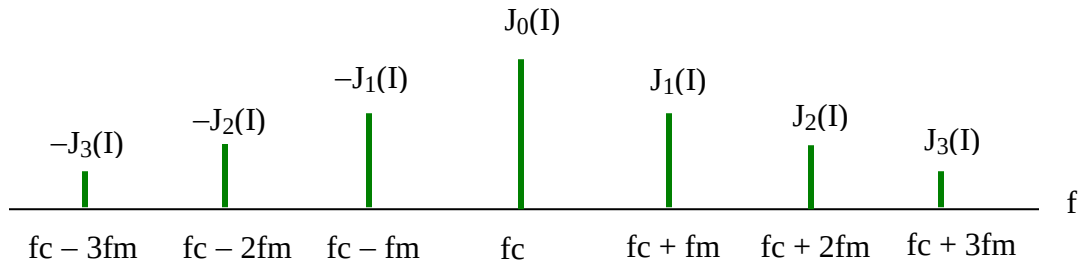


Figure 2.23: Spectrum of FM Signal with Sine Wave Modulating Signal

Notice that although the FM signal is generated with just two oscillators, the spectrum of the FM signal is very rich with spectral components at f_c , $f_c \pm f_m$, $f_c \pm 2f_m$, $f_c \pm 3f_m$ and so on. The amplitudes of the various spectral components are determined by Bessel functions. $J_K(I)$ is a Bessel function of the first kind of order K with argument I equal to the modulation index. Table 2.4 lists the values of Bessel Functions up to order 8 for various modulation indices. It is important to note that increasing the modulation index will create a richer signal with more spectral components and a larger bandwidth ($BW \approx 2(I + 1)f_m$). Values for Bessel functions can be easily calculated in MATLAB using the `besselj` function. Example 2.15 illustrates some basic properties and sounds associated with FM signals.

Example 2.15: Simple FM Synthesis Model

Open the Simple_FM.mdl shown in Figure 2.24 from the text website. The signal generator for the modulating signal should be set up for a 440 Hz sine wave with amplitude zero; that is, there is no modulation signal initially. The ADSR envelope is similar to the one in Figure 2.17 but extends for a longer period of time.

- Run the model. With no modulation, a pure 440 Hz tone should be audible and this single frequency component will also be displayed by the FFT scope.
- Open the Signal Generator for the Modulating Signal Block. Leave the frequency of the modulating signal at 440 Hz, but vary the amplitude (try values of 1, 3, 5, and 10). As the amplitude is increased, the bandwidth of the signal should increase (more spectral components). Since the modulating frequency, f_m , is exactly the same as the carrier frequency, all the spectral components are harmonic (440, 880, 1320, ...). Increasing the amplitude simply adds more harmonics and changes the characteristic of the sound.
- Now change the modulating frequency to 480 Hz and experiment with amplitude. The sound is now inharmonic since the modulating frequency is not harmonically related to the carrier frequency. As before, increasing amplitude will increase the signal bandwidth adding more spectral components. Some of the sounds aren't particularly pleasant.

- (d) Change the modulation frequency to 616 Hz and the amplitude to one. A 5:7 ratio between the carrier frequency and the modulating frequency should sound somewhat bell-like, although the attack is not correct in this particular model.

Table 2.4: Bessel Functions

Modulation Index	$J_0(I)$	$J_1(I)$	$J_2(I)$	$J_3(I)$	$J_4(I)$	$J_5(I)$	$J_6(I)$	$J_7(I)$	$J_8(I)$
0.00	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
0.25	0.9844	0.1240	0.0078	0.0003	0.0000	0.0000	0.0000	0.0000	0.0000
0.50	0.9385	0.2423	0.0306	0.0026	0.0002	0.0000	0.0000	0.0000	0.0000
0.75	0.8642	0.3492	0.0671	0.0085	0.0008	0.0001	0.0000	0.0000	0.0000
1.00	0.7652	0.4401	0.1149	0.0196	0.0025	0.0002	0.0000	0.0000	0.0000
1.25	0.6459	0.5106	0.1711	0.0369	0.0059	0.0007	0.0001	0.0000	0.0000
1.50	0.5118	0.5579	0.2321	0.0610	0.0118	0.0018	0.0002	0.0000	0.0000
1.75	0.3690	0.5802	0.2940	0.0919	0.0209	0.0038	0.0006	0.0001	0.0000
2.00	0.2239	0.5767	0.3528	0.1289	0.0340	0.0070	0.0012	0.0002	0.0000
2.25	0.0827	0.5484	0.4047	0.1711	0.0515	0.0121	0.0023	0.0004	0.0001
2.50	-0.0484	0.4971	0.4461	0.2166	0.0738	0.0195	0.0042	0.0008	0.0001
2.75	-0.1641	0.4260	0.4739	0.2634	0.1007	0.0297	0.0071	0.0015	0.0003
3.00	-0.2601	0.3391	0.4861	0.3091	0.1320	0.0430	0.0114	0.0025	0.0005
3.50	-0.3801	0.1374	0.4586	0.3868	0.2044	0.0804	0.0254	0.0067	0.0015
4.00	-0.3971	-0.0660	0.3641	0.4302	0.2811	0.1321	0.0491	0.0152	0.0040
4.50	-0.3205	-0.2311	0.2178	0.4247	0.3484	0.1947	0.0843	0.0300	0.0091
5.00	-0.1776	-0.3276	0.0466	0.3648	0.3912	0.2611	0.1310	0.0534	0.0184
5.50	-0.0068	-0.3414	-0.1173	0.2561	0.3967	0.3209	0.1868	0.0866	0.0337
6.00	0.1506	-0.2767	-0.2429	0.1148	0.3576	0.3621	0.2458	0.1296	0.0565
6.50	0.2601	-0.1538	-0.3074	-0.0353	0.2748	0.3736	0.2999	0.1801	0.0880
7.00	0.3001	-0.0047	-0.3014	-0.1676	0.1578	0.3479	0.3392	0.2336	0.1280
7.50	0.2663	0.1352	-0.2303	-0.2581	0.0238	0.2835	0.3541	0.2832	0.1744
8.00	0.1717	0.2346	-0.1130	-0.2911	-0.1054	0.1858	0.3376	0.3206	0.2235
8.50	0.0419	0.2731	0.0223	-0.2626	-0.2077	0.0671	0.2867	0.3376	0.2694
9.00	-0.0903	0.2453	0.1448	-0.1809	-0.2655	-0.0550	0.2043	0.3275	0.3051
9.50	-0.1939	0.1613	0.2279	-0.0653	-0.2691	-0.1613	0.0993	0.2868	0.3233
10.00	-0.2459	0.0435	0.2546	0.0584	-0.2196	-0.2341	-0.0145	0.2167	0.3179
10.50	-0.2366	-0.0789	0.2216	0.1633	-0.1283	-0.2611	-0.1203	0.1236	0.2851
11.00	-0.1712	-0.1768	0.1390	0.2273	-0.0150	-0.2383	-0.2016	0.0184	0.2250
11.50	-0.0677	-0.2284	0.0279	0.2381	0.0963	-0.1711	-0.2451	-0.0846	0.1421
12.00	0.0477	-0.2234	-0.0849	0.1951	0.1825	-0.0735	-0.2437	-0.1703	0.0451
12.50	0.1469	-0.1655	-0.1734	0.1100	0.2262	0.0347	-0.1984	-0.2252	-0.0538
25.00	0.0963	-0.1254	-0.1063	0.1083	0.1323	-0.0660	-0.1587	-0.0102	0.1530

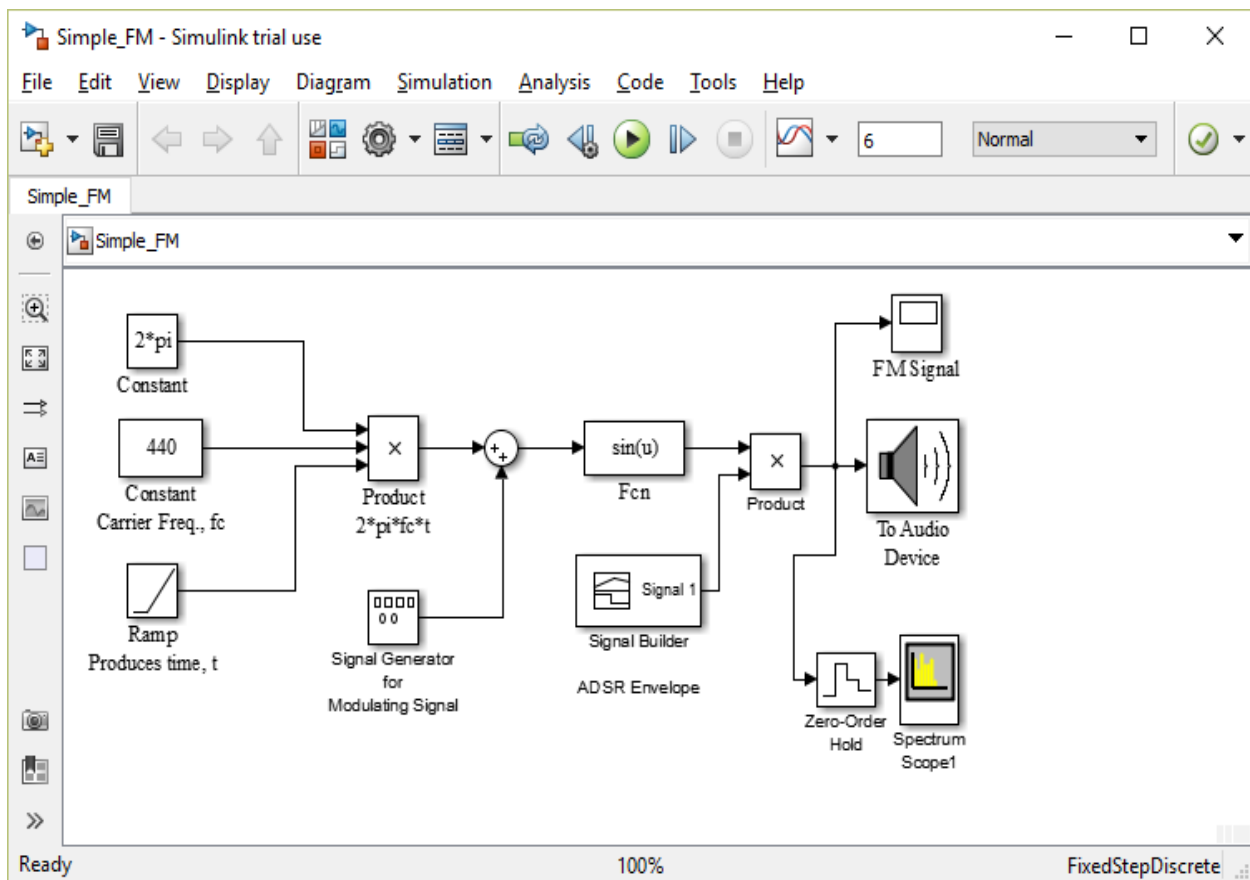


Figure 2.24: Simple_FM.mdl (Example 2.15)

In the early 1970s, John Chowning varied the carrier frequency, modulating frequency, and amplitude envelopes of both the FM signal and the modulating signal to create a variety of musical instruments. The next example simulates Chowning's bell and wood drum instruments.

Example 2.16: FM Synthesis of Selected Chowning Instruments

(a) FM Synthesis of Bell Instrument

The Simulink model used to create Chowning's FM bell instrument is shown in Figure 2.25. Before running the model, the amplitude envelopes, e1 and e2, must be created. Load the bell_FM_env.m file and the Bell_FM.mdl file from the text website into your current MATLAB directory, then run the m-file by typing the command:

```
[e1 e2] = bell_FM_env;
```

After creating the envelopes, turn the volume down on your computer. Then open and run the Bell_FM.mdl in Simulink. The model will produce a nice bell tone.

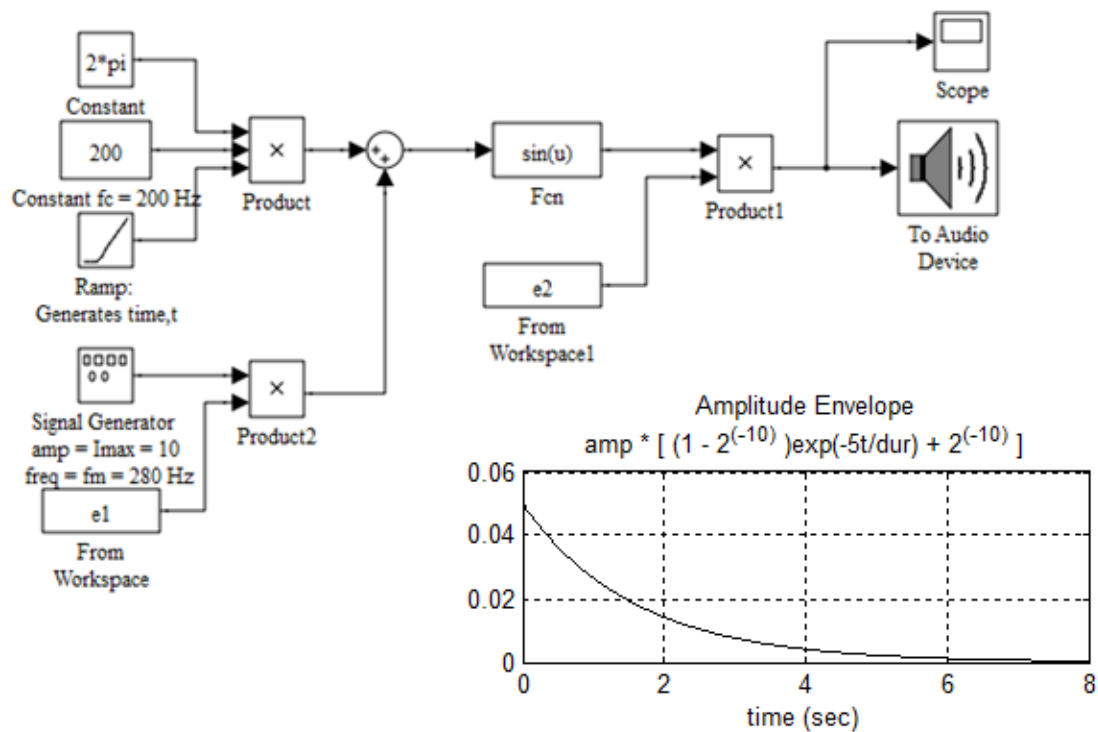


Figure 2.25: Simulink Model of Chowning's FM Bell Instrument

(b) FM Synthesis of Wood Drum

The Simulink model used to create Chowning's FM wood drum instrument and the amplitude envelopes for the modulating waveform and the FM waveform are shown in Figure 2.26. Copy the WoodDrum_FM.mdl file from the text website into your current MATLAB directory, open the model in Simulink, and then run the model.

How does Chowning's FM synthesis of a bell compare to Risset's additive synthesis of a bell? Both use the same type of exponential envelope which is characteristic of a bell sound (quick attack followed by an exponential decay). Risset's bell instrument requires eleven oscillators because in additive synthesis an oscillator is required for each frequency in the desired spectrum. Chowning's bell only requires two oscillators because FM synthesis creates additional frequencies beyond the oscillator frequencies. So FM synthesis is computationally simpler than additive synthesis.

How does the bell instrument compare to the wood drum instrument? The basic structure of the two instruments is the same; that is, the modulating signal, created from an oscillator and an index envelope, is used to vary the frequency of a second oscillator, while a second envelope signal controls the amplitude of the final FM signal. Altering the frequencies, the modulation index envelope, and the final amplitude envelope dramatically changes the sound. The frequencies for the wood drum are much lower than the bell frequencies, but like the bell, are non-harmonic (not integer multiples of one another). In both instruments, the modulation index

envelope is initially high providing many non-harmonic spectral components. The bell instrument has a wider bandwidth than the wood drum since the modulating frequency is higher for the bell (280 Hz vs. 55 Hz). The very quick decay of the modulation index envelope for the wood drum creates a quick burst of energy followed by resonance (80 Hz). The bell instrument has a much more gradual (exponential) decay in the modulation index envelope resulting in a smoother transition through simpler spectrums to a predominant component at the carrier frequency of 200 Hz.

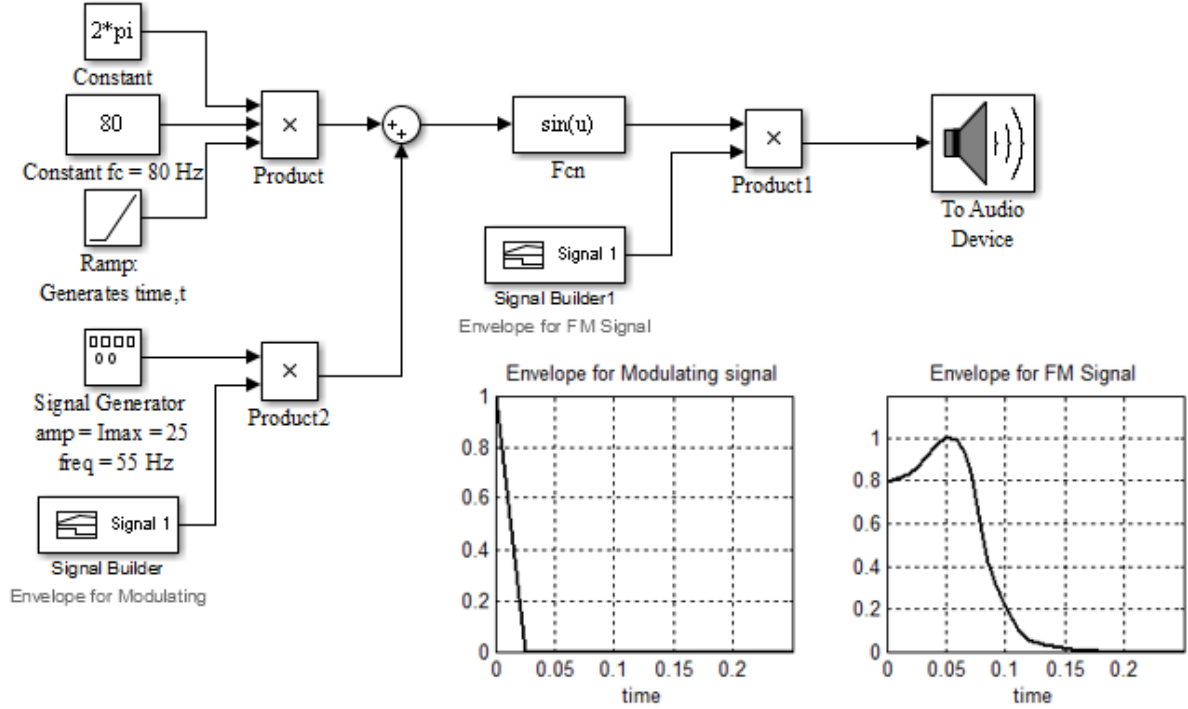


Figure 2.26: Simulink Model of Chowning's FM Wood Drum Instrument

There are several other synthesis methods for creating musical instruments and musical effects including waveshaping synthesis and granular synthesis. These synthesis methods are beyond the scope of this text but are described in detail by Dodge and Jerse. A few more of Chowning's FM instruments will be explored in the lab at the end of this chapter.

2.10 APPLICATION: SIMPLE AUDIO EFFECTS

Tremolo

Tremolo adds a “trembling” or “warbling” effect to an audio signal. Tremolo is a form of amplitude modulation. The amplitude (volume) of an audio signal is varied cyclically at a sub-audio frequency typically less than 10 Hz. The tremolo signal is of the form

$$y(k) = \text{Amp} * [1 + m * \sin(2\pi f_m k T_s)] * x_{\text{audio}}(k) \quad (2.9)$$

The modulation index, m , which controls the range of amplitude variation, is chosen to be between zero and one. A modulation index equal to one (100% modulation) provides the widest

swing in amplitude, while a modulation index close to zero provides very little amplitude variation. The modulation frequency, f_m , controls the rate of amplitude variation and is typically chosen below 10 Hz.

The spectrum of the tremolo signal contains all of the frequencies present in the original audio signal plus two smaller sidebands located f_m Hz above and below each audio frequency. The strength of the sidebands depends directly on the modulation index, m . A sample spectrum is shown in Figure 2.27.

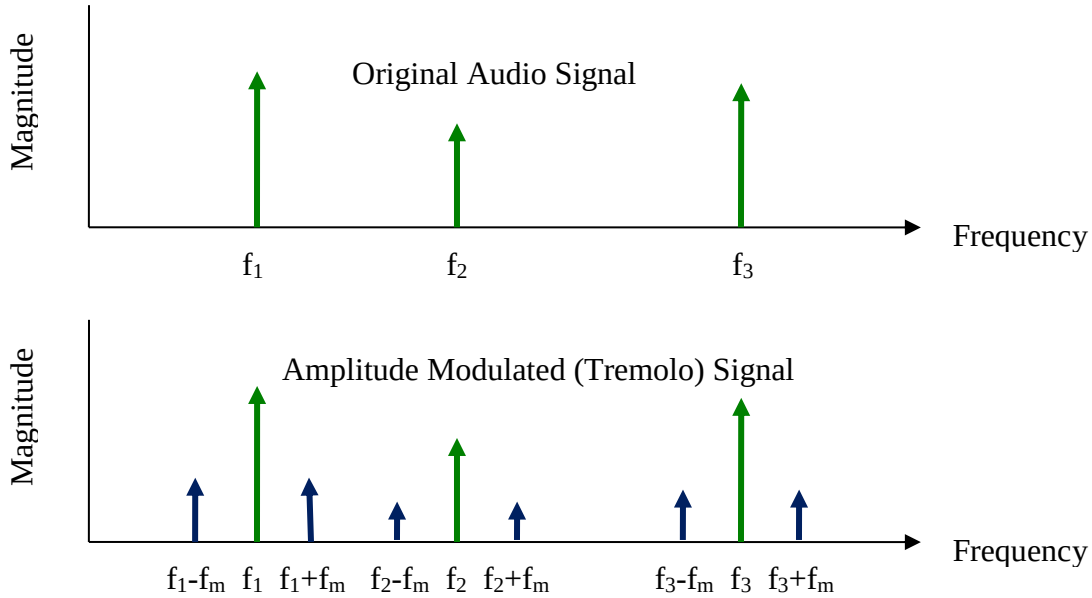


Figure 2.27: Spectrum of Tremolo Signal

Ring Modulation

Similar to tremolo, ring modulation is also a form of amplitude modulation where the amplitude of the audio signal is varied cyclically. A ring modulated signal is simply the product of the original audio signal and a sinusoidal modulating signal:

$$y(k) = \text{Amp} * [\sin(2\pi f_m k T_s)] * x_{\text{audio}}(k) \quad (2.10)$$

Comparing the tremolo signal described in equation 2.9 with the ring modulated signal described in equation 2.10, the tremolo signal simply mixes the original audio signal back in with the product of the audio signal and the oscillator. For readers familiar with AM radio, tremolo is amplitude modulation with a transmitted carrier (except the carrier and message switch roles and the frequencies are much lower), and ring modulation is double sideband suppressed carrier (DSB-SC) modulation.

A sample spectrum of a ring modulated signal is shown in Figure 2.28. Comparing the ring modulated spectrum to the tremolo spectrum in Figure 2.27, the original audio signal frequencies are no longer present in the ring modulated signal. Only the sideband frequencies are present in the ring modulated signal, and these sideband frequencies are stronger than the sideband frequencies in the tremolo signal.

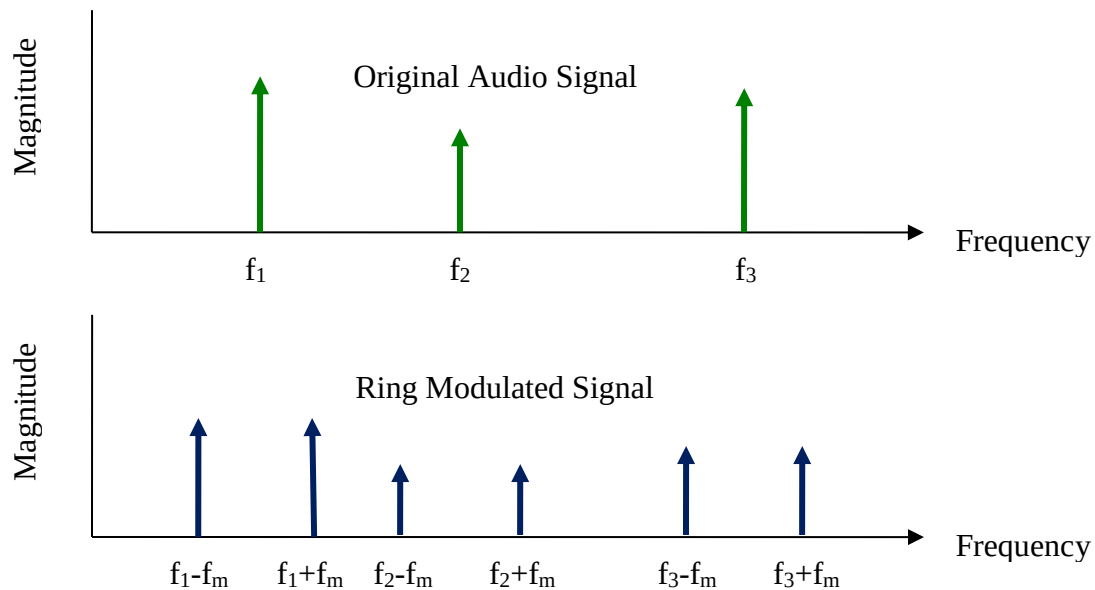


Figure 2.28: Spectrum of Ring Modulated Signal

Example 2.17: Listening to Tremolo and Ring Modulation Effects

Open the AM_RING.mdl shown in Figure 2.29 from the text website. Also, copy the BACH PRELUDE.wav file from the text website into your current MATLAB directory. The model is initially set up for a tremolo effect (via the manual switch) with an amplitude variation rate of 6 Hz. Since the modulation index is initially set to zero, the wav file will play unaltered.

- (a) Double click on the Slider Gain – Modulation Index block and slide it off to the side of the model window. Start the simulation and listen to the wav file unaltered for a few seconds. Then adjust the slider gain to increase the modulation index and observe the “trembling” effect created. Note that the Audio Device has a queue of 1 second so there will be a slight delay for any changes to take effect. In the author’s opinion, a small modulation index of 0.2 to 0.3 produces a pleasant tremolo effect. The effect becomes a bit too pronounced when the modulation index is set to a mid-range value of 0.4 to 0.6 and is downright obnoxious in the higher range of 0.8 and above. Of course, audio effects are a matter of personal preference, so the reader may certainly disagree with the author!
- (b) Set the modulation index on the slider to 0.3 then close the slider. Double click on the Sine Wave 1 block to open it then slide it off to the side of the model window. Re-run the simulation while varying the modulation frequency. Increasing the modulating frequency

spreads the sidebands further apart. A modulation frequency around 15 Hz will cause the beating effect explored in Example 2.15. Choosing a very low modulation frequency of 1 Hz or less makes the amplitude variation definitely distinguishable to the ear. It sounds as if someone is taking the volume control and cycling it up and down.

- (c) Now double click on the manual switch to create the ring modulation effect. The ringing effect should be very apparent at the model modulating frequency of 4 Hz.
- (d) Double click on the Sine Wave block. Vary the modulating frequency above and below 4 Hz. Higher frequencies will create a very dissonant, non-harmonic effect that will change radically with frequency. Lower frequencies will again sound like someone is turning the volume control up and down.

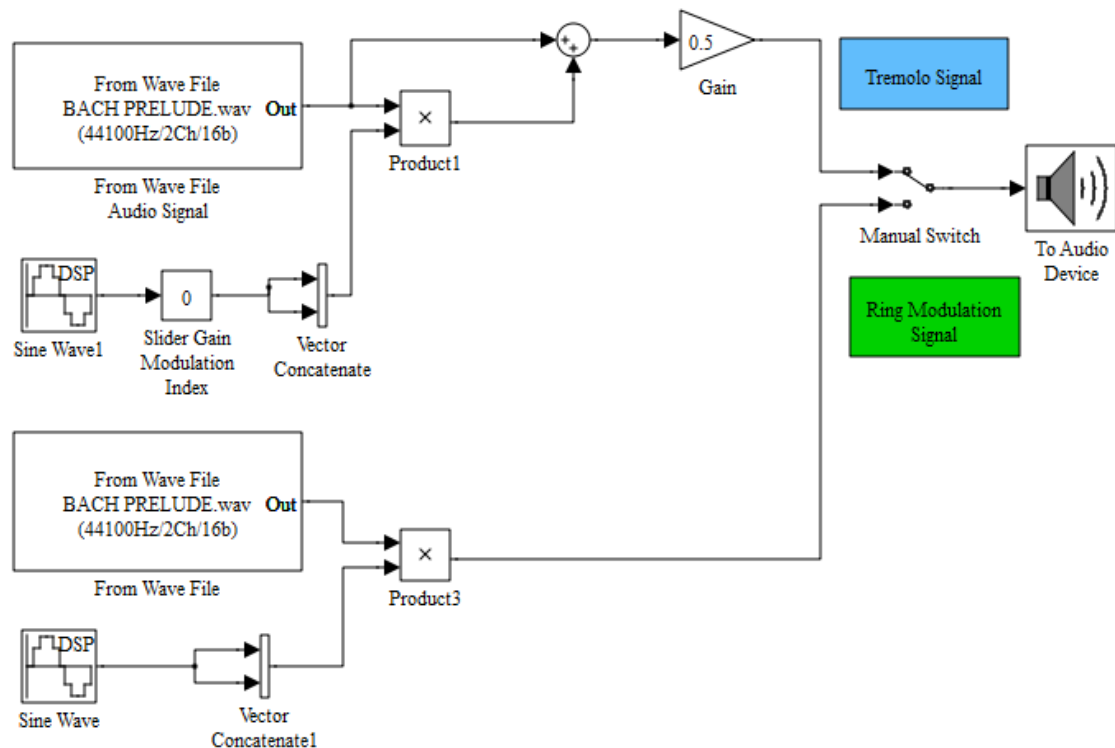


Figure 2.29: Simulink Model for Tremolo and Ring Modulation Effects

Other types of audio effects including chorus, flanging, phasing, and reverb will be explored in Chapters five and six.

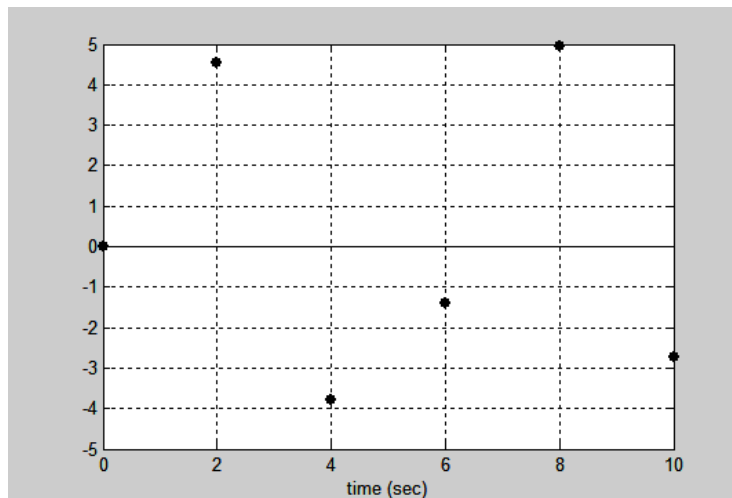
Answers to Chapter 2 Challenge Questions

Question 2.1 Is the signal, y , as defined and stored in MATLAB really an analog signal?

No. The signal y in MATLAB only exists at discrete times: $t = 0, 0.01, 0.02, \dots, 9.99, 10$. The signal is not defined for times in between these values. The plot command in MATLAB connects the sample values (by drawing a line between adjacent values) to make the signal appear analog. The signal, y , in MATLAB is a sampled version of the analog signal.

Question 2.2 If the sampling interval is increased to 2.0 seconds, does the plot of the discrete signal look like the plot of the original analog signal?

No. A plot of the discrete signal with a sampling interval of 2.0 seconds is shown below. It is pretty difficult to see the original sine wave in this plot. Amazingly enough, it will be shown in Chapter 3 that a very nice replica of the original analog signal can be created from samples spaced apart by 2.0 seconds.



Question 2.3 How could the digital signal in Example 2.3 be changed or re-defined to reduce the quantization error?

More levels could be used to define the digital signal. Instead of rounding the signal to the nearest tenth, the signal could be rounded to the nearest hundredth. This would reduce the roundoff error but increase the number of levels or values by a factor of 10. This concept will be discussed further in Chapter 3.

Question 2.4 Look at the MATLAB code for Example 2.4. How many samples of the sine wave are being taken in each cycle?

There are 80 samples per cycle. The period of the sine wave is 0.0008 seconds. The time increment used in the MATLAB code is $1e-5 = 0.00001$ sec. Therefore, the number of samples per cycle is $0.0008/0.00001 = 80$.

Question 2.5 How is the period of the composite signal related to the periods of the two individual sine waves added together to produce the composite signal?

The period of the composite signal is the least common multiple (lcm) of the two individual periods. Least common multiple (lcm) and greatest common factor (gcf) are concepts from elementary school (nice to know the time you spent in 4th grade math didn't go to waste). The least common multiple is the smallest number that each of the individual numbers divide into without a remainder.

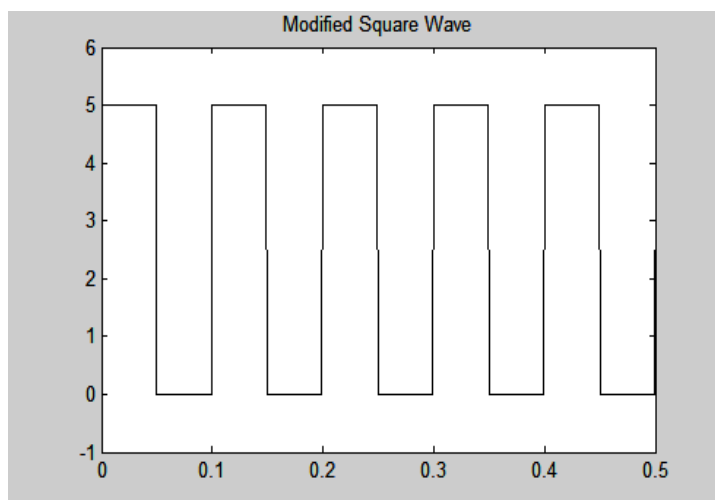
In part (a), the two individual periods are 1 second and $\frac{1}{4}$ second. The smallest number that is divisible by both of these numbers without a remainder is 1 second. Another way to look at this: one second is the smallest period of time in which I can fit complete cycles of both sine waves (one cycle of the 1Hz and four cycles of the 4 Hz).

In part (b), the two individual periods are 1 second and $\frac{2}{3}$ second. The least common multiple for these two numbers is 2. That is, 2 seconds is the smallest interval of time in which I can fit complete cycles of both sine waves (2 cycles of the 1 Hz and 3 cycles of the $\frac{2}{3}$ Hz).

The same concept extends to more than two sine waves added together. Simply find the least common multiple for all the individual periods. Note: if you like greatest common factor (gcf) better, find the gcf for all the individuals frequencies then invert it to find the resulting period.

Question 2.6 How would you modify the MATLAB code above to create a 10 Hz square wave that ranged from 0 to 5 instead of -1 to $+1$?

The 10 Hz is easy – we just replace the 3 with a 10 in the square function. You might want to consider the time range though. Since 10 Hz means 10 cycles/sec, a 2 second time range would display 20 cycles which might be a bit crowded on the plot. A 0.5 second time range would show 5 cycles. What about modifying the range? Well, if we multiply the square wave by 2.5, the range of the square wave will expand to -2.5 to $+2.5$. Adding 2.5 will then shift the range to the required range of 0 to 5.

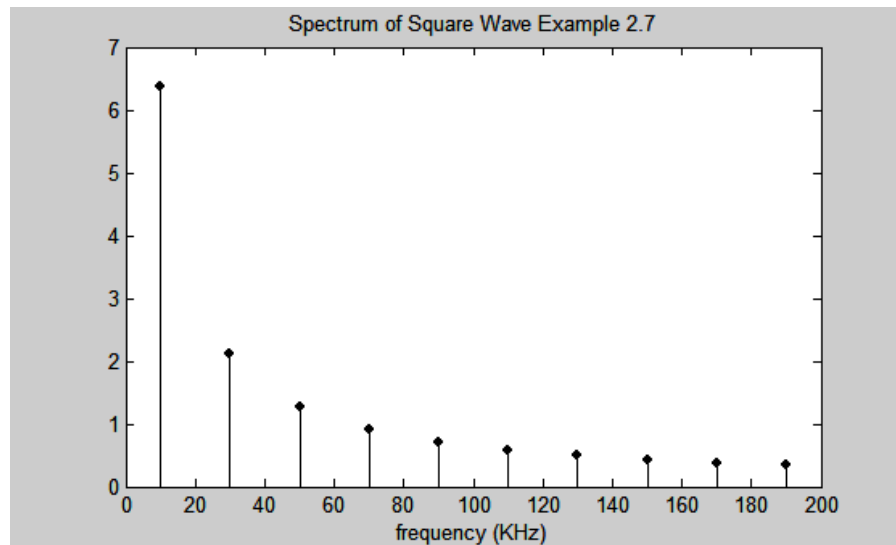


MATLAB Commands:

```
t = 0:0.001:0.5;  
y = 2.5*square(2*pi*10*t) + 2.5;  
plot(t,y); title('Modified Square Wave')
```

Question 2.7 What does the spectrum of the square wave in Example 2.7 look like?

The square wave has a frequency of 10 kHz. According to Table 2.2, the square wave can be expressed as the sum of sine waves at frequencies of 10 kHz, 30 kHz, 50 kHz, ...; that is, the fundamental frequency of 10 kHz plus odd harmonics (multiples) of the fundamental frequency. The amplitude of the sine waves decreases with increasing frequency so the higher frequency components are significantly weaker than the lower frequency components.



Question 2.8 What is the 3dB bandwidth of the 10 kHz square wave from Example 2.7? Does the 3dB bandwidth seem like a good practical bandwidth for this square wave? Why or why not? (Hint: take a look at the spectrum of this square wave in Challenge Question 2.7).

Using the spectrum shown in the answer to challenge question 2.7, the 3dB bandwidth for the 10 kHz square wave would be 10 kHz. This is true because the 10 kHz component has the peak magnitude of $20/\pi = 6.4$. The peak magnitude multiplied by 70.7% equals 4.5. All of the other frequency components have magnitude lower than 4.5.

Is 10 kHz a good practical bandwidth for the square wave? The practical bandwidth is supposed to be a measure of the frequency range that adequately captures the original signal. In this case, if we cut out all frequency components except the 10 kHz component (fundamental frequency), what we have left is a 10 kHz sine wave (not a good match to the square wave). To adequately represent the square wave, some of the higher frequency components must be included. This concept is illustrated in Figure 2.12. For the square wave, a much better measure of practical signal bandwidth would be derived by determining the bandwidth required to capture 98% of the original signal power.

Question 2.9

A 440 Hz sine wave sounds like a pure tone. What makes the 440 Hz sawtooth and square waves sound different than the 440 Hz sine wave?

The harmonics present in a sawtooth wave and in a square wave make these waveforms sound very different than a sine wave. A 440 Hz sine wave has only a single frequency component at 440 Hz causing the sine wave to sound like a pure tone. A 440 Hz sawtooth wave has frequency components at the fundamental frequency, 440 Hz, and at the harmonic frequencies (880 Hz, 1320 Hz, 1760 Hz, etc). The 440 Hz square wave has frequency components at the fundamental frequency, 440 Hz, and at the odd harmonic frequencies (1320 Hz, 2200 Hz, 3080 Hz, etc.).

Chapter 2 Problems

Problem 2.1: Determine the period then plot four cycles of the following signals in MATLAB.

- (a) $\sin(2\pi(400)t)$
- (b) $\sin(2\pi(400)t) + \sin(2\pi(600)t)$
- (c) $\sin(2\pi(400)t) * \sin(2\pi(600)t)$

Problem 2.2: Plot the following signals in MATLAB. Print out both the plots and the MATLAB code used to generate your plots.

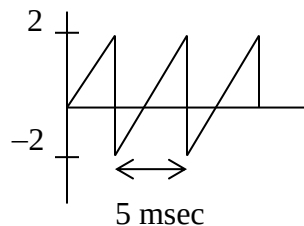
- (a) 6 cycles of a 440 Hz square wave with a 75% duty cycle and a voltage range of -5 to $+5V$.
- (b) 3 cycles of a 500 Hz triangular wave with a voltage range of $+1$ to $+4V$

Problem 2.3: Sketch the spectrum of the following signals.

- (a) $\sin(2\pi(400)t)$
- (b) $\sin(2\pi(400)t) + \sin(2\pi(600)t)$
- (c) $\sin(2\pi(400)t) * \sin(2\pi(600)t)$

Problem 2.4:

- (a) Sketch the spectrum of the sawtooth waveform shown below. Include the first seven frequency components.
- (b) Plot the sum of the first three terms and the sum of the first seven terms of the Fourier series representation of the sawtooth waveform in MATLAB and comment on the match to the original waveform.



Problem 2.5:

- (a) Sketch the spectrum of a 500 Hz triangular wave with a voltage range of $+1$ to $+4V$ (include components to the 9th harmonic frequency).
- (b) Plot the sum of the first two terms and the sum of the first five terms of the Fourier series representation of the triangular wave in MATLAB and comment on the match to the original waveform.

Problem 2.6:

Plot the spectrum of $4\sin^3(2\pi(120)t)$. (Hint: need a trig identity)

Problem 2.7:

The rectangular pulse function is defined as

$$\text{rect}\left(\frac{t}{\tau}\right) = \begin{cases} 1 & |t| < \tau/2 \\ 0 & |t| > \tau/2 \end{cases}$$

- (a) Sketch the rectangular pulse for $\tau = 1$ and for $\tau = 10$.
- (b) Using the Fourier Transform table in the Appendix, find the Fourier Transform of the rectangular pulse for $\tau = 1$ and for $\tau = 10$.
- (c) Using MATLAB, plot the spectrum of the rectangular pulse for $\tau = 1$ and for $\tau = 10$.
- (d) Estimate the 3dB bandwidth for each of the pulses. How do they compare?

Problem 2.8:

Consider the function, $f(t) = te^{-at}u(t)$.

- (a) Plot the signal, $f(t)$, in MATLAB for $a = 0.5$ and $a = 4$.
- (b) Using the Fourier Transform table in the Appendix, find the Fourier Transform of the signal for $a = 0.5$ and $a = 4$.
- (c) Using MATLAB, plot the spectrum of the signal for $a = 0.5$ and $a = 4$.

Problem 2.9: For each of the following two tone audio signals, describe what you expect to hear and explain your answer?

- (a) $\sin(2\pi(294)t) + \sin(2\pi(392)t)$
- (b) $\sin(2\pi(294)t) + \sin(2\pi(300)t)$

Problem 2.10: Using Table 2.4, sketch the spectrum for each of the following FM audio signals.

- (a) $f_{FM}(t) = \sin(2\pi(800)t + 0.25 \sin(2\pi(200)t))$
- (b) $f_{FM}(t) = \sin(2\pi(600)t + 2.0 \sin(2\pi(300)t))$

Problem 2.11: Sketch the spectrum for the tremolo and ring-modulation signals shown below.

- (a) $y_{trem}(t) = [1 + 0.2\sin(2\pi(6)t)] * \sin(2\pi(440)t)$
- (b) $y_{ring}(t) = [\sin(2\pi(4)t)] * \sin(2\pi(440)t)$

CHAPTER 2 LAB EXERCISE

Sinusoidal Signals and Sound Synthesis

Materials

A computer with MATLAB and a soundcard/speaker. A microphone.

Objectives

1. Review properties of sinusoidal signals and how to plot these signals using MATLAB.
2. Write a MATLAB m-function to create a sine wave.
3. Generate tones using the MATLAB sound function.
4. Experiment with FM synthesis to create electronic instrument sounds.
5. Observe the effect of ring modulation on a voice signal.

Procedure

A. Properties of Sinusoidal Functions and Plotting in MATLAB

A sinusoidal function is of the form: $A\sin(2\pi ft + \phi)$

$$\left\{ \begin{array}{l} A = \text{amplitude} \\ f = \text{frequency (Hz)} \\ \phi = \text{phase shift} \\ T = 1/f = \text{period of the sine wave} \end{array} \right.$$

A sinusoidal function is easily plotted in MATLAB by defining an appropriate time vector. It is important to remember that in MATLAB, angles must be entered in radians instead of degrees unless you use the special trig functions designed for degrees (**sind**, **cosd**, ...) Work through the following example to review how to plot sinusoids.

Example Plot three periods of a sinusoidal signal with an amplitude of 10, a frequency of 1.0 kHz, and a phase shift of -60° .

First, calculate the period of the sine wave: $T = 1/f = 1/1000 = 0.001$ sec.

```
T = 0.001;
t = 0:T/100:3*T; %Duration: 3 Periods with 100 samples/cycle
y = 10*sin(2*pi*1000*t - 60*pi/180);
plot(t,y); grid
```

The plot should show three cycles of a sinusoidal signal with amplitude 10 and a period of 1ms. Phase shift can be determined from the plot by zooming in and finding the first peak. (Try it!) The first peak occurs at $t \approx 0.17$ ms. So, phase shift $\cong (0.17 \text{ ms}/T) \times 360^\circ \cong -61^\circ$. The small error is a resolution error occurring because we have sampled the sine wave every 0.01ms to plot it.

1. Let $x_1 = 10\sin(2\pi(200)t)$ and $x_2 = 15\sin(2\pi(200)t - 45^\circ)$. Plot $x_1 + x_2$ then determine the amplitude, frequency, and phase shift of the resulting signal.

Amp = _____ Frequency = _____ Phase Shift = _____

2. Let $x_1 = 10\sin(2\pi(200)t)$ and $x_2 = 15\sin(2\pi(400)t)$. Plot $x_1 + x_2$.

What is the period of the signal? _____

How is the period related to the periods of x_1 and x_2 ? _____

Comment: x_2 is a harmonic of x_1 because its frequency (400 Hz) is an integer multiple of the frequency of x_1 (200Hz). 200 Hz is the fundamental frequency in this exercise.

3. Let $x_1 = 10\sin(2\pi(200)t)$ and $x_2 = 15\sin(2\pi(300)t)$. Plot $x_1 + x_2$.

What is the period of the signal? _____

How is the period related to the periods of x_1 and x_2 ? _____

4. Let $x_1 = 10\sin(2\pi(200)t)$ and $x_2 = 15\sin(2\pi(300)t)$. Plot $x_1 \cdot x_2$.

What is the period of the signal? _____

What two frequencies are present in the signal $x_1 \cdot x_2$? _____

B. Function to generate sine wave

In part A of this lab, we have created and plotted several sinusoidal signals. In doing so, many of the same commands have been used multiple times.

1. Following the procedure outlined in the lab exercise from Chapter 1, write an m-function titled `sine_wave` to create a sinusoidal signal. The function should have five input arguments: amplitude, frequency, phase, duration, and sampling frequency and two output arguments: signal (the values of the sine wave) and time (the sample times). Be sure to include comments on how the function is executed.
2. Save the file as `sine_wave` (Save it in your current director folder).
3. Now verify that your `sine_wave` function works properly by creating then plotting three periods of a sinusoidal signal with an amplitude of 10, a frequency of 1 kHz, a phase shift of -60° , and a sampling frequency of 100,000 Hz using the following commands.

```
[y,t]=sine_wave(10, 1000, -60*pi/180, 3*(1/1000), 100000);
plot(t,y); grid
```

C. Generating Tones using MATLAB

The layout of a piano keyboard is shown in Figure 1. The A key above middle C produces a tone with a frequency of 440 Hz. The frequencies of the other keys can be computed from the following formula:

$$f = 440 * 2^{(\text{keynum} - 49)/12}$$

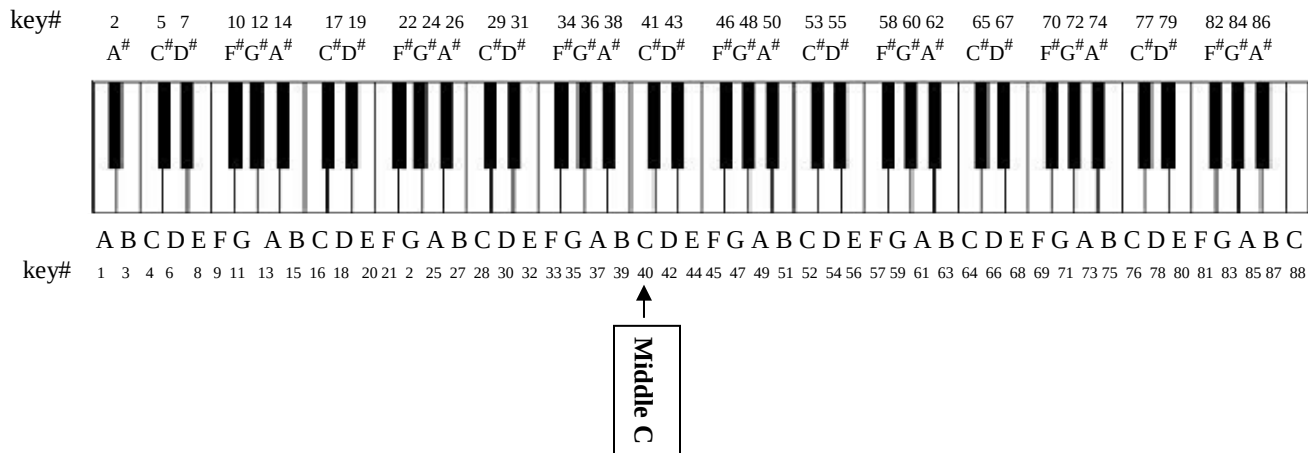


Figure 1: Piano Keyboard with Key Numbers

1. Create a 440 Hz sinusoidal signal (named `tone_A`) with an amplitude of 1, a duration of 2 sec, and a sampling frequency of 11025 Hz using your `sine_wave` function. Then execute the following MATLAB command:

```
sound(tone_A, 11025)
```

2. In order to illustrate the importance of matching the sampling rate of the signal to the sampling rate of the D/A converter, do the following. Without changing the signal itself, execute the following MATLAB command

```
sound(tone_A, 0.5*11025)
```

What happens to the pitch and the duration of the signal? Why?

Now double the output frequency in the sound command (2×11025). What happens to the pitch? Why?

3. Using your `sine_wave` function, create the following three signals:

tone_C: 261.6 Hz sine wave with a duration of 2 seconds, an amplitude of 1/3 and a sampling frequency of 11025 Hz.

tone_E: 329.6 Hz sine wave with a duration of 2 seconds, an amplitude of 1/3 and a sampling frequency of 11025 Hz.

tone_G: 392 Hz sine wave with a duration of 2 seconds, an amplitude of 1/3 and a sampling frequency of 11025 Hz.

At the MATLAB command prompt, type the following commands:

```
tone = [tone_C, tone_E, tone_G];  
sound(tone, 11025)
```

What happens? Why?

At the MATLAB command prompt, type the following commands:

```
chord = tone_C + tone_E + tone_G;  
sound(chord, 11025)
```

What happens? Why?

4. Clear `tone_C` and then recreate `tone_C` with a duration of 1 second. Don't change `tone_E` or `tone_G`. Repeat the `tone` and `chord` commands in the previous step. What happens? Why?
5. The first few measures of Prelude in C Major by Johann Sebastian Bach are shown in Figure 2 along with the accompanying key numbers to accommodate those students who do not read music. Write an m-function that will convert the key numbers to the appropriate frequencies, create the tones, and play the first few measures. A duration of 0.125 seconds for each note works well for this piece.



```
key_num = [40 44 47 52 56 47 52 56 40 44 47 52 56 47 52 56 40 42 49 54 57 49 54 57
40 42 49 54 57 49 54 57];
```

Figure 2: Bach's Prelude in C

E. Voice Effects using Ring Modulation

1. Build the ring modulator shown in Figure 3 in Simulink using the block parameters displayed in Figure 4. The From Audio Device and Sine Wave blocks are in the DSP Toolbox under Sources. The To Audio Device block in the DSP Toolbox under Sinks.

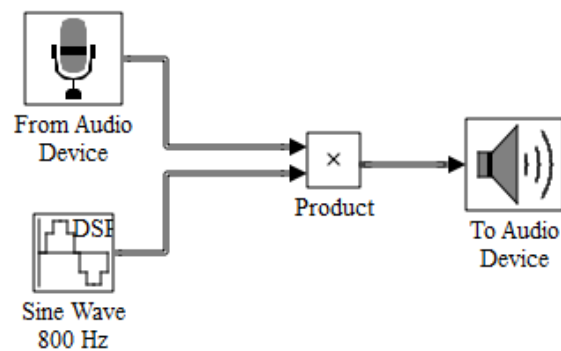


Figure 3: Ring Modulation of Voice

2. Talk into the microphone and listen through the headphones to hear the effect of ring modulation on your voice – there will be a 3 second delay due to the queue.
3. What does your voice sound like? Modify the modulating frequency, 800 Hz, and describe the effect.
4. Describe how the frequency spectrum of your voice is altered by the ring modulator.

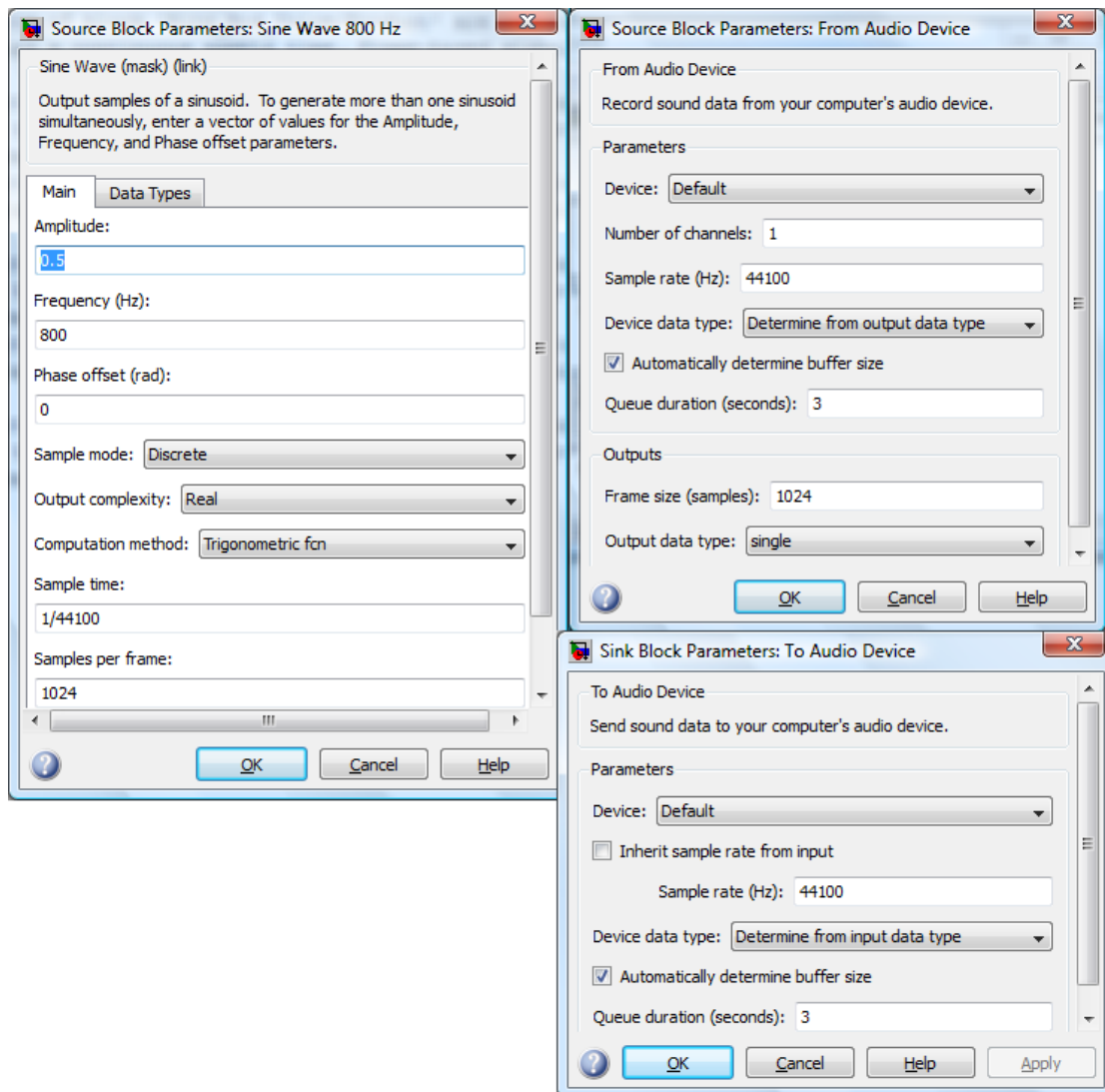


Figure 4: Block Parameters for Ring Modulator

CHAPTER 3 SAMPLING AND RECONSTRUCTING SIGNALS

Many DSP applications begin with analog signals. In order to process these analog signals, the signals must first be sampled and converted to digital signals. In some cases, after the digital signals are processed, they must then be converted back to analog signals. The process of creating an analog signal from a digital signal is referred to as reconstruction. Figure 3.1 is a generic illustration of a DSP system. This chapter explains the concepts of sampling analog signals and reconstructing an analog signal from digital samples.

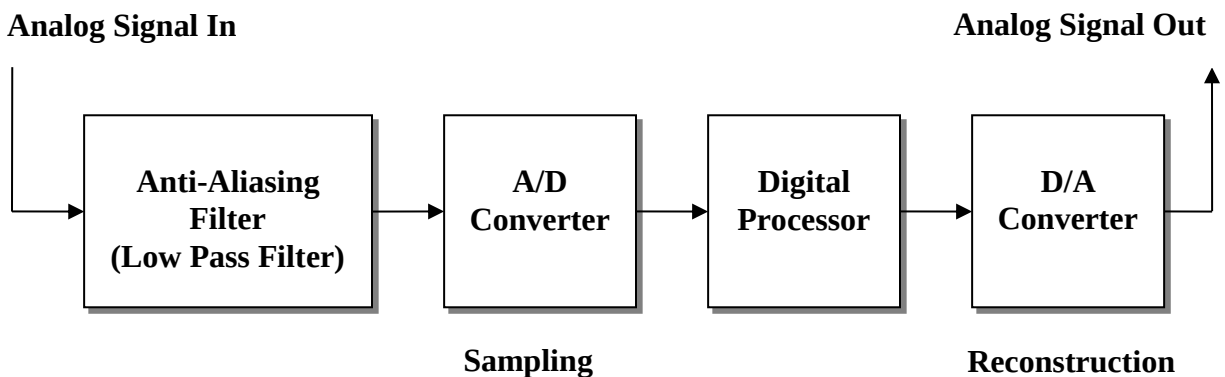


Figure 3.1: Block Diagram of a DSP System

3.1 THE BASICS OF SAMPLING

Analog signals are converted to digital signals using an A/D converter. The A/D converter acquires a sample of the analog signal every T_s seconds and converts the signal level (often voltage) to a set of bits – a digital representation of the analog signal level.

Figure 3.2 shows a sine wave signal being sampled every 0.2 seconds. The square markers indicate the times at which the signal is sampled. The dotted horizontal lines indicate the available discrete levels for the A/D converter. In order to keep things simple, a 3-bit converter is used; that is, every analog sample will be represented by a 3-bit digital code. So, there are eight discrete levels (more about this later) marked as dashed lines. The 3-bit code associated with each discrete level is shown to the right of each level. Table 3.1 lists the 3-bit digital code corresponding to each of the sampling times.

There are a couple of key observations to make here:

- The signal information between sampling intervals is lost in the sampling process.
- The signal level at the sampling instant is rounded (or sometimes truncated) to the nearest discrete level resulting in rounding (quantization) error.

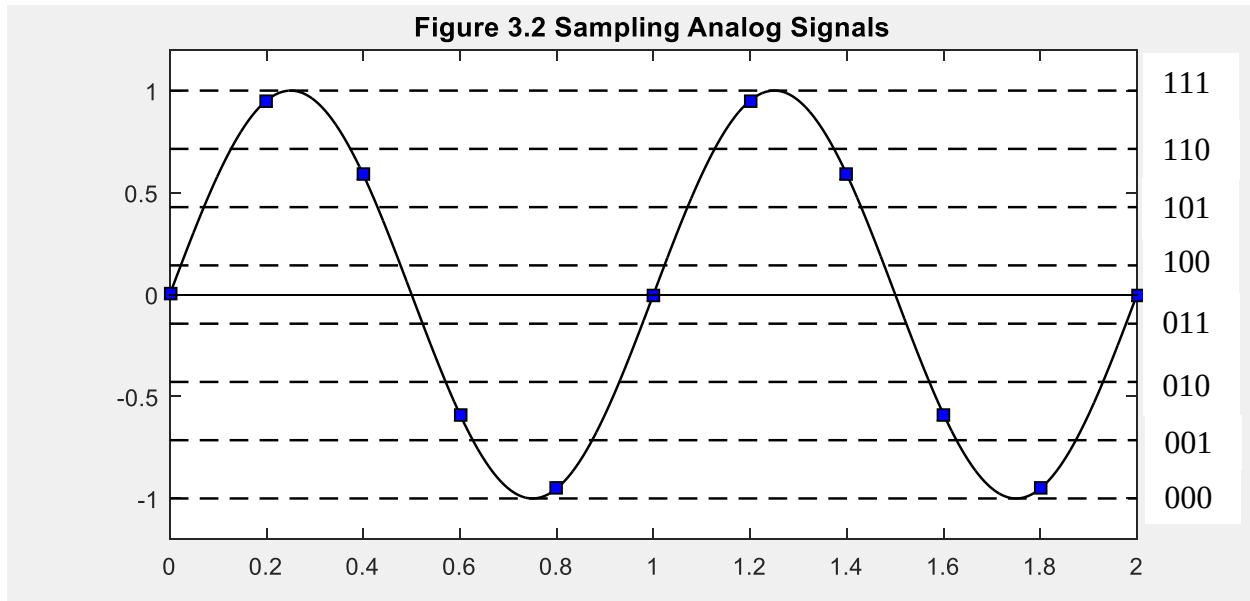


Figure 3.2: Sampling Analog Signals

Table 3.1: 3-bit Digital Code at each Sampling Interval

SAMPLE TIME (SEC)	3-BIT DIGITAL CODE
0	100
0.2	111
0.4	110
0.6	001
0.8	000
1.0	011
1.2	111
1.4	110
1.6	001
1.8	000
2.0	011

3.2 A/D CONVERSION: TERMINOLOGY AND EQUATIONS

This section describes the common terminology and equations associated with A/D converters. Table 3.2 defines the notation that will be used throughout the remainder of this text.

Sampling Frequency

The sampling frequency is given by:

$$F_s = 1/T_s \quad (3.1)$$

Table 3.2: Notation for A/D Conversion

SYMBOL	DEFINITION	UNIT
T_s	Sampling Interval	sec
F_s	Sampling Frequency	Hz
B	Number of Bits for A/D	bits
L	Number of Levels for A/D	levels
V_{FS}	Full-swing Voltage	volts
q	Quantization level (step size)	volts
SNR	Signal to Noise Ratio	dB

Number of Discrete Levels

The number of discrete levels for a B-bit A/D converter is given by:

$$L = 2^B \quad (3.2)$$

The 3-bit A/D in the previous section had $2^3 = 8$ levels; that is, the number of binary numbers you can create using only 3 bits. A 12-bit A/D would have 4096 discrete levels.

Quantization Level (Step Size)

The quantization level (or difference between two adjacent discrete levels) is given by:

$$q = \frac{V_{FS}}{(2^B - 1)} = \frac{V_{max} - V_{min}}{(2^B - 1)} \quad (3.3)$$

The full swing voltage is simply the range of analog voltage inputs the A/D converter is designed to accept. An A/D converter that accepts analog input voltages between 0 and +5V has a full-swing voltage of 5V. An A/D converter that accepts analog input voltages between -2V and +2V has a full-swing voltage of 4V.

Resolution

The term resolution refers to the number of bits in the A/D converter or to the step size.

Quantization Error

Quantization error is the error that occurs when an analog voltage level is rounded (or truncated) to the nearest discrete voltage level. If an A/D converter rounds to the nearest level, then the maximum quantization error is half the step size ($q/2$). If the A/D converter truncates (always rounds down), then the maximum quantization error is equal to the step size (q).

Signal to Noise Ratio (SNR)

The signal to noise ratio is the ratio (in dB) of the signal level (rms) to the noise level (rms). Taking only the quantization error noise into account, the best (theoretical) signal to noise ratio for a B-bit A/D converter is calculated using Equation 3.4.

$$\text{SNR}_{\text{dB}} (\text{Best}) = 6.02B + 1.76 \quad (3.4)$$

The actual SNR will be about 2 to 10 dB lower than this theoretical best value due to the nonlinearities and on-chip noise associated with practical design of IC chips.

Example 3.1: A/D Converter Characteristics

A 16 bit A/D converter has a sampling frequency of 48 kHz, accepts analog input voltages ranging from -5V to $+5\text{V}$, and rounds to the nearest discrete level. Calculate the sampling interval, T_s , the number of discrete levels, L , the step size, q , the maximum quantization error, and the SNR (Best) for this A/D converter.

Solution:

$$\text{Sampling Interval} \quad T_s = 1/F_s = 1/(48000 \text{ Hz}) = 0.000020833 \text{ seconds} = 20.833 \mu\text{s}$$

$$\text{Number of Levels} \quad L = 2^B = 2^{16} = 65536 \text{ levels}$$

$$\text{Step Size} \quad q = V_{\text{FS}}/(2^B - 1) = (10\text{V}) / 65535 = 0.0001526 \text{ V} = 0.1526 \text{ mV}$$

$$\text{Maximum Quantization Error} = q/2 = 0.07513 \text{ mV}$$

$$\text{SNR (Best)} \quad \text{SNR (Best)} = 6.02 \cdot 16 + 1.76 = 98.08 \text{ dB}$$

Challenge Question 3.1

How could the quantization error in an A/D converter be reduced? Are there any negative consequences to this change?

3.3 TYPES OF A/D CONVERTERS

This section describes several different types of A/D converters: successive approximation, flash, and sigma-delta as well as the advantages and disadvantages of each.

Successive Approximation A/D Converter

A schematic diagram for an 8-bit successive approximation A/D converter is shown in Figure 3.3. This converter works by determining what each bit should be, one at a time, beginning with the most significant bit (D_7). It takes one clock cycle to determine each bit, so an 8-bit converter would require 8 clock cycles to determine the 8-bit code for the analog sample. The successive approximation A/D is very similar to the number-guessing game described in the next challenge question.

Challenge Question 3.2

I am thinking of a number from 1 to 64. You are to guess this number as quickly as possible. Each time you guess, I will only tell you whether your guess is high or low. How would you play the game to minimize the number of guesses you would have to make? If you play it smart, what is the most number of guesses you would ever have to make?

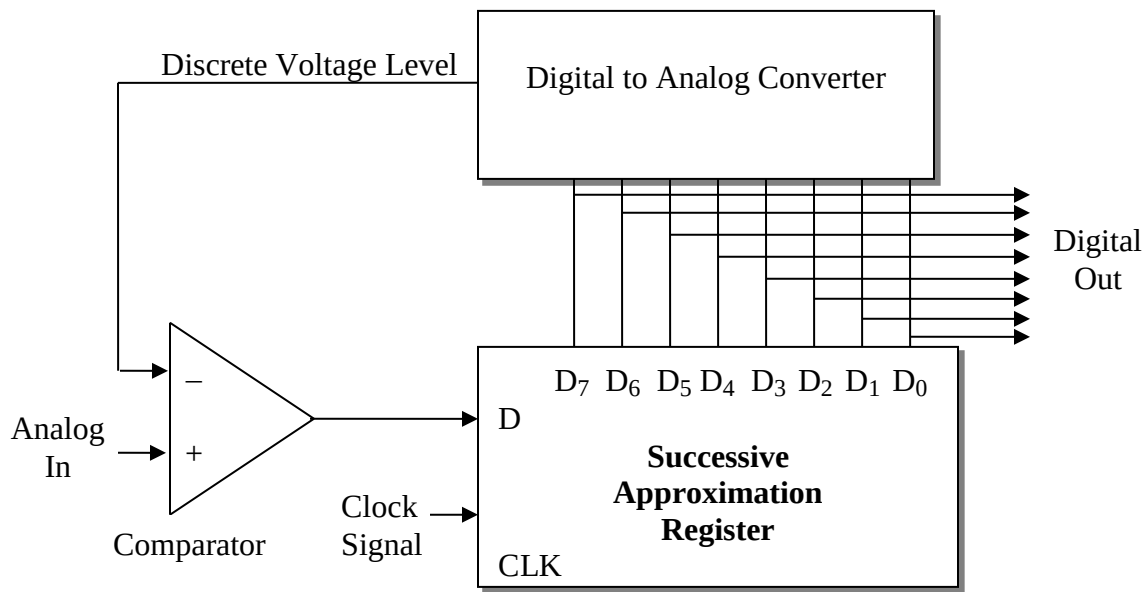


Figure 3.3: Successive Approximation A/D Converter

Here's what happens in each clock cycle:

In the first clock cycle, the successive approximation register sets the most significant bit (MSB) to 1 and all other bits to 0. The D/A converter sends out the discrete voltage level corresponding to 10000000 (the halfway point in the guessing game). The comparator compares the analog voltage level to this discrete voltage level. If the analog voltage is higher than this level, the comparator outputs a HIGH and the successive approximation register keeps the MSB (D_7) equal to 1. If the analog voltage is lower than the discrete level, the comparator outputs a LOW and the successive approximation register sets the MSB (D_7) equal to 0. At the end of this cycle, the proper value for the most significant bit has been determined.

In the second clock cycle, the successive approximation register sets the next bit down (D_6) to 1. The MSB stays set at the value determined in the first clock cycle. The rest of the bits stay at 0. The D/A converter sends out the discrete voltage level corresponding to $*1000000$ (* represents the value of the MSB determined in the first clock cycle). The comparator compares the analog voltage level to this discrete voltage level. If the analog voltage is higher than this level, the comparator outputs a HIGH and the successive approximation register keeps D_6 equal to 1. If the analog voltage is lower than the discrete level, the comparator outputs a LOW and the successive approximation register sets D_6 equal to 0. At the end of this cycle, the proper value for D_6 has been determined.

In the third clock cycle, D_5 will be determined following the same procedure as described in the previous paragraph for D_6 . After eight clock cycles, the 8-bit code for the incoming analog sample has been determined. This entire process must occur within one sampling interval, T_s .

Flash (Parallel) A/D Converter

A schematic diagram for a 3-bit flash A/D converter is shown in Figure 3.4. The voltage divider circuit is used to create the following discrete voltage levels: $(7/8)V_{cc}$, $(6/8)V_{cc}$, $(5/8)V_{cc}$, $(4/8)V_{cc}$, $(3/8)V_{cc}$, $(2/8)V_{cc}$, $(1/8)V_{cc}$. These discrete voltage levels are used as inputs into the comparators. The analog input voltage is compared to all of the discrete levels at the same time. All of the comparators with discrete voltage levels smaller than the analog input voltage put out a HIGH. For example, suppose the analog input voltage is $(3/4)V_{cc}$. This voltage level is higher than all the discrete levels except $7/8V_{cc}$. So, all of the comparators except the top one ($7/8V_{cc}$) would output a HIGH.

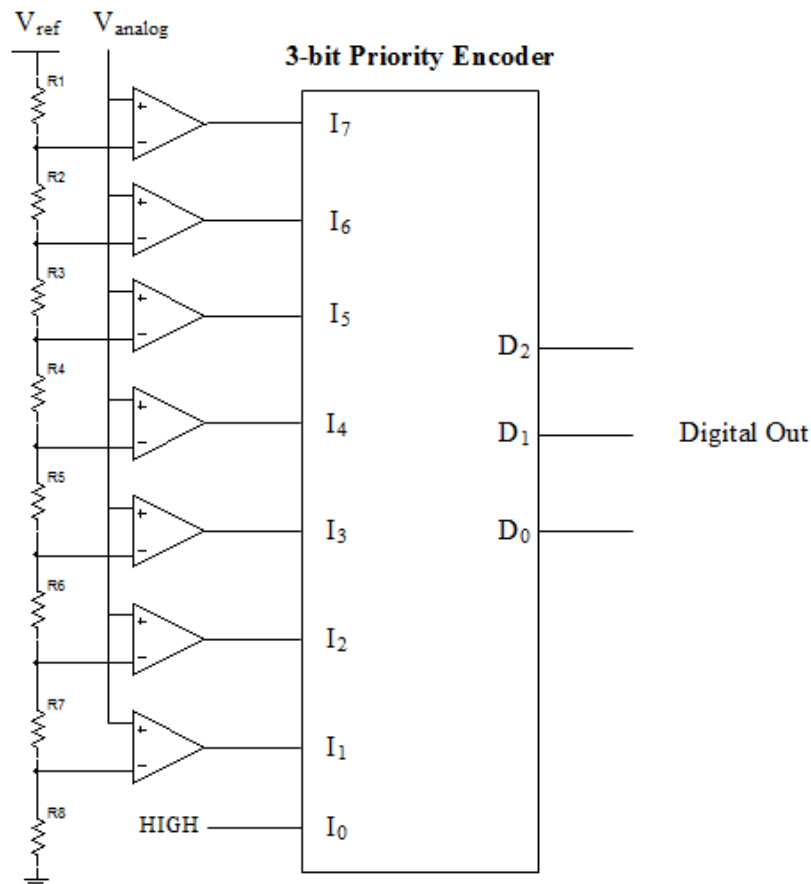


Figure 3.4: 3-Bit FLASH A/D Converter

The 3-bit priority encoder creates a 3-bit code for an input line with a HIGH on it. If there are multiple input lines with a high, it selects the largest input line. So, if I_6 , I_5 , I_4 , I_3 , I_2 , I_1 , I_0 are all HIGH, then the encoder will output the 3-bit binary code for 6 (i.e. 110).

The flash A/D converter is much faster than the successive approximation A/D because it compares the analog voltage signal to all the discrete levels at the same time and outputs the B-bit code for that analog sample. The successive A/D converter sets one bit at a time and requires B clock cycles to output a B-bit code. So what is the downside of a flash A/D converter? The

flash A/D is considerably more expensive than a successive approximation A/D. The simple 3-bit converter shown in Figure 3.4 requires seven comparators. An 8-bit flash converter would require $2^8 - 1 = 255$ comparators while a 12-bit flash A/D would need $2^{12} - 1 = 4095$ comparators. So, there is a tradeoff between speed and cost.

Pipeline A/D Converter

A simple schematic diagram of an 8-bit pipeline ADC is shown in Figure 3.5. Like the successive approximation (SAR) converter, the pipeline A/D converter determines the bits in stages starting with the most significant bit. Unlike the SAR, which must hold each analog voltage sample until all the bits have been determined, the pipeline ADC can begin processing a new analog voltage sample as soon as the most significant bit and the voltage input into the next stage have been determined. As indicated in Figure 3.5, the input voltage to each stage is compared to $\frac{1}{2}$ of the full-swing voltage. The comparator output determines the bit for that stage and feeds a simple 1-bit DAC circuit. A residual voltage is computed then multiplied by a residual gain of 2 to determine the input voltage for the next stage.

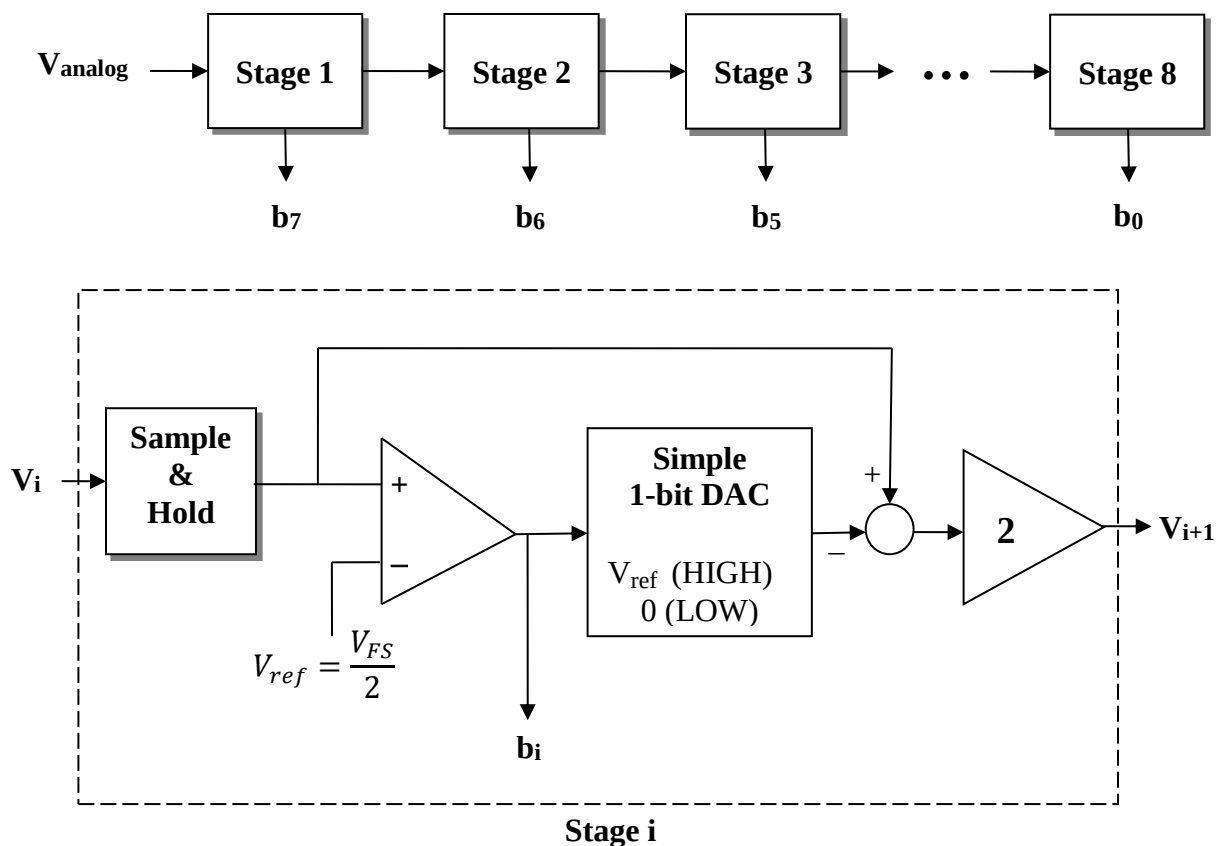


Figure 3.5: Pipeline ADC

Example 3.2: Pipeline ADC

Suppose the full-swing voltage is 5V and the analog input sample is 3.2V. Determine the input and output of each stage of the simple 8-bit pipeline ADC shown in Figure 3.5

Solution

The analog input exceeds half the full-swing voltage so the comparator output in the first stage would be high and the most significant bit would be set to 1. The 1-bit DAC output would be 2.5V (half the full-swing voltage), the residual voltage would be 0.7V ($3.2\text{V} - 2.5\text{V}$), and the input voltage to the next stage would be double the residual voltage or 1.4V. This process would be repeated in the second stage to find the next bit using 1.4V as an input while the first stage could begin processing a new analog input sample. The results from each stage are summarized in Table 3.3. As indicated in Table 3.3, the input analog sample of 3.2V is converted (correctly) to the 8-bit code 10100011.

Table 3.3: Pipeline ADC Example

Stage	Input Voltage	Comparator Output	DAC Output	Residual Voltage	Input to Next Stage
1	3.2	High; b7=1	2.5	0.7	1.4
2	1.4	Low; b6=0	0	1.4	2.8
3	2.8	High; b5 = 1	2.5	0.3	0.6
4	0.6	Low; b4=0	0	0.6	1.2
5	1.2	Low; b3=0	0	1.2	2.4
6	2.4	Low; b2=0	0	2.4	4.8
7	4.8	High; b1=1	2.5	2.3	4.6
8	4.6	High; b0=1			

A pipeline ADC requires more circuitry than an SAR but as indicated previously allows for higher sampling rates due to the parallel nature of the architecture. A pipeline ADC requires considerably less circuitry than a flash ADC but cannot achieve sampling rates as high as a flash ADC. The diagram in Figure 3.5 is a very simplified schematic of a pipeline ADC. There are considerable variations in the architecture of pipeline ADCs. For example, some pipeline ADCs determine two bits per stage by replacing the single comparator shown in Figure 3.5 with a 3-bit flash converter. The residual voltage going into the next stage is multiplied by 4 instead of 2. Also, most pipelined ADCs use digital error correction algorithms to reduce the accuracy requirements for the comparators in each stage.

Sigma-Delta A/D Converter

A simplified schematic diagram of a Sigma-Delta A/D converter is shown in Figure 3.6. The sigma-delta A/D converter produces one bit per sample but samples at a much higher rate than required (oversampling). The integrator block adds the output of the summer to the previous integrator output (accumulator). The comparator block compares the integrator output to ground and produces a 1 (High) if the integrator output is at least 0V and a 0 (Low) if the integrator output falls below 0V. The output of the comparator is fed back to a 1-bit DAC which produces the maximum reference voltage for a 1 and the minimum voltage level for a 0. The digital low-pass filter and decimator extracts information from the stream of bits, and reduces the sampling

rate by averaging the data stream and filtering out quantization noise outside the band of interest. Digital filters and decimation will be discussed in Chapter 5.

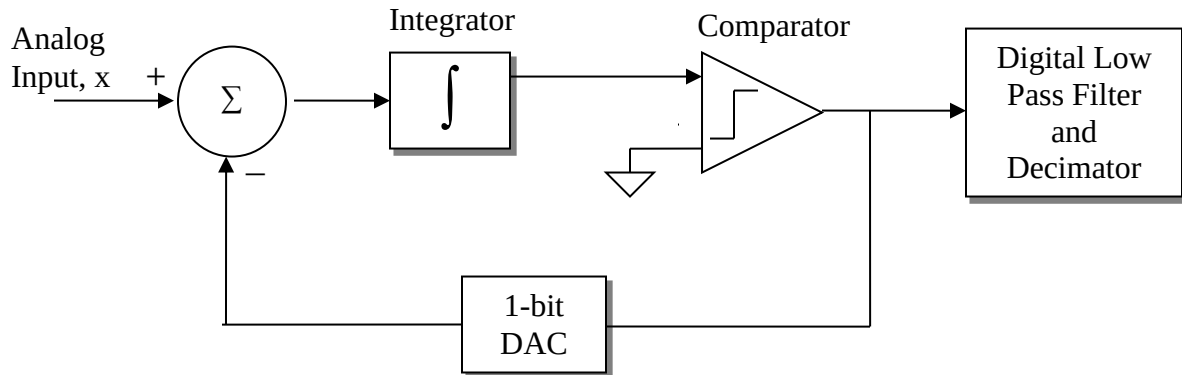


Figure 3.6: Sigma-Delta A/D Converter

Table 3.4 shows an example of the outputs of the various blocks in the Sigma-Delta converter for an input voltage level of 1.5V and an allowable input signal range of -2.5V to 2.5V . The integrator is initialized to 0.

Table 3.4: Sigma-Delta Converter Example

X: INPUT VOLTAGE	SUMMER OUTPUT	INTEGRATOR OUTPUT	COMPARATOR OUTPUT	1-BIT DAC OUTPUT
1.5V	1.5V	1.5V	1	2.5V
1.5V	-1.0V	0.5V	1	2.5V
1.5V	-1.0V	-0.5V	0	-2.5V
1.5V	4.0V	3.5V	1	2.5V
1.5V	-1.0V	2.5V	1	2.5V
1.5V	-1.0V	1.5V	1	2.5V
1.5V	-1.0V	0.5V	1	2.5V
1.5V	-1.0V	-0.5V	0	-2.5V
1.5V	4.0V	3.5V	1	2.5V
1.5V	-1.0V	2.5V	1	2.5V
1.5V	-1.0V	1.5V	1	2.5V
1.5V	-1.0V	0.5	1	2.5V
1.5V	-1.0V	-0.5V	0	2.5V
1.5V	4.0V	3.5V	1	2.5V
1.5V	-1.0V	2.5V	1	2.5V
1.5V	-1.0V	1.5V	1	2.5V
1.5V	-1.0V	0.5V	1	2.5V
1.5V	-1.0V	-0.5V	0	-2.5V
1.5V	4.0V	3.5V	1	2.5V
1.5V	-1.0V	2.5V	1	2.5V

From Table 3.4, it can be seen that a pattern emerges every ten cycles (bits). 8 of 10 bits or 80% of the bit stream is 1s. How does this relate to the input voltage?

Voltage Range = 5.0V (−2.5V to 2.5V)

Input Voltage = 1.5V

Input Voltage is 4.0V (out of a possible 5V) above the minimum voltage of -2.5V

Input Voltage is $4.0\text{V}/5.0\text{V} \times 100\% = 80\%$ of the full-scale voltage.

Therefore 80% of the bitstream will be 1s.

Table 3.5 shows the output bit stream for several different voltage levels between -2.5V and 2.5V . These can be derived using the same approach used for the input voltage of 1.5V in Table 3.4. Notice that the density of 1s is directly proportional to the input voltage.

Table 3.5: Bit Patterns for Various Input Voltage Levels

[illegible]

Sigma-delta converters are low cost, high resolution A/D converters with wide dynamic range. The integrator acts as a noise shaping filter – pushing more of the quantization noise into a higher frequency range that will then be filtered out by the digital low-pass filter and decimator. Most commercial sigma-delta ADCs use multiple integrator loops to push more of the noise into the higher frequency band thus improving the SNR. A very nice interactive design tool that illustrates the how sigma-delta converters work is available on Analog Devices website at: <http://designtools.analog.com/dt/sdtutorial/sdtutorial.html> .

3.4 NYQUIST-SHANNON SAMPLING THEOREM

The Nyquist-Shannon sampling theorem describes the conditions under which an analog signal can be reconstructed from its sampled values. This theorem is fundamental to digital signal processing applications including digital radio, HDTV, and cell phones to name a few. Before proceeding, you may want to review Section 2.7 on signal bandwidth.

Nyquist-Shannon Sampling Theorem

An analog signal, $x(t)$, that is band-limited to f_B Hz can be perfectly reconstructed from its sampled values if the sampling frequency $F_s > 2 \cdot f_B$ Hz.

Comments:

- $F_s = 2 \cdot f_B$ Hz is known as the Nyquist rate (the minimum sampling rate).
- The theorem assumes the sampled values for the analog signal, $x(t)$ are exact. However, from the previous section on A/D conversion, we know the samples are actually quantized to a finite number of discrete levels and do not match the analog signal exactly.
- The theorem assumes the analog signal is band-limited which may not be the case. A signal cannot be both band-limited and time-limited.

The next several sections will illustrate that even if a signal must be pre-filtered in order to band-limit the signal, and the sampled values are not an exact match to the original signal, it is still possible to reconstruct an analog signal with sufficient fidelity to satisfy practical system performance criteria.

Spectrum of Sampled Signals (Ideal Sampling)

To promote an understanding of the sampling theorem, it is useful to look at the spectrum of sampled signals. Ideal sampling means capturing an exact sample value of the signal at the exact instant of time the sample is needed. The practical reality of quantization noise (limited resolution) and acquisition time associated with A/D converters is ignored.

Figure 3.7a shows the spectrum of a hypothetical analog signal. Figure 3.7b shows the spectrum of the analog signal sampled at the Nyquist rate ($F_s = 2 \cdot f_B$). The original analog signal spectrum plus spectral images centered at F_s , $2F_s$, $3F_s$... appear in the spectrum. Notice that the original signal could be recovered with a low-pass filter of bandwidth B ; that is, the spectral images can be filtered out.

Figure 3.7c shows the spectrum of the analog signal sampled above the Nyquist rate. The spectral images are further apart because the sampling frequency F_s is larger. Again, the original signal could be recovered with a low-pass filter. It is worth noting that the low-pass filter would not need to be nearly of the quality of that required in Figure 3.7b.

Finally, Figure 3.7d shows the spectrum of the analog signal sampled below the Nyquist rate. In this case, the spectral images overlap the original analog signal spectrum making it impossible to recover the original signal. This overlap distortion, or foldover of the higher frequencies into the lower frequency band, is called aliasing.

Spectrum of Sampled Signals (Practical Sampling)

Ideal sampling assumes that an exact value of the analog signal can be captured at precisely the exact point in time when it is needed. In reality, A/D conversion involves quantization error and acquisition time limitations. These practical considerations have the effect of warping the ideal spectrum – in effect the ideal spectrum is multiplied by a sinc function ($\text{sinc}(f/F_s)$). This function was described in Section 2.4.

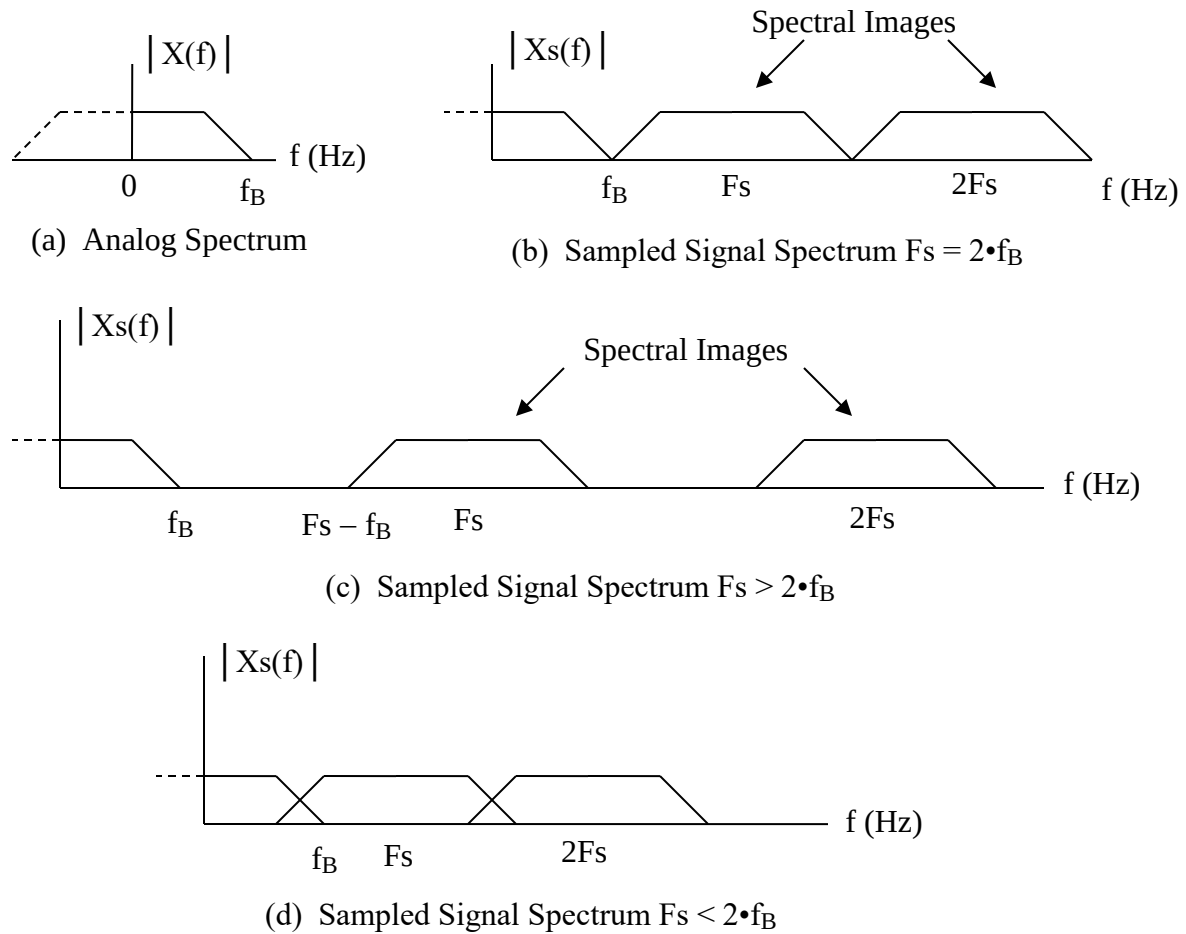


Figure 3.7: Spectra of Ideally Sampled Signals

The effect of non-ideal sampling is illustrated in Figure 3.8. The top plot shows the ideal sampled signal spectrum and is very similar to the signal in Figure 3.7 with $f_B = 1000$ Hz and $F_s = 3000$ Hz. The middle plot shows the sinc function ($|\text{sinc}(f/3000)|$). The non-ideal (practical) sampled signal spectrum is the product of the top two graphs and is shown in the bottom plot. The spectral images are significantly distorted but the original signal spectrum is only slightly distorted. This distortion can be reduced further by increasing the sampling frequency, F_s , several times above the Nyquist rate. As long as the distortion is not significant, a good approximation to the original analog signal can be recovered with a low-pass filter. In fact, it will be shown in Section 3.8 that a well-designed interpolation (reconstruction) filter can reverse the distortion effects.

Challenge Question 3.3

Increasing the sampling frequency above the Nyquist rate will separate the spectral images from the baseband signal and reduce the distortion of the baseband signal. What are some disadvantages of increasing the sampling frequency?

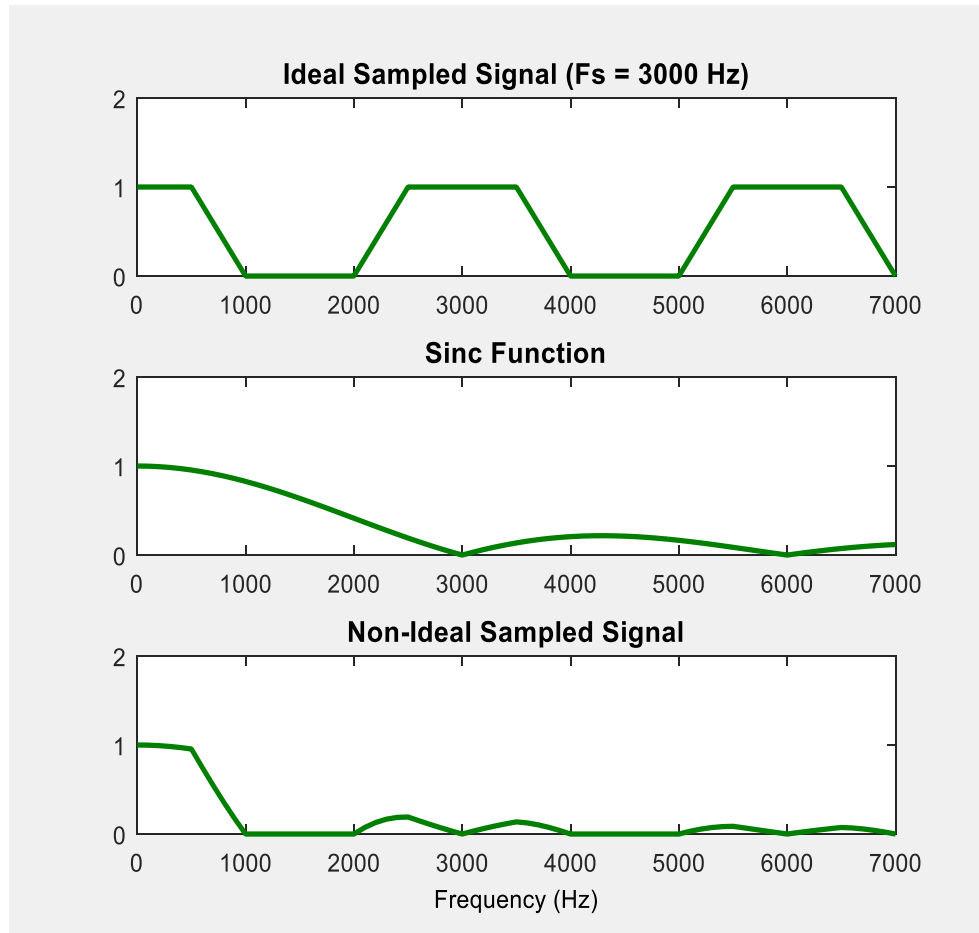


Figure 3.8: Effect of Non-Ideal Sampling

3.5 ALIASING

If a signal is not band-limited or the sampling rate does not exceed the minimum sampling rate (Nyquist rate), then aliasing occurs. Aliasing or foldover distortion was illustrated in the previous section in Figure 3.7. A simple example will further illustrate the concept of aliasing.

Example 3.3: Illustration of Aliasing

Using MATLAB®, three different sine waves, 2 kHz, 8 kHz, and 12 kHz, are sampled at a rate of 10 kHz (10,000 samples/second). According to the Nyquist-Shannon sampling theorem, only the 2 kHz wave is being sampled fast enough. The 8 kHz wave should be sampled above 16 kHz while the 12 kHz wave should be sampled above 24 kHz. The MATLAB statements are shown below and the plots of the sampled signals appear in Figure 3.9.

```
Fs = 10000;           % Set the sampling frequency
t = 0:1/Fs:0.001;     % Time increment must be 1/Fs
y1 = sin(2*pi*2000*t);
y2 = sin(2*pi*8000*t);
y3 = sin(2*pi*12000*t);
```

```
subplot(3,1,1); plot(t,y1); title('2 kHz wave');
subplot(3,1,2); plot(t,y2); title('8 kHz wave');
subplot(3,1,3); plot(t,y3); title('12 kHz wave');
```

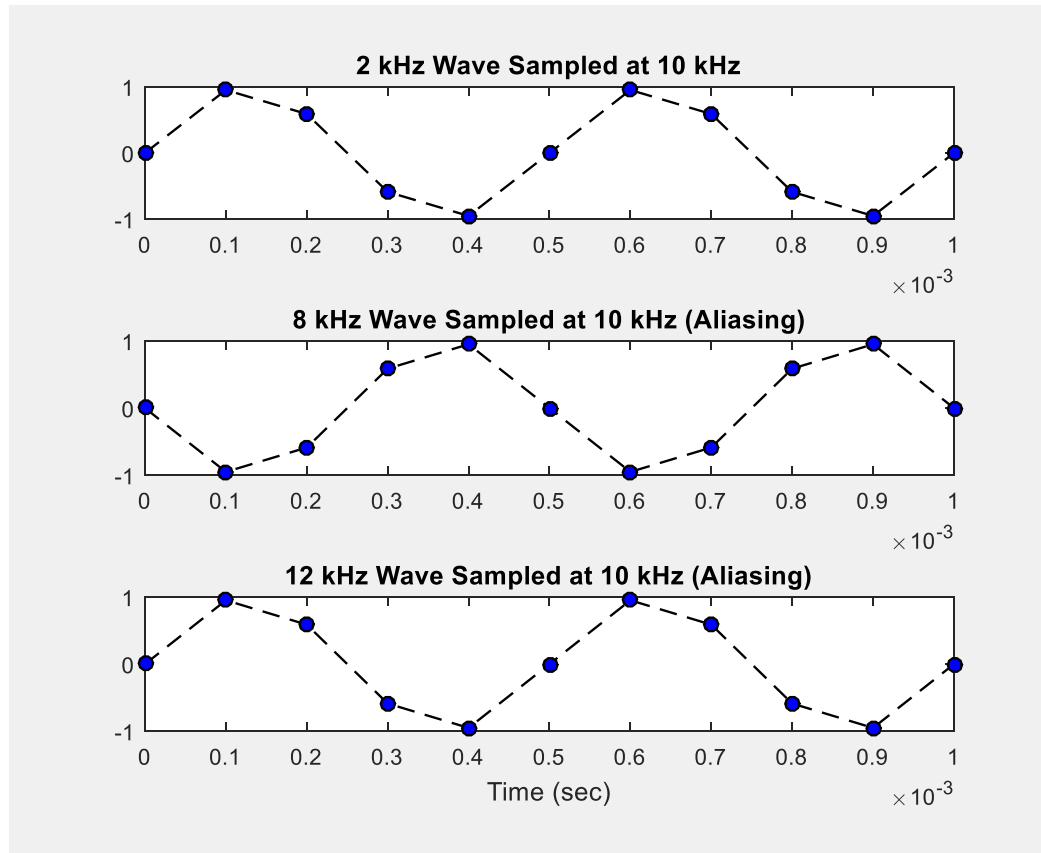


Figure 3.9: Aliasing Example 3.3

What happens? The 2 kHz signal still looks like a 2 kHz sine wave albeit a little jagged. The period of this signal is 0.5 ms which corresponds to a frequency of 2 kHz. The 8 kHz waveform is an inverted version of the 2 kHz wave and the 12 kHz waveform matches the 2 kHz sampled waveform.

The two higher frequency signals are no longer distinguishable from a 2 kHz wave – both signals “alias” as a lower frequency signal. It is not possible to reconstruct the original signals from the sampled values – the signals are simply not being sampled fast enough!

It is useful to look at the two higher frequency signals and see where the samples are being taken. The following MATLAB code generates Figure 3.10.

```
Fs = 10000;           % Set the sampling frequency
t = 0:1/Fs:0.001;     % Time increment must be 1/Fs
y1 = sin(2*pi*2000*t);
y2 = sin(2*pi*8000*t);
y3 = sin(2*pi*12000*t);
```

```

% Use Finer Time Increment for Analog Signals
ta = 0:0.000001:0.001;
y4 = sin(2*pi*2000*ta);
y5 = sin(2*pi*8000*ta);
y6 = sin(2*pi*12000*ta);
subplot(3,1,1); plot(ta,y4,'k-',t,y1,'b. ');
title(' 2 kHz wave sampled at 10 kHz ' )
subplot(3,1,2); plot(ta,y5,'k-',t,y2,'b. ');
title('8 kHz wave sampled at 10 kHz');
subplot(3,1,3); plot(ta,y6,'k-',t,y3,'b. ');
title('12 kHz wave sampled at 10 kHz');

```

As indicated in Figure 3.10, the samples for the 8 kHz and 12 kHz waves are spaced too far apart in time and do not adequately represent the original analog signals. In fact, the samples for these signals trace out a 2 kHz signal.

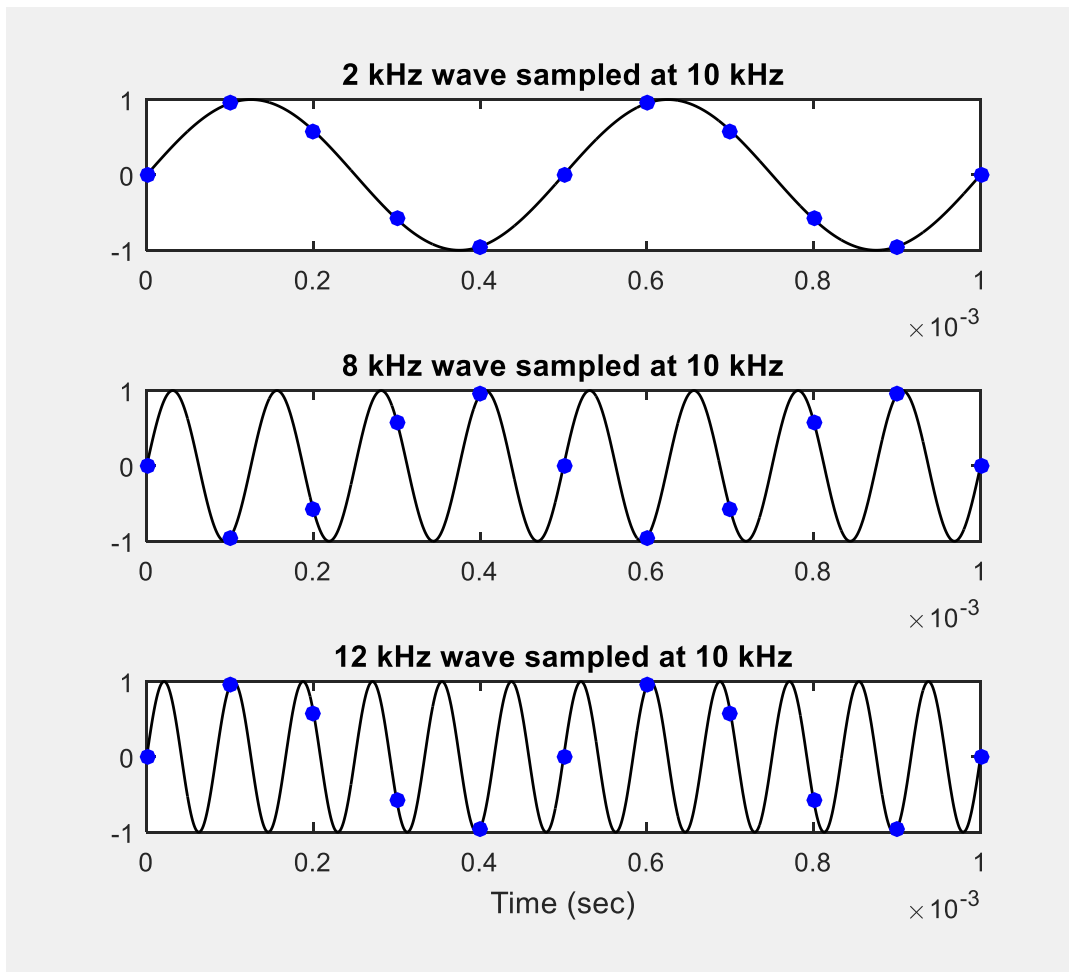


Figure 3.10: Aliasing Example 3.3 (continued)

Example 3.4: Aliasing of Sinusoidal Signals

Suppose a 2 kHz signal (amplitude = 2) and a 12 kHz signal (amplitude = 1) are added together then sampled at 10 kHz. What do you suppose would happen? The answer is illustrated using the MATLAB commands listed below to produce Figure 3.11.

```
Fs = 10000;           % Set the sampling frequency
t = 0:1/Fs:0.001;     % Time increment must be 1/Fs
y = 2*sin(2*pi*2000*t) + sin(2*pi*12000*t);
subplot(2,1,1); plot(t,y,'k-');
title('Signal sampled at 10 kHz')
ta = 0:0.000001:0.001;
y_analog = 2*sin(2*pi*2000*ta) + sin(2*pi*12000*ta);
subplot(2,1,2); plot(ta,y_analog,'k-',t,y,'b.');
```

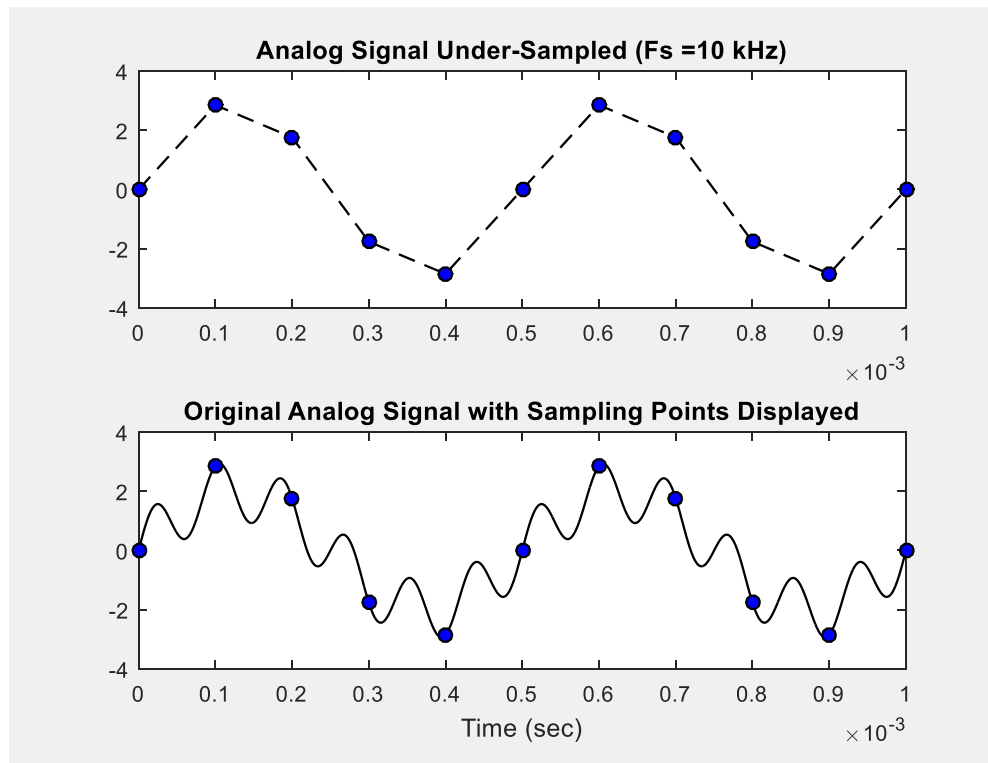


Figure 3.11: Aliasing Example 3.4

What happens? The 12 kHz signal sampled at 10 kHz becomes a 2 kHz signal with amplitude equal to 1. It is added to the 2 kHz signal with amplitude of 2 to produce a 2 kHz signal with amplitude equal to 3.

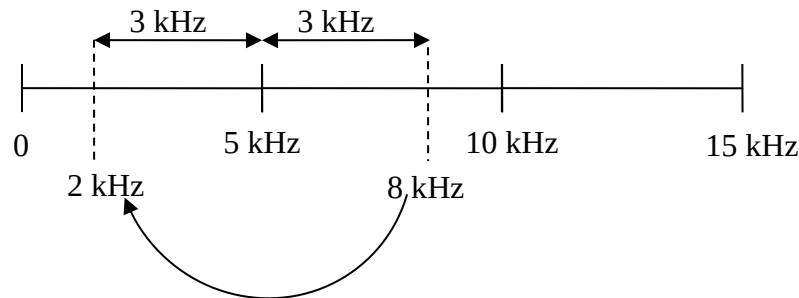
Challenge Question 3.4

What happens if a 2 kHz signal (amplitude =1) and an 8 kHz signal (amplitude = 1) are added together then sampled at 10 kHz?

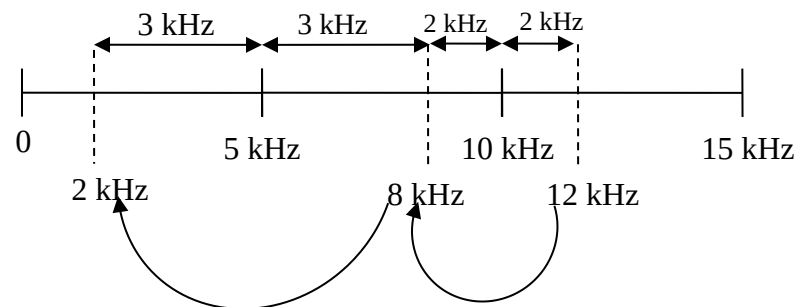
The examples in this section have illustrated that sine waves that are not sampled sufficiently fast will alias as lower frequency sine waves. The alias frequency can be determined by drawing a “number line” with divisions that are multiples of $F_s/2$, then folding the higher frequency signal across the divisions until it fits in the range $[0, F_s/2]$. With each foldover, the signal is inverted. Therefore, an odd number of folds results in an inverted signal and an even number of folds results in a non-inverted signal. Example 3.5 illustrates the process.

Example 3.5: Determining Fold-over Frequency

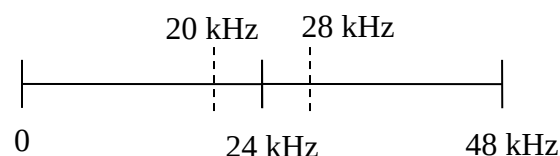
- (a) In Example 3.4, it was demonstrated using MATLAB that an 8 kHz signal sampled at 10 kHz will alias as a 2 kHz (inverted) signal. The alias frequency can also be determined using the number line approach shown in the following diagram. The number line is first divided into sections that are $F_s/2$ or 5 kHz wide. Then the signal frequency is folded down over each division until it falls into the lowest range of 0 to 5 kHz. Since there is only one (odd number) fold, the signal is inverted.



- (b) In Example 3.4, it was shown that a 12 kHz signal sampled at 10 kHz will also alias as a 2 kHz (non-inverted) signal. This can be shown using the same number line approach used in part (a). There are two (even number) folds so the signal is not inverted.



- (c) Now suppose a 28 kHz signal is sampled at 48 kHz. What would the alias frequency be? This time, the number line needs to be divided into sections that are 24 kHz wide ($F_s/2$). As indicated in the diagram below, the signal will alias as a 20 kHz signal. It will be inverted as well since there is only one (odd number) fold.



3.6 DESIGN OF ANTI-ALIASING FILTERS

As shown in the previous section, any analog signal frequencies above $F_s/2$ will alias to a lower frequency potentially corrupting the signal spectrum. An anti-aliasing filter is an analog low-pass filter which band-limits the analog signal to $F_s/2$ prior to sampling to avoid aliasing.

Ideal Filter Characteristics

Figure 3.12 shows a sketch of the magnitude and phase response for an ideal low-pass filter as well as a magnitude response for a non-ideal or practical low-pass filter. An ideal low-pass filter would pass all signal frequencies up to some frequency, f_{pass} , without change and would completely reject all signal frequencies above f_{pass} . The flat magnitude response guarantees that all frequencies in the passband have exactly the same gain (a gain of 1 in the diagram). The linear phase response guarantees that all signal frequencies in the passband experience an identical time delay. Thus any combination of signal frequencies in the passband would appear unchanged at the filter output. Any signal frequencies outside the passband would be completely blocked by the filter and not appear at the filter output.

It is not possible to design an ideal low-pass filter. Figure 3.12 also shows a rough sketch of a magnitude response for a realizable low-pass filter. There is a transition region between the passband and stopband of the filter. Signal frequencies in the transition region would pass through the filter but would be attenuated (reduced). Signal frequencies in the stopband are significantly attenuated by the filter. The passband frequency is often defined as the frequency at which the magnitude is 3dB down from the maximum passband magnitude.

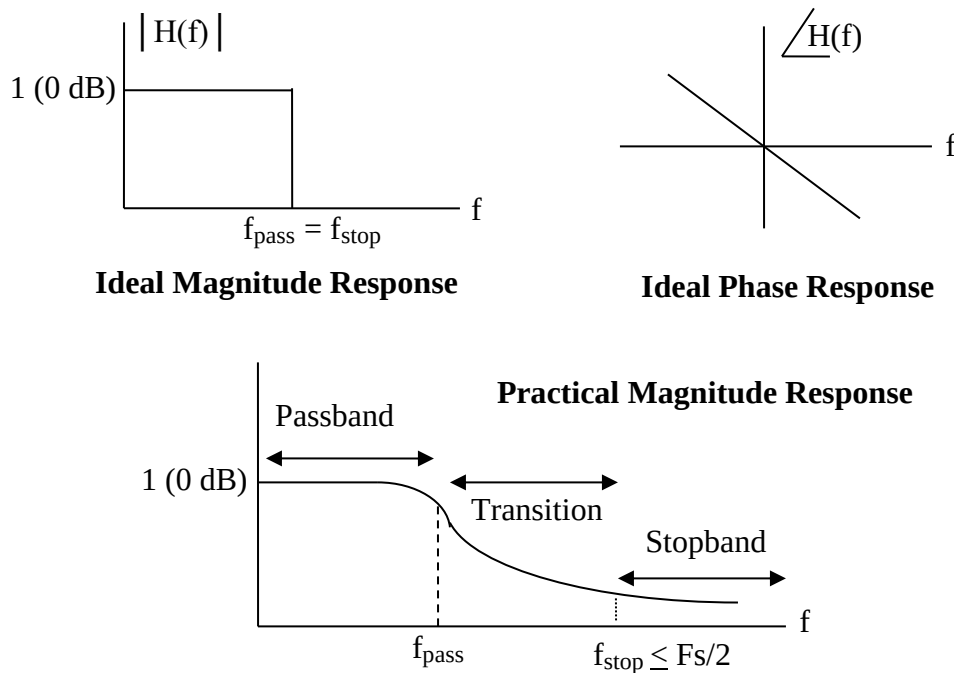


Figure 3.12: Low-Pass Filter Frequency Response

Types of Low-Pass Filters

Three commonly used analog filters are Butterworth, Chebychev, and Bessel. Each type of filter has its own unique characteristics that make it suitable for certain applications. The Butterworth filter is maximally flat in the passband and has a smooth transition into the stopband. The Chebychev filter has a sharp roll-off between the passband and stopband (narrow transition region) but has ripples in the passband or stopband. The Bessel filter has the worst magnitude response of the three (furthest from ideal) but has the most linear phase response.

There are many resources including texts and filter design tables to aid in designing these analog filters. Texas Instruments' WEBENCH[®] Filter Designer, a free on-line program for designing analog filters, can be accessed at this link:

<http://www.ti.com/lscs/ti/analog/webench/webench-filters.page>

The WEBENCH Filter Designer was used to design a circuit for an analog low-pass filter. The filter specifications were: a cutoff frequency of 2000 Hz, a stopband frequency of 8000 Hz, a gain of 1 in the passband, and a maximum passband ripple of 1 dB. The software produces multiple options for types of filters and displays the filter order, the frequency response (magnitude and phase), and the step response for each option. The circuit for one of the options, a 4th order Butterworth low-pass filter which consists of two cascaded 2nd order op-amp sections, is shown in Figure 3.13.

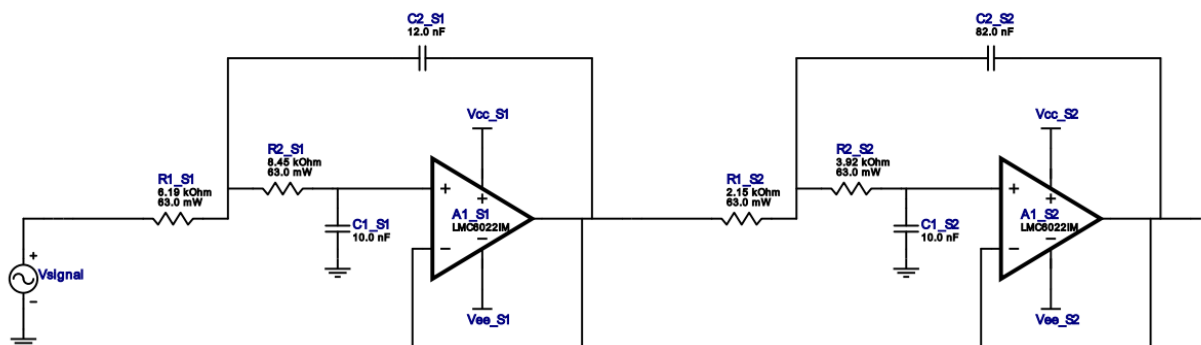


Figure 3.13: 4th Order Low-Pass Sallen Key Butterworth Filter Designed with Texas Instrument WEBENCH Filter Designer

Design Requirements for Anti-aliasing Filter

The design specifications for the low-pass anti-aliasing filter should include the following:

- The filter passband should be wide enough to pass all significant frequency components of the input signal being sampled; that is f_{pass} or f_{cutoff} should be greater than or equal to the signal bandwidth.

- The stopband attenuation for all frequencies $f \geq F_s/2$ should be at least $20 \log(\sqrt{1.5} \cdot 2^B)$. This will ensure that the foldover distortion will be below the level that the B-bit ADC can detect; that is, less than half the step size of the A/D converter.

The following formula is useful for designing Butterworth filters:

$$|H(f)|_{\text{BUTTERWORTH}} = \frac{1}{\sqrt{1 + (f / f_c)^{2N}}} \quad (3.5)$$

$|H(f)|$ = the filter magnitude at frequency f
 f_c = the filter cutoff or passband frequency
 N = the order of the filter (number of filter poles)

Example 3.6: Design an Anti-Aliasing Filter

In a digital signal processing application, audio signals with a bandwidth of 4 kHz will be sampled by an 8-bit ADC at a rate of 16 kHz. Design an anti-aliasing Butterworth low-pass filter for the input audio signals.

Solution:

The passband or cutoff frequency for the filter should be at least 4 kHz (signal bandwidth) to ensure that the signals pass through the filter. Choose $f_c = 4$ kHz.

The stopband attenuation requirement assuming an 8-bit ADC ($B=8$) with a sampling frequency of 16 kHz is given by:

$$20 \log(\sqrt{1.5} \cdot 2^B) = 20 \log(\sqrt{1.5} \cdot 2^8) = 50 \text{ dB} \quad \text{for } f \geq 8 \text{ kHz}$$

The formula for the magnitude of the Butterworth filter can be used to determine the filter order. The 50 dB of attenuation must be converted to straight magnitude of $10^{(-50/20)} = 0.00316228$.

$$|H(f)|_{\text{BUTTERWORTH}} = \frac{1}{\sqrt{1 + (f / f_c)^{2N}}} \leq -50 \text{ dB} \quad \text{for } f \geq 8 \text{ kHz}$$

$$|H(f)|_{\text{BUTTERWORTH}} = \frac{1}{\sqrt{1 + (8\text{kHz} / 4\text{kHz})^{2N}}} \leq 0.00316228$$

$$|H(f)|_{\text{BUTTERWORTH}} = \frac{1}{\sqrt{1 + (2)^{2N}}} \leq 0.00316228$$

$$\sqrt{1 + (2)^{2N}} \geq 1/0.00316228 = 316.22777$$

$$1 + (2)^{2N} \geq 100,000$$

$$(2)^{2N} \geq 99,999$$

$$(2N) \cdot \log(2) \geq \log(99,999)$$

$$N \geq \frac{1}{2} \cdot \log(99,999)/\log(2) = 8.3$$

A 9th order low-pass Butterworth filter with a cutoff frequency of 4 kHz will satisfy the design requirements. Component values can then be determined from filter reference tables or using software such as Texas Instruments' WEBENCH Filter Designer.

In Example 3.6, the required filter order was determined using a formula for the magnitude of a Butterworth filter. The order could also have been determined using a filter design tool and experimenting with order to find the smallest order to satisfy the stopband filter requirements.

3.7 PRACTICAL CONSIDERATIONS

Choosing an Anti-Aliasing Filter Type

Three types of low pass filters, Butterworth, Chebychev, and Bessel, were discussed in the previous section. The best type of anti-aliasing filter to use depends on the application. For audio signals, the signal information is contained in the frequency of the signal. A filter that preserves the frequency information (Butterworth or Chebychev) is therefore the best choice for audio applications.

For video signals, brightness and color vary over space and time. A filter that preserves the shape of the waveform (Bessel) is therefore the best choice for image applications. Figure 3.14 shows the pulse response for an 8-pole Chebychev and an 8-pole Bessel Filter. Notice that the output signal from the Chebychev filter has ringing on both sides. For a video signal, this would affect the brightness of objects – the left side would be brighter than the center while the right side would be darker than the center. In comparison, the Bessel filter, with its linear phase characteristics, produces a smooth symmetrical response with no overshoot.

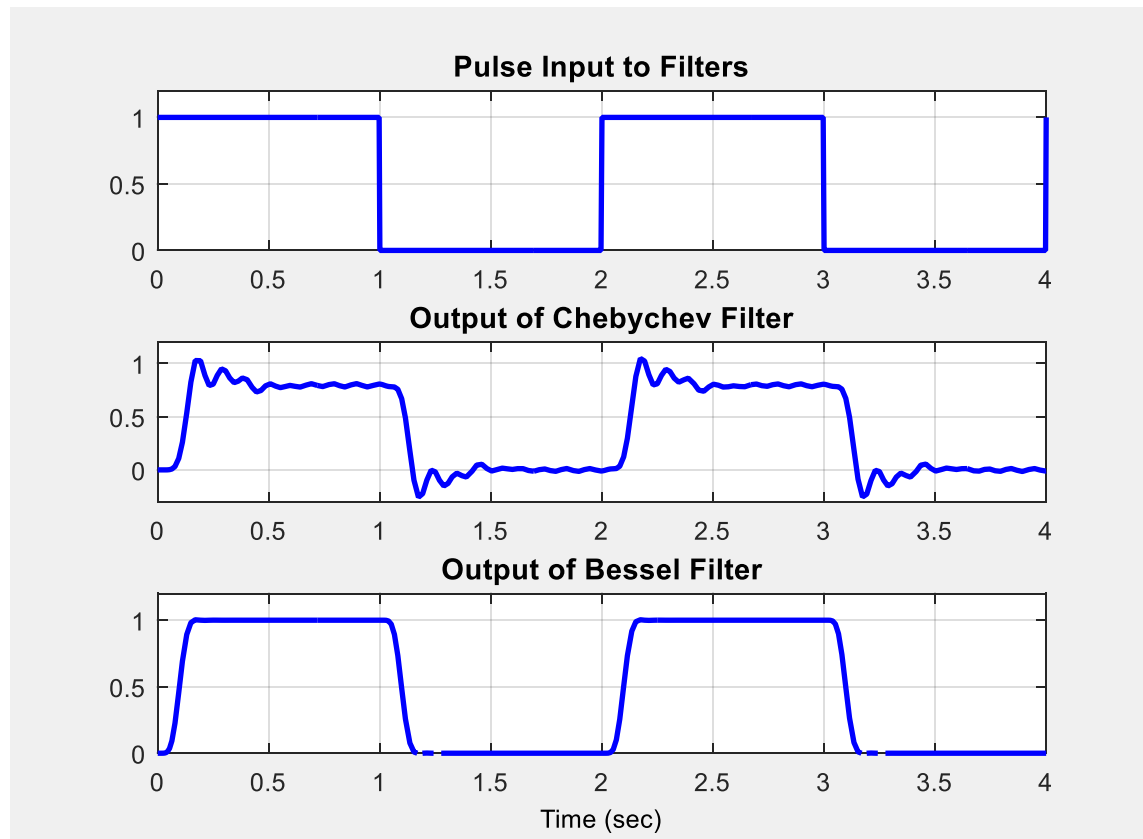


Figure 3.14: Pulse Responses of Low-Pass Filters

Matching the Input Signal to A/D converter

Every A/D converter has an allowable input voltage range. Some ADCs allow only positive input voltages such as 0 to +5V while others allow for both positive and negative input voltages such as -2V to +2V. In order to maximize signal-to-noise ratio (SNR), the input signal should be conditioned to match the voltage range of the ADC.

If the input signal has voltages outside the range of the ADC, these values will be clipped by the ADC or even worse, damage an ADC that doesn't have protection for out of range input voltages. If the input signal range is much smaller than the voltage range of the ADC, then several discrete levels of the ADC will not be used. This effectively reduces the number of bits which in turn reduces the SNR.

Figure 3.15 illustrates the importance of matching the input signal range to the ADC voltage range. Assume that the ADC range is -5V to +5V. In the top graph, the input signal range matches the ADC range and the sampled values reflect the variation in the analog signal. In the middle graph, the input signal exceeds the range of the ADC: the ADC range is -5V to +5V, but the input signal varies from -10V to +10V. This results in clipping of the input signal which causes critical signal information to be lost. In the 3rd plot, the input signal only ranges from -0.1V to +0.1V. Therefore all the ADC discrete levels from +0.1V to +5V and from -0.1V

to -5V are never used (98% of the levels). This effectively reduces the number of bits in the ADC and lowers the SNR.

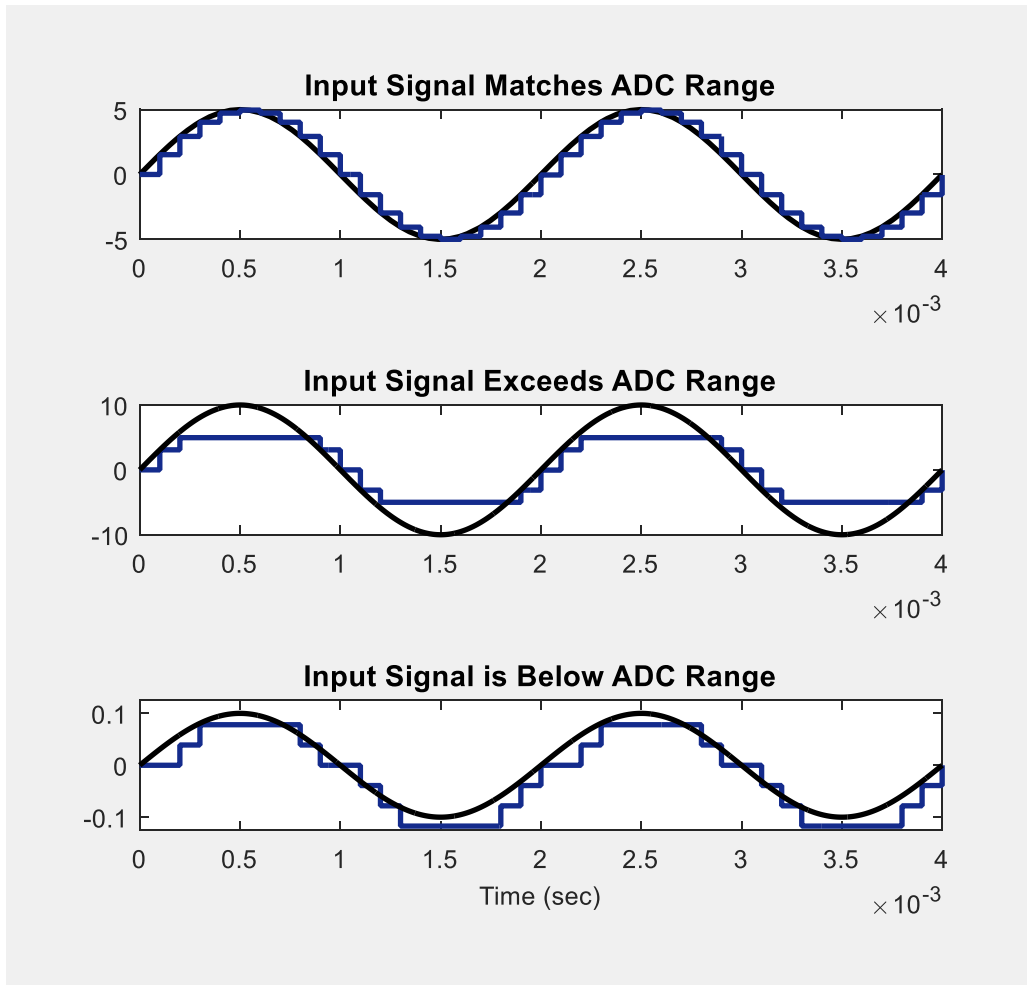


Figure 3.15: Matching Input Signal to ADC

Challenge Question 3.5

A 12-bit ADC with an allowable input voltage range of -5V to $+5\text{V}$ is used to sample an input signal that ranges from -1V to $+1\text{V}$. What is the effective number of bits for this input signal?

Some input signals, such as voice signals, are in a certain voltage range most of the time but on occasion have spikes outside this range. Since these spikes must be covered by the ADC full-swing voltage, the majority of the signal is using only a very small range of the ADC. As discussed previously, this effectively reduces the SNR. Companding is a technique used to modify or condition a signal with occasional spikes. The signal is passed through a logarithmic amplifier which linearizes the signal – spreading out the lower voltage range and flattening the higher voltage range. The digital equivalent of this is an uneven spacing between discrete levels (non-uniform step size). At lower voltage ranges, the discrete levels would be closer together

(smaller step size) while in the higher voltage ranges, discrete levels would be further apart. Obviously, passing a signal through a logarithmic amplifier will distort the signal. So at the receiver or signal recovery end, the signal must be passed through an inverse function to cancel the distortion introduced at the front end. Compandors are used in the telephone system, concert audio systems, wireless microphones, and in noise reduction schemes such as DBX and Dolby C.

Oversampling to Simplify Analog Anti-Aliasing Filter

In some applications, the analog anti-aliasing filter design requirements can result in an unreasonably high order analog filter. In cases such as these, it is much more practical to reduce the complexity of the analog filter by sampling at a higher rate (oversampling), then reducing the sampling rate digitally.

Challenge Question 3.6

An 8-bit ADC has a sampling frequency of 8 kHz. The input signal is an audio signal with a bandwidth of 3.4 kHz. What order would the anti-aliasing Butterworth filter need to be? If an 8-bit ADC with a sampling frequency of 48 kHz were used instead, what order would the anti-aliasing Butterworth filter need to be?

The Butterworth filter in the Challenge Question 3.6 for the 8-bit ADC sampling at 8 kHz would have to be 36th order – way too large for an analog filter! An 8-bit ADC sampling at 48 kHz would only require a 3rd order anti-aliasing Butterworth filter which is much more practical. In order to reduce the sampling rate to 8 kHz, the ADC would be followed by a digital low-pass filter with a cutoff frequency of 3.4 kHz and an attenuation of at least 50 dB for all frequencies higher than 4 kHz. High order digital filters are easy to implement and will be discussed in Chapters 5 and 6. The output signal from the digital filter would be decimated by a factor of 6 (48 kHz / 8 kHz) to reduce the sampling rate to 8 kHz prior to processing.

3.8 RECONSTRUCTION

This section discusses how analog signals can be recovered or reconstructed from sampled values. Figures 3.7 and 3.8 illustrated the spectrum of a sampled signal. Assuming the sampling rate is sufficiently high (above the Nyquist rate), the spectral images will be separated from the original signal and the original signal can be recovered using a low-pass filter. Figure 3.12 showed the spectrum for an ideal low-pass filter. As discussed in Section 3.6, it is not possible to design an ideal low-pass filter. There is a transition phase between the passband and stopband and phase is not necessarily linear. So, the reconstruction filter will be at best an approximation to an ideal low-pass filter. Examples of practical reconstruction filters include step interpolation filters, linear interpolation filters, sinc interpolation filters, and raised cosine interpolation filters.

Step Interpolation Filter (Zero Order Hold)

A step interpolation filter or zero order hold, simply holds a signal constant until the next sample value becomes available, then steps up or down to the next value. This results in the stair-step type analog signal shown in Figure 3.16. This is the only interpolation scheme which can be implemented in real time (no time delay), but it doesn't produce a good quality analog signal unless the sampling rate is really fast.

Linear Interpolation Filter

A linear interpolation filter simply connects adjacent sample values with a straight line producing the piecewise linear analog signal shown in Figure 3.16. The linear interpolation filter has a time delay of one sampling instant, T_s . In order to create the analog signal between $t = 0$ and $t = T_s$, we need to wait for the sample value at T_s . In general, linear interpolation does not produce a good quality analog signal unless the sampling rate is really fast.

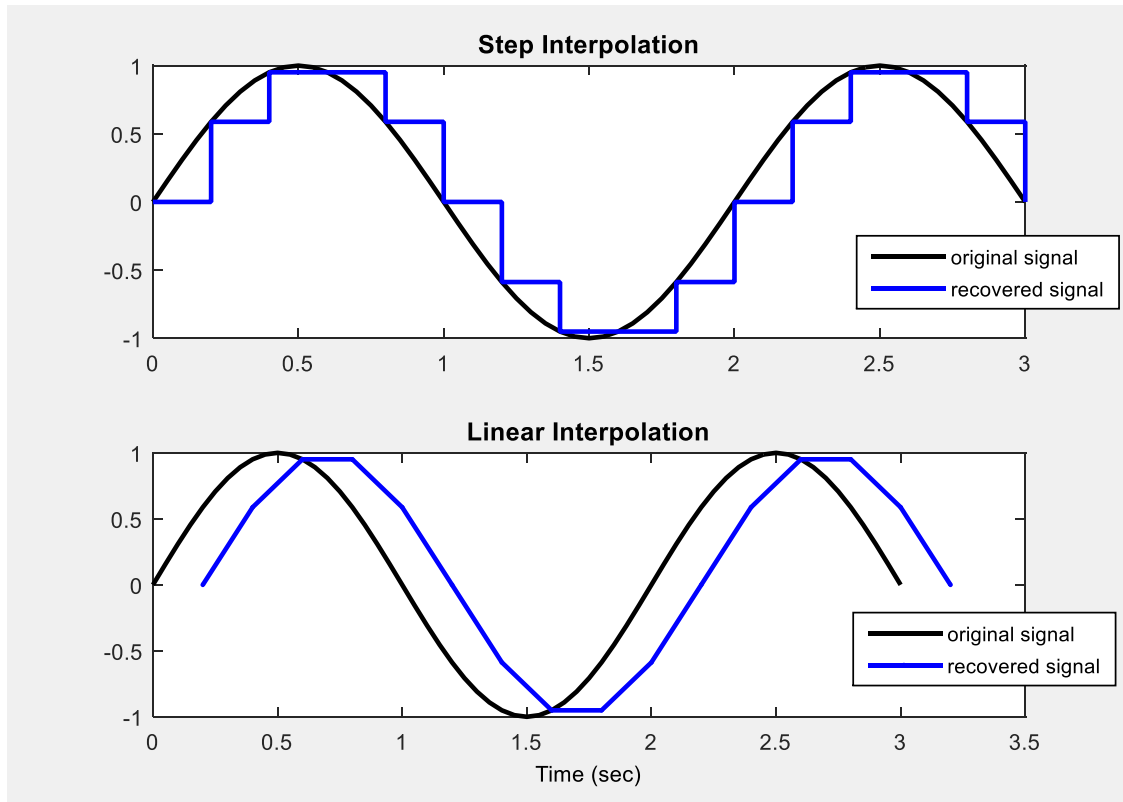


Figure 3.16: Step Interpolation and Linear Interpolation

Sinc Interpolation Filter

The sinc function was defined in Section 2.4 as $\text{sinc}(t) = \sin(\pi t) / \pi t$. A sinc interpolation filter takes a finite number, N , of adjacent signal sample values, multiplies these sample values by N values of the $\text{sinc}(t/T_s)$ function, then adds the results of the N multiplications to create an analog signal. The time delay associated with the sinc interpolation filter is $N \cdot T_s$. It will be shown in Chapter 5 that the higher the value of N , the closer the sinc interpolation filter will approximate an ideal low-pass filter. So, a higher value of N results in a better quality analog signal, but will also require more processing and introduce a longer time delay. The sinc function is shown in Figure 3.17 along with the result of using a sinc interpolation filter to reconstruct a sine wave from sampled values. Sinc filters will be explored in much greater detail in Chapter 5.

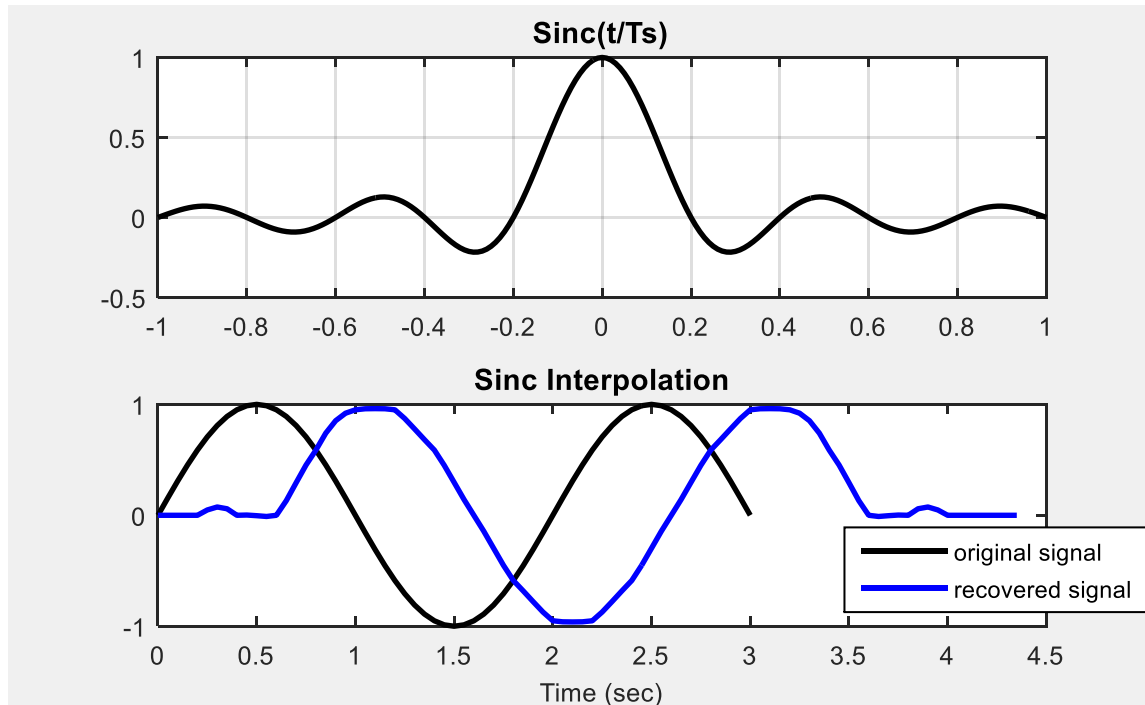


Figure 3.17: Sinc Interpolation

Raised Cosine Interpolation Filter

A raised cosine interpolation filter is similar to sinc interpolation except the roll-off is faster so fewer sample values (smaller N) are needed for good interpolation. This results in a shorter time delay. The raised cosine function is defined as

$$\text{sinc}(t/T_s) \cdot \frac{\cos(\pi R t/T_s)}{[1 - (2R t/T_s)^2]} \quad (3.6)$$

The parameter R is the roll-off factor and can be chosen as any value from 0 to 1. Notice that if $R = 0$, then the function becomes the $\text{sinc}(t/T_s)$ function. The raised cosine function is shown in Figure 3.18 along with the result of using a raised cosine interpolation filter to reconstruct a sine wave from sampled values. Notice the improvement in quality of the reconstructed analog signal when compared to the results of the sinc interpolation filter of the same order shown in Figure 3.17.

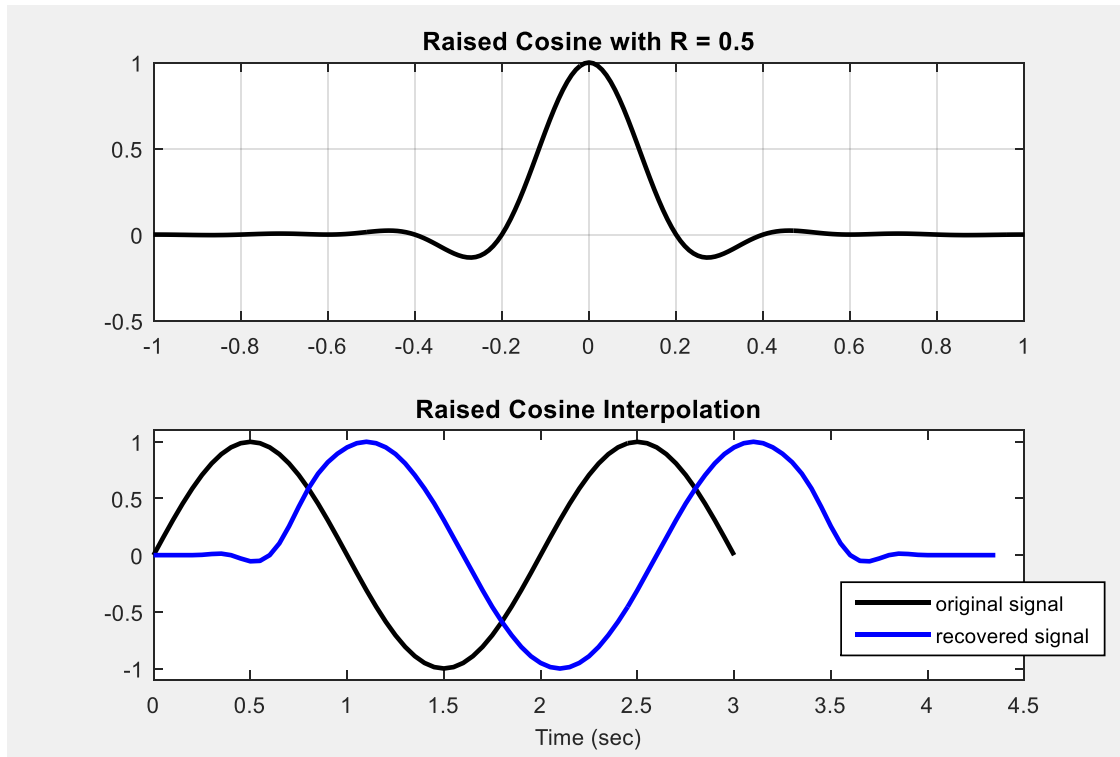


Figure 3.18: Raised Cosine Interpolation

3.9 PRACTICAL APPLICATION FOR UNDER-SAMPLING

If the sample rate is below twice the highest signal frequency, the signal frequencies will fold-over to lower frequencies. This fold-over effect is used to great advantage in software defined radio. A general explanation is included here. In an analog radio receiver, FM or AM signals are shifted down to an intermediate (IF) frequency using non-linear mixer circuits. After additional filtering, the signal is shifted down to a baseband signal prior to demodulation. This same effect is accomplished digitally by undersampling. Suppose we have an FM signal with a bandwidth of 0.2 MHz centered at 100.1 MHz and the signal is sampled at 29.8 MHz. Notice that the sampling rate is above the Nyquist rate (twice the signal bandwidth is 0.4 MHz) but significantly below the actual signal frequencies (undersampling). Where does the signal end up? Using the fold-over number line approach outlined in Section 3.5, the FM signal will end up at the IF frequency of 10.7 MHz where it can be filtered to eliminate adjacent FM signals, shifted to baseband, and then demodulated.

Answers to Chapter 3 Challenge Questions

Question 3.1 How could the quantization error in an A/D converter be reduced? Are there any negative consequences to this change?

The quantization error could be decreased by using an A/D converter with more bits. This would result in more discrete levels spaced closer together (smaller step size). Increasing the number of bits required for an A/D converter will in general increase the cost of the converter and could also increase conversion time.

Question 3.2 I am thinking of a number from 1 to 64. You are to guess this number as quickly as possible. Each time you guess, I will only tell you whether your guess is high or low. How would you play the game to minimize the number of guesses you would have to make? If you play it smart, what is the most number of guesses you would ever have to make?

The best first guess would be 32 (halfway between 1 and 64). If I were told 32 was too low, then I would guess 48 (halfway between 32 and 64). If I were told 32 was too high, I would guess 16 (halfway between 1 and 32). If I continued this pattern of cutting the remaining range in half, I would arrive at the correct number using at most six guesses. To arrive at 6 guesses as being the most required, just follow through any scenario where your guesses are wrong till it is not possible to be wrong. For example, guess 1 is 32 (HIGH), so guess 2 is 48 (LOW), so guess 3 is 56 (HIGH), so guess 4 is 52 (LOW), so guess 5 is 54 (LOW), so guess 6 is 55 (CORRECT).

Aside: $2^6 = 64$. If I wanted to know the maximum number of guesses to correctly guess a number from 1 to 1000, I would find the smallest number N that satisfied $2^N \geq 1000$. So the maximum number of guesses for a number from 1 to 1000 would be 11. If you really enjoy number games, you could determine the set of numbers that would require the maximum number of guesses to get to!

Question 3.3 Increasing the sampling frequency above the Nyquist rate will separate the spectral images from the baseband signal and reduce the distortion of the baseband signal. What are some disadvantages of increasing the sampling frequency?

Increasing the sampling frequency could result in a more expensive A/D converter. Also, any processing done on the incoming signal sample (filtering, correlation, etc.) must be completed within one sampling interval. Increasing the sampling frequency (decreasing the sampling interval) could result in a more expensive digital signal processor.

Question 3.4 What happens if a 2 kHz signal (amplitude =1) and an 8 kHz signal (amplitude = 1) are added together then sampled at 10 kHz?

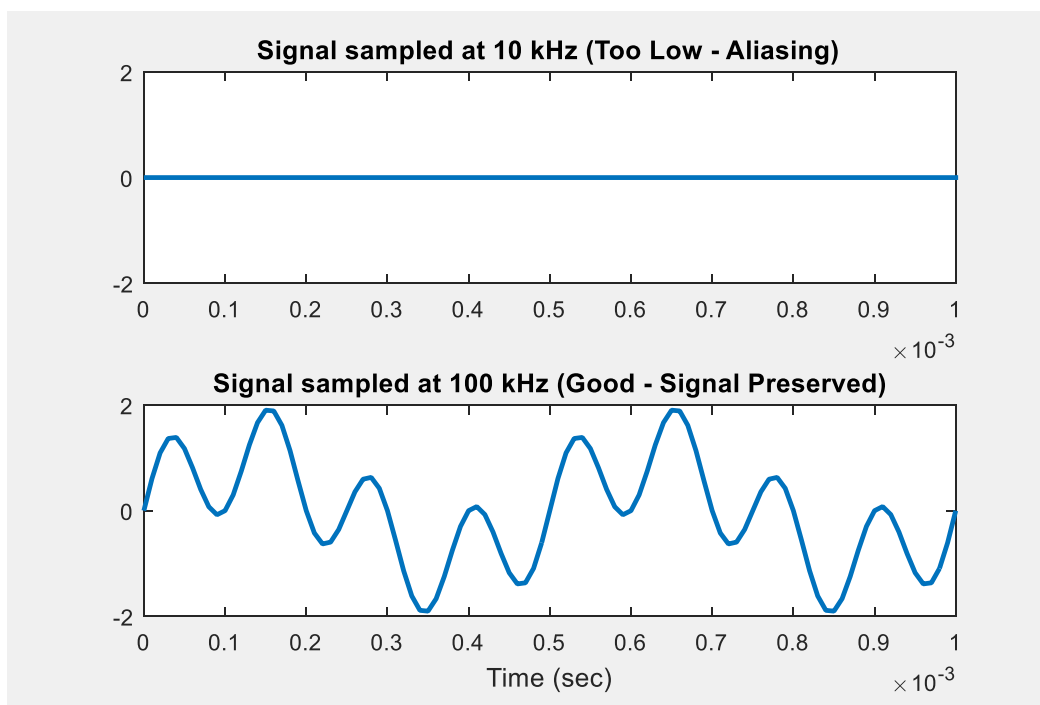
Looking back at Example 3.2, an 8 kHz signal sampled at 10 kHz becomes an inverted 2 kHz sine wave. If both waves have the same amplitude, then they should cancel and produce no output. This can be verified in MATLAB with the following code:

```

Fs = 10000;           % Sampling frequency - Too Low!
t = 0:1/Fs:0.001;    % Time increment must be 1/Fs
y = sin(2*pi*2000*t) + sin(2*pi*8000*t);
subplot(2,1,1); plot(t,y); ylim([-2 2]);
title('Signal sampled at 10 kHz (Too Low - Aliasing)');

% Choose A Higher Sampling Frequency - Greater than 2*8000
New_Fs = 100000;
t = 0:1/New_Fs:0.001; %Time increment must be 1/New_Fs
y_analog = sin(2*pi*2000*t)+sin(2*pi*8000*t);
subplot(2,1,2); plot(t,y_analog);
title('Signal sampled at 100 kHz (Good - Signal Preserved)');
xlabel('Time (sec)')

```



Question 3.5 A 12-bit ADC with an allowable input voltage range of -5V to $+5\text{V}$ is used to sample an input signal that ranges from -1V to $+1\text{V}$. What is the effective number of bits for this input signal?

A 12-bit ADC has $2^{12} = 4096$ discrete levels. The input signal voltage swing is 2V (-1V to 1V) which is only 20% of the ADC full-swing voltage of 10V . Therefore the input signal uses only 20% of the discrete levels or 820 discrete levels. The effective number of bits for this input signal can be calculated by solving $2^B \geq 820$ for B . The effective number of bits is 10. Recall from Section 3.2 the SNR (Best) = $6.02B + 1.76$ (dB). So the SNR will be reduced by about 12.04 dB (actually reduced a bit more because of rounding for B).

Question 3.6 An 8-bit ADC has a sampling frequency of 8 kHz. The input signal is an audio signal with a bandwidth of 3.4 kHz. What order would the anti-aliasing Butterworth filter need to be? If an 8-bit ADC with a sampling frequency of 48 kHz were used instead, what order would the anti-aliasing Butterworth filter need to be?

The design requirements for the 8-bit ADC sampling at 8 kHz are:

- Cutoff frequency = 3.4 kHz
- Attenuation of $20 \log(\sqrt{1.5} \cdot 2^B) = 20 \log(\sqrt{1.5} \cdot 2^8) = 50 \text{ dB}$ for all $f \geq F_s/2$ (4 kHz)

$$|H(f)|_{\text{BUTTERWORTH}} = \frac{1}{\sqrt{1+(f/f_c)^{2N}}} \leq -50 \text{ dB} \quad \text{for } f \geq 4 \text{ kHz}$$

$$|H(f)|_{\text{BUTTERWORTH}} = \frac{1}{\sqrt{1+(4\text{kHz}/3.4\text{kHz})^{2N}}} \leq 0.00316228$$

$$|H(f)|_{\text{BUTTERWORTH}} = \frac{1}{\sqrt{1+(1.1765)^{2N}}} \leq 0.00316228$$

$$\sqrt{1+(1.1765)^{2N}} \geq 1/0.00316228 = 316.22777$$

$$1 + (1.1765)^{2N} \geq 100,000$$

$$(1.1765)^{2N} \geq 99,999$$

$$(2N) \cdot \log(1.1765) \geq \log(99,999)$$

$$N \geq \frac{1}{2} \cdot \log(99,999)/\log(1.1765) = 35.4$$

The Butterworth filter would need to be 36th order – totally unreasonable for an analog filter because it would require eighteen 2nd order op-amp sections! A Chebychev filter would be smaller than 36th order but still unreasonably complicated.

The design requirements for an 8-bit ADC sampling at 48 kHz would be a cutoff frequency of 3.4 kHz and an attenuation of $20 \log(\sqrt{1.5} \cdot 2^B) = 20 \log(\sqrt{1.5} \cdot 2^8) = 50 \text{ dB}$ for all $f \geq F_s/2$ (24 kHz). Repeating the calculations above using $f = 24 \text{ kHz}$ rather than 4 kHz will yield a filter order $N = 3$ which is certainly realizable.

Chapter 3 Problems

Problem 3.1: A 12 bit A/D converter has a sampling frequency of 44.1 kHz, accepts analog input voltages ranging from 0V to +5V, and rounds to the nearest discrete level. Calculate the sampling interval, T_s , the number of discrete levels, L , the step size, q , the maximum quantization error, and the SNR (Best) for this A/D converter.

Problem 3.2: A 12 bit successive approximation A/D converter has a clock rate of 96 kHz.

- (a) How much time is required to convert each analog sample to a 12 bit digital code?
- (b) What is the maximum bandwidth of the incoming signal for this A/D converter?

Problem 3.3: A Sigma-Delta A/D converter has an input voltage range of 0 to 3V. Assuming the integrator is initially equal to zero, create a table similar to Table 3.3 to determine to output bit pattern for the following voltage levels:

- (a) 1.0 V
- (b) 2.25 V
- (c) 2.5 V

Problem 3.4:

An analog signal is sampled at a rate of 48 kHz. Complete the following table showing the frequency components of the sampled signal based on the frequency components of the incoming signal.

Frequency Components Of Input Signal (kHz)	Frequency Components Of Sampled Signal (kHz)
15	
30	
15, 30, 60	

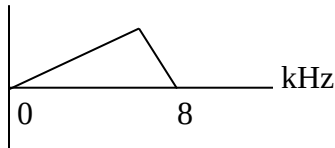
Problem 3.5:

Using MATLAB, plot each of the following signals using the given sampling rate. In each case, find the frequency(ies) of the sampled signal and state whether aliasing has occurred or not.

- (a) $\sin(2\pi(3000)t)$ at $F_s = 8$ kHz
- (b) $\sin(2\pi(6000)t)$ at $F_s = 8$ kHz
- (c) $\sin(2\pi(3000)t) + 3\sin(2\pi(11000)t)$ at $F_s = 8$ kHz
- (d) $\sin(2\pi(3000)t) + 3\sin(2\pi(11000)t)$ at $F_s = 48$ kHz
- (e) $\sin(2\pi(3000)t) \cdot 3\sin(2\pi(11000)t)$ at $F_s = 8$ kHz
- (f) $\sin(2\pi(3000)t) \cdot 3\sin(2\pi(11000)t)$ at $F_s = 24$ kHz

Problem 3.6:

- (a) Assuming a sampling rate of 44.1 kHz and the signal spectrum shown below, sketch the spectrum of the sampled signal (assume ideal sampling).



- (b) Repeat part (a) for a sampling rate of 12 kHz.
 (c) What is the minimum sampling frequency required for this signal to avoid aliasing?

Problem 3.7: A 12 bit A/D converter has a voltage range of 0V to +5V and a sampling frequency of 44 kHz.

- (a) Find the minimum order for an anti-aliasing Butterworth filter with a cutoff frequency of 8 kHz.
 (b) Roughly sketch the frequency response of the anti-aliasing filter including passband region, stopband region, and expected stopband attenuation.
 (c) If the input signal range is only 0 to +1.5V, calculate the effective number of bits for the A/D converter.

Problem 3.8: An 8 bit A/D converter has a voltage range of -2V to +2V and a sampling frequency of 16 KHz.

- (a) Find the minimum order for an anti-aliasing Butterworth filter with a cutoff frequency of 3.4 kHz.
 (b) Sketch the frequency response for the Butterworth low-pass filter including passband region, stopband region, and expected stopband attenuation.
 (c) Use Texas Instrument's FilterPro to design Butterworth and Chebychev anti-aliasing filters to meet the given specifications. How do the two filters compare in terms of magnitude response and filter order? What is the attenuation of each filter at the stopband frequency of 8 kHz?

Problem 3.9:

- (a) Why is the anti-aliasing filter designed with a minimum stopband attenuation of $20\log\sqrt{1.5} * 2^B$? In other words, what does this formula guarantee?
 (b) Suppose the analog anti-aliasing filter needed for a DSP application has an unreasonably high order. Describe an alternative.

CHAPTER 3 LAB EXERCISE

Sampling, Aliasing, and Reconstruction

Objectives

1. To view signal sampling and reconstruction in the time domain.
2. To illustrate the concepts of aliasing and quantization error.

Background

The sampling theorem states that an analog signal can be reconstructed from its sampled values if the sampling frequency exceeds twice the highest frequency component present in the analog signal (Nyquist frequency).

Considerations:

- Many signals are not band-limited but can be made so by passing the signal through a low-pass (anti-aliasing) filter. Filtering will result in a loss of high frequency signal information.
- A/D converters round signal sample values to the nearest quantization level which introduces quantization or round-off error.
- Perfect reconstruction of an analog signal requires exact sample values (non-existent due to round-off errors) and an ideal low-pass filter (also non-existent).

Some Key Formulas:

- Sampling Frequency: $F_s > 2F_B$ (F_B is highest frequency of analog signal)
- Number of Quantization Levels: 2^B (B is number of bits for A/D converter)
- Quantization Level (Step Size): $V_{FS} / (2^B - 1)$

Procedure

A. Build the Simulink® model shown in Figure 1. Double click on each of the blocks listed below and adjust settings where necessary.

1. Signal Generator (Located in Sources Library): Sine wave, Amplitude = 2, Frequency = $2\pi \cdot 100$ rad/sec or 100 Hz.
2. Zero-Order Hold (Located in Discrete Library): Sample Time = 1/1000 ($F_s = 1000$ Hz)
3. Analog Filter (Located in Signal Processing blockset under filtering): Filter Type = Butterworth, Edge Frequency = $F_s/2 = 500$ Hz = $2\pi \cdot 500$ rad/sec, leave everything else in default values.
4. Multiplexer (Located in Signal Routing Library)
5. Simulation Parameters: Set End Time = 5/100 (Displays 5 cycles of 100 Hz waveform)

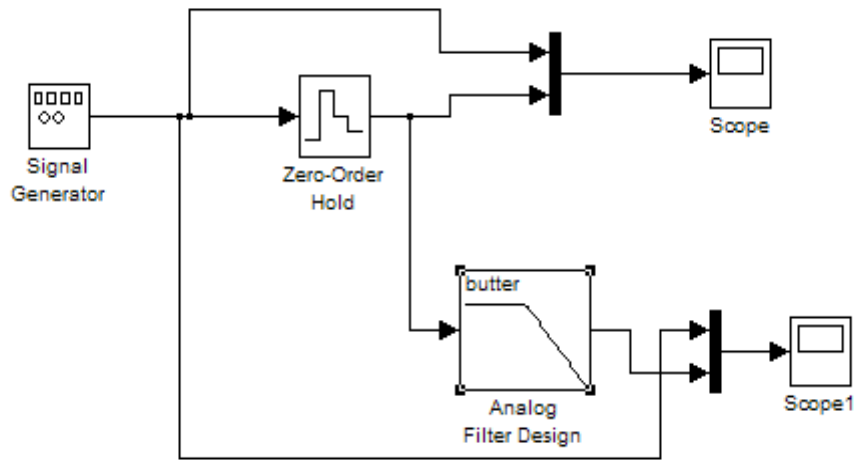


Figure 1: Sampling Analog Waveforms

B. Effect of Sampling Frequency

Run the simulation and view the scope outputs. Since the sine wave is being sampled suitably fast and the bit resolution is sufficient, the sine wave will be reconstructed nicely from the sampled values. The time delay in reconstruction occurs because the filter uses a finite number of past inputs and outputs (depends on filter order) to compute the next filter output.

Notice that the output of the low-pass filter is considerably smoother than the input signal to the low-pass filter. The low-pass filter acts as an interpolation filter: smoothing the data between samples. Interpolation will be investigated further in Section G.

Now, vary the sampling frequency (set in the Zero Order Hold) and comment on the effects on the reconstructed signal. ***Remember to change the edge frequency ($F_s/2$) of the low-pass filter each time you change the sampling frequency!*** In particular, comment on the effects of reducing the sampling frequency below the Nyquist rate (200 Hz for the 100 Hz sine wave). Print a few graphs to show effects of sampling rate.

C. Aliasing

Set the sampling frequency back to 1000 Hz and the edge frequency of the low-pass filter to $2\pi \cdot 500$. What should be the maximum frequency of the incoming signal?

Maximum frequency of input signal _____

Double click on the signal generator and set the frequency to 600 Hz (or $2\pi \cdot 600$ rad/sec). Set the final time under Simulation Parameters to 10/600. Record your observations about the frequency of the reconstructed signal in the table below. Then complete the table using the other input frequencies indicated.

Input Frequency	Output Frequency
600 Hz	
800 Hz	
1000 Hz	
1200 Hz	

D. Sampling Signals which are not band-limited

Change the Signal Generator to produce a 200 Hz square wave with amplitude of 2. The sampling time of the ZOH should be set to 1/1000 ($F_s = 1000$ Hz). The edge frequency of the low-pass filter should be 500 Hz. Set the final time under Simulation Parameters back to 5/200. Run the simulation and record your observations below.

What's going on?

A 200 Hz square wave has frequency components at 200 Hz, 600 Hz, 1000 Hz, 1400 Hz, ... (the fundamental frequency and odd harmonics) The higher frequencies produce the sharp angles on the square wave. These higher frequencies are lost in the sampling process – they alias as lower frequencies.

Increase the sampling frequency (ZOH) remembering to also adjust the edge frequency of the filter to $F_s/2$. What effect does the increase in sampling frequency have on the reconstructed signal? How much do you have to increase the sampling frequency to get a decent square wave?

E. Effect of Quantization Level

1. Modify the Simulink model to match that in Figure 2. The Encoder and Decoder are located in the Signal Processing Blockset in the Quantizer folder. For each set Peak = 2 , Bits = 8.
2. Change the sampling frequency back to 1000 Hz in the ZOH and change the filter edge frequency back to $2\pi \cdot 500$ rad/sec. **Change the signal generator back to a 100 Hz sine wave with an amplitude of 2.**
3. Run the model and observe the scope outputs. Since the input signal peak matches the DAC peaks and the signal is being sampled sufficiently fast, the results should look quite nice.
4. Now change the amplitude of the signal from the signal generator to 10. What do you observe? Why?
5. Now change the amplitude of the signal from the signal generator to 0.1. What do you observe? Why?
6. Set the amplitude of the incoming signal back to 2.0. Vary the number of bits. Observations?

Comment: You have just seen it is important to match the ADC to the actual peak-to-peak variation in an analog signal. Some analog signals will remain within a certain range most of the time with occasional spikes out of range. To maintain good precision at the lower signal level without losing the occasional high signal spike, companding can be used. The public telephone system in the U.S. uses μ -law companding to compress the higher levels in a signal and leave the lower levels virtually unchanged. The signal is then expanded at the receiver end using the inverse of the companding function.

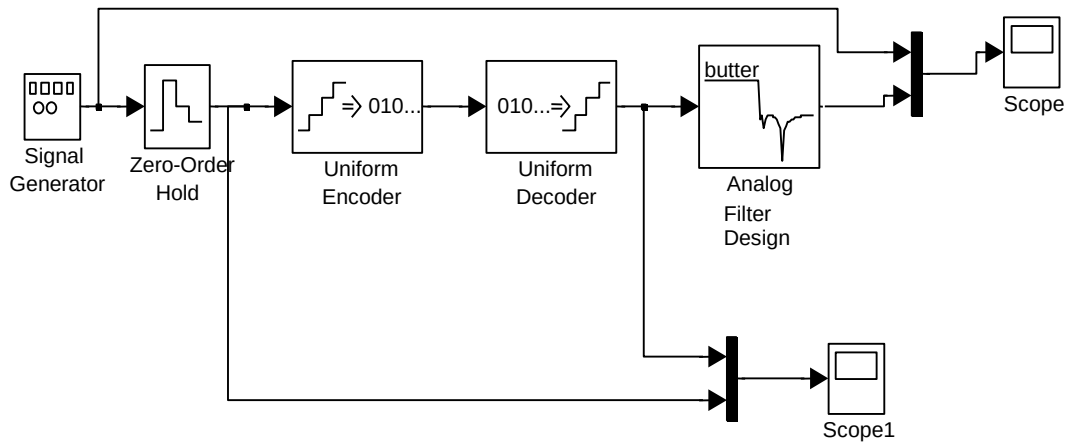


Figure 2: Simulink Model of Signal Sampling, Encoding, and Reconstruction

G. Interpolation and Decimation

In some applications, it is necessary to change the sampling rate of digital data. For example:

- Digital data may be stored at a sampling rate that is lower than that at which it will be played (audio) or displayed (image).
- It may be necessary to interface between two systems that operate at different sampling frequencies.

Interpolation – the process of adding data points between adjacent samples of a digital signal thus increasing the effective sampling rate.

Decimation – the process of removing sample points from a digital signal thus decreasing the effective sampling rate (and the number of samples to be stored).

MATLAB Functions

NewSignal = interp(signal,R);

Inserts data points between adjacent samples of signal. NewSignal will be R times as long as signal and have a sampling rate R times that of signal.

NewSignal = decimate(signal,R);

Produces (through filtering) one sample for every R original samples of signal. NewSignal will be 1/R times as long as signal and have a sampling rate 1/R times that of signal.

1. Connect the speaker to the computer.
2. Load and play a snippet of Handel's Hallelujah Chorus using the following MATLAB commands:

```
load handel;  
%Loads a snippet of chorus in vector y using Fs = 8192 Hz  
sound(y, Fs)
```

3. Suppose we have a soundboard that samples at a fixed rate of $2 \cdot F_s$. Attempt to play the samples at this new frequency using the command

```
sound(y, 2*Fs)
```

Explain what happens.

4. Interpolation can be used to match the original signal, y , to the new sampling rate. Try the following set of statements:

```
y_int = interp(y, 2);  
sound(y_int, 2*Fs)
```

Observations?

Length of vector y = _____ Length of vector y_{int} = _____

5. The original signal is sampled at 8192 Hz. This is pretty low for a music signal. Lower it even more through decimation as follows:

```
y_dec = decimate(y, 3);  
sound(y_dec, Fs/3)
```

Observations?

Length of vector y = _____ Length of vector y_{dec} = _____

This chapter provides the mathematical background needed for the DSP applications covered in the rest of the text. Topics covered include fixed-point and floating-point number systems and arithmetic operations, difference equations, Z-Transforms, convolution and correlation.

4.1 FIXED POINT REPRESENTATION

A fixed-point processor uses an integer number system for data and arithmetic operations. Common formats for representing integers are unsigned integers, offset binary, sign-magnitude, and twos complement. These formats are summarized for 4-bit numbers in Table 4.1.

Table 4.1: Integer Number Systems

4-BIT NUMBER	UNSIGNED INTEGER	OFFSET BINARY	SIGN- MAGNITUDE	TWOS COMPLEMENT
0000	0	-8	0	0
0001	1	-7	1	1
0010	2	-6	2	2
0011	3	-5	3	3
0100	4	-4	4	4
0101	5	-3	5	5
0110	6	-2	6	6
0111	7	-1	7	7
1000	8	0	0	-8
1001	9	1	-1	-7
1010	10	2	-2	-6
1011	11	3	-3	-5
1100	12	4	-4	-4
1101	13	5	-5	-3
1110	14	6	-6	-2
1111	15	7	-7	-1

Unsigned integer is a straight binary representation with all bits used as magnitude bits. This system does not allow for negative numbers. Offset binary simply shifts the values to allow for both positive and negative numbers. In the sign-magnitude representation, the most significant bit (MSB) is used as a sign bit (0=positive and 1=negative) with the remaining bits used for magnitude (straight binary). The twos complement representation also uses the MSB as a sign bit (0=positive and 1=negative) and the remaining bits to represent numeric value but it is not a straight binary representation. Twos complement lends itself very well to arithmetic operations and is therefore the most common format for fixed-point processors.

Twos Complement Format

One nice way to think about twos complement is as follows. Look at Table 4.1 and imagine starting at 0 (0000). If we add 1, we count up one row to 0001 or +1. If we add 2 to 0, we count up two rows to 0010 or +2. Suppose we start at 0 (0000) and subtract 1. In the twos complement system, we count down one row (circle back to the bottom of the table) to 1111 or -1. If we subtract 2 from 0, we count down two rows back to 1110 or -2. The procedure for converting between integer and twos complement format is outlined below and illustrated in Example 4.1.

Integer format → Twos complement format:

- For positive numbers, the sign bit (MSB) is set to 0 and the remaining bits are a straight binary representation of the number.
- For negative numbers, represent the absolute value of the number in twos complement, then invert all the bits and add 1 to the result.

Twos Complement format → Integer format:

- If MSB = 0, the number is positive. Simply convert the magnitude bits to a number using straight binary conversion.
- If MSB = 1, the number is negative. Use straight binary conversion except put a negative in front of the MSB value.

Example 4.1: Converting Between Twos Complement Format and Decimal Format

(a) Find the 16-bit twos complement representation for +95.

$$95 = 64 + 16 + 8 + 4 + 2 + 1$$

Setting the MSB to 0 (positive number) and using the remaining 15 bits to represent the magnitude gives

$$95 = 0000\ 0000\ 0101\ 1111 = 005FH \text{ (Hexadecimal)}$$

(b) Find the 16-bit twos complement representation for -95.

First represent the absolute value of the number which is +95. This was done in part (a)

$$+95 = 0000\ 0000\ 0101\ 1111$$

Invert all the bits then add 1:

$$\begin{array}{r} 1111\ 1111\ 1010\ 0000 \\ + \quad \quad \quad \quad \quad \quad 1 \\ \hline 1111\ 1111\ 1010\ 0001 \end{array}$$

$$-95 = 1111\ 1111\ 1010\ 0001 = FFA1H \text{ (Hexadecimal)}$$

(c) What number does 0010 0000 1100 0101 represent assuming twos complement?

Since the MSB = 0, the number is positive. The remaining bits are converted to magnitude using straight binary.

$$0010\ 0000\ 1100\ 0101 = 2^{13} + 2^7 + 2^6 + 2^2 + 2^0 = 8192 + 128 + 64 + 4 + 1 = 8389$$

(d) What number does 1000 1100 1101 1110 represent assuming twos complement?

Since the MSB = 1, the number is negative. Use straight binary except put a negative in front of the MSB.

$$\begin{aligned} 1000\ 1100\ 1101\ 1110 &= -2^{15} + 2^{11} + 2^{10} + 2^7 + 2^6 + 2^4 + 2^3 + 2^2 + 2^1 \\ &= -32768 + 2048 + 1024 + 128 + 64 + 16 + 8 + 4 + 2 \\ &= -29474 \end{aligned}$$

Challenge Question 4.1

What is the range of numbers (negative to positive) that can be represented using the 16-bit twos complement format?

Arithmetic Operations

To understand why twos complement works so well for arithmetic operations, consider the following addition problems:

$$\begin{array}{r} 48 \\ + 95 \\ \hline \end{array} \qquad \begin{array}{r} -37 \\ + -5 \\ \hline \end{array} \qquad \begin{array}{r} -37 \\ + 5 \\ \hline \end{array} \qquad \begin{array}{r} -48 \\ + 95 \\ \hline \end{array}$$

To do these problems by hand (no calculator), you would look at each problem and first figure out what the sign of the answer would be. In the first problem, both numbers are positive so the answer would be positive. In the second problem, both numbers are negative which would result in a negative answer. In the third problem, the negative number has the largest magnitude so the answer would be negative. Finally, in the fourth problem, the answer would be positive because the positive number has the largest magnitude. Once a decision about sign has been made, magnitude is determined in the first two problems by simply adding the two magnitudes. Magnitude is determined in the last two problems by subtracting the smaller magnitude from the larger. A processor using a sign-magnitude representation would be required to use this same decision making and comparison process to complete an addition operation. Addition using an offset binary format would be considerably more complicated.

The twos complement representation does not require the decision making and comparison process described above. Numbers are added using straight binary addition regardless of sign. This is illustrated in the following example.

Example 4.2: Arithmetic Operations using Twos Complement Format

Using the twos complement representation, complete the following arithmetic operations.

$$\begin{array}{llll} \text{(a)} & 48 & \text{(b)} & -37 \\ & + \underline{95} & & + \underline{-5} \\ & & \text{(c)} & -37 \\ & & & + \underline{5} \\ & & \text{(d)} & -48 \\ & & & + \underline{95} \end{array}$$

Solution:

$$\begin{array}{l} \text{(a)} \quad 48 = 0000\ 0000\ 0011\ 0000 \\ \quad 95 = \underline{0000\ 0000\ 0101\ 1111} \\ \quad 0000\ 0000\ 1000\ 1111 = 2^7 + 2^3 + 2^2 + 2^1 + 2^0 = 128 + 8 + 4 + 2 + 1 = 143 \end{array}$$

$$\begin{array}{l} \text{(b)} \quad -37 = 1111\ 1111\ 1101\ 1011 \\ \quad -5 = \underline{1111\ 1111\ 1111\ 1011} \\ \quad 1111\ 1111\ 1101\ 0110 = -2^{15} + 2^{14} + 2^{13} + 2^{12} + 2^{11} + 2^{10} + 2^9 + 2^8 + 2^7 + 2^6 + 2^4 + 2^2 + 2^1 \\ \quad = -42 \end{array}$$

$$\begin{array}{l} \text{(c)} \quad -37 = 1111\ 1111\ 1101\ 1011 \\ \quad 5 = \underline{0000\ 0000\ 0000\ 0101} \\ \quad 1111\ 1111\ 1110\ 0000 = -2^{15} + 2^{14} + 2^{13} + 2^{12} + 2^{11} + 2^{10} + 2^9 + 2^8 + 2^7 + 2^6 + 2^5 \\ \quad = -32 \end{array}$$

$$\begin{array}{l} \text{(d)} \quad -48 = 1111\ 1111\ 1101\ 0000 \\ \quad 95 = \underline{0000\ 0000\ 0101\ 1111} \\ \quad 0000\ 0000\ 0010\ 1111 = 2^5 + 2^3 + 2^2 + 2^1 + 2^0 = 32 + 8 + 4 + 2 + 1 = 47 \end{array}$$

Note: the overflow bit occurring in parts (b) and (d) is simply discarded.

What about other arithmetic operations? For subtraction, the number being subtracted is inverted (invert all bits and add 1) then added to the other number. For multiplication, straight binary multiplication is performed. However, when two 16-bit numbers are multiplied together, the result is a 32-bit number which must then be reduced to 16 bits (more on this later).

Range and Arithmetic Overflow

The range of a B-bit twos complement system is -2^{B-1} to $+2^{B-1} - 1$. For a 16-bit system, the range would be from -32768 to $+32767$. The following example illustrates what happens when this range is exceeded in an arithmetic operation.

Example 4.3: Overflow in Twos Complement

Using the 16-bit twos complement representation, add 18000 and 20000.

$$18000 = 0100\ 0110\ 0101\ 0000$$

$$20000 = \underline{0100\ 1110\ 0010\ 0000}$$

$$1001\ 0100\ 0111\ 0000 = -2^{15} + 2^{12} + 2^{10} + 2^6 + 2^5 + 2^4 = -27536$$

The sum of 18000 and 20000 exceeds the range of the 16-bit twos complement representation by 5233 (38000 – 32767). The addition operation wraps the result to $-32768 + 5232 = -27536$ resulting in an answer that is way off from the correct answer. Overflow could also easily occur with multiplication operations.

Arithmetic overflow is a serious concern for fixed-point processors and must be avoided. There are several methods for eliminating or minimizing overflow that will be discussed throughout this text.

Fractional Representation (Q15 Format)

Scaling is an effective method for eliminating or minimizing arithmetic overflow in fixed-point operations. In a fractional representation, all numbers are scaled to be between -1 and $+1$. If numbers between -1 and $+1$ are multiplied together, the answer will always lie between -1 and $+1$ so overflow will never occur in multiplication operations using the fractional representation. However, when numbers between -1 and $+1$ are added together, the result can be outside the range of -1 to $+1$ so addition operations still need to be treated carefully, possibly using additional scaling operations. Methods for scaling digital filter coefficients to minimize or prevent arithmetic overflow are covered in Chapters 5 and 6.

The Q15 format is a 16-bit fractional representation and is a very common format for fixed-point DSP processors. In Q15 format, the MSB is still a sign bit (0=positive and 1=negative) but the remaining bits represent negative powers of 2 as shown in Table 4.2.

Table 4.2: Q15 Format

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Power	Sign 0=Pos. 1=Neg.	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}	2^{-13}	2^{-14}	2^{-15}

Fraction between -1 and $+1 \rightarrow$ Q15 format:

- For positive fractions, set MSB = 0 (positive). The remaining 15 bits are a straight binary representation of the fraction multiplied by 2^{15} .
- For negative fractions, represent the absolute value of the fraction as described above for positive fractions then invert all the bits and add 1.

Q15 format $\rightarrow$ Fraction between -1 and $+1$:

- If MSB = 0, the number is positive. Simply convert the magnitude bits to a number using straight binary conversion. Then divide the result by 2^{15} . Table 4.2 could also be used for the conversion.
- If MSB = 1, the number is negative. Use straight binary conversion except put a negative in front of the MSB value. Then divide the result by 2^{15} . Table 4.2 could also be used for the conversion if -1 is substituted for the MSB.

Example 4.4: Converting between Q15 Format and Decimal Format

(a) Convert 0.75 to Q15 format.

- Set MSB = 0 (positive)
- $0.75 \cdot 2^{15} = 24576$ Converting to 15 bit straight binary $\rightarrow 110\ 0000\ 0000\ 0000$
- $0.75 = 0110\ 0000\ 0000\ 0000 = 6000H$ in Q15 format

Shortcut: $0.75 = 0.5 + 0.25 = 2^{-1} + 2^{-2} \rightarrow 0110\ 0000\ 0000\ 0000$ using Table 4.2. Of course, this only works if you can easily express the number as the sum of negative powers of 2.

(b) Convert -0.75 to Q15 format.

- Represent $+0.75$ in Q15 $\rightarrow 0110\ 0000\ 0000\ 0000$
- Invert all the bits and add 1:

$$\begin{array}{r} 1001\ 1111\ 1111\ 1111 \\ + \quad \quad \quad \quad \quad 1 \\ \hline 1010\ 0000\ 0000\ 0000 \end{array}$$
- $-0.75 = 1010\ 0000\ 0000\ 0000 = A000H$ in Q15 format

(c) What fraction does 0010 1100 0000 0000 represent using Q15 format?

- $0010\ 1100\ 0000\ 0000 = 2^{13} + 2^{11} + 2^{10} = 8192 + 2048 + 1024 = 11264$
- Dividing by $2^{15} \rightarrow 0.34375$

Alternative using Table 4.2:

$$0010\ 1100\ 0000\ 0000 = 2^{-2} + 2^{-4} + 2^{-5} = 0.25 + 0.0625 + 0.03125 = 0.34375$$

(d) What fraction does 1101 0101 1000 0000 represent using Q15 format?

- $1101\ 0101\ 1000\ 0000 = -2^{15} + 2^{14} + 2^{12} + 2^{10} + 2^8 + 2^7$
 $= -32768 + 16384 + 4096 + 1024 + 256 + 128 = -10880$
- Dividing by $2^{15} \rightarrow -0.3320312$

Alternative using Table 4.2

$$\begin{aligned} 1101\ 0101\ 1000\ 0000 &= -1 + 2^{-1} + 2^{-3} + 2^{-5} + 2^{-7} + 2^{-8} \\ &= -0.3320312 \end{aligned}$$

Can 0.1 be represented exactly using Q15 format? If not, how close can we get?

Multiplication in Q15

Example 4.5: Multiplication in Q15 Format

The Q15 representation for 0.75 was determined in Example 4.4 and the Q15 representation of 0.1 was determined in Challenge Question 4.2.

[illegible]
$$0000100110011001 = (2^{11} + 2^8 + 2^7 + 2^4 + 2^3 + 2^0) / 2^{15} = 0.074981689453125$$

The actual product of 0.1 and 0.75 is 0.075. What are the sources of error in this computation? There are two sources of error. The first is the fact that 0.1 cannot be represented exactly in Q15 format as indicated in Challenge Question 4.2. The second is that only 16 bits of the 32 bit result will be retained. Looking at the 32-bit product, there are two bits of value 1 that are dropped off in the process of reducing the result to a 16-bit answer which introduces additional rounding error.

It is important to note that while these errors seem very small, digital signal processing applications are often computationally intensive involving lots of arithmetic operations. A very small round-off error in each computation can lead to a very significant error after hundreds of thousands of such operations. Round-off error in arithmetic operations is a critical design consideration for DSP applications.

4.2 FLOATING POINT REPRESENTATION

A floating-point processor uses a decimal (real) number system for data and arithmetic operations. Numbers are represented in a form similar to scientific notation except powers of 2 are used in place of powers of 10. Floating-point format can vary from processor to processor; however, once you understand one format, it is not difficult to learn another format. This section will cover the IEEE 754 standard for 32-bit single precision floating-point numbers.

IEEE 754 32-bit Single Precision Floating-point Format

In the IEEE 754 standard, the MSB is the sign bit (0=positive and 1=negative), the next 8 bits are exponent bits, and the remaining 23 bits are mantissa bits as shown in Table 4.3.

Table 4.3: IEEE 754 32-bit Single Precision Floating-Point Format

SIGN BIT	8 EXPONENT BITS	23 MANTISSA BITS
Bit 31 (MSB)	Bits 30 29 ... 23	Bits 22 21 ... 1 0

Straight binary format is used for the exponent bits while the mantissa bits are in Q23 format (same as Q15 except the powers of 2 go from 2^{-1} to 2^{-23}). The relationship between the decimal format and the floating-point format is described in Equation 4.1:

$$\text{Decimal Number} = (-1)^S \cdot M \cdot 2^{(E - 127)} \quad (4.1)$$

S = Sign Bit (0=positive and 1 = negative)

M = $1 + (\text{Integer value for 23 mantissa bits} / 2^{23})$

E = integer value for 8 exponent bits

The procedure for converting between decimal format and floating point format is outlined then illustrated in Example 4.6.

Decimal Number → Floating-point format:

- Determine the sign bit (0=positive and 1=negative).
- Divide the decimal number by the largest power of 2 that does not exceed the decimal number.
- E = the power of 2 from the previous step plus 127. Convert E to 8 exponent bits.
- M = the result of the division operation in the second step. Subtract 1 from M and convert the result to Q23 format (i.e., multiply by 2^{23} and convert to 23 mantissa bits).
- Put the sign bit, exponent bits, and mantissa bits together to create the 32-bit value.

Floating-point Format → Decimal Number:

- Determine the sign of the number from the MSB (0=positive and 1=negative).
- Determine the integer value for E using straight binary to integer conversion.
- Determine the value of $M = 1 + (\text{Integer value for 23 mantissa bits} / 2^{23})$.
- Plug into the formula: $\text{Decimal Number} = (-1)^S \cdot M \cdot 2^{(E - 127)}$.

Example 4.6: Converting between Floating-Point Format and Decimal Format

(a) Convert 72 to IEEE 754 32-bit floating-point format.

- $S = 0$
- The largest power of 2 that does not exceed 72 is $2^6 = 64$. $72/64 = 1.125$
- $E = 6 + 127 = 133 = 1000\ 0101$
- $M = 1.125$, so $M - 1 = 0.125$ and $0.125 \cdot 2^{23} = 1048576 = 001\ 0000\ 0000\ 0000\ 0000\ 0000$
- $72 = 01000010100100000000000000000000 = 42900000H$

Note: Don't convert to hex until the sign bit, 8 exponent bits, and 23 mantissa bits have been combined.

(b) Convert -0.40625 to IEEE 754 32-bit floating-point format.

- $S = 1$
- The largest power of 2 that does not exceed 0.40625 is $2^{-2} = 0.25$. $0.40625/0.25 = 1.625$
- $E = -2 + 127 = 125 = 0111\ 1101$
- $M = 1.625$, so $M - 1 = 0.625$ and $0.625 \cdot 2^{23} = 5242880 = 101\ 0000\ 0000\ 0000\ 0000\ 0000$
- $-0.40625 = 10111101101000000000000000000000 = BED00000H$

(c) What decimal number does DF520000H represent assuming IEEE 754 32-bit single precision floating-point format?

- $DF520000H = 10111110010100100000000000000000$
- $S = 1 \rightarrow$ number is negative
- Exponent bits = $0111\ 1100 \rightarrow E = 124$
- Mantissa Bits = $101001000000000000000000 = 5373952$
- $M = 1 + (5373952 / 2^{23}) = 1.640625$
- $\text{Decimal Value} = (-1)^S \cdot M \cdot 2^{(E - 127)} = (-1)^1 \cdot 1.640625 \cdot 2^{(124 - 127)} = -0.205078125$

Challenge Question 4.3

Is it possible to represent 0.1 exactly in IEEE 754 32-bit single precision floating-point format? If not, how close can we get?

Range and Conventions of IEEE Standard

Using the definition for the IEEE 754 standard in the previous subsection, the largest possible 32-bit floating point numbers would be $\pm 6.8 \cdot 10^{38}$ and the smallest floating point numbers would be $\pm 5.4 \cdot 10^{-39}$. Note that there would be no exact representation for 0 which is a real problem since 0 is a fairly useful number. The IEEE 754 standard adds a number of conventions to handle very small numbers (including an exact representation for 0) and other computational issues.

Some of these conventions are:

- ± 0 is represented by all zeros in the exponent bits and all zeros in the mantissa bits
- $\pm \infty$ is represented by all ones in the exponent bits and all zeros in the mantissa bits
- NaN (Not A Number) produced for example by attempting to divide by zero is represented by all ones in the exponent and anything besides all zeros in the mantissa bits.
- De-normalized numbers (used to represent very small numbers) have all zeros in the exponent bits and an implied 0. (instead of 1.) for the fraction piece.

The added conventions result in a range of approximately 10^{-45} to 10^{38} . Clearly, the dynamic range of a floating-point processor is much greater than that of a fixed-point processor.

Some floating-point considerations:

- All integers between ± 16.8 million can be represented exactly.
- Fractions that can be expressed as negative powers of 2 or linear combinations of negative powers of 2 from $(2^{-1}$ to $2^{-23})$ can be represented exactly.
- The spacing between consecutive floating-point numbers is not uniform; the gap between adjacent numbers gets larger as numbers become bigger. Fixed-point numbers are uniformly spaced over their dynamic range.
- Unlike fixed-point systems, it is difficult (but not impossible) to exceed the dynamic range of a floating-point system.
- Round-off error is a design issue in floating-point systems just as it is in fixed-point systems. Round-off error can occur when using numbers that cannot be represented exactly in floating-point format (like 0.1) and as a result of arithmetic operations. If the error tends to be random (both positive and negative), then very little drift occurs. However, if the error tends to be additive (predominantly positive or predominantly negative), then significant drift can occur with multiple arithmetic operations.

4.3 FIXED POINT vs. FLOATING POINT PROCESSORS

Choosing between a floating-point processor and a fixed-point processor is very application dependent. Floating-point processors have greater dynamic range than fixed-point processors and therefore tend to require less design effort since overflow is normally not an issue with floating-point operations and values can be represented with greater accuracy. On the other hand, floating-point processors tend to be somewhat more expensive, consume more power, and are slower than fixed-point processors because of more functionality (complexity) and wider data buses (32-bit vs. 16-bit).

In choosing a processor type, it is important to consider the type of signal (data) being processed, the number and type of arithmetic operations being performed, the speed at which data must be processed, and how the processed signal will be used. Fixed-point processors work well for video applications where the pixel data is in integer format and the compression algorithms utilize DCTs (discrete-cosine transforms) which can be handled effectively using integer operations. Floating-point processors work well for high fidelity audio applications which require intensive filtering, often involving very large FFTs (fast-Fourier transforms). The human ear is more difficult to deceive than the human eye, so output signal quality specifications are higher for audio signals than for video signals.

4.4 DISCRETE SYSTEMS

A linear discrete system processes a digital input signal linearly to produce a digital output signal as illustrated in Figure 4.1.



Figure 4.1: Discrete System

Input/Output Difference Equation

The process can be described (modeled) using an input/output difference equation:

$$\begin{aligned} y_1(k) + a_1y(k-1) + a_2y(k-2) + \cdots + a_Ny(k-N) \\ = b_0x(k) + b_1x(k-1) + b_2x(k-2) + \cdots + b_Lx(k-L) \end{aligned} \quad (4.2)$$

The **order of the discrete system** is the maximum of N and L ; that is, the largest delay of the input or output signal.

Solving for $y(k)$ gives:

$$\begin{aligned} y_1(k) = -a_1y(k-1) - a_2y(k-2) - \cdots - a_Ny(k-N) + b_0x(k) + \\ b_1x(k-1) + b_2x(k-2) + \cdots + b_Lx(k-L) \end{aligned} \quad (4.3)$$

The output, $y(k)$, is a linear combination of past outputs ($y(k-1)$, $y(k-2)$, ...), the current input ($x(k)$), and past inputs ($x(k-1)$, $x(k-2)$, ...). The processor computes $y(k)$ by storing the numerical coefficients ($a_1, a_2, \dots, b_0, b_1, b_2, \dots$), and N past output and input values. Each time a new input value comes in, a new output value is computed. The newest input and output values are stored in memory for the next computation while the oldest input and output values are discarded. The output value is computed using multiplication and addition operations typically referred to as multiply and accumulate (MAC). Multiply and accumulate operations are used extensively in DSP applications including filtering, correlation, and fast-Fourier transforms.

Example 4.7: Computing the Output Signal Recursively

A discrete system is modeled by the difference equation

$$y(k) + 0.2y(k-1) + 0.5y(k-2) = x(k) - x(k-1)$$

Find the output signal for an input signal $x(0) = 1$, $x(1) = 3$, $x(k) = 0$ for $k \geq 2$. Assume both the output and input signal are zero for $k < 0$; that is, the system is initially at rest.

Solution:

Solving the equation for $y(k)$ yields

$$y(k) = -0.2y(k-1) - 0.5y(k-2) + x(k) - x(k-1)$$

Plugging in values for k starting at $k = 0$ will allow us to solve for $y(k)$ recursively

$$y(0) = -0.2y(-1) - 0.5y(-2) + x(0) - x(-1) = 0 + 0 + 1 + 0 = 1$$

$$y(1) = -0.2y(0) - 0.5y(-1) + x(1) - x(0) = -0.2(1) - 0 + 3 - 1 = 1.8$$

$$y(2) = -0.2y(1) - 0.5y(0) + x(2) - x(1) = -0.2(1.8) - 0.5(1) + 0 - 3 = -3.86$$

$$y(3) = -0.2y(2) - 0.5y(1) + x(3) - x(2) = -0.2(-3.86) - 0.5(1.8) + 0 - 0 = -0.128$$

The process of stepping through the difference equation starting at $k = 0$ to solve for $y(k)$ is called **recursion**. This recursion process is exactly how digital filters are implemented using a time domain approach.

Impulse Response

The impulse response of a system is the output response of a system initially at rest when the input signal is a discrete impulse, $\delta(k)$, defined as

$$\text{Discrete Impulse } \delta(k) = \begin{cases} 1 & k = 0 \\ 0 & k \neq 0 \end{cases} \quad (4.4)$$

Example 4.8: Computing the Impulse Response of Discrete Systems Recursively

Find the impulse response for each of the discrete systems.

$$(a) \quad y(k) = 0.25x(k) + 0.5x(k-1) + 0.3x(k-2)$$

$$(b) \quad y(k) = 0.1y(k-1) + x(k) - 0.5x(k-1)$$

Solution:

$$(a) \quad y(0) = 0.25x(0) + 0.5x(-1) + 0.3x(-2) = 0.25$$

$$y(1) = 0.25x(1) + 0.5x(0) + 0.3x(-1) = 0.5$$

$$y(2) = 0.25x(2) + 0.5x(1) + 0.3x(0) = 0.3$$

$$y(k) = 0 \quad \text{for } k \geq 3$$

Notice that the impulse response is equal to the numerical coefficients in the difference equation. This will always be the case when the output does not depend on past output values but only on input and past input values.

$$(b) \quad y(0) = 0.1y(-1) + x(0) - 0.5x(-1) = 0 + 1 - 0 = 1$$

$$y(1) = 0.1y(0) + x(1) - 0.5x(0) = 0.1(1) + 0 - 0.5 = -0.4$$

$$y(2) = 0.1y(1) + x(2) - 0.5x(1) = 0.1(-0.4) = -0.04$$

$$y(3) = 0.1y(2) + x(3) - 0.5x(2) = 0.1(-0.04) = -0.004$$

$$\vdots$$

$$y(k) = -0.4 \cdot (0.1)^{k-1} \quad \text{for } k \geq 1$$

An easy pattern develops in this case because the difference equation was chosen to be simple.

Stability of Discrete Systems

A discrete system is stable if for any set of initial conditions and any bounded input signal, the output signal remains bounded. A discrete system is stable if and only if the impulse response goes to zero as time goes to infinity. Both of the systems in Example 4.8 are stable.

4.5 Z-TRANSFORMS

The Z-transform is a very useful tool for describing and analyzing discrete systems in terms of stability, output response and frequency response. The Z-transform of a discrete signal, $f(k)$, is defined as

$$\mathcal{Z}\{f(k)\} = F(z) = \sum_{k=-\infty}^{\infty} f(k)z^{-k} \quad (4.5)$$

Example 4.9 illustrates how to find the Z-transform of some discrete signals using this definition.

Example 4.9: Z-Transforms of Basic Signals

(a) Using the definition, find the Z-transform of a discrete unit impulse, $\delta(k)$.

$$\mathcal{Z}\{\delta(k)\} = \sum_{k=-\infty}^{\infty} \delta(k)z^{-k} = 1z^{-0} = 1 \quad \delta(k) \leftrightarrow 1$$

(b) Using the definition, find the Z-transform of a discrete unit step, $u(k) = 1$ for $k \geq 0$ and $u(k) = 0$ for $k < 0$.

$$\mathcal{Z}\{u(k)\} = \sum_{k=-\infty}^{\infty} u(k)z^{-k} = 1z^{-0} + z^{-1} + z^{-2} + \dots = \frac{z}{z-1} \quad u(k) \leftrightarrow \frac{z}{z-1}$$

The infinite series summation simplifies to $z/(z-1)$. This can be easily verified by long division (or by looking it up in a math book).

(c) Find the Z-transform of a discrete exponential sequence, $x(k) = a^k u(k)$.

$$\mathcal{Z}\{a^k u(k)\} = \sum_{k=-\infty}^{\infty} a^k u(k)z^{-k} = \sum_{k=-\infty}^{\infty} u(k) \left[\frac{z}{a} \right]^{-k} = \frac{z/a}{(z/a)-1} \quad a^k u(k) \leftrightarrow \frac{z}{z-a}$$

In general, Z-Transforms for signals are not computed using the definition but are found using Z-Transform tables. A table of Z-Transforms for several common discrete signals is included in the Appendix.

Properties of Z-Transforms

Z-transforms have several properties only two of which will be considered here.

1. Linearity Property:

$$\begin{aligned} &\text{If } f_1(k) \leftrightarrow F_1(z) \text{ and } f_2(k) \leftrightarrow F_2(z) \text{ then} \\ &a_1 f_1(k) + a_2 f_2(k) \leftrightarrow a_1 F_1(z) + a_2 F_2(z) \text{ for any real numbers } a_1 \text{ and } a_2 \end{aligned} \quad (4.6)$$

The Z-Transform of any linear combination of discrete signals is the same linear combination of the Z-Transforms of the individual discrete signals.

2. Time Delay Property:

$$f(k-n) \leftrightarrow z^{-n} F(z) \quad (4.7)$$

The Z-Transform of a discrete signal that has been delayed by n sampling instants is the Z-Transform of the signal without delay multiplied by z^{-n} . The operator z^{-1} will be used throughout the text to indicate a time delay of one sampling instant.

Transfer Function

The transfer function of a linear system describes how the system transfers an input signal, x , into an output signal, y , assuming the system is initially at rest. For discrete systems, the transfer function is derived by taking the Z-Transform of the input/output difference equation and putting the result in the form

$$Y(z) = H(z) \cdot X(z) \quad (4.8)$$

$H(z)$ is the transfer function of the discrete linear system.

System Poles – roots of the denominator of the transfer function.

System Zeros – roots of the numerator of the transfer function.

The transfer function can be derived starting with the input/output difference equation from Equation 4.2:

$$y(k) + a_1y(k-1) + a_2y(k-2) + \dots + a_Ny(k-N) = b_0x(k) + b_1x(k-1) + b_2x(k-2) + \dots + b_Lx(k-L)$$

Using the linearity and time delay properties for Z-Transforms gives

$$Y(z) + a_1z^{-1} Y(z) + a_2z^{-2} Y(z) + \dots + a_Nz^{-N} Y(z) = b_0X(z) + b_1z^{-1}X(z) + b_2z^{-2}X(z) + \dots + b_Lz^{-L}X(z) \quad (4.9)$$

Factoring out $Y(z)$ from the left-hand side and $X(z)$ from the right-hand side gives

$$Y(z) [1 + a_1z^{-1} + a_2z^{-2} + \dots + a_Nz^{-N}] = [b_0 + b_1z^{-1} + b_2z^{-2} + \dots + b_Lz^{-L}] X(z) \quad (4.10)$$

Dividing both sides by the polynomial on the left then eliminating the negative powers of z through multiplication by z^N gives

$$Y(z) = \frac{b_0 + b_1z^{-1} + \dots + b_Lz^{-L}}{1 + a_1z^{-1} + \dots + a_Nz^{-N}} X(z) = \frac{b_0z^N + b_1z^{N-1} + \dots + b_Lz^{N-L}}{z^N + a_1z^{N-1} + \dots + a_N} X(z) \quad (4.11)$$

So, the transfer function of the linear discrete system is

$$H(z) = \frac{b_0 + b_1z^{-1} + \dots + b_Lz^{-L}}{1 + a_1z^{-1} + \dots + a_Nz^{-N}} = \frac{b_0z^N + b_1z^{N-1} + \dots + b_Lz^{N-L}}{z^N + a_1z^{N-1} + \dots + a_N} \quad (4.12)$$

Comparing the transfer function to the original input/output difference equation, it should be clear that the transfer function can be derived immediately from the input/output difference equation without all the intervening algebra. This is illustrated in Example 4.10.

Example 4.10: Determining the Transfer Function of Discrete Systems

Find the transfer functions for the following discrete systems (taken from Examples 4.7 and 4.8).

$$(a) \quad y(k) + 0.2y(k-1) + 0.5y(k-2) = x(k) - x(k-1)$$

$$(b) \quad y(k) = 0.25x(k) + 0.5x(k-1) + 0.3x(k-2)$$

$$(c) \quad y(k) = 0.1y(k-1) + x(k) - 0.5x(k-1)$$

Solution:

- (a) The numerator of the transfer function is determined from the right-hand (input) side of the difference equation and the denominator is determined from the left-hand (output) side of the difference equation. Time delays are represented using the z^{-1} operator.

$$H(z) = \frac{1 - z^{-1}}{1 + 0.2z^{-1} + 0.5z^{-2}} = \frac{z^2 - z}{z^2 + 0.2z + 0.5}$$

$$(b) \quad H(z) = \frac{0.25 + 0.5z^{-1} + 0.3z^{-2}}{1} = \frac{0.25z^2 + 0.5z + 0.3}{z^2}$$

- (c) The output terms need to be on the left-hand side prior to determining the transfer function:

$$y(k) - 0.1y(k-1) = x(k) - 0.5x(k-1)$$

$$H(z) = \frac{1 - 0.5z^{-1}}{1 - 0.1z^{-1}} = \frac{z - 0.5}{z - 0.1}$$

The transfer function is useful for determining stability of a system, output response, and frequency response. The transfer function is important for designing and analyzing digital filters and for applications involving Fast-Fourier transforms and correlation or convolution operations. In Examples 4.7 and 4.8, an output response was determined using recursion; that is, stepping through the difference equation and solving for $y(k)$ starting at $k = 0$. An alternative for finding the output response is to use the transfer function and the **filter** function in MATLAB®. This alternative is demonstrated in Example 4.11.

Example 4.11: Computing Output Signal using the Transfer Function

- (a) A discrete system is modeled by the difference equation:

$$y(k) + 0.2y(k-1) + 0.5y(k-2) = x(k) - x(k-1)$$

Use the transfer function and MATLAB to find the output signal for an input signal $x(0) = 1$, $x(1) = 3$, $x(k) = 0$ for $k \geq 2$.

- (b) Using MATLAB, find the impulse response for the discrete system:

$$y(k) = 0.1y(k-1) + x(k) - 0.5x(k-1)$$

Solution:

(a) The transfer function for the system was determined in Example 4.10(a) as

$$H(z) = \frac{1 - z^{-1}}{1 + 0.2z^{-1} + 0.5z^{-2}} = \frac{z^2 - z}{z^2 + 0.2z + 0.5}$$

This transfer function is entered into MATLAB along with the input signal then processed by the **filter** function as follows

```
num_tf = [1 -1 0];
den_tf = [1 0.2 0.5];
x = [1 3 0 0 0 0];
y=filter(num_tf,den_tf,x)
```

```
y = 1.0000 1.800 -3.860 -0.1280 1.955 -0.3271
```

This result matches the output response values calculated using recursion in Example 4.7.

(b) The transfer function for the system was determined in Example 4.10(c) as

$$H(z) = \frac{1 - 0.5z^{-1}}{1 - 0.1z^{-1}} = \frac{z - 0.5}{z - 0.1}$$

This transfer function is entered in MATLAB. While it is certainly possible to define an input signal $x = [1 \ 0 \ 0 \ 0]$ to represent a discrete impulse and use the **filter** function, it is faster to use the specialized **dimpulse** function.

```
num_tf=[1 -0.5];
den_tf = [1 -0.1];
y=dimpulse(num_tf, den_tf,5)
```

```
y = 1.0000 -0.4000 -0.0400 -0.0040 -0.00040
```

The 3rd argument in **dimpulse** indicates the number of output values to be calculated. This impulse response matches the response calculated in Example 4.8 using recursion.

Stability of Discrete-Time Systems

A discrete system is stable if for any set of initial conditions and any bounded input signal, the output signal remains bounded. Recall that a discrete system is stable if and only if the impulse response goes to zero as time goes to infinity. It is also possible to determine stability of a discrete system by looking at the transfer function because the transfer function is the Z-Transform of the impulse response; that is, $H(z) = \mathcal{Z}\{h(k)\}$.

System Stability Theorem:

A discrete system will be stable if and only if all of the system poles have magnitude strictly less than 1. In other words, all the system poles must lie within a unit circle in order for a discrete system to be stable.

Example 4.12: Stability of Discrete Systems

Determine which of the following discrete systems are stable.

- (a) $y(k) + 0.2y(k-1) + 0.5y(k-2) = x(k) - x(k-1)$
- (b) $y(k) = 0.25x(k) + 0.5x(k-1) + 0.3x(k-2)$
- (c) $y(k) = 0.1y(k-1) + x(k) - 0.5x(k-1)$

Solution:

The transfer functions for all three of these systems were determined in Example 4.10

$$(a) \quad H(z) = \frac{z^2 - z}{z^2 + 0.2z + 0.5} = \frac{z^2 - z}{(z + 0.1 - j0.7)(z + 0.1 + j0.7)}$$

The poles for this 2nd order system are $-0.1 \pm j0.7 = 0.7071 \angle 1.7127$ rad. Since these poles have a magnitude less than 1, the system in (a) is stable.

$$(b) \quad H(z) = \frac{0.25z^2 + 0.5z + 0.3}{z^2}$$

The poles for this 2nd order system are 0, 0. These poles have magnitude less than 1 so the system is stable.

$$(c) \quad H(z) = \frac{z - 0.5}{z - 0.1}$$

The pole for this 1st order system is 0.1 (magnitude less than 1) so this system is also stable.

MATLAB can be used to determine the poles of the system for denominators that are difficult to factor by inspection. The following MATLAB commands will determine the poles and the magnitude of the poles for the system in Example 4.12(a).

```
poles=roots([1 0.2 0.5])
mag_poles = abs(poles)

poles =  -0.1000 + 0.7000i
         -0.1000 - 0.7000i

mag_poles =  0.7071
            0.7071
```


Challenge Question 4.4

What can be said about the stability of discrete systems when the output depends only on the current input and past inputs and has no dependence on past outputs? Why?

4.6 CONVOLUTION

Convolution is a mathematical operation that takes two signals and produces a third signal. It has many applications in digital signal processing and in other areas as well. Convolution can be done with either continuous signals or discrete signals. In this text, only discrete convolution will be considered. The convolution operation is symbolized using the symbol, $*$. This should not be confused with multiplication! The convolution of two discrete signals is defined as follows

$$y(k) = x(k) * h(k) = \sum_{n=0}^k x(n)h(k-n) \quad (4.13)$$

The summation in Equation 4.13 can be broken up to get a better picture as follows

$$\begin{aligned} y(0) &= x(0)h(0) \\ y(1) &= x(0)h(1) + x(1)h(0) \\ y(2) &= x(0)h(2) + x(1)h(1) + x(2)h(0) \\ y(3) &= x(0)h(3) + x(1)h(2) + x(2)h(1) + x(3)h(0) \end{aligned}$$

Notice the pattern that the time indices in x and h sum to the time index for y .

Example 4.13: Computing Convolution using the Formula

Compute $y(k) = x(k) * h(k)$ for $x(k) = \{1 \ -1 \ 2 \ 0 \ 0 \ \dots\}$ and $h(k) = \{3 \ -2 \ 0 \ 0 \ \dots\}$.

Solution:

$$\begin{aligned} y(0) &= x(0)h(0) = 1(3) = 3 \\ y(1) &= x(0)h(1) + x(1)h(0) = 1(-2) + (-1)(3) = -5 \\ y(2) &= x(0)h(2) + x(1)h(1) + x(2)h(0) = 1(0) + (-1)(-2) + 2(3) = 8 \\ y(3) &= x(0)h(3) + x(1)h(2) + x(2)h(1) + x(3)h(0) = 1(0) + (-1)(0) + 2(-2) + 0(3) = -4 \\ y(4) &= x(0)h(4) + x(1)h(3) + x(2)h(2) + x(3)h(1) + x(4)h(0) = 1(0) + (-1)(0) + 2(0) + 0(-2) + 0(3) = 0 \\ y(k) &= 0 \text{ for } k \geq 4 \end{aligned}$$

It is easy to make errors computing the convolution by hand using the summation definition, particularly when the sequences get longer. A simpler way to compute convolution by hand is a graphical method which involves flipping (reversing) one of the two sequences and sliding it

under the other sequence. A multiply and accumulate calculation is done where the sequences overlap. This graphical method is shown in Example 4.14.

Example 4.14: Computing Convolution Graphically

Compute $y(k) = x(k) * h(k)$ using the graphical method for convolution given that:

$$x(k) = \{1 \ -1 \ 2 \ 0 \ 0 \ \dots\} \text{ and } h(k) = \{3 \ -2 \ 0 \ 0 \ \dots\}.$$

Solution:

x		1	-1	2	0	0	
h , flipped	0	0	-2	3	$\rightarrow$		$y(0)$
Sum of Overlap products				3			$= 3$

x		1	-1	2	0	0	
h , flipped	0	0	-2	3	$\rightarrow$		$y(1)$
Sum of Overlap products			-2	-3			$= -5$

x		1	-1	2	0	0	
h , flipped	0	0	-2	3	$\rightarrow$		$y(2)$
Sum of Overlap products		0	2	6			$= 8$

x		1	-1	2	0	0	
h , flipped		0	0	-2	3	$\rightarrow$	$y(3)$
Sum of Overlap products		0	0	-4	0		$= -4$

x		1	-1	2	0	0	
h , flipped		0	0	-2	3	$\rightarrow$	$y(4)$
Sum of Overlap products		0	0	0	0		$= 0$

Of course the easiest method for computing convolution is using a software program such as MATLAB as illustrated in the next example.

Example 4.15: Computing Convolution using MATLAB

Compute $y(k) = x(k) * h(k)$ for $x(k) = \{1 \ -1 \ 2 \ 0 \ 0 \ \dots\}$ and $h(k) = \{3 \ -2 \ 0 \ 0 \ \dots\}$ using MATLAB.

Solution:

```
x = [1 -1 2];
h = [3 -2];
y = conv(x, h)
```

```
y = 3 -5 8 -4
```

Useful Facts about convolution in MATLAB:

1. The length of $y(k)$ = length of $x(k)$ + length of $h(k)$ - 1
2. The start time for $y(k)$ = start time for $x(k)$ + start time for $h(k)$. In other words, if both $x(k)$ and $h(k)$ have non-zero values at $k=0$, then $y(k)$ will also have a non-zero value starting at $k=0$. If $x(k)$ has its first non-zero value at $k=1$ and $h(k)$ has its first non-zero value at $k=2$, then $y(k)$ will remain zero until $k=3$.

There are several applications for convolution. In digital image processing, convolutional filtering is used in edge detection and other processes. In audio applications, convolution can be used to map the impulse response of an actual theatre onto a digital audio signal to create reverb (the echo effects that occur when music is performed in a room or theatre). This makes the digital audio signal sound more realistic. A few simple applications will be considered in this section with more applications to come later.

Multiplying Polynomials using Convolution

Polynomials can be multiplied by simply convolving the coefficient sequences for each polynomial as shown in the following example.

Example 4.16: Computing the Product of Polynomials using Convolution

Compute the following products

- (a) $(z^3 + 4z^2 - 5z + 10) \cdot (z^2 - 7z + 16)$
- (b) $(2z^3 - z^2 + 5z + 10) \cdot (z^2 - 16) \cdot (z + 4)$

Solution:

(a) Simply convolve the coefficient sequences: $\{1 \ 4 \ -5 \ 10\} * \{1 \ -7 \ 16\}$. This can be done by hand graphically or using MATLAB. Here we will use MATLAB.

```
x = [1 4 -5 10];
h = [1 -7 16];
conv(x,h)
ans = 1 -3 -17 109 -150 160
```

$$\Rightarrow (z^3 + 4z^2 - 5z + 10) \cdot (z^2 - 7z + 16) = z^5 - 3z^4 - 17z^3 + 109z^2 - 150z + 160$$

(b) Again, convolve the coefficient sequences: $\{2 \ -1 \ 5 \ 10\} * \{1 \ 0 \ -16\} * \{1 \ 4\}$. Notice the placeholder of 0 in the second sequence to account for the missing z -term. To convolve three sequences, compute the convolution of any pair of sequences and convolve the result with the third sequence. This can be done in MATLAB using a nested convolution command.

```
a = [2 -1 5 10]; b = [1 0 -16]; c = [1 4];
conv(a, conv(b, c))
ans = 2 7 -31 -82 24 -480 -640
```

$$\Rightarrow (2z^3 - z^2 + 5z + 10) \cdot (z^2 - 16) \cdot (z + 4) = 2z^6 + 7z^5 - 31z^4 - 82z^3 + 24z^2 - 480z - 640$$

Computing the Output Response of a Discrete System Using Convolution

In Section 4.4, the output response of a discrete system was determined by solving the input/output difference equation using recursion. The output response can also be computed by convolving the input sequence with the impulse response of the system.

$$y(k) = x(k) * h(k) = \sum_{n=0}^k x(n)h(k-n) \quad (4.14)$$

$y(k)$ is output sequence

$x(k)$ is input sequence

$h(k)$ is system impulse response

Example 4.17: Summary of Methods for Computing the Output Response

A discrete system is modeled by the input/output difference equation

$$y(k) = 0.25x(k) + 0.5x(k-1) + 0.3x(k-2)$$

Compute the output response for an input sequence $x(k) = \{1 \ 0 \ -1 \ 2\}$.

Solution using Convolution:

Since this system does not depend on past outputs (only inputs and past inputs), the impulse response is simply the coefficients of the input terms; that is, $h(k) = \{0.25 \ 0.5 \ 0.3\}$. So the output $y(k) = x(k) * h(k)$.

```
x = [1 0 -1 2];  
h = [0.25 0.5 0.3];  
conv(x, h)
```

```
ans =    0.2500    0.5000    0.0500    0    0.7000    0.6000
```

```
y(k) = {0.25 0.5 0.05 0 0.7 0.6 0 0 ...}
```

Solution using Recursion:

```
y(0) = 0.25x(0) + 0.5x(-1) + 0.3x(-2) = 0.25(1) = 0.25  
y(1) = 0.25x(1) + 0.5x(0) + 0.3x(-1) = 0.25(0) + 0.5(1) = 0.5  
y(2) = 0.25x(2) + 0.5x(1) + 0.3x(0) = 0.25(-1) + 0.5(0) + 0.3(1) = 0.05  
y(3) = 0.25x(3) + 0.5x(2) + 0.3x(1) = 0.25(2) + 0.5(-1) + 0.3(0) = 0  
y(4) = 0.25x(4) + 0.5x(3) + 0.3x(2) = 0.25(0) + 0.5(2) + 0.3(-1) = 0.7  
y(5) = 0.25x(5) + 0.5x(4) + 0.3x(3) = 0.25(0) + 0.5(0) + 0.3(2) = 0.6  
y(k) = 0 for k ≥ 6
```

Solution using Transfer Function and filter command in MATLAB:

As shown in Section 4.5, the transfer function can be derived by inspection from the difference equation and is given by

$$H(z) = \frac{0.25 + 0.5z^{-1} + 0.3z^{-2}}{1} = \frac{0.25z^2 + 0.5z + 0.3}{z^2}$$

```
num_tf=[0.25 0.5 0.3];  
den_tf=[1 0 0];  
x = [1 0 -1 2 0 0];  
y=filter(num_tf, den_tf, x)
```

```
y = 0.2500 0.5000 0.0500 0 0.7000 0.6000
```

Comments

- Not surprisingly, all three methods produce the same output sequence.
- When using the filter command, the output sequence produced will be exactly the same length as the input sequence. So in order to get the whole output sequence, trailing zeros were added to the input sequence.

4.7 INVERSE Z-TRANSFORMS

In Example 4.17, three different methods were used to find the output response of a discrete system. Each of these methods produced a sequence of values for the output response, but did not yield a closed-form expression or equation for the system output. In this particular example, the input sequence is very short and the system order is small, so the output sequence can be completely computed in only a few iterations. For longer input sequences and higher order systems, a closed-form expression for the output sequence may prove useful, particularly in analyzing system behavior.

An equation for the output of a discrete system can be derived using inverse Z-Transforms. In Section 4.5, it was shown that the Z-transform of the output response is the product of the system transfer function, $H(z)$, and the Z-transform of the input; that is, $Y(z) = H(z) \cdot X(z)$. So, the output sequence, $y(k)$, can be determined by finding the inverse Z-Transform of $Y(z)$. There are several methods for finding an inverse Z-Transform. Only one method, partial fraction expansion followed by table look-up, will be included in this section. The procedure for computing an equation for $y(k)$ is:

1. Find the transfer function, $H(z)$, of the discrete system.
2. Find the Z-Transform of the input, $X(z)$, using the Z-transform table in the appendix.
3. Find the Z-Transform of the output, $Y(z) = H(z)X(z)$.
4. Find a partial fraction expansion for $Y(z)/z$ using the MATLAB function **residue**.
5. Find a partial fraction expansion for $Y(z)$ by multiplying the result in Step 4 by z .
6. Use the Z-transform table in the appendix to find an equation for $y(k)$.

Example 4.18: Computing Step Response using Inverse Z-Transform

Find the unit step response for each of the discrete systems.

$$(a) \quad y(k) + 0.2y(k-1) + 0.5y(k-2) = x(k) - x(k-1)$$

$$(b) \quad y(k) = 0.25x(k) + 0.5x(k-1) + 0.3x(k-2)$$

$$(c) \quad y(k) = 0.1y(k-1) + x(k) - 0.5x(k-1)$$

Solution:

The transfer function for each of these systems was determined in Example 4.10. The input to the system is a unit step which has a Z-Transform of $X(z) = z/(z-1)$. The Z-transform of a unit step was derived in Example 4.9 and is also included in the appendix.

$$(a) \quad Y(z) = \left[\frac{z^2 - z}{z^2 + 0.2z + 0.5} \right] \left[\frac{z}{z-1} \right] \quad Y(z)/z = \left[\frac{z^2 - z}{z^2 + 0.2z + 0.5} \right] \left[\frac{1}{z-1} \right]$$

In MATLAB, enter the numerator and denominator of $Y(z)/z$ then use **residue** as follows:

```
num = [1 -1 0]; den = conv([1 0.2 0.5], [1 -1]);  
[coeff poles] = residue(num, den)  
  
coeff =    0.0000          poles =    1.0000  
         0.5000 + 0.0714i          -0.1000 + 0.7000i  
         0.5000 - 0.0714i          -0.1000 - 0.7000i
```

Matching the numerator coefficients to the poles gives the partial fraction expansion for $Y(z)/z$:

$$Y(z)/z = \frac{0}{z-1} + \frac{0.5 + 0.0714i}{z - (-0.1 + 0.7i)} + \frac{0.5 - 0.0714i}{z - (-0.1 - 0.7i)}$$

Converting the complex numbers to polar form then multiplying by z gives the partial fraction expansion for $Y(z)$:

$$Y(z) = \frac{0z}{z-1} + \frac{(0.5051 \angle 0.142 \text{ rad})z}{z - (0.7071 \angle 1.71 \text{ rad})} + \frac{(0.5051 \angle -0.142 \text{ rad})z}{z - (0.7071 \angle -1.71 \text{ rad})}$$

Using the Z-transform table in the appendix gives the output sequence:

$$y(k) = 2 \cdot (0.5051) \cdot (0.7071)^k \cdot \cos(1.71k + 0.142)$$

$$(b) \quad Y(z) = \left[\frac{0.25z^2 + 0.5z + 0.3}{z^2} \right] \left[\frac{z}{z-1} \right] \quad Y(z)/z = \left[\frac{0.25z^2 + 0.5z + 0.3}{z^2} \right] \left[\frac{1}{z-1} \right]$$

```
num = [0.25 0.5 0.3]; den = conv([1 0 0], [1 -1]);  
[coeff poles] = residue(num, den)
```

$$\begin{array}{rcl} \text{coeff} = & 1.0500 & \text{poles} = 1 \\ & -0.8000 & 0 \\ & -0.3000 & 0 \end{array}$$

The partial fraction for $Y(z)/z$ is:

$$Y(z)/z = \frac{1.05}{z-1} + \frac{-0.8}{(z-0)} + \frac{-0.3}{(z-0)^2} \quad \text{Note the power of 2 for 2nd pole at zero}$$

$$Y(z) = \frac{1.05z}{z-1} - 0.8 + \frac{-0.3}{z}$$

$$y(k) = (1.05) \cdot u(k) - (0.8) \cdot \delta(k) - (0.3) \cdot \delta(k-1)$$

$$(c) \quad Y(z) = \left[\frac{(z-0.5)}{(z-0.1)} \right] \left[\frac{z}{z-1} \right] \quad Y(z)/z = \left[\frac{(z-0.5)}{(z-0.1)} \right] \left[\frac{1}{z-1} \right]$$

```
num = [1 -0.5]; den = conv([1 -0.1], [1 -1]);
[coeff poles] = residue(num, den)
```

$$\begin{array}{rcl} \text{coeff} = & 0.5556 & \text{poles} = 1.0000 \\ & 0.4444 & 0.1000 \end{array}$$

$$Y(z)/z = \frac{0.5556}{z-1} + \frac{0.4444}{(z-0.1)}$$

$$Y(z) = \frac{0.5556z}{z-1} + \frac{0.4444z}{z-0.1}$$

$$y(k) = (0.5556) \cdot u(k) - 0.4444 \cdot (0.1)^k$$

Computing a closed-form expression for the output response of a system is useful for analyzing system behavior. As illustrated in Example 4.18, the equation for the output response of a discrete system will have terms corresponding to each of the poles of the discrete system and a term corresponding to the input. If a discrete system has poles at 0.1, -0.5, and 0.25, the equation for the output response will have decaying exponential terms corresponding to these poles; that is, $C_1(0.1)^k$, $C_2(-0.5)^k$, and $C_3(0.25)^k$. Complex poles at $0.7 \angle 1.7$ cause a damped oscillatory response in the output of the form $C_1(0.7)^k \cos(1.7k + \theta)$. It should be clear at this point why a discrete system is stable if and only if all of the system poles have magnitude strictly less than 1.

4.8 CORRELATION

Correlation is a measure of how well two signals match one another. Correlation is used in speech recognition, pattern recognition, radar tracking, and many other applications. The cross-correlation between two discrete signals is a measure of how well the signals are related in a linear manner and is computed as

$$R_{xy}(n) = \sum_{k=0}^{N-1} x(k)y(k-n) = \sum_{k=0}^{N-1} x(k+n)y(k) \quad n = -M+1, \dots, N-1 \quad (4.15)$$

M = length of signal $y(k)$

N = length of signal $x(k)$

Expanding the summation for a couple values of n will help in understanding the correlation calculation a bit better.

$$R_{xy}(0) = x(0)y(0) + x(1)y(1) + x(2)y(2) + \dots + x(N-1)y(N-1)$$

$$R_{xy}(1) = x(1)y(0) + x(2)y(1) + x(3)y(2) + \dots + x(N-1)y(N-2)$$

$$R_{xy}(2) = x(2)y(0) + x(3)y(1) + x(4)y(2) + \dots + x(N-1)y(N-3)$$

$$R_{xy}(3) = x(3)y(0) + x(4)y(1) + x(5)y(2) + \dots + x(N-1)y(N-4)$$

$$R_{xy}(-1) = x(0)y(1) + x(1)y(2) + x(2)y(3) + \dots + x(N-2)y(N-1)$$

To compute, $R_{xy}(0)$, signals $x(k)$ and $y(k)$ are lined up in time and $R_{xy}(0)$ is simply the sum of the overlapping products. To compute $R_{xy}(1)$, the signal $y(k)$ is shifted one space to the right of signal $x(k)$ and $R_{xy}(1)$ is simply the sum of the overlapping products. To compute $R_{xy}(-1)$, the signal $y(k)$ is shifted one space to the left of signal $x(k)$ and $R_{xy}(-1)$ is simply the sum of the overlapping products.

Challenge Question 4.5

Why would we want to compare one signal to a shifted version of another signal?

Example 4.19: Computing Cross-Correlation of Two Sequences

Compute the cross-correlation for the two sequences $x(k) = \{1 \ 2 \ 5\}$ and $y(k) = \{2 \ 4 \ 9\}$.

Solution:

$x(k)$			1	2	5	
$y(k)$ shifted left by 2	2	4	9			$R_{xy}(-2)$
Sum of Overlapping Products			9			9
$x(k)$			1	2	5	
$y(k)$ shifted left by 1	2	4	9			$R_{xy}(-1)$
Sum of Overlapping Products			4	18		22

x(k)	1	2	5	
y(k)	2	4	9	$R_{xy}(0)$
Sum of Overlapping Products	2	8	45	55
x(k)	1	2	5	
y(k) shifted right by 1	2	4	9	$R_{xy}(1)$
Sum of Overlapping Products	4	20		24
x(k)	1	2	5	
y(k) shifted right by 2	2	4	9	$R_{xy}(2)$
Sum of Overlapping Products	10			10

Challenge Question 4.6

Looking at Example 4.19, how does the correlation of sequences $x(k)$ and $y(k)$ compare to the convolution of sequences $x(k)$ and $y(k)$?

So what do these numbers tell us in terms of a match between $x(k)$ and $y(k)$? Well, the closest match between $x(k)$ and $y(k)$ occurs when the signals are lined up in time; that is, $y(k)$ does not appear to be a time-shifted version of $x(k)$. Beyond that, it is extremely difficult to describe how close a linear relationship there is between $x(k)$ and $y(k)$ based on these numbers.

A much better measure can be derived by normalizing the cross-correlation function as follows:

$$\text{Normalized } R_{xy}(n) = \frac{\sum_{k=0}^{N-1} x(k)y(k-n)}{\sqrt{R_{xx}(0)R_{yy}(0)}} \quad n = -M+1, \dots, N-1 \quad (4.16)$$

$R_{xx}(0)$ is the correlation of x with itself and $R_{yy}(0)$ is the correlation of y with itself. Computing the correlation of a signal with itself is referred to as an auto-correlation.

The normalized $R_{xy}(n)$ will always range from -1 to $+1$. A value of $+1$ means there is a 100% positive correlation (linear relation) between the two sequences; that is, $x(k) = Ay(k)$ where A is some positive real number. A value of -1 means there is a 100% negative correlation (linear relationship) between the two sequences; that is, $x(k) = -Ay(k)$ where A is some positive real number. A value of 0 means there is no linear relationship between the two sequences.

Example 4.20: Normalized Cross-Correlation for Two Sequences

Compute the normalized cross-correlation for the sequences $x(k) = \{1 \ 2 \ 5\}$ and $y(k) = \{2 \ 4 \ 9\}$.

Solution:

$$R_{xx}(0) = x(0)x(0) + x(1)x(1) + x(2)x(2) = 1^2 + 2^2 + 5^2 = 30$$

$$R_{yy}(0) = y(0)y(0) + y(1)y(1) + y(2)y(2) = 2^2 + 4^2 + 9^2 = 101$$

$$\sqrt{R_{xx}(0)R_{yy}(0)} = \sqrt{30(101)} = 55.0454$$

Dividing the results of Example 4.19 by the normalization factor yields

Norm. $R_{xy}(-2)$	Norm. $R_{xy}(-1)$	Norm. $R_{xy}(0)$	Norm. $R_{xy}(1)$	Norm. $R_{xy}(2)$
0.1635	0.3997	0.9992	0.4360	0.1817

Now we can conclude that while there is not a perfect linear relationship between $x(k)$ and $y(k)$, the relationship is very close to linear. Look back at the sequences $x(k)$ and $y(k)$. You should be able to see a strong linear relationship between the two sequences.

The normalized cross-correlation can be computed quite easily in MATLAB by first entering the sequences $x(k)$ and $y(k)$ and then using the command

$$\text{xcorr}(x, y) / \text{sqrt}(\text{xcorr}(x, x, 0) * \text{xcorr}(y, y, 0))$$

MATLAB will generate the $R_{xy}(n)$ vector for $n = -N+1$ to $N-1$ where N is the maximum length of sequences $x(k)$ and $y(k)$. Therefore, $R_{xy}(0)$ always ends up at the center of the vector $R_{xy}(n)$.

Example 4.21: Normalized Cross-Correlation for Sequences

Compute the cross-correlation for the following sequences and comment on the results.

- (a) $x(k) = \{2 \ -1 \ -3 \ -2 \ 1\}$ and $ya(k) = \{10 \ -5 \ -15 \ -10 \ 5\}$
 (b) $x(k) = \{2 \ -1 \ -3 \ -2 \ 1\}$ and $yb(k) = \{-6 \ 3 \ 9 \ 6 \ -3\}$
 (c) $x(k) = \{2 \ -1 \ -3 \ -2 \ 1\}$ and $yc(k) = \{4 \ 3 \ -6 \ 5 \ 3\}$

Solution:

(a) $x = [2 \ -1 \ -3 \ -2 \ 1]; \quad ya = [10 \ -5 \ -15 \ -10 \ 5];$
 $Ra = \text{xcorr}(x, ya) / \text{sqrt}(\text{xcorr}(x, x, 0) * \text{xcorr}(ya, ya, 0))$

$$Ra = 0.1053 \quad -0.2632 \quad -0.3684 \quad 0.2632 \quad \mathbf{1.0000} \quad 0.2632 \quad -0.3684 \quad -0.2632 \quad 0.1053$$

The results show a 100% positive correlation between $x(k)$ and $ya(k)$ when both signals are lined up in time ($R_{xya}(0)$). This should not be surprising since $ya(k) = 5x(k)$.

(b) $yb = [-6 \quad 3 \quad 9 \quad 6 \quad -3];$
 $Rb = \text{xcorr}(x, yb) / \text{sqrt}(\text{xcorr}(x, x, 0) * \text{xcorr}(yb, yb, 0))$

$$Rb = -0.1053 \quad 0.2632 \quad 0.3684 \quad -0.2632 \quad \mathbf{-1.0000} \quad -0.2632 \quad 0.3684 \quad 0.2632 \quad -0.1053$$

The results show a 100% negative correlation between $x(k)$ and $yb(k)$ when both signals are lined up in time. Again, not surprising since $yb(k) = -3x(k)$.

```
(c)   yc = [4    3   -6    5    3];
       Rc = xcorr(x,yc)/sqrt(xcorr(x,x,0)*xcorr(yc,yc,0))
```

```
Rc = 0.1412  0.1648 -0.6120 -0.2118  0.3766  0.0942 -0.5649 -0.1177  0.0942
```

The results show a 37.66% positive correlation between $x(k)$ and $yc(k)$ when the signals are lined up in time. There is a 61.2% negative correlation between $x(k)$ and $yc(k+2)$ (i.e., when $yc(k)$ leads $x(k)$ by two sampling instants). These two signals do not correlate well.

Figure 4.2 shows scatterplots for the signals in this example. Scatterplots can be created in MATLAB using the command ***scatter(x,y)***. It is easy to see the linear relationship between both $x(k)$ and $ya(k)$ and $x(k)$ and $yb(k)$. The lack of a linear relationship between $x(k)$ and $yc(k)$ is also apparent.

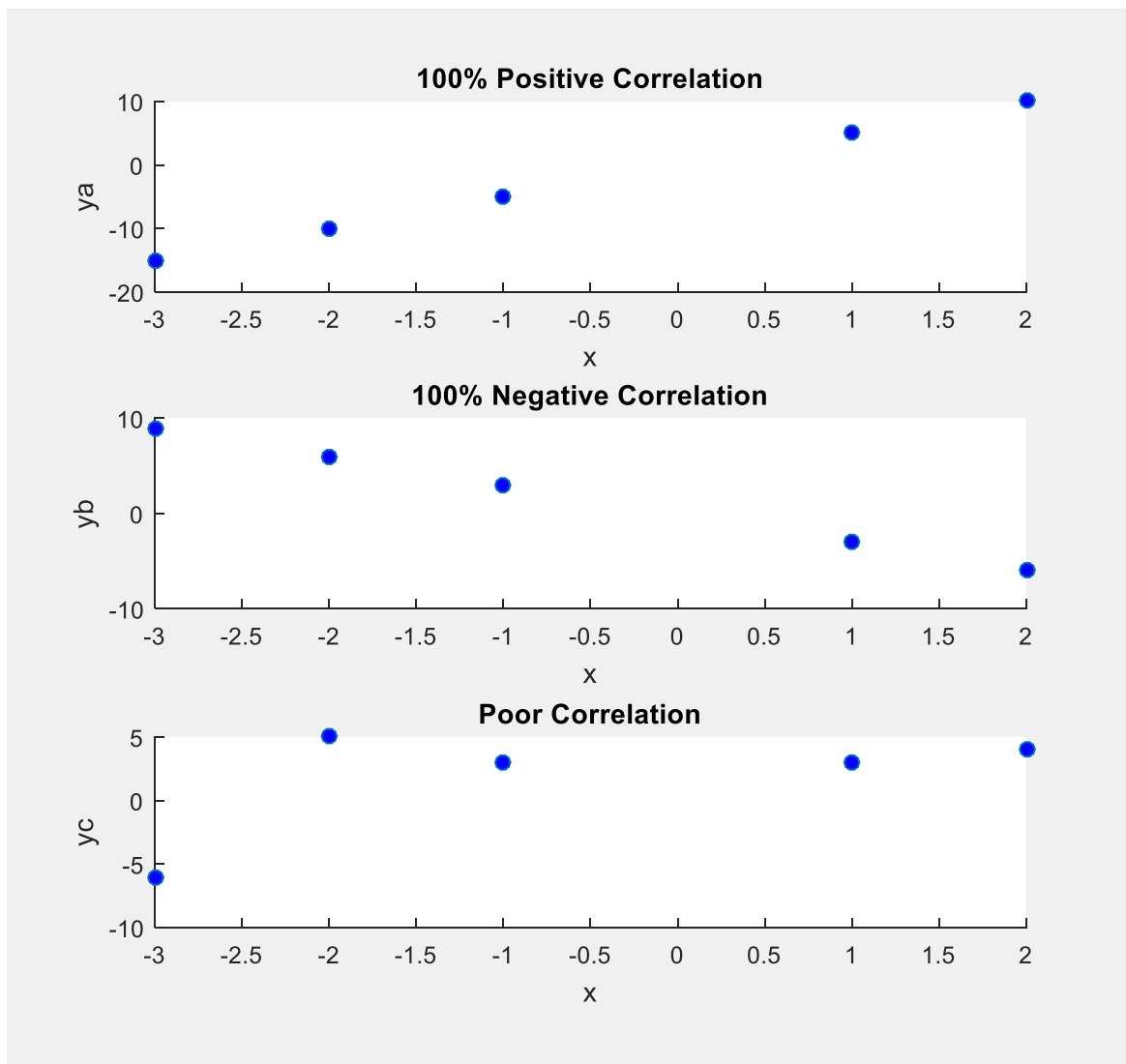


Figure 4.2: Scatterplots for Example 4.21

Answers to Chapter 4 Challenge Questions

Question 4.1 What is the range of numbers (negative to positive) that can be represented using the 16-bit twos complement format?

The largest positive number would be $0111\ 1111\ 1111\ 1111 = 32767$

The most negative number would be $1000\ 0000\ 0000\ 0000 = -32768$

So, the range is -32768 to 32767.

Question 4.2 Can 0.1 be represented exactly using Q15 format? If not, how close can we get?

$$0.1 \cdot 2^{15} = 3276.8$$

Since this is not an integer, 0.1 cannot be represented exactly in Q15 format. How close can we get? Round the product to the nearest integer which is $3277 \rightarrow 0000\ 1100\ 1100\ 1101$.

$$3277 / 2^{15} = 0.100006103515625$$

So representing 0.1 in Q15 format will result in a quantization error of $6.103515625 \cdot 10^{-6}$.

Question 4.3 Is it possible to represent 0.1 exactly in IEEE 754 32-bit single precision floating-point format? If not, how close can we get?

$$S = 0$$

The largest power of 2 that does not exceed 0.1 is $2^{-4} = 0.0625$. $0.1/0.0625 = 1.6$.

$$E = -4 + 127 = 123$$

$M = 1.6$. $M - 1 = 0.6$. $0.6 \cdot 2^{23} = 5033164.8$ This value is not an integer and must be rounded so 0.1 cannot be represented exactly in this floating point format.

How close can we get?

$$\begin{aligned}\text{Decimal Value} &= (-1)^S \cdot M \cdot 2^{(E-127)} = (-1)^0 \cdot (1 + 5033165/2^{23}) \cdot 2^{(123-127)} \\ &= 1.0000000149011612\end{aligned}$$

This is certainly closer to 0.1 than the Q15 format but still not exact. The increase in accuracy is due not only to the change in format but also to the increased number of bits.

Question 4.4 What can be said about the stability of discrete systems when the output depends only on the current input and past inputs and has no dependence on past outputs? Why?

A discrete system with an output that depends only on the current input and past inputs with no feedback from the output is always stable. Systems of this type have all N poles at the origin (zero) and therefore satisfy the pole criterion for stability.

To see this, just test an example:

$$y(k) = 0.1x(k) + 0.2x(k-1) - x(k-2) + 0.2x(k-3) + 0.1x(k-4)$$

The transfer function of this system is given by:

$$H(z) = \frac{0.1 + 0.2z^{-1} - z^{-2} + 0.2z^{-3} + 0.1z^{-4}}{1} = \frac{0.1z^4 + 0.2z^3 - z^2 + 0.2z + 0.1}{z^4}$$

This 4th order system has all four poles at the origin and is stable. It should be clear that any example picked with no dependence on past outputs must have all poles at the origin.

Question 4.5 Why would we want to compare one signal to a shifted version of another signal?

It is possible that one signal could be a time-delayed version of the other signal. For example, a radar return signal is a time-delayed (and noisy) version of the original transmitted signal.

Question 4.6 Looking at Example 4.19, how does the correlation of sequences $x(k)$ and $y(k)$ compare to the convolution of sequences $x(k)$ and $y(k)$?

A correlation computation is very similar to a convolution computation except that in a convolution, one of the two sequences is reversed, which is not the case for a correlation. We get a completely different set of numbers but the computation itself is very similar. It is also interesting to note that signal order (which signal is reversed) is not important in a convolution computation but signal order (which signal is shifted) does matter in a correlation computation.

$$R_{xy}(n) = [9 \quad 22 \quad 55 \quad 24 \quad 10] \neq R_{yx}(n) = [10 \quad 24 \quad 55 \quad 22 \quad 9]$$

$$\text{conv}(x,y) = [2 \quad 8 \quad 27 \quad 38 \quad 45] = \text{conv}(y,x) = [2 \quad 8 \quad 27 \quad 38 \quad 45]$$

Chapter 4 Problems

Problem 4.1: Convert the following fractions to Q15 format. Put your final answer in HEX. Also indicate which of the numbers can be represented exactly in Q15 format and which cannot.

(a) 0.125	(d) 0.265625
(b) 0.5625	(e) 0.075
(c) 0.4	(f) 0.7578125

Problem 4.2: Convert each of the following numbers to the 32-bit floating-point format defined in the IEEE 754 Standard. Your final answer should be in HEX. Also, indicate which of the numbers can be represented exactly in 32-bit floating-point format and which cannot.

(a) 53	(d) 85
(b) -159	(e) -1346
(c) 0.175	(f) 0.275

Problem 4.3: Use recursion to find the first eight terms ($y(k)$ for $k = 0 \dots 7$) of the impulse response ($x(k) = \delta(k)$) and the first eight terms of the step response ($x(k) = 1$ for $k \geq 0$) for each of the following discrete systems.

- (a) $y(k) = 0.6x(k) + 0.4x(k-1) + 0.6x(k-2)$
- (b) $y(k) = 0.25x(k) - 0.2x(k-1) + 0.8x(k-2) - 0.4x(k-3) + 0.25x(k-4)$
- (c) $y(k) - 0.5y(k-1) = x(k)$
- (d) $y(k) + 2y(k-1) = x(k)$
- (e) $y(k) + 0.2y(k-1) + 0.26y(k-2) = x(k) - 0.5x(k-1)$

Problem 4.4: For each of the following discrete systems, find the transfer function, the system order, the system poles, the system zeros, and state whether or not the system is stable.

- (a) $y(k) = 0.6x(k) + 0.4x(k-1) + 0.6x(k-2)$
- (b) $y(k) = 0.25x(k) - 0.2x(k-1) + 0.8x(k-2) - 0.4x(k-3) + 0.25x(k-4)$
- (c) $y(k) - 0.5y(k-1) = x(k)$
- (d) $y(k) + 2y(k-1) = x(k)$
- (e) $y(k) + 0.2y(k-1) + 0.26y(k-2) = x(k) - 0.5x(k-1)$

Problem 4.5: For each of the following discrete systems, use MATLAB to find the first eight terms of the impulse response (*dimpulse*) and the first eight terms of the step response (*dstep*). Compare your answers to Problem 4.3.

- (a) $y(k) = 0.6x(k) + 0.4x(k-1) + 0.6x(k-2)$
- (b) $y(k) = 0.25x(k) - 0.2x(k-1) + 0.8x(k-2) - 0.4x(k-3) + 0.25x(k-4)$
- (c) $y(k) - 0.5y(k-1) = x(k)$
- (d) $y(k) + 2y(k-1) = x(k)$
- (e) $y(k) + 0.2y(k-1) + 0.26y(k-2) = x(k) - 0.5x(k-1)$

Problem 4.6: Convolve the following sequences (by hand). Use MATLAB (*conv*) to check your answers.

- (a) $x(k) = \{1 \ -1 \ 3 \ 6\}$ and $h(k) = \{2 \ 4 \ 5\}$
- (b) $x(k) = \{1 \ 1 \ 1 \ 1 \ 1\}$ and $h(k) = \{0.2 \ -0.5 \ -0.5 \ 0.2\}$
- (c) $x(k) = \{-2 \ 1 \ -2 \ 1\}$ and $h(k) = \{0.3 \ 0.5 \ 0.8 \ 0.5 \ 0.3\}$

Problem 4.7: Using MATLAB, find an equation or closed-form expression for the unit step response for each of the following discrete systems. Use your equation to find $y(0)$, $y(1)$, $y(2)$, and $y(3)$ then check your results by using your answers from Problem 4.3 or Problem 4.5.

- (a) $y(k) = 0.6x(k) + 0.4x(k-1) + 0.6x(k-2)$
- (b) $y(k) = 0.25x(k) - 0.2x(k-1) + 0.8x(k-2) - 0.4x(k-3) + 0.25x(k-4)$
- (c) $y(k) - 0.5y(k-1) = x(k)$
- (d) $y(k) + 2y(k-1) = x(k)$
- (e) $y(k) + 0.2y(k-1) + 0.26y(k-2) = x(k) - 0.5x(k-1)$

Problem 4.8: Using MATLAB, compute the cross-correlation for the following sequences and comment on the results.

- (a) $x(k) = \{1 \ -2 \ 5 \ -2 \ 1\}$ and $y(k) = \{-5 \ 10 \ -25 \ 10 \ -5\}$
- (b) $x(k) = \{3 \ -2 \ -5 \ -2 \ 3\}$ and $y(k) = \{0 \ 0 \ 6 \ -4 \ -10 \ -4 \ 6\}$
- (c) $x(k) = \{3 \ -2 \ -5 \ -2 \ 3\}$ and $y(k) = \{0 \ 0 \ 6 \ -4.1 \ -10.5 \ -4 \ 6.2\}$

Problem 4.9: Create the following discrete sequences in MATLAB then compute the cross-correlation for the sequences and comment on the results.

- (a) A 10Hz sine wave and a 10.1 Hz sine wave both sampled at $F_s = 50$ Hz for $\frac{1}{2}$ second.
- (b) A 10Hz sine wave and a 12 Hz sine wave both sampled at $F_s = 50$ Hz for $\frac{1}{2}$ second.
- (c) A 10 Hz sine wave and a second 10 Hz sine wave corrupted by noise both sampled at $F_s = 50$ Hz for $\frac{1}{2}$ second. *Note: To create the noisy signal, first create the sine wave without noise, say x . The noisy signal, y , would be $y = x + 0.3*randn(1, \text{length}(x))$.*

CHAPTER 5 FINITE IMPULSE RESPONSE (FIR) FILTERS

This chapter introduces finite impulse response (FIR) digital filters. Several methods for designing FIR filters are covered. The Filter Design and Analysis Tool (FDAT) is introduced and several filter design examples using this versatile MATLAB® design tool are included.

5.1 FINITE IMPULSE RESPONSE (FIR) DIGITAL FILTERS

FIR (Finite Impulse Response) digital filters can be described by the input/output difference equation:

$$y(k) = b_0x(k) + b_1x(k-1) + b_2x(k-2) + \cdots + b_Nx(k-N) \quad (5.1)$$

- The constants $b_0, b_1, \dots, b_N$ are the filter coefficients. There are $N+1$ filter coefficients.
- N is the filter order. So, the number of filter coefficients = Filter Order + 1.
- Designing an FIR filter involves determining the filter order and a set of filter coefficients that meet the given filter specifications.
- The output of the filter is a multiply and accumulate (MAC) calculation: the current input and N past inputs are multiplied by the filter coefficients and the resulting products are added together.
- To implement the filter, the filter coefficients, the current input, and a finite number of past inputs must be stored in memory. The filter output does not depend on past filter outputs.

Impulse Response of FIR Filters

The impulse response was defined in Chapter 4 as the output response when the input signal is a discrete unit impulse; that is $\delta(k) = 1$ for $k = 0$ and $\delta(k) = 0$ for $k \neq 0$. Also covered in Chapter 4 was the fact that a system with an output that depends only on the current input and past inputs and does not depend on past outputs (a non-recursive system) has an impulse response equal to the coefficients of the difference equation:

$$h(k) = [b_0 \ b_1 \ b_2 \ \dots \ b_N] \quad (5.2)$$

Transfer Function of FIR Filters

The transfer function of an FIR filter can be derived by inspection from the input/output difference equation using the time shift property of Z-transforms.

$$H(z) = \frac{b_0 + b_1z^{-1} + b_2z^{-2} + \cdots + b_Nz^{-N}}{1} = \frac{b_0z^N + b_1z^{N-1} + b_2z^{N-2} + \cdots + b_N}{z^N} \quad (5.3)$$

Stability of FIR Filters

Discrete-time systems are stable if the magnitude of each of the poles is strictly less than one. An N th order FIR filter has N poles at the origin (zero). Therefore, all FIR filters are stable.

This is one of the advantages of FIR filters over other types of filters that can exhibit stability problems when not implemented carefully.

Output of an FIR Filter

The output of an FIR filter can be computed using several different methods including the following:

- Recursively using the input/output difference equation
- Through convolution using $y(k) = x(k)*h(k)$
- Taking an inverse Z-Transform of $Y(z) = X(z)H(z)$
- Using the MATLAB command $y = \text{filter}(\text{num\_tf}, \text{den\_tf}, x)$
- Using Fast Fourier Transforms (FFTs) to be discussed in Chapter 7

Example 5.1: Impulse Response, Transfer Function, and Output Response for FIR Filter

An FIR filter is described by

$$y(k) = \frac{1}{3}x(k) + \frac{1}{3}x(k-1) + \frac{1}{3}x(k-2)$$

- (a) Find the filter order and the impulse response of the filter.
- (b) Find the transfer function for the FIR filter and the filter poles.
- (c) Find the output response for the FIR filter if the input sequence $x(k) = \{1 \ 3 \ 6 \ -1 \ 5\}$.

Solution

- (a) The filter order for the FIR filter is the largest delay on the input so $N = 2$ (2nd order)
The impulse response is equal to the filter coefficients: $h(k) = [1/3 \ 1/3 \ 1/3]$.

- (b) The transfer function is derived by taking the Z-Transform of the difference equation

$$H(z) = \frac{\frac{1}{3} + \frac{1}{3}z^{-1} + \frac{1}{3}z^{-2}}{1} = \frac{\frac{1}{3}z^2 + \frac{1}{3}z + \frac{1}{3}}{z^2}$$

The filter poles (roots of the denominator) are at 0 and 0. FIR filters always have all N poles at the origin.

- (c) The output response can be computed by multiple methods. In this case, convolution (in MATLAB) will be used.

```
x = [1  3  6  -1  5];
h = [1/3 1/3 1/3];
y = conv(x,h)
```

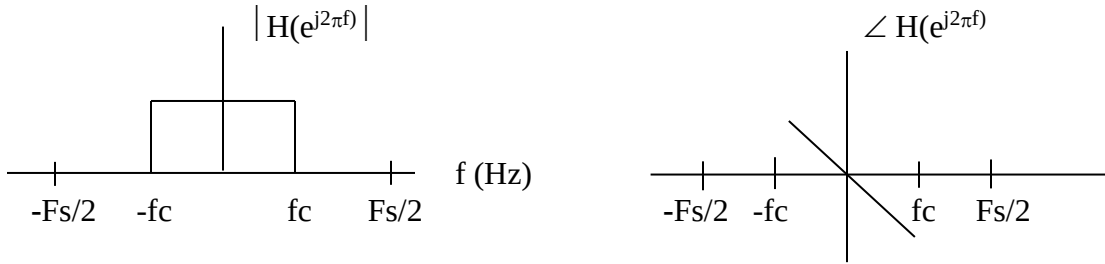
```
y =  0.3333  1.3333  3.3333  2.6667  3.3333  1.3333  1.6667
```

Challenge Question 5.1

Look back at the difference equation in Example 5.1. What does this filter do numerically to an input sequence?

5.2 DESIGNING LOW-PASS FIR FILTERSWindow-Based Design Using Fourier Series

The window-based design begins with an ideal filter then interjects reality to design a physically realizable filter. The magnitude and phase response for an ideal low-pass digital filter is shown below.



f_c = desired analog cutoff frequency
 F_s = sampling frequency.

The impulse response for this ideal filter can be computed using an inverse Fourier Series (equation 5.4).

$$h(k) = \int H(e^{j2\pi f k}) e^{j2\pi f k} df \quad (5.4)$$

This results in the impulse response for an ideal low-pass filter (details of the integration are not included here) shown in equation 5.5.

$$h(k) = \frac{\sin(2\pi \frac{f_c}{F_s} k)}{\pi k} = 2 \left(\frac{f_c}{F_s} \right) \text{sinc} \left(2 \cdot \frac{f_c}{F_s} k \right) \quad k = 0, \pm 1, \pm 2, \dots, \pm \infty$$

$$\text{sinc}(x) = \sin(\pi x) / (\pi x) \quad (5.5)$$

We now have the filter coefficients, $h(k)$, required to create a perfect low-pass digital filter. Unfortunately, even though we can describe and compute the filter coefficients for an ideal low-pass filter, we can't build one. Why? Two reasons: there are an infinite number of filter coefficients, and the impulse response starts before time equals zero (a non-causal system). Here is the required computation for an ideal low-pass filter output:

$$h(k) = \dots h(-2)x(k+2) + h(-1)x(k+1) + h(0)x(k) + h(1)x(k-1) + h(2)x(k-2) + \dots$$

To compute the filter output, we would need to do an infinite number of multiplies and adds every single sampling instant (clearly not possible). We would also need to know all future values of the input signal; whereas, in real-time applications, we only have access to the current filter input and past filter inputs.

These problems can be addressed by first cutting off $h(k)$ to make it finite, then shifting $h(k)$ to the right to make it causal. Obviously, we will no longer have an ideal low-pass filter.

Truncating the filter coefficients to $k = -N/2, \dots, 0, \dots, +N/2$ then shifting to the right by $N/2$ gives the filter coefficients for an N th order physically realizable low-pass filter:

$$h(k) = \frac{\sin\left(2\pi\left(\frac{f_c}{F_s}\right)\left(k - \frac{N}{2}\right)\right)}{\pi\left(k - \frac{N}{2}\right)} = 2\left(\frac{f_c}{F_s}\right) \operatorname{sinc}\left(2\left(\frac{f_c}{F_s}\right)\left(k - \frac{N}{2}\right)\right) \quad k = 0, 1, 2, \dots, N$$

$$\text{Note: } h\left(\frac{N}{2}\right) = 2\left(\frac{f_c}{F_s}\right) \quad (5.6)$$

The process of truncating $h(k)$ to allow for a finite number of computations and shifting $h(k)$ to make a causal system is illustrated in Figure 5.1. Figure 5.1a shows the ideal filter coefficients extended only to $k = \pm 20$. The ideal filter coefficients would actually extend out to $\pm \infty$ but that's a little difficult to graph. In Figure 5.1b, the filter is made finite order by truncating the filter coefficients beyond $k = \pm 10$. Finally, in Figure 5.1c, the filter coefficients are shifted to the right by 10 to produce a 20th order physically realizable (causal) FIR low-pass filter.

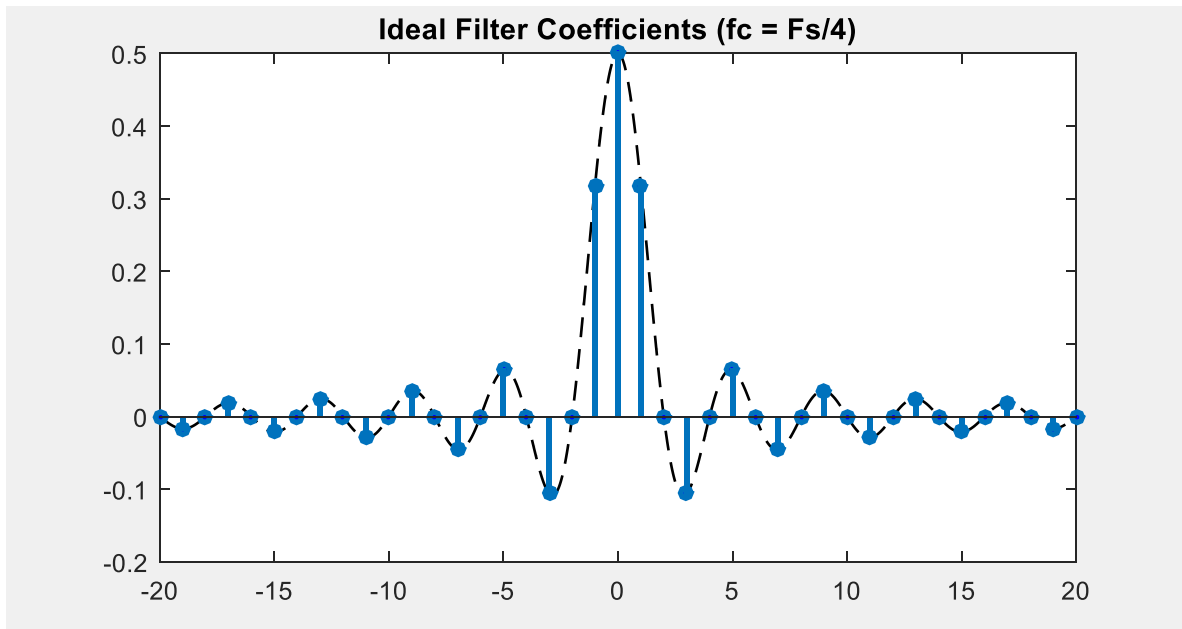


Figure 5.1a: Ideal Low-Pass Filter Coefficients ($f_c = F_s/4$)

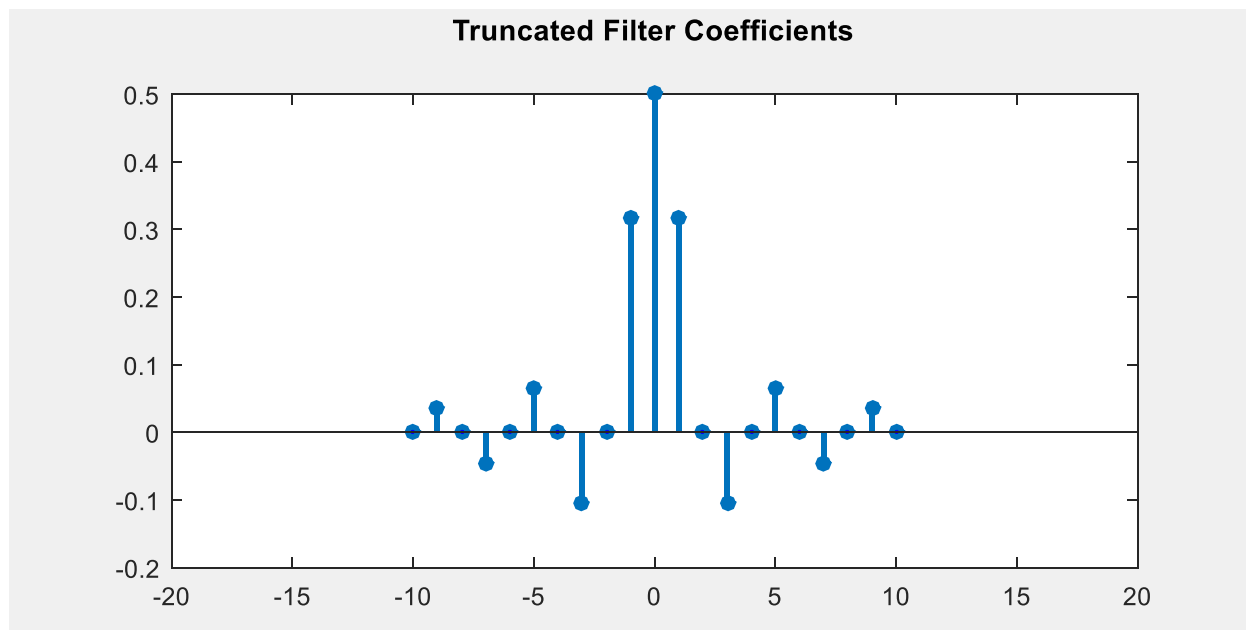


Figure 5.1b: Truncated Filter Coefficients

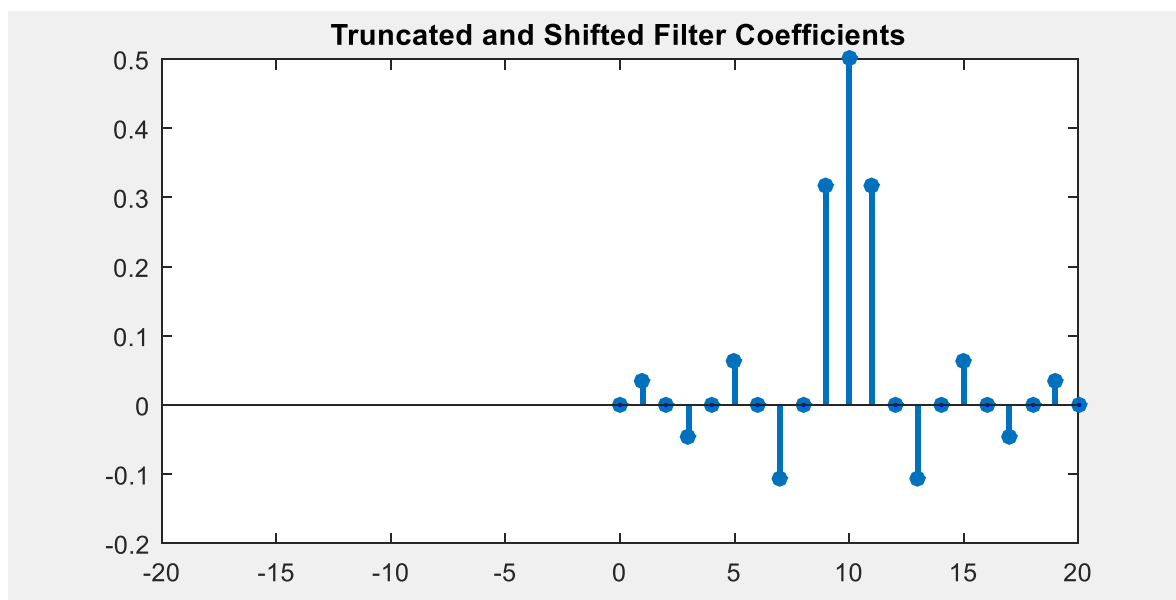


Figure 5.1c: Truncated and Shifted Filter Coefficients

Example 5.2: Design of Low-pass FIR Filter using Fourier Series

- (a) Compute the filter coefficients for a 10th order digital low-pass FIR filter assuming a sampling frequency of 44.1 kHz and a cut-off frequency of 20 kHz.
- (b) Write the difference equation for computing the filter output $y(k)$.
- (c) Find the transfer function for the filter.
- (d) Plot the frequency response for the 10th order FIR filter and compare it to an ideal low-pass filter.

Solution

- (a) The filter coefficients can be computed from the truncated and shifted ideal impulse response (equation 5.6).

$$h(k) = \frac{\sin(2\pi \left(\frac{f_c}{F_s}\right) \left(k - \frac{N}{2}\right))}{\pi \left(k - \frac{N}{2}\right)} = 2 \left(\frac{f_c}{F_s}\right) \text{sinc} \left(2 \left(\frac{f_c}{F_s}\right) \left(k - \frac{N}{2}\right)\right) \quad k = 0, 1, 2, \dots, N$$

$$h(k) = 2 \left(\frac{20}{44.1}\right) \text{sinc} \left(2 \left(\frac{20}{44.1}\right) (k - 5)\right) \quad k = 0, 1, 2, \dots, 10$$

These coefficients can be found using a calculator (make sure the calculator is in radian mode) or perhaps more easily using MATLAB:

```
k=0:1:10;
h=2*(20/44.1)*sinc(2*(20/44.1)*(k - 5))

h = 0.0633 -0.0732  0.0815 -0.0878  0.0917  0.9070  0.0917
    -0.0878  0.0815  -0.0732  0.0633
```

- (b) The filter coefficients for an FIR filter are the numerical coefficients of the input/output difference equation. Therefore the difference equation for computing the filter output is:

$$y(k) = 0.0633x(k) - 0.0732x(k-1) + 0.0815x(k-2) - 0.0878x(k-3) \\ + 0.0917x(k-4) + 0.9070x(k-5) + 0.0917x(k-6) - 0.0878x(k-7) \\ + 0.0815x(k-8) - 0.0732x(k-9) + 0.0633x(k-10)$$

The filter output can be computed recursively from the current input and the ten most recent past inputs using a finite number of computations (eleven multiplications and 10 additions).

- (c) The transfer function is derived by taking the Z-Transform of the difference equation:

$$H(z) = [0.0633z^{10} - 0.0732z^9 + 0.0815z^8 - 0.0878z^7 + 0.0917z^6 + 0.9070z^5 + \\ 0.0917z^4 - 0.0878z^3 + 0.0815z^2 - 0.0732z + 0.0633] / z^{10}$$

(d) The frequency response can be easily plotted in MATLAB using the *freqz* command:

```
num_tf = h;
den_tf = zeros(size(h)); den_tf(1) = 1;
% Creates den_tf = [1 0 ... 0] of correct size
Fs = 44100;
[H,f] = freqz(num_tf,den_tf,200,Fs);
% Calculates frequency response for 200 frequencies
subplot(2,1,1); plot(f/1000, 20*log10(abs(H)));
title('Magnitude Response (dB)'); grid
subplot(2,1,2); plot(f/1000, angle(H));
title('Phase Response (rad)'); grid
xlabel('Frequency (kHz)');
```

The resulting frequency response plot for the 10th order FIR filter is shown in Figure 5.2. How does the response compare to an ideal low-pass filter with a cutoff frequency of 20 kHz? The ideal low-pass filter would have a flat magnitude response over the entire passband; that is, the magnitude would be 0 dB (or 1) from 0 to 20 kHz then drop immediately to $-\infty$ dB (or 0) for all frequencies above 20 kHz. The 10th order FIR filter has ripple (oscillation) in the passband and the magnitude begins gradually dropping off at about 17.5 kHz. The 3 dB cutoff frequency is roughly 18 kHz. The phase response of the 10th order FIR matches the ideal response – it is linear. Another nice feature of FIR filters (in addition to inherent stability) is a linear (ideal) phase response.

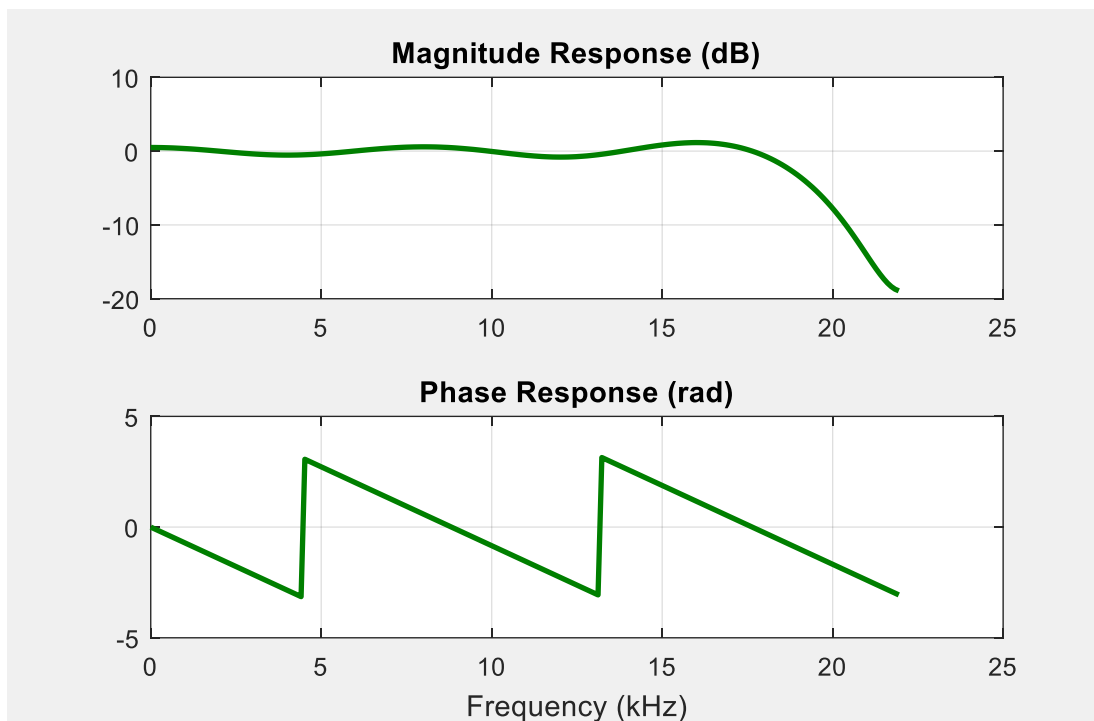


Figure 5.2: Frequency Response for 10th order FIR Filter

Example 5.3: Frequency Response of Low-Pass FIR Filters

Plot the frequency response (magnitude only) for a 30th order and 50th order FIR filter assuming a sampling frequency of 44.1 kHz and a cutoff frequency of 20 kHz.

Solution

The following MATLAB commands compute the filter coefficients, the corresponding frequency response, then plot the magnitude response for the 30th and 50th order filters.

```
k=0:1:30; h_30=2*(20/44.1)*sinc(2*(20/44.1)*(k-15));  
k=0:1:50; h_50=2*(20/44.1)*sinc(2*(20/44.1)*(k-25));  
Fs = 44100;  
num_30 = h_30; den_30 = zeros(size(h_30)); den_30(1) = 1;  
num_50 = h_50; den_50 = zeros(size(h_50)); den_50(1) = 1;  
[H_30,f_30] = freqz(num_30,den_30,200,Fs);  
[H_50,f_50] = freqz(num_50,den_50,200,Fs);  
subplot(2,1,1);plot(f_30/1000,20*log10(abs(H_30)));  
title('Magnitude (dB) of 30th Order FIR Filter');grid  
subplot(2,1,2);plot(f_50/1000,20*log10(abs(H_50)));  
title('Magnitude (dB) of 50th Order FIR Filter');grid  
xlabel('Frequency (kHz)')
```

The magnitude responses for the two filters are shown in Figure 5.3. Notice that the higher order filter has a flatter passband and a narrower transition region (i.e., closer to ideal) than the lower order filter.

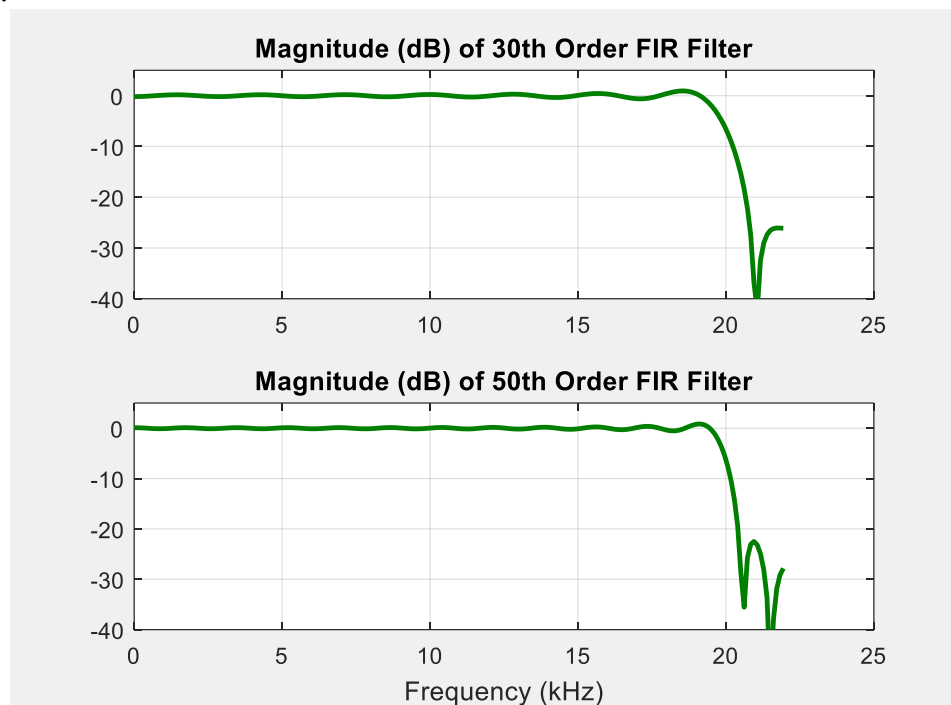


Figure 5.3: Magnitude Response for FIR Filters (Example 5.3)

Challenge Question 5.2

Increasing filter order clearly improves the frequency response and brings the filter closer to an ideal low-pass filter. However, increasing the filter order will require more processing (multiplications and additions) to produce the filter output. How many multiplications are required for an N^{th} order FIR filter?

Windows

Increasing the order of the FIR filter definitely improves the frequency response, but there are still ripples present in both the passband and the stopband. These ripples are due to the abrupt truncation of the filter coefficients and can be reduced by using a window function. A window function gradually attenuates the filter coefficients at the edges rather than abruptly truncating the coefficients. Figure 5.4 shows a variety of window functions for a 20th order FIR filter.

The rectangular window simply represents abrupt truncation of the ideal filter coefficients. The Hamming, Blackman, and Kaiser windows gradually attenuate the filter coefficients; each using a different function to do so. Each of these windows will reduce the ripple effect caused by abrupt truncation. However, the transition from passband to stopband will widen with a window function (always a catch or compromise). Example 5.4 illustrates the effects of the windowing on the frequency response of the filter.

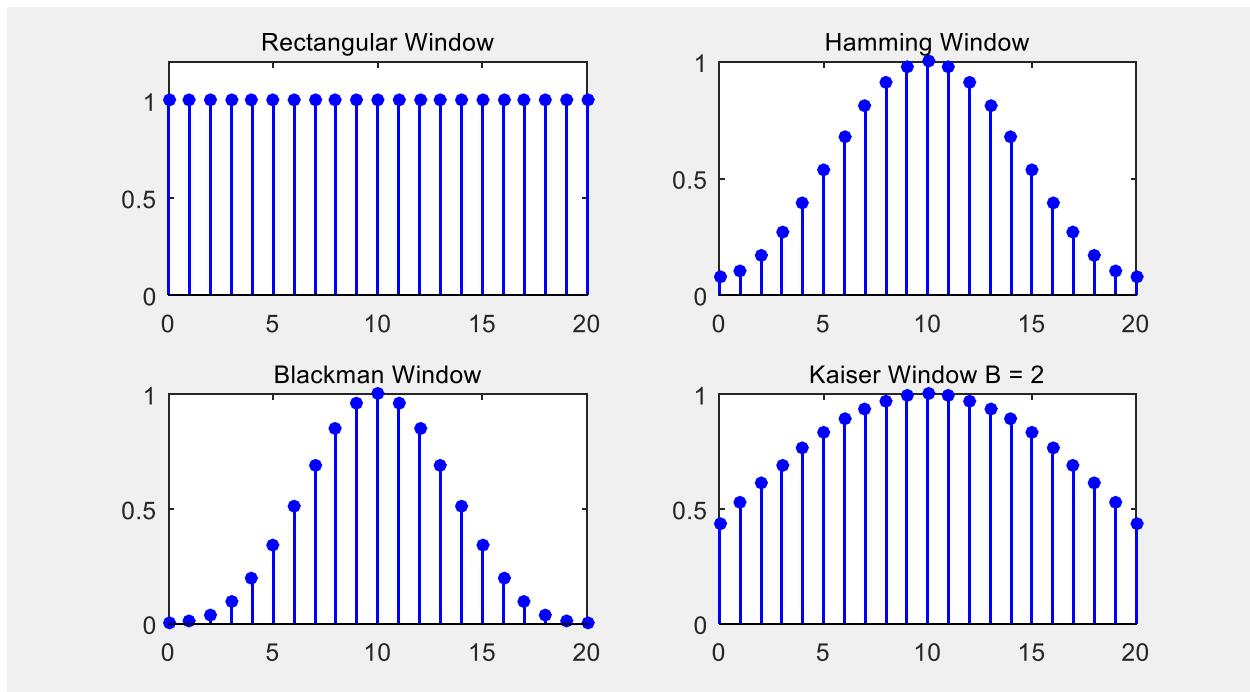


Figure 5.4: Examples of Window Functions

Example 5.4: FIR Filter Design using Windows

Design a 50th order low-pass FIR filter using the Hamming window, the Blackman window, and the Kaiser window. Assume that the sampling frequency is 44.1 kHz and the desired cutoff frequency is 20 kHz. Plot the frequency response (magnitude only) for each filter then compare each filter to the 50th order FIR filter designed in Example 5.3 using abrupt truncation.

Solution

MATLAB code:

```
%Calculate filter coefficients:
k = 0:1:50; h = 2*(20/44.1)*sinc(2*(20/44.1)*(k-25));

% Multiply the coefficients by each window function:
h_hamming = h.*window(@hamming,51)';
h_blackman = h.*window(@blackman,51)';
h_kaiser = h.*window(@kaiser,51,3)';

% Transfer Function for Hamming Filter:
num_hamm = h_hamming;
den_hamm = zeros(size(h_hamming)); den_hamm(1)=1;
% Transfer Function for Blackman Filter:
num_black = h_blackman;
den_black = zeros(size(h_blackman)); den_black(1)=1;
% Transfer Function for Kaiser Filter:
num_kais = h_kaiser;
den_kais = zeros(size(h_kaiser)); den_kais(1)=1;

% Compute & plot frequency response for each FIR filter:
Fs = 44100;
[H_hamm,f] = freqz(num_hamm,den_hamm,200,Fs);
subplot(3,1,1); plot(f/1000,20*log10(abs(H_hamm)));
title('Magnitude (dB) Hamming Window'); grid
[H_black,f] = freqz(num_black,den_black,200,Fs);
subplot(3,1,2); plot(f/1000,20*log10(abs(H_black)));
title('Magnitude (dB) Blackman Window'); grid
[H_kais,f] = freqz(num_kais,den_kais,200,Fs);
subplot(3,1,3); plot(f/1000,20*log10(abs(H_kais)));
title('Magnitude (dB) Kaiser Window'); grid
xlabel('Frequency (kHz)')
```

The frequency response for each of the windowed filters is shown in Figure 5.5. Each of the window functions reduces the amount of passband and stopband ripple exhibited in the 50th order FIR filter in Example 5.3 using abrupt truncation of the ideal filter coefficients. The Hamming and Blackman windows show very little ripple but exhibit a wider transition region from

passband to stopband. The Kaiser window creates a filter with a sharper transition than the other two windowed filters but there is some remaining ripple in the passband and stopband of this filter.

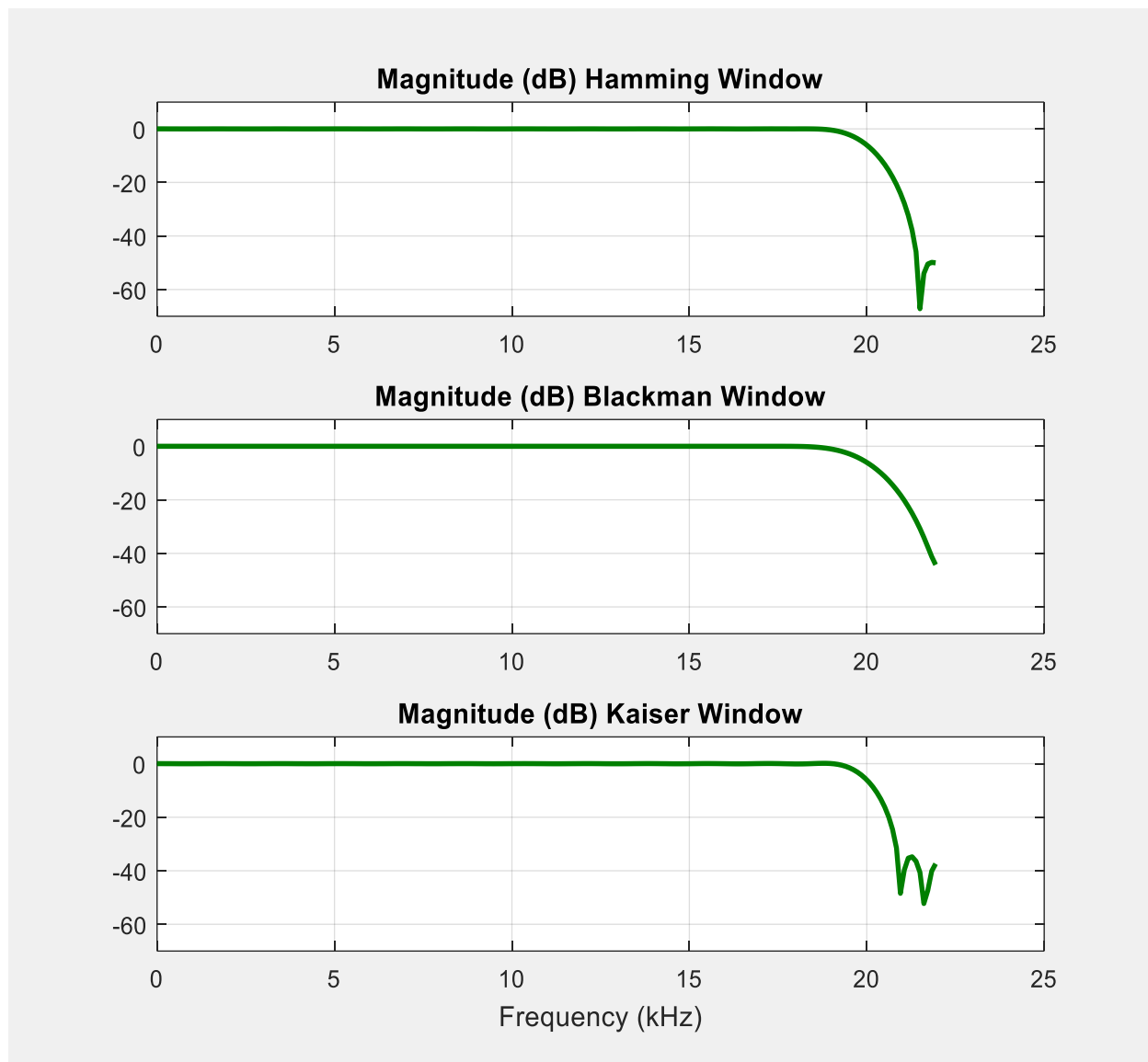


Figure 5.5: Frequency Responses for Various Window Types (Example 5.4)

Half of the filter coefficients for each windowing function used in Example 5.4 are shown in Table 5.1. Due to symmetry, it is only necessary to show half the coefficients. Note that the coefficients are very small and in many cases don't vary much from window to window. However, the frequency responses are dramatically different. Small changes in the filter coefficients can lead to big changes in filter performance – an extremely important fact to keep in mind!

Table 5.1: Filter Coefficients for Example 5.4

	No Window	Hamming Window	Blackman Window	Kaiser Window
b(0)	0.01084072	0.000867258	-1.50445E-19	0.002221098
b(1)	-0.008811246	-0.00073686	-1.25939E-05	-0.002138065
b(2)	0.005826503	0.000550323	3.38692E-05	0.00164337
b(3)	-0.002054463	-0.000230722	-2.75899E-05	-0.000663284
b(4)	-0.002259125	-0.000309272	-5.58293E-05	-0.000824127
b(5)	0.006803149	0.001141923	0.000273574	0.00277317
b(6)	-0.011218913	-0.002296225	-0.000679181	-0.005060409
b(7)	0.015122244	0.003731943	0.001305012	0.007482408
b(8)	-0.018128538	-0.005321083	-0.002139607	-0.009762738
b(9)	0.0198791	0.006841224	0.003103454	0.011568659
b(10)	-0.020066987	-0.007983695	-0.004028852	-0.012536774
b(11)	0.018460564	0.008377489	0.004650719	0.01230579
b(12)	-0.014923044	-0.007627412	-0.004614736	-0.010553504
b(13)	0.00942657	0.005362622	0.00350693	0.007034359
b(14)	-0.002059688	-0.001289767	-0.000904838	-0.001613547
b(15)	-0.006972481	-0.004756262	-0.003554481	-0.005706333
b(16)	0.017355583	0.012771253	0.01009914	0.014769501
b(17)	-0.028680625	-0.022556748	-0.018752857	-0.025264635
b(18)	0.040464999	0.033716046	0.029285389	0.036737115
b(19)	-0.052178722	-0.04567337	-0.041195495	-0.048615608
b(20)	0.063274278	0.057715495	0.053734406	0.060251151
b(21)	-0.073218182	-0.069052146	-0.065971008	-0.070965542
b(22)	0.081522252	0.078888853	0.076892255	0.080104686
b(23)	-0.087772578	-0.086504111	-0.085525259	-0.087091929
b(24)	0.091654292	0.09132184	0.091062572	0.091476241
b(25)	0.907029478	0.907029478	0.907029478	0.907029478
b(26) = b(24) ⋮				

There are many different window functions to choose from. MATLAB offers seventeen different options for windows. In addition, the Filter Design and Analysis Tool (FDAT) allows the user to create a custom window. All window functions attempt to deal with the compromise between a sharp transition between passband and stopband and ripple in the stopband and passband.

The Hamming window function was graphed in Figure 5.4 and is described by the equation:

$$\text{Hamming Window: } w(k) = 0.54 - 0.46 \cos(2\pi k/N) \quad k = 0, \dots, N \quad (5.7)$$

The Blackman window tweaks the Hamming function somewhat to give better stopband attenuation and less passband ripple. However, the Blackman window results in a filter with a wider transition from passband to stopband (a filter using a Hamming window has approximately 20% faster roll off than a filter designed with the Blackman window). The Blackman window function is graphed in Figure 5.4. The coefficients can be calculated from the following equation:

$$\text{Blackman Window: } w(k) = 0.42 - 0.5 \cos(2\pi k/N) + 0.08 \cos(4\pi k/N) \quad k = 0, \dots, N \quad (5.8)$$

The Kaiser window coefficients are computed as follows:

$$\text{Kaiser Window: } w(k) = \frac{I_0\left(\pi\beta\sqrt{1 - \left(\frac{2k}{N}\right)^2}\right)}{I_0(\pi\beta)} \quad k = 0, \dots, N$$

where $I_0(\cdot)$ is a modified Bessel function (5.9)

The parameter, β , is a design parameter that allows a tradeoff between ripple and transition. If β is chosen to be 0, the Kaiser window becomes a rectangular window. If $\beta = 5.44$, the Kaiser window becomes a Hamming window. The Kaiser window coefficients are shown in Figure 5.4 for $\beta = 2$.

Optimal Design of FIR Filters

In the windows-based design of FIR filters, the filter coefficients representing an ideal filter are truncated and shifted to make the filter finite order and causal, then adjusted with a window function to improve the filter characteristics. In optimal design, the coefficients of FIR filters are determined using numerical algorithms that minimize, in some sense, the difference between the ideal frequency response and the actual frequency response.

$$\text{Minimize } E = \mathcal{F} [H_{\text{ideal}}(f) - H_{\text{actual}}(f)] \quad (5.10)$$

For example, the least squares approach minimizes the square of the difference between the ideal and the actual frequency response. The Parks-McClellan approach utilizes a Remez algorithm to create an optimal elliptic FIR filter. The Remez algorithm was designed to find a polynomial of specified order that would provide a best fit to a function over a given range. In both of these optimal approaches (least squares and elliptic), weighting functions can be introduced that allow the designer to put heavier emphasis on a match to the ideal filter in certain frequency ranges. For example, putting a higher weighting in the passband means matching the ideal filter in the passband is more critical than a match to the ideal filter in the stopband.

The optimization algorithms are extremely complicated but yield very nice filter responses. There are several good software programs, including MATLAB, for calculating optimal FIR filter coefficients. The filter order, N , and the weighting functions need to be adjusted to meet

the filter performance specifications. These optimal algorithms will be explored further in Section 5.4 using the Filter Design and Analysis Tool (FDAT) in MATLAB.

Frequency Sampling Approach for FIR Filter Design

The final method for FIR filter design to be discussed in this section uses frequency sampling. The method is fairly simple to explain.

- Start with a desired frequency response, $H_{\text{des}}(f)$.
- Sample the frequency response with sufficient resolution in the frequency domain (typically 512 equally spaced samples).
- Do an inverse Fast Fourier Transform (Chapter 7) on the frequency samples to produce the filter coefficients, $h(k)$. The number of filter coefficients is exactly equal to the number of samples taken in the frequency domain.
- Window $h(k)$ if necessary to reduce the filter order to an acceptable size.

This approach works well when the desired filter is some sort of a custom filter; that is, it doesn't look like a typical low-pass, high-pass, band-pass, or notch filter.

5.3 DESIGNING HIGH-PASS, BAND-PASS, AND NOTCH FIR FILTERS

In the previous section, several low-pass FIR filters were designed. High-pass, band-pass, and notch filters can easily be designed by transforming the filter specifications to an “equivalent” low-pass filter, designing the low-pass filter, then transforming the low-pass filter coefficients to the desired filter type. Figure 5.6 shows how high-pass, band-pass, and notch filters are translated into “equivalent” low-pass filters.

Recall that the low-pass filter coefficients (using the windows-based Fourier Series approach) were computed as:

$$h_{\text{LP}}(k) = 2 \left(\frac{f_c}{F_s} \right) \text{sinc} \left(2 \left(\frac{f_c}{F_s} \right) \left(k - \frac{N}{2} \right) \right) \quad k = 0, 1, 2, \dots, N$$

Using Figure 5.6 as a guide, the transformations needed to get the other filter types are:

$$h_{\text{HP}}(k) = \begin{cases} 1 - h_{\text{LP}}(k) = 1 - 2 \left(\frac{f_h}{F_s} \right) & k = \frac{N}{2} \quad (\text{N must be even}) \\ -h_{\text{LP}}(k) = -2 \left(\frac{f_h}{F_s} \right) \text{sinc} \left(2 \left(\frac{f_h}{F_s} \right) \left(k - \frac{N}{2} \right) \right) & k \neq N/2 \end{cases} \quad (5.11)$$

$$h_{\text{BP}}(k) = 2 \cos \left(2\pi \left(\frac{f_0}{F_s} \right) \left(k - \frac{N}{2} \right) \right) \cdot \left(\frac{BW}{F_s} \right) \cdot \text{sinc} \left(\left(\frac{BW}{F_s} \right) \left(k - \frac{N}{2} \right) \right) \quad k = 0, 1, 2, \dots, N \quad (5.12)$$

$$h_{Notch}(k) = \begin{cases} 1 - h_{BP}(k) = 1 - 2 \left(\frac{BW}{F_s} \right) & k = \frac{N}{2} \quad (N \text{ must be even}) \\ -h_{BP}(k) = -2 \cos \left(2\pi \left(\frac{f_0}{F_s} \right) \left(k - \frac{N}{2} \right) \right) \cdot \left(\frac{BW}{F_s} \right) \cdot \text{sinc} \left(\left(\frac{BW}{F_s} \right) \left(k - \frac{N}{2} \right) \right) & k \neq \frac{N}{2} \end{cases} \quad (5.13)$$

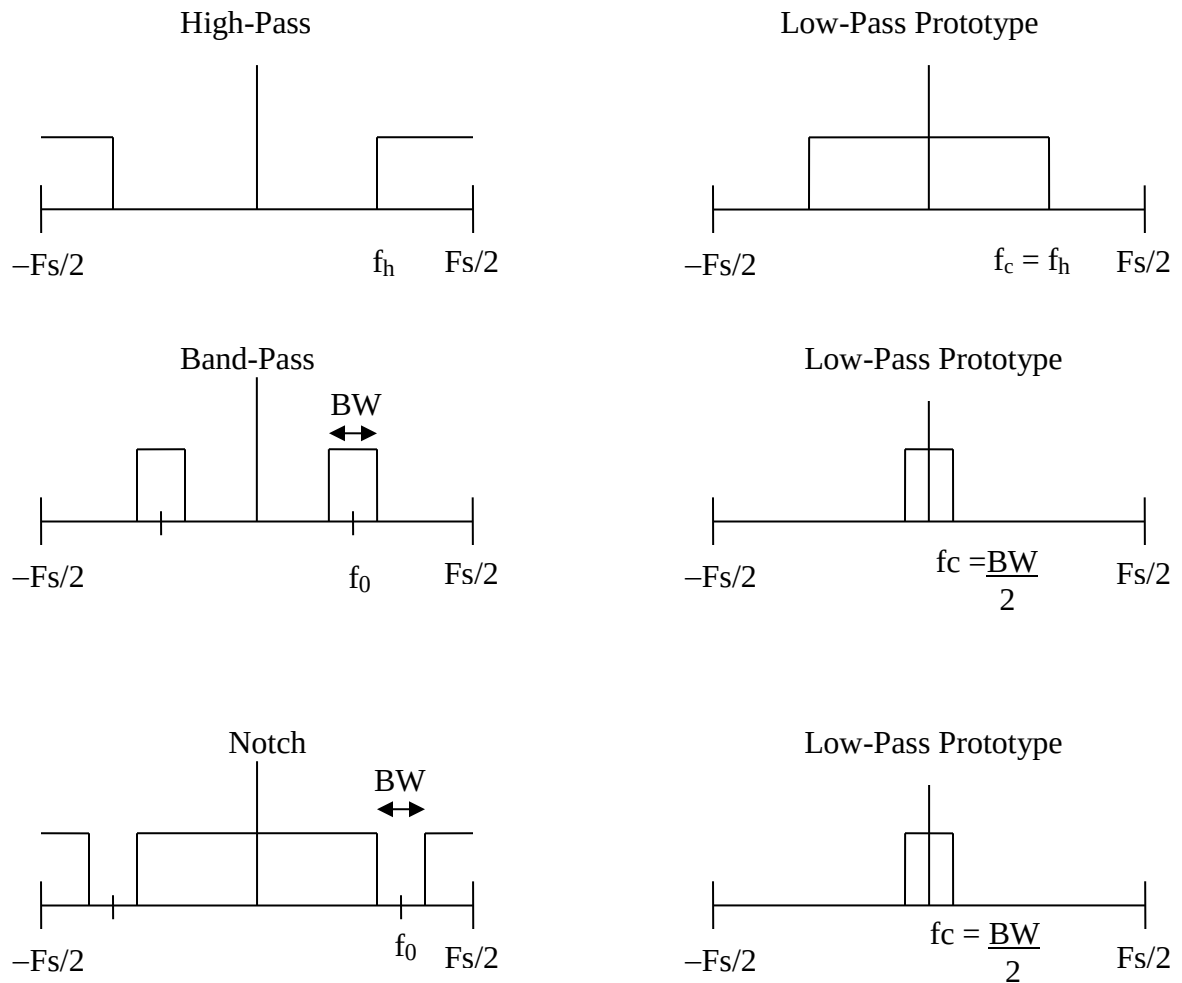


Figure 5.6: Transforming Other Filter Types to Low-Pass Specifications

Windowing can easily be added to the filter design by multiplying the resulting filter coefficients by the desired window function in exactly the same way the low-pass filter coefficients were windowed in Example 5.4

Example 5.5: High-Pass, Band-Pass, and Notch FIR Filter Design

Assume a sampling frequency of 48 kHz. Design the following filters and plot the frequency response (magnitude only).

- A 50th order high-pass filter with a cutoff frequency of 16 kHz.
- A 50th order band-pass filter with a center frequency of 10 kHz and a bandwidth of 4kHz.
- A 50th order notch filter with a center frequency of 10 kHz and a bandwidth of 4 kHz.

Solution

- The filter coefficients for the high-pass filter are calculated using the formula in equation 5.11:

$$h_{HP}(k) = -1 \cdot h_{LP}(k) \quad \text{for all } k \neq \frac{N}{2}$$

$$h_{HP}(k) = 1 - h_{LP}(k) = 1 - 2f_h/F_s \quad \text{for } k = \frac{N}{2}$$

Using MATLAB:

```
Fs = 48000; % Sampling Frequency
fh=16000; % High-Pass Cutoff Frequency
k=0:1:50;
N = 50;
h_HP = -2*(fh/Fs)*sinc(2*(fh/Fs)*(k - N/2));

% Now fix the coefficient for k = N/2
% Be careful here with indexing - see note below!
h_HP(26)= 1 - 2*fh/Fs;

num_HP = h_HP;
den_HP = zeros(size(h_HP)); den_HP(1)=1;
[H_HP,f_HP]=freqz(num_HP,den_HP,200,Fs);
subplot(3,1,1);plot(f_HP/1000,20*log10(abs(H_HP)));
title('High Pass Filter')
```

Note: According to the formula, $h_{HP}(N/2) = h_{HP}(25) = 1 - 2f_h/F_s$. So why in the MATLAB code do we use the command $h_{HP}(26) = 1 - 2f_h/F_s$? We start numbering (indexing) the filtering coefficients at 0; that is, $h_{HP}(0), h_{HP}(1), \dots, h_{HP}(50)$. However, in MATLAB, vector indexing must start at 1. Therefore $h_{HP}(0)$ is the 1st entry in the vector h_{HP} which makes $h_{HP}(25)$ the 26th entry in the MATLAB vector h_{HP} . So, exercise caution!

(b) The filter coefficients for the band-pass filter are calculated using the formula in equation 5.12:

$$h_{BP}(k) = 2 \cos \left(2\pi \left(\frac{f_0}{F_s} \right) \left(k - \frac{N}{2} \right) \right) \cdot \left(\frac{BW}{F_s} \right) \cdot \text{sinc} \left(\left(\frac{BW}{F_s} \right) \left(k - \frac{N}{2} \right) \right) \quad k = 0, 1, 2, \dots, N$$

Using MATLAB:

```
Fs=48000; % Sampling Frequency
fo=10000; % Center of Pass-Band Filter
BW=4000; % Bandwidth
k=0:1:50;
N=50;
h_BP = 2*cos(2*pi*(fo/Fs)*(k - N/2))*(BW/Fs).*sinc((BW/Fs)*(k-N/2));
num_BP = h_BP;
den_BP = zeros(size(h_BP)); den_BP(1)=1;
[H_BP,f_BP]=freqz(num_BP,den_BP,200,Fs);
subplot(3,1,2);plot(f_BP/1000,20*log10(abs(H_BP)));
title('Band Pass Filter')
```

(c) The filter coefficients for the notch filter are calculated using the formula in equation 5.13:

$$h_{\text{Notch}}(k) = -1 \cdot h_{BP}(k) \quad \text{for all } k \neq \frac{N}{2}$$

$$h_{\text{Notch}}(k) = 1 - h_{BP}(k) = 1 - 2BW/F_s \quad \text{for } k = \frac{N}{2}$$

Using MATLAB:

```
k=0:1:50;
h_N = (-1)* h_BP;

% Correct the k = N/2 filter coefficient
% Same issue as with High Pass - be careful with indexing!
h_N(N/2 + 1)= 1 - 2*BW/Fs;

num_N = h_N;
den_N = zeros(size(h_N)); den_N(1)=1;
[H_N,f_N]=freqz(num_N,den_N,200,Fs);
subplot(3,1,3);plot(f_N/1000,20*log10(abs(H_N)));
title('Notch Filter')
xlabel('Frequency (kHz)')
```

The frequency responses of the three filter types are shown in Figure 5.7. These filters were all designed with a rectangular window (abrupt truncation). The ripple in the response could be reduced by using a windowing function as in Example 5.4.

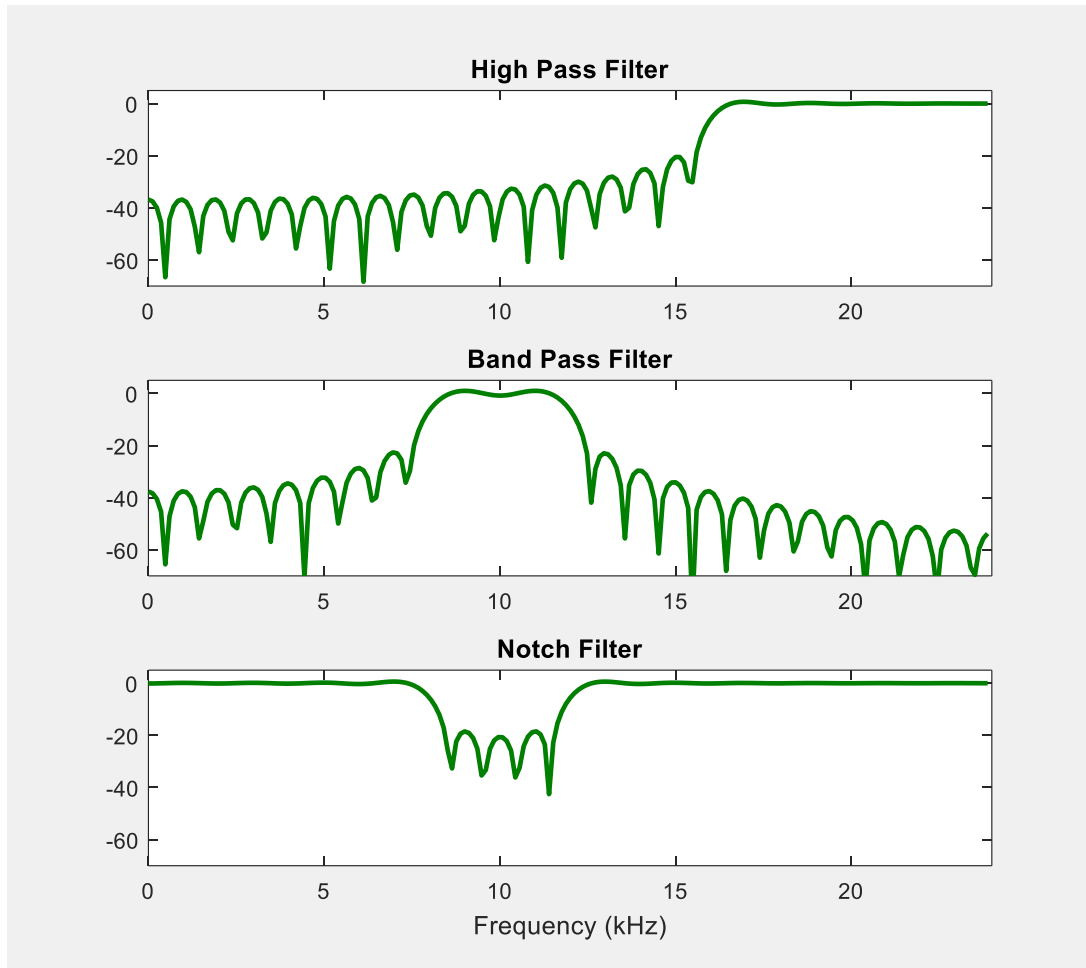


Figure 5.7: Filter Responses for Example 5.5

5.4 DESIGNING FIR FILTERS USING FDAT

MATLAB has a very nice tool for designing digital filters, the Filter Design and Analysis Tool (FDAT). It can be launched in MATLAB by typing `fdatool` in the command window. Figure 5.8 shows a screen shot of the Filter Design and Analysis Tool. The FDAT has several features including:

1. Choice of Filter Response Type (Low-pass, High-pass, Band-pass, Notch, and a few others)
2. Design Choice (FIR Windowing Design, FIR Optimal Design, IIR)
3. Filter Specifications Options (Sampling Frequency, Cutoff Frequencies, Center Frequencies, etc)
4. Ability to specify Stopband Attenuation and Pass-band Ripple constraints
5. Choice of Filter Order, N . For some designs, the FDAT will compute the minimum order to achieve the filter specifications.

- Ability to display magnitude response, phase response, group delay, phase delay, filter impulse response, filter step response, pole/zero plot, and list of filter coefficients using the icons across the top.

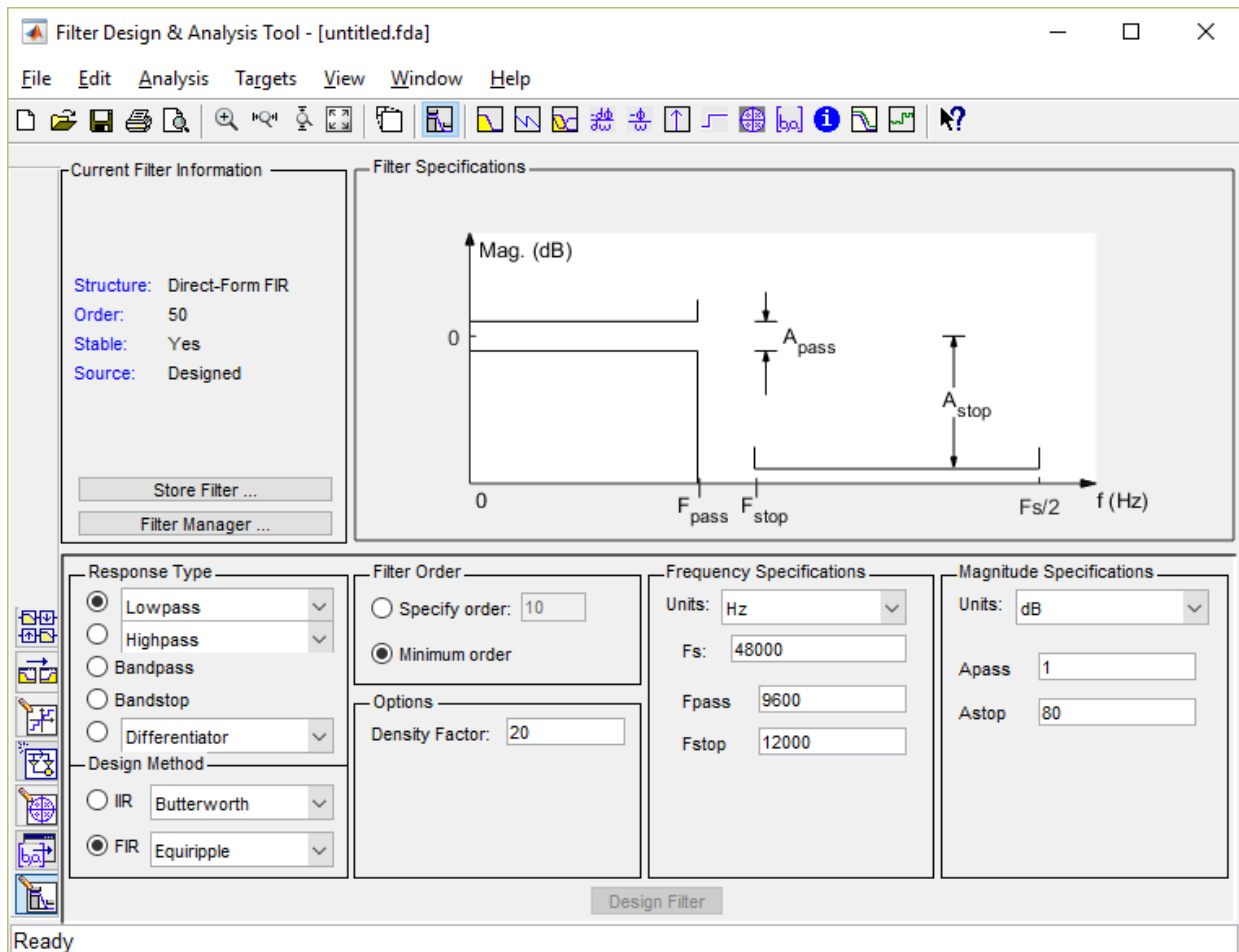


Figure 5.8: The Filter Design and Analysis Tool

Example 5.6: Designing FIR Filters using the Filter Design and Analysis Tool

Using the FDAT, design a 50th order low-pass FIR filter using the Rectangular window (abrupt truncation) and the Blackman window. Assume that the sampling frequency is 44.1 kHz and the desired cutoff frequency is 20 kHz. Compare the designs to those completed in Examples 5.3 and 5.4.

Solution:

Figure 5.9 shows the magnitude response of a 50th order low-pass FIR filter with abrupt truncation of coefficients (rectangular window). Notice the selections made in the FDAT to

specify this 50th order filter with a cutoff of 20 kHz assuming a sampling frequency of 44.1kHz. Go back to Example 5.3 (Figure 5.3) and look at the magnitude response for the 50th order filter with the same specifications. The responses look the same.

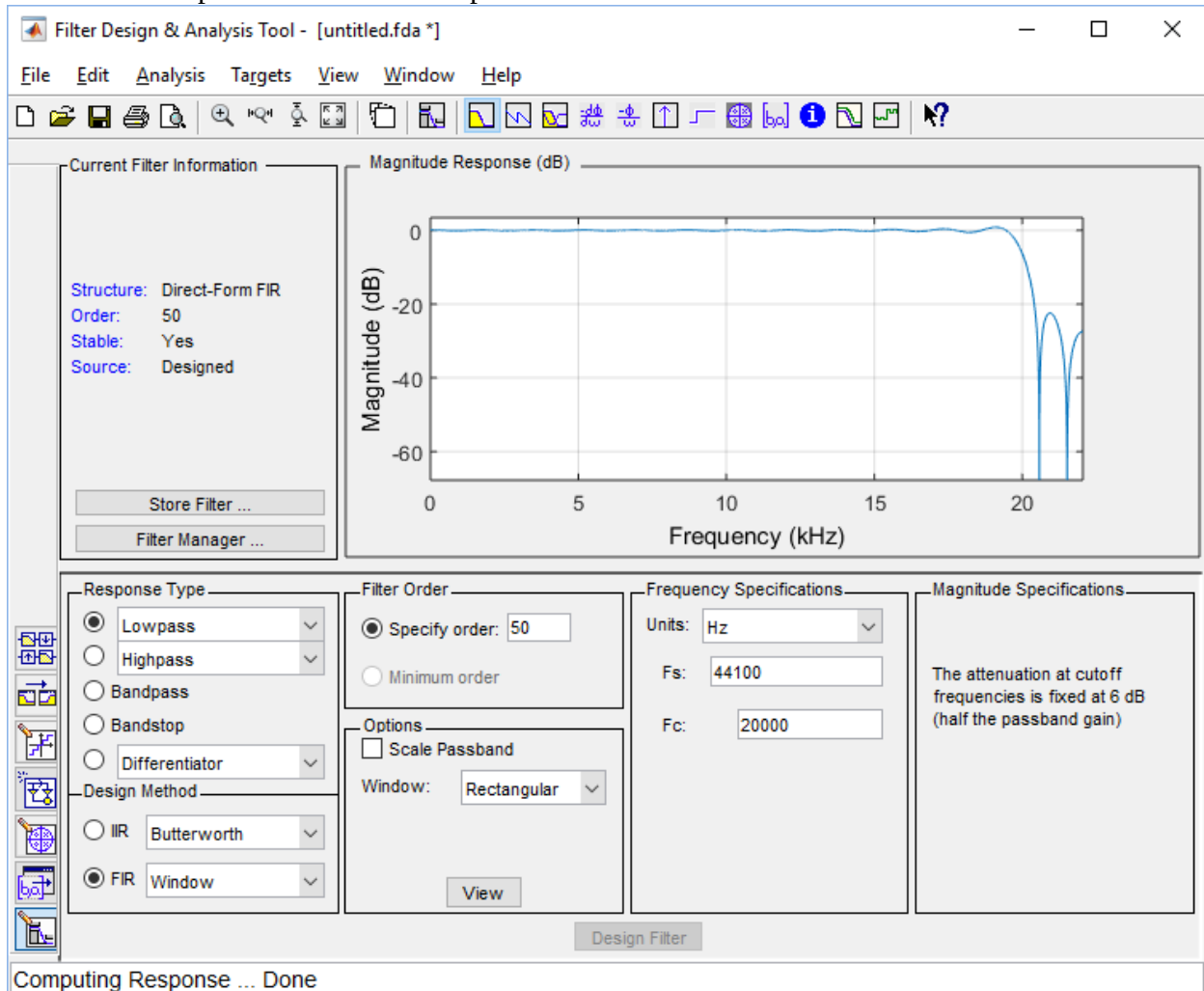


Figure 5.9: 50th Order FIR Filter – No Windowing (Rectangular)

Figure 5.10 displays a portion of the filter coefficients. Compare these filter coefficients to the coefficients listed in Table 5.1 for no windowing. The filter coefficients are identical - they are simply displayed with more precision in the FDAT.

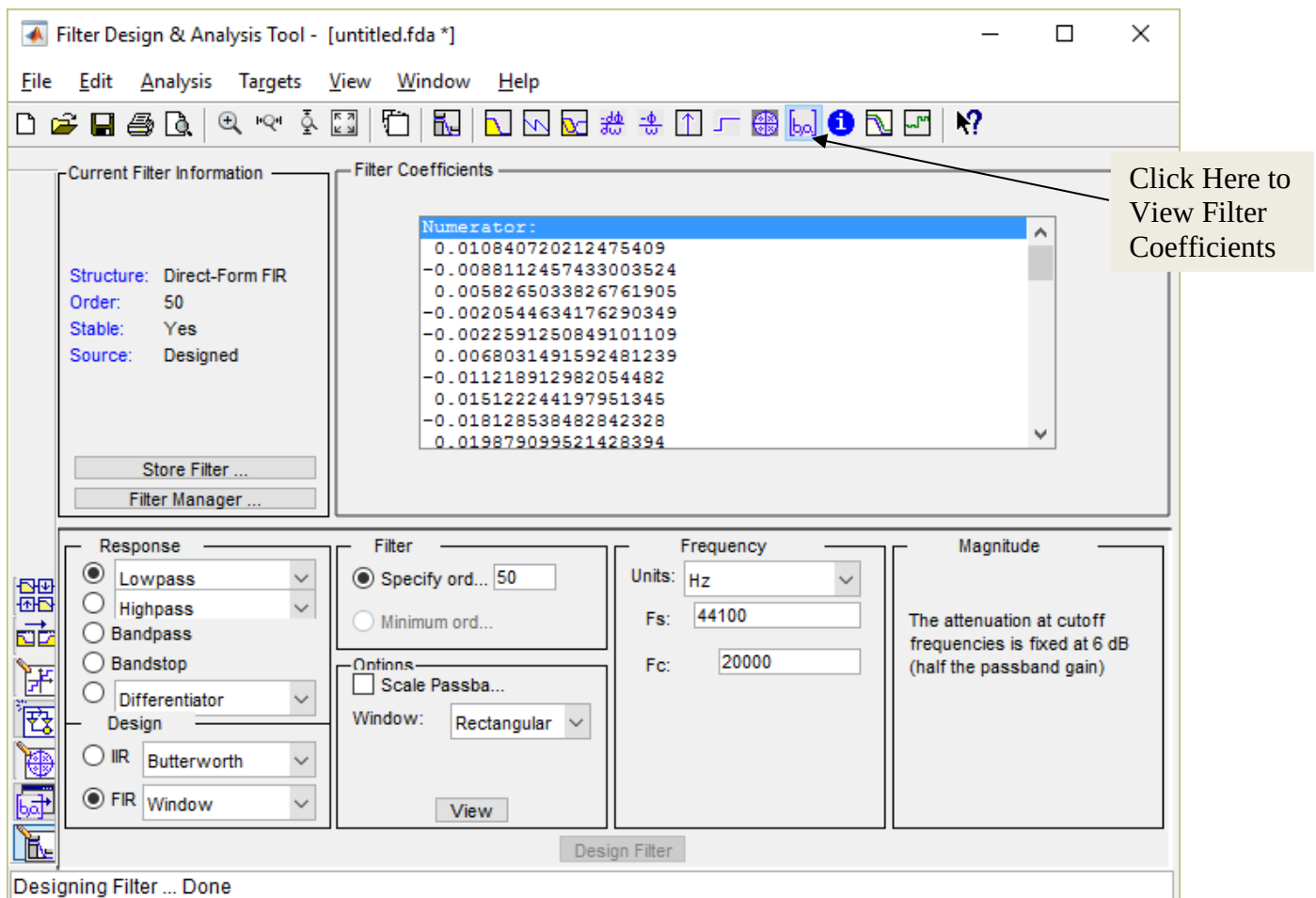


Figure 5.10: Filter Coefficients for 50th order FIR Low-pass filter with no windowing

Figure 5.11 shows the magnitude response and the first seven filter coefficients for a 50th order low-pass FIR filter using the Blackman window. The magnitude response matches the response in Example 5.4. The filter coefficients displayed in the FDAT are identical to those listed in the Table 5.1 for the Blackman window.

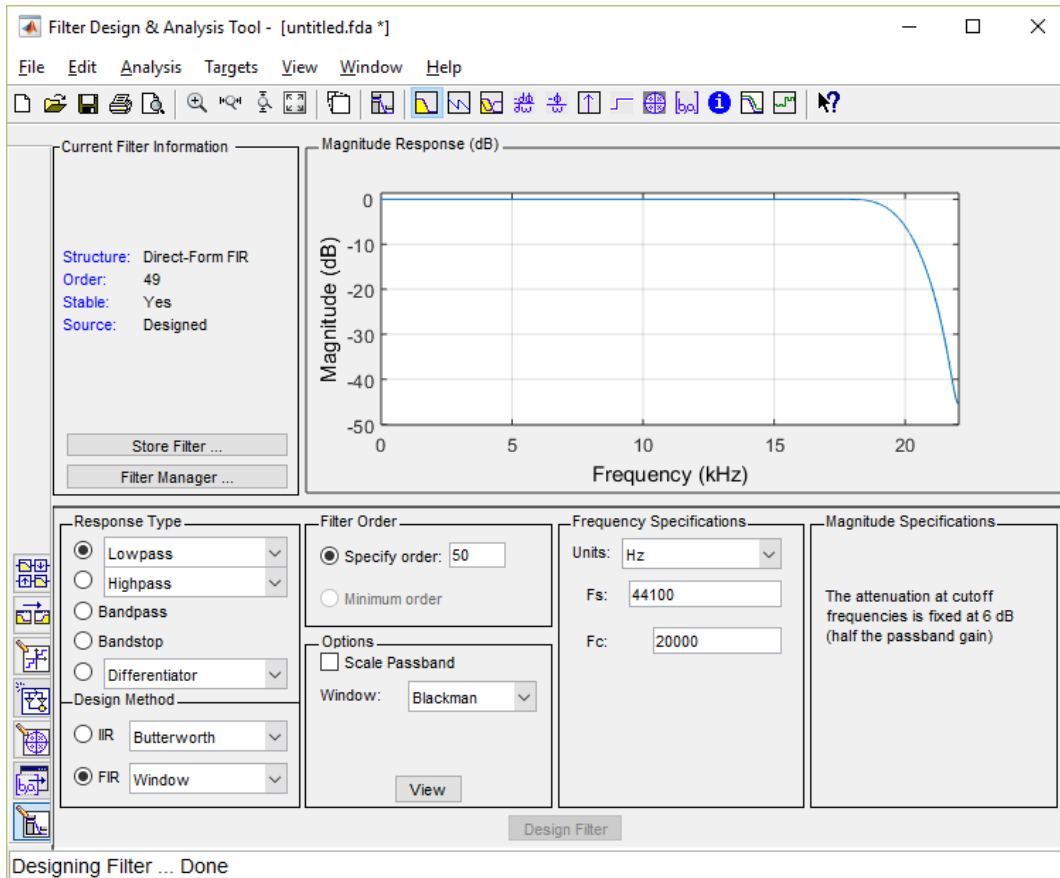


Figure 5.11 (a): Magnitude Response for 50th Order FIR Low-pass Filter using Blackman Window

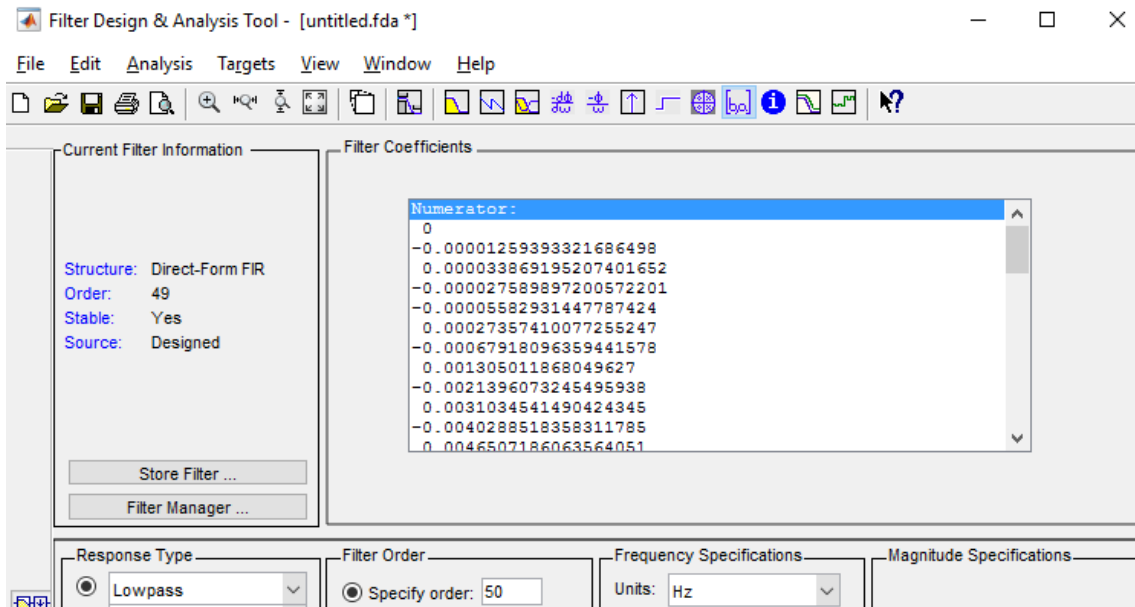


Figure 5.11 (b): Filter Coefficients for 50th Order FIR Low-pass Filter

Example 5.7: Designing a Bandpass Filter using FDAT

Using the FDAT, design a 50th order band-pass filter with a center frequency of 10 kHz and a bandwidth of 4 kHz. Assume a sampling frequency of 48 kHz. Compare the resulting design to the band-pass filter designed in Example 5.5.

Solution:

The band-pass filter design using the FDAT is shown in Figure 5.12. The magnitude response looks the same as the one generated in Example 5.5. The FDAT uses Equation 5.12 to calculate the filter coefficients.

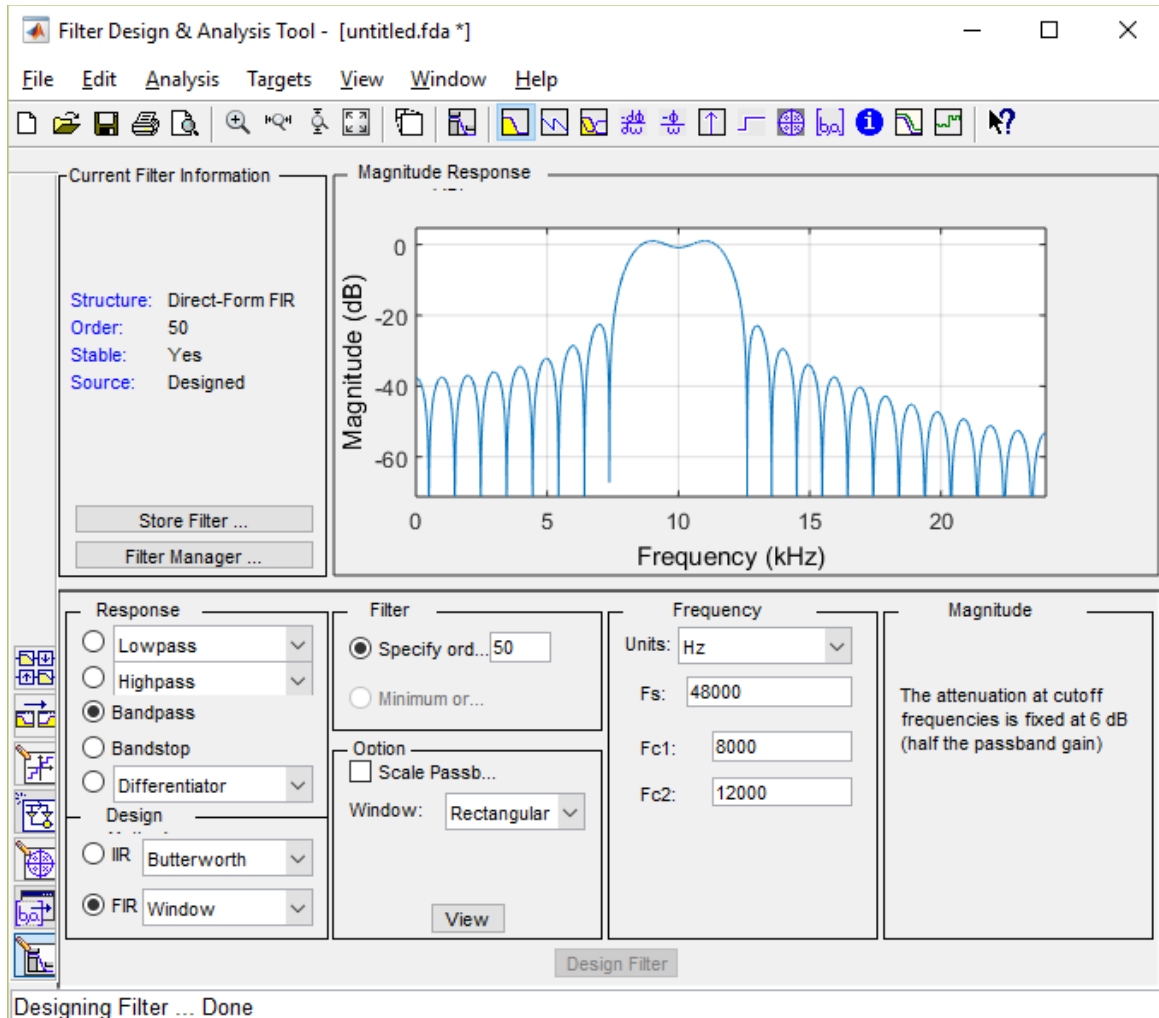


Figure 5.12: 50th Order FIR Band-Pass Filter

Examples 5.6 and 5.7 demonstrate that the FDAT simply implements the equations for windowed FIR filters discussed in the previous section. Optimal filters were also discussed in the previous section but no examples were given. That is because the optimization algorithms are extremely complicated and filter coefficients are difficult to calculate without software tools. Fortunately, there are now many software tools, including the FDAT, that perform the optimization algorithms to produce filter coefficients.

Example 5.8: Optimal FIR Filter Design using FDAT

Using the FDAT, design an optimal equiripple band-pass filter with a center frequency of 10 kHz and a bandwidth of 4 KHz. Assume a sampling frequency of 48 kHz. Compare the resulting design to the windowed band-pass filter designed in Example 5.7.

Solution:

There are many options available for the optimal equiripple design. These options will be explored in this example. We begin with specifying a 50th order filter with equal weighting in the passband and stopband. The resulting magnitude response is shown in Figure 5.13. How does this design compare to the windowed bandpass filter? The optimal design has a sharper roll-off. The ripple is spread evenly in the stop-band and in the pass-band (hence the name).

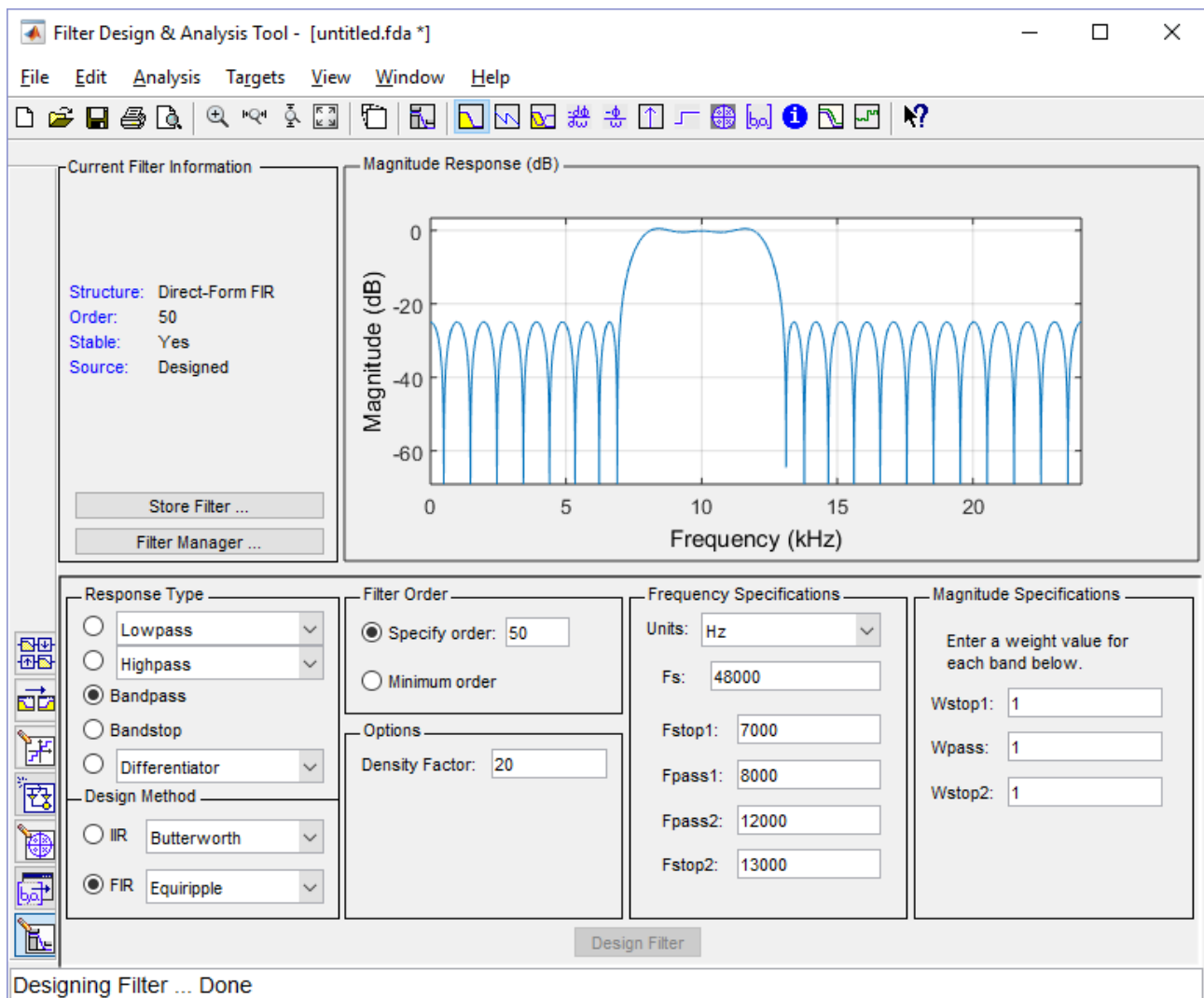


Figure 5.13: Equiripple Design – Specified Filter Order – Equal Weighting

The passband and stopband weights (W_{pass} and W_{stop}) can be varied to force a closer match to an ideal low-pass filter in either the passband or the stopband. Figure 5.14 illustrates the effect of weighting the passband match higher (ten times the weight on the stopband match). Comparing this filter response with the response in Figure 5.13, it is apparent that weighting the passband higher will result in less ripple in the passband but the stopband attenuation is not as large.

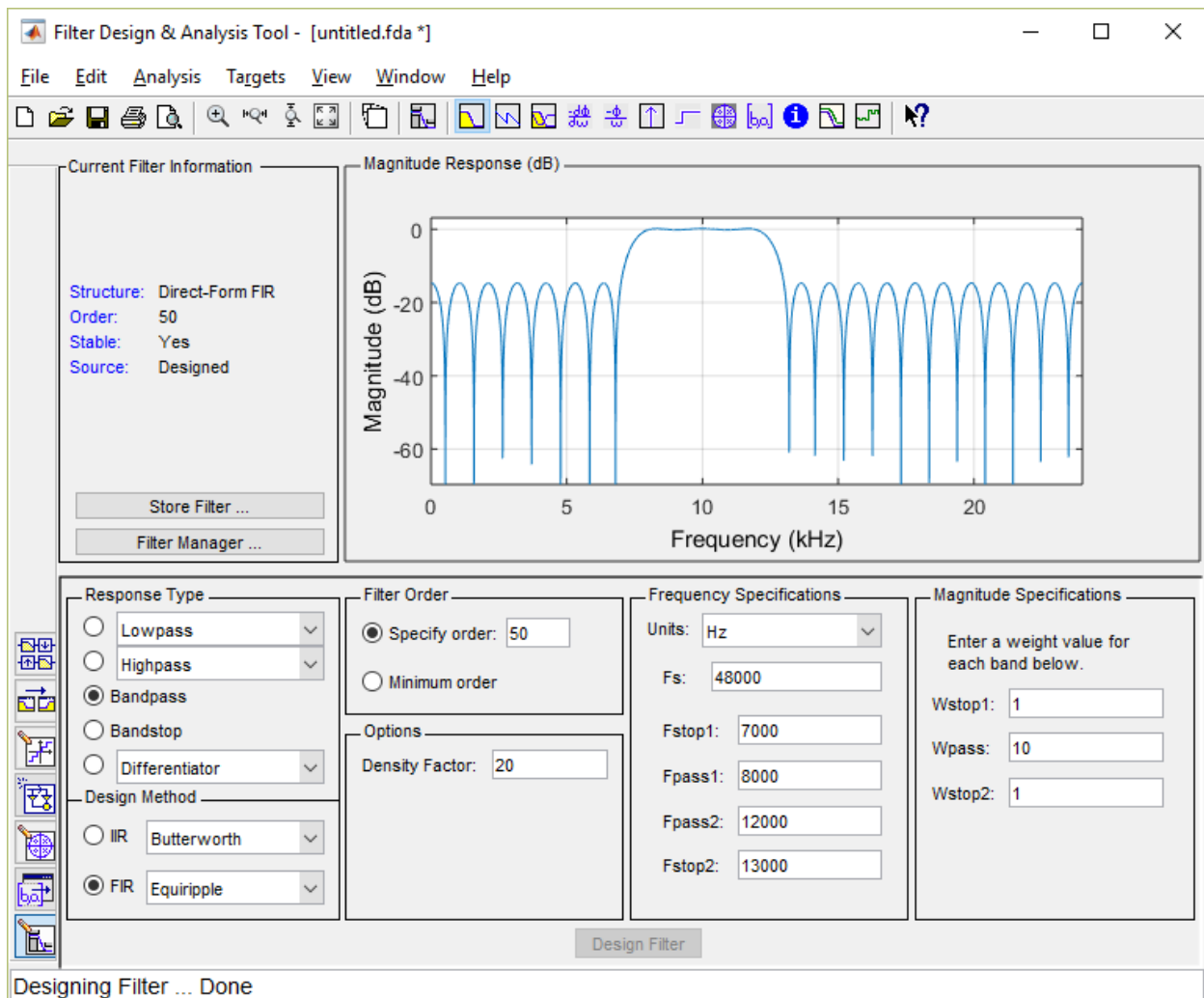


Figure 5.14: Equiripple Design – Specified Order – Higher Passband Weighting

Figure 5.15 illustrates the effect of a higher weight on the stopband. Not surprisingly, the attenuation in the stopband increases but there is more ripple in the passband.

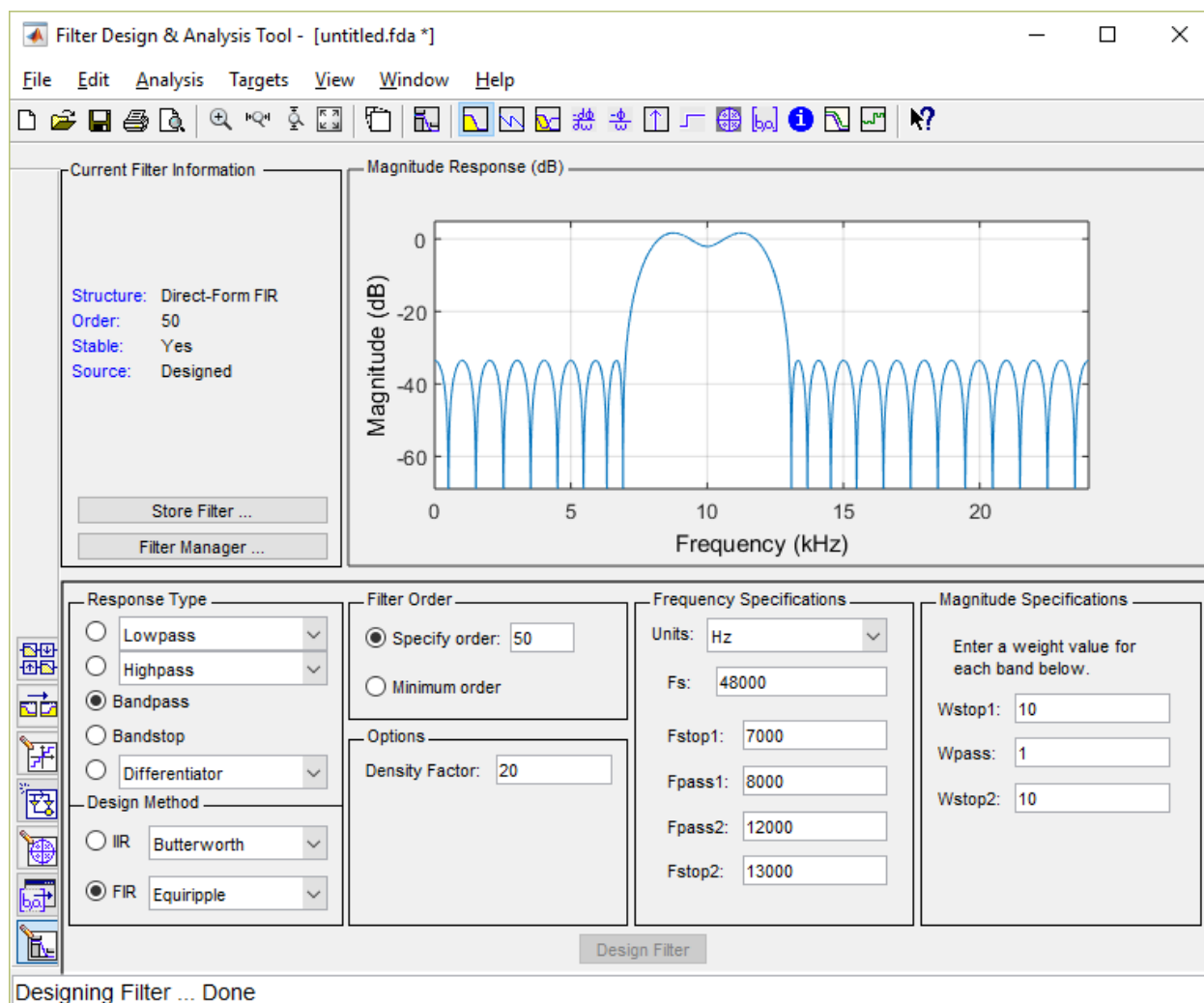


Figure 5.15: Equiripple Design – Specified Order – Higher Stopband Weighting

Thus far, the filter order has been specified. There is also an option to find the minimum order design that meets a certain set of specifications. As illustrated in Figure 5.16, the optimal filter allows the user to specify a desired stopband attenuation and acceptable passband ripple. The FDAT will then determine the minimum filter order required to satisfy the specifications. The filter is 87th order but has only 0.1 dB of ripple in the passband and at least 30 dB of attenuation in the stopband. Of course, reducing the required attenuation in the stopband or allowing more ripple in the passband would reduce the required filter order. It is worth noting that an order of 87 for a digital filter is very reasonable while an order of 87 for an analog filter is prohibitive.

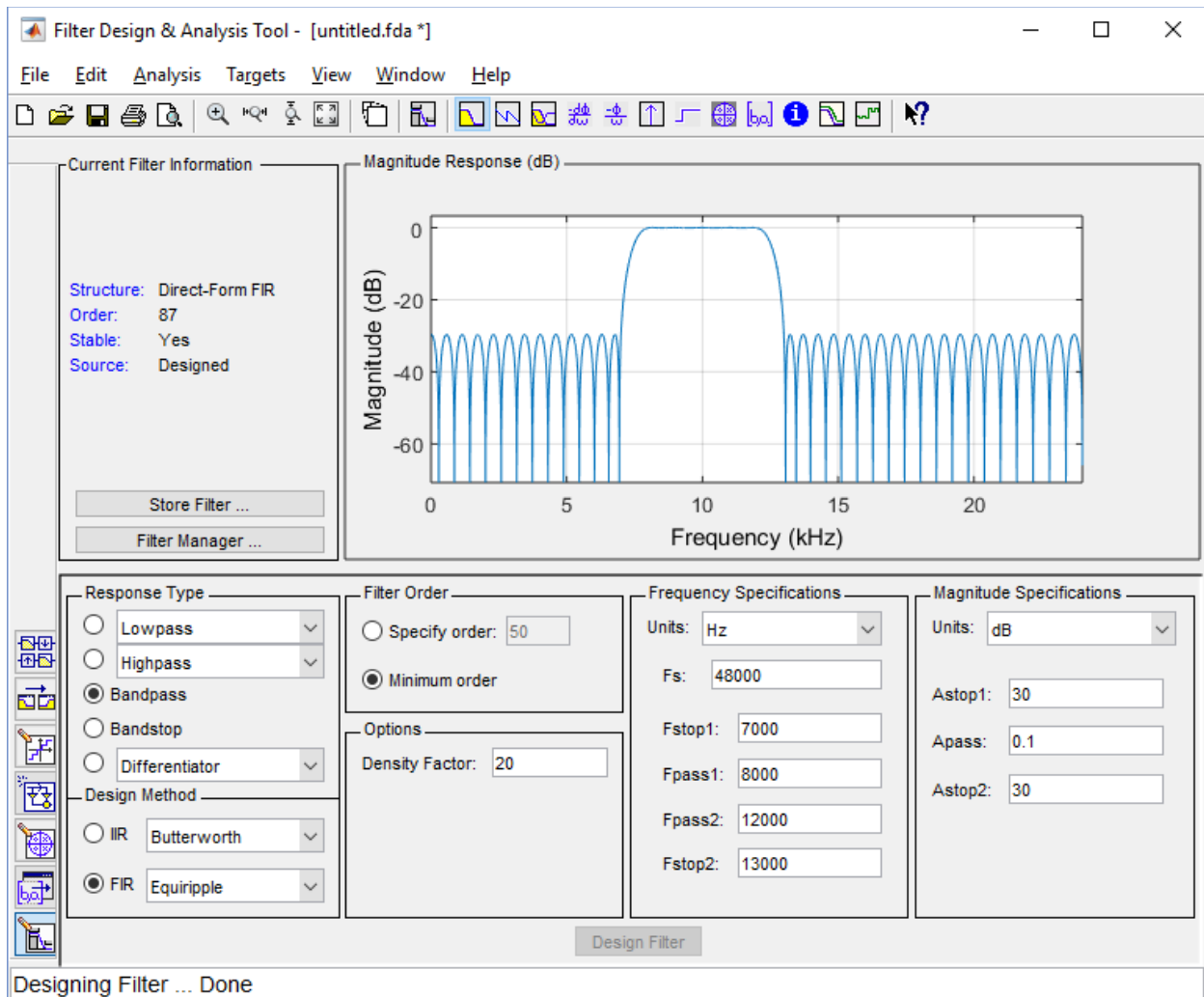


Figure 5.16: Equiripple Design – Minimum Order to meet Specifications

Example 5.9: Exporting Filter Coefficients from FDAT to the MATLAB Workspace

Consider the 87th order band-pass FIR filter designed in Example 5.8. Plot the output response if the input to the filter is:

$$x(t) = \sin(2\pi(5000)t) + \sin(2\pi(9000)t) + \sin(2\pi(15000)t)$$

Solution:

Using the FDAT, design the 87th order band-pass filter shown in Figure 5.16. In the FDAT window, choose File → Export and an Export window will pop up.

Click on the Export button and the filter coefficients will be exported to the MATLAB Workspace under the variable name of Num. In the MATLAB workspace, execute the following commands:

```

% Filter Coefficients for numerator have been exported under Num
% Define the denominator of the filter transfer function
Den = zeros(size(Num)); Den(1)=1;
Fs = 48000; t = 0:1/Fs:0.003;
x = sin(2*pi*5000*t)+sin(2*pi*9000*t)+sin(2*pi*15000*t);
y = filter(Num,Den,x);
subplot(2,1,1); plot(t,x);
title('Input to the Filter');
subplot(2,1,2); plot(t,y);
title('Output of the Filter'); xlabel('time (sec)');

```

The input signal and the resulting output signal for this filter are shown in Figure 5.17. As expected, the band-pass filter passes the 9000 Hz component of the input signal and rejects (significantly attenuates) the 5000 Hz and 15,000 Hz components of the input signal.

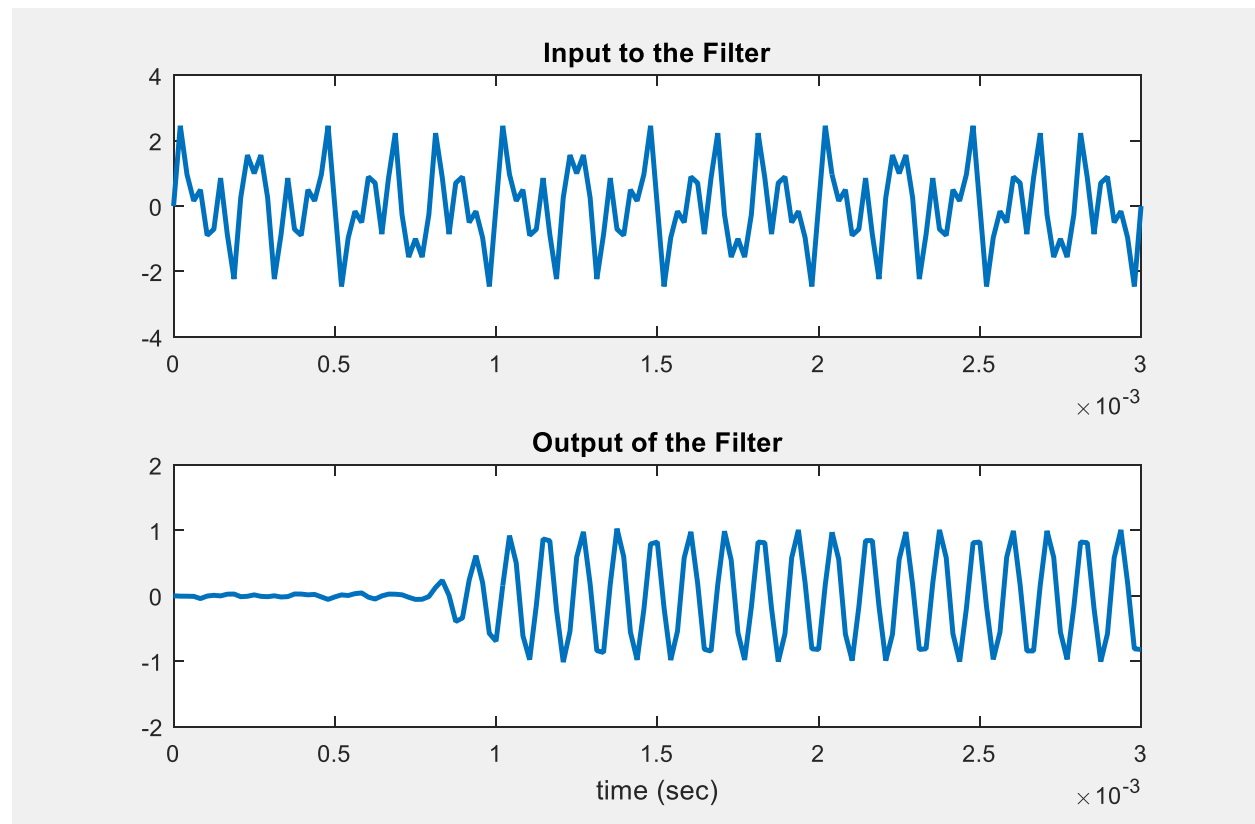


Figure 5.17: Input and Output Signals for Band-Pass Filter

Challenge Question 5.3

Why is the output response in Figure 5.17 very small from the initial time of zero to about 0.8 ms? In other words, why does it appear that there is a time delay in the output response?

5.5 PRACTICAL CONSIDERATIONS FOR FIR FILTERS

Word-length Effects (Filter Coefficient Quantization Errors)

In order to implement a filter on a DSP processor, the filter coefficients must be stored in memory. In a floating-point processor, the coefficients would be stored in floating-point format and for a fixed-point processor, the filter coefficients would be stored in fixed-point format. Floating-point and fixed-point formats were covered in Chapter 4. It was shown through many examples that quantization error often occurs when representing numbers in a specified format. The filter coefficients derived using the FDAT (or some other design tool) will very likely be quantized when stored in the memory of a DSP processor. *In digital filter design, it is imperative to check the effect of filter coefficient quantization on the performance of a filter!* Example 5.10 illustrates how quantization of the coefficients can impact filter performance.

Example 5.10: Effect of Coefficient Word-length on Filter Performance

Consider the 87th order elliptical FIR band-pass filter designed in Example 5.8.

- (a) Compute the filter coefficients for a 16-bit processor using Q15 format. Plot the frequency response for the quantized filter and compare it to the original frequency response.
- (b) Repeat part (a) assuming the filter coefficients are stored as 8-bit values.

Solution

- (a) Using the FDAT, design the 87th order band-pass filter shown in Figure 5.16. In the FDAT window, choose File → Export and export the filter coefficients to the MATLAB workspace using the default variable Num. The following set of MATLAB commands quantize the filter coefficients to 16-bits (1 sign bit, 15 magnitude bits) then plot the resulting filter frequency response.

```
% Define denominator for Filter transfer function:
Den=zeros(size(Num)); Den(1) = 1;
% Number of bits:
B = 16;
% Quantize the Filter Coefficients to Q15 format:
Numq = round(Num*2^(B-1))/(2^(B-1));
Fs = 48000;
% Frequency response for un-quantized filter:
[H,f]=freqz(Num,Den,200,Fs);
% Frequency Response for 16-bit Filter
[Hq,f]=freqz(Numq,Den,200,Fs);
subplot(2,1,1);plot(f/1000,20*log10(abs(H)));
title('Unquantized Filter Coefficients'); grid
ylabel('Magnitude (dB)');
subplot(2,1,2);plot(f/1000,20*log10(abs(Hq)));
title('16-Bit Filter Coefficients'); grid
xlabel('Frequency (kHz)'); ylabel('Magnitude (dB)');
```

Figure 5.18 shows the frequency response (magnitude) for both the un-quantized and quantized (16-bit) filters. The responses are nearly identical.

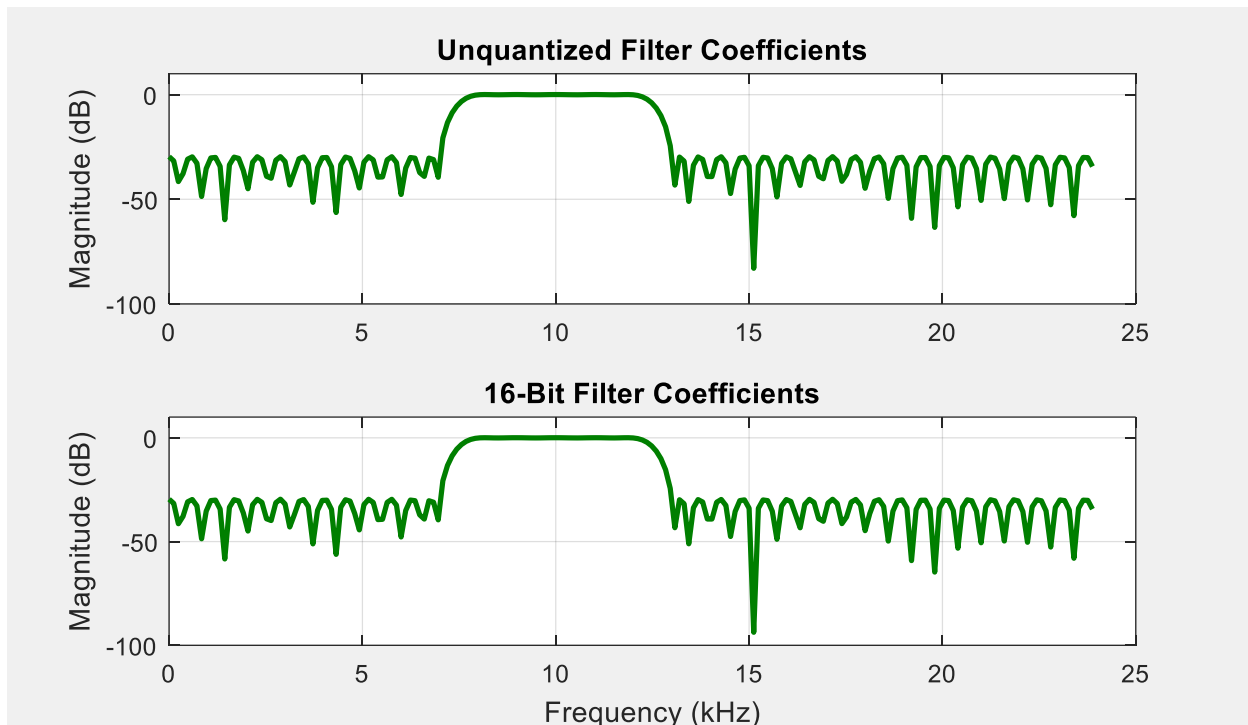


Figure 5.18: Filter Magnitude Response Using 16-Bit Filter Coefficients

(b) For 8-bits, the MATLAB commands are easily modified to

```
% Define denominator for Filter transfer function:
Den=zeros(size(Num)); Den(1) = 1;
% Number of bits:
B = 8;
% Quantize the Filter Coefficients to 8 bits:
Numq = round(Num*2^(B-1))/(2^(B-1));
Fs = 48000;
% Frequency response for un-quantized filter:
[H,f]=freqz(Num,Den,200,Fs);
% Frequency Response for 8-bit Filter
[Hq,f]=freqz(Numq,Den,200,Fs);
subplot(2,1,1);plot(f/1000,20*log10(abs(H)));
title('Unquantized Filter Coefficients'); grid
ylabel('Magnitude (dB)');
subplot(2,1,2);plot(f/1000,20*log10(abs(Hq)));
title('8-Bit Filter Coefficients'); grid
xlabel('Frequency (kHz)'); ylabel('Magnitude (dB)');
```

Figure 5.19 shows the frequency responses for the un-quantized and quantized (8-bits) filters. Quantizing the filter coefficients to 8-bits definitely degrades the performance of this band-pass filter. Increased ripple is observed in the pass-band and less attenuation occurs in the stop-band.

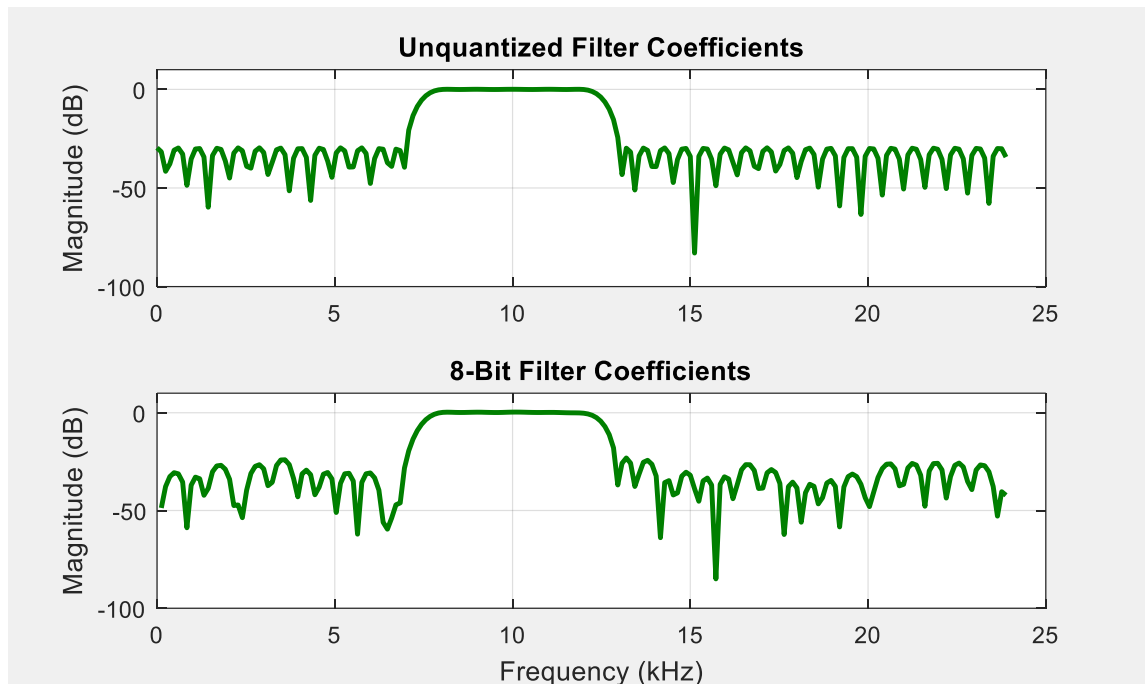


Figure 5.19: Filter Magnitude Response Using 8-Bit Filter Coefficients

Table 5.2 lists the first twelve un-quantized filter coefficients, the filter coefficients quantized to 16-bits, and the filter coefficients quantized to 8-bits. The loss in precision is readily apparent. When quantized to 8-bits, several of the original filter coefficients become zero.

Table 5.2: First 12 Filter Coefficients for Example 5.10

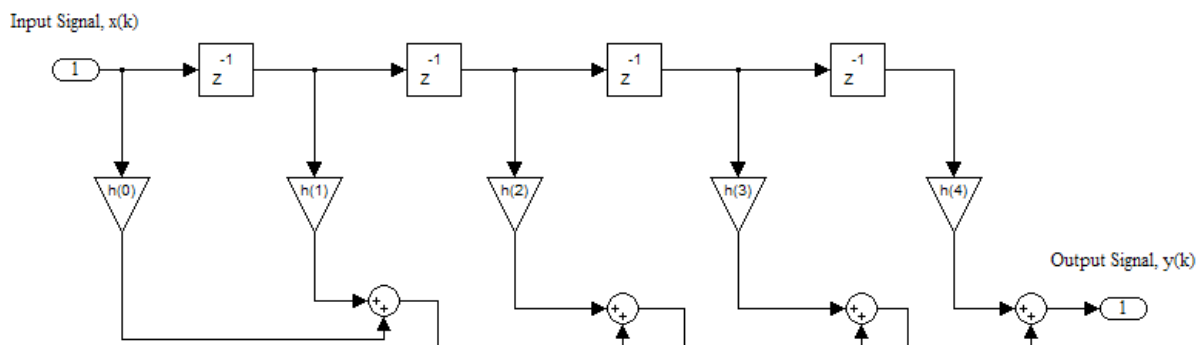
Un-Quantized Coefficients	16-Bit Coefficients	8-Bit Coefficients
-0.00261674337283854	-0.00262451171875000	0.0000000
-0.00179960986044774	-0.00180053710937500	0.0000000
-0.00288475539268499	-0.00289916992187500	0.0000000
-0.01577921015881180	-0.01577758789062500	-0.0156250
0.00389085557827262	0.00387573242187500	0.0000000
0.00151759269512286	0.00152587890625000	0.0000000
0.00082359680536102	0.00082397460937500	0.0000000
0.00175909004813907	0.00177001953125000	0.0000000
0.00448762339792434	0.00448608398437500	0.0078125
-0.00313446488889777	-0.00314331054687500	0.0000000
-0.00967046889150305	-0.00967407226562500	-0.0078125
-0.00144632666869306	-0.00143432617187500	0.0000000

The band-pass filter in Example 5.10 could be implemented on a 16-bit fixed point processor with no noticeable degradation in filter performance due to coefficient quantization. Other digital filters may experience an unacceptable degradation in performance if the coefficients were quantized to 16-bits. Checking the effect of coefficient quantization on filter performance is a critical step in digital filter design!

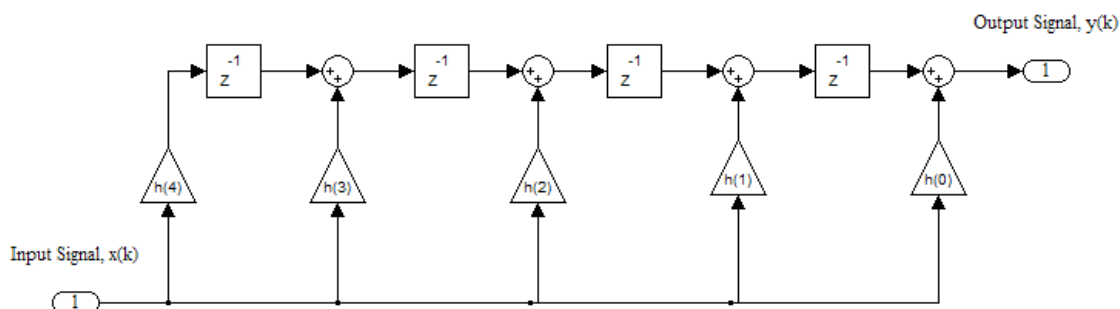
FIR Filter Realizations

Several different realizations for an FIR filter are shown in Figure 5.20. For simplicity, these realizations are shown for a 4th order filter but can easily be extended to any order. The Direct form is just a straight forward implementation of the difference equation for an FIR filter. An Nth order FIR filter in Direct form has N filter taps (coefficient/delay terms).

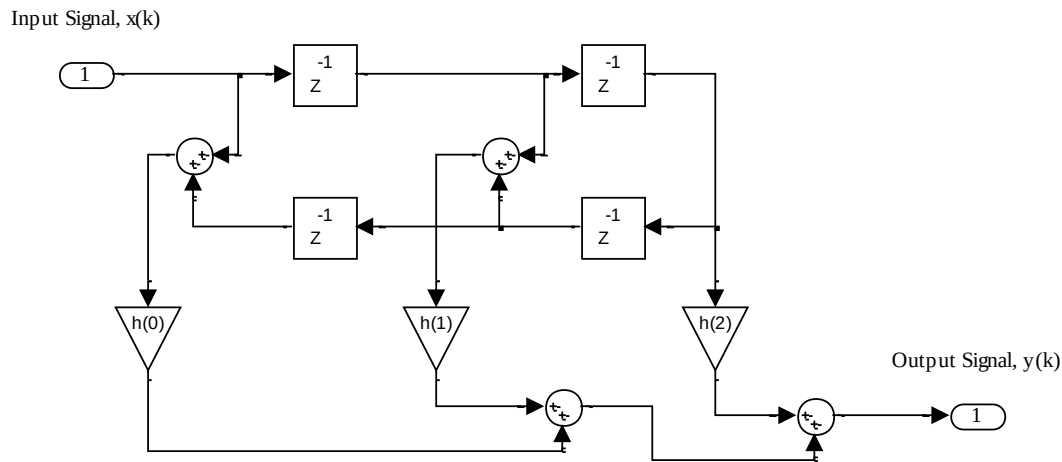
The Direct-Transposed form performs the multiplications first then sends the results through a series of adders and delays. Mathematically, the Direct and the Direct-Transposed realizations are identical; in other words, the output signal produced should be the same. The differences lie in how each realization is actually implemented using a DSP processor or an FPGA. In the Direct form, an input buffer or register is required to save the N past inputs to the filter. In the Direct-Transposed form, assuming a parallel pipelined architecture, the input signal can be applied in parallel to all the tap multipliers which eliminates the need for an input register. Also, the filter order can easily be expanded by simply cascading additional multiply/accumulate taps.



(a) Direct Form



(b) Direct Form - Transposed



(c) Direct Form - Symmetric

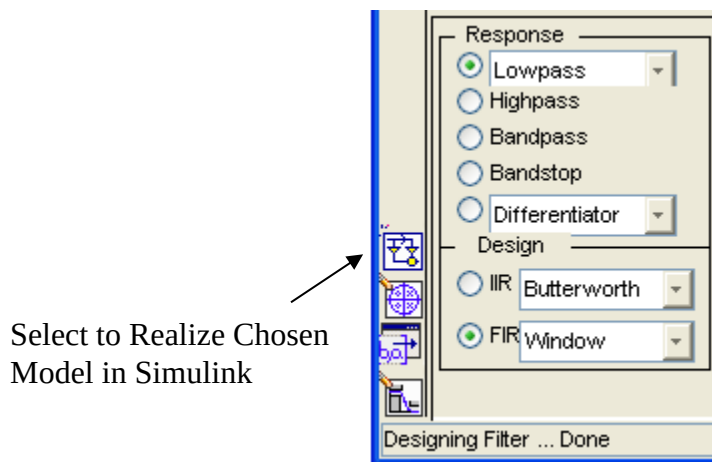
Figure 5.20: FIR Filter Realizations

The Direct Symmetric-Form reduces the number of multiplies (filter taps) by exploiting the symmetry of FIR filter coefficients. Since $h(0) = h(N)$, $h(1) = h(N - 1) \dots$, the number of multiplies can be cut in half by computing the filter output as

$$y(k) = h(0)[x(k) + x(k - N)] + h(1)[x(k - 1) + x(k - (N - 1))] + \dots$$

Cutting the number of multiplications in half significantly reduces the computational complexity of the filter and the processor requirements.

The FDAT defaults to a Direct-Form realization. This is easily changed by clicking on Edit → Convert Structure and switching to a Direct-Transposed or a Direct-Symmetric realization. The filter realization can then be exported to Simulink® by clicking on the Realize Model icon at the bottom left of the FDAT as shown in the following screenshot.



Select to Realize Chosen Model in Simulink

Roundoff Errors in Arithmetic Operations

The output signal of an FIR filter is computed using the following equation:

$$y(k) = \sum_{i=0}^N b_i x(k-i) = b_0 x(k) + b_1 x(k-1) + b_2 x(k-2) + \cdots b_N x(k-N)$$

Suppose the filter coefficients are 16 bits and the processor utilizes a 16-bit ADC. Each multiplication will be 32 bits (size of h + size of x). Eventually, this must be truncated or rounded to 16-bits. Rounding the result will result in a reduction of the Signal-Noise Ratio (SNR). To minimize the effect of rounding, it is much better to store each product in a double precision register and round after the final summation instead of rounding to 16 bits after each individual multiplication.

Arithmetic Overflow in Fixed Point Processors

As discussed in Chapter 4, fixed point processors typically use a Q15 format in which all numbers are scaled to be between -1 and $+1$. Any multiplication operations on numbers in Q15 format results in a number that is also between -1 and $+1$ (no overflow). However, addition operations on numbers in Q15 format can result in overflow. Arithmetic overflow will cause either a sign change in the result (undesirable) or clipping (saturating at the maximum or minimum value). Arithmetic overflow can be eliminated (or significantly reduced) by scaling the filter coefficients and/or the input data.

Suppose the filter coefficients are scaled using an $\mathcal{L}_1$ norm as follows:

$$b_{i\text{scaled}} = \frac{b_i}{\sum_{i=0}^N |b_i|} \quad i = 0, 1, \dots, N \quad (5.14)$$

The $\mathcal{L}_1$ scaled coefficients are then stored in the processor memory instead of the original filter coefficients. Arithmetic overflow will not occur in this case. To see this, recall that

$$y(k) = \sum_{i=0}^N b_i x(k-i) = b_0 x(k) + b_1 x(k-1) + b_2 x(k-2) + \cdots + b_N x(k-N)$$

Worst case would occur if all the products were the same sign (let's assume positive) and as large as possible. This would happen if each input, $x(k-i)$, were the same sign as the filter coefficient, b_i , that it was being multiplied by and had a magnitude of one (largest possible magnitude in Q15 format). In this case, the output would be:

$$\text{Maximum Value for } y(k) \text{ with no scaling} = |b_0| + |b_1| + |b_2| + \cdots + |b_N|$$

However, if the $\mathcal{L}_1$ scaled coefficients were used, then the worst case (largest possible) output would be:

$$\begin{aligned}\text{Maximum Value for } y(k) \text{ with scaling} &= \frac{|b_0|}{\sum_{i=0}^N |b_i|} + \frac{|b_1|}{\sum_{i=0}^N |b_i|} + \frac{|b_2|}{\sum_{i=0}^N |b_i|} + \cdots + \frac{|b_N|}{\sum_{i=0}^N |b_i|} \\ &= \frac{1}{\sum_{i=0}^N |b_i|} [|b_0| + |b_1| + |b_2| + \cdots + |b_N|] = 1\end{aligned}$$

Challenge Question 5.4

What is a possible disadvantage of scaling down the filter coefficients to avoid arithmetic overflow?

$\mathcal{L}_1$ scaling guarantees that arithmetic overflow will never occur when calculating the FIR filter output; however, as addressed in the challenge question, $\mathcal{L}_1$ scaling may have a negative impact on the coefficient quantization error. The following example illustrates scaling and the effect on filter performance.

Example 5.11: Effect of Scaling on FIR Filter Performance

Consider the 87th order band-pass FIR filter designed in Example 5.8 (Figure 5.16). Scale the filter coefficients using the $\mathcal{L}_1$ norm. Then quantize the coefficients to 8-bits and compare the filter magnitude response to the unscaled but quantized filter performance.

Solution:

Using the FDAT, design the 87th order band-pass filter shown in Figure 5.16. In the FDAT window, choose File → Export to export the filter coefficients to the MATLAB workspace under the default variable name of Num. The following MATLAB commands complete the example:

```
Den=zeros(size(Num)); Den(1)=1;
B=8;
% Quantize filter coefficients to 8-bits
Numq=round(Num*2^(B-1))/(2^(B-1));
Fs=48000;
[Hq,f]=freqz(Numq,Den,200,Fs);
subplot(2,1,1); plot(f/1000,20*log10(abs(Hq)));
title('8-Bit Filter Coefficients (Unscaled)'); grid
ylabel('Magnitude (dB)');
Scale_Factor = sum(abs(Num));
Num_scaled=Num/Scale_Factor;
Num_scaled_q = round(Num_scaled*2^(B-1))/(2^(B-1));
[H_scaled_q,f]=freqz(Num_scaled_q,Den,200,Fs);
subplot(2,1,2); plot(f/1000,20*log10(abs(H_scaled_q)));
title('8-Bit Filter Coefficients (Scaled)'); grid
xlabel('Frequency (kHz)'); ylabel('Magnitude (dB)');
```

The results are shown in Figure 5.21. In addition to the expected loss in filter gain due to scaling, the quantization error increases when attempting to represent the smaller (scaled) coefficients with 8-bits. Other scaling methods to reduce the occurrence of arithmetic overflow will be explored in Chapter 6.

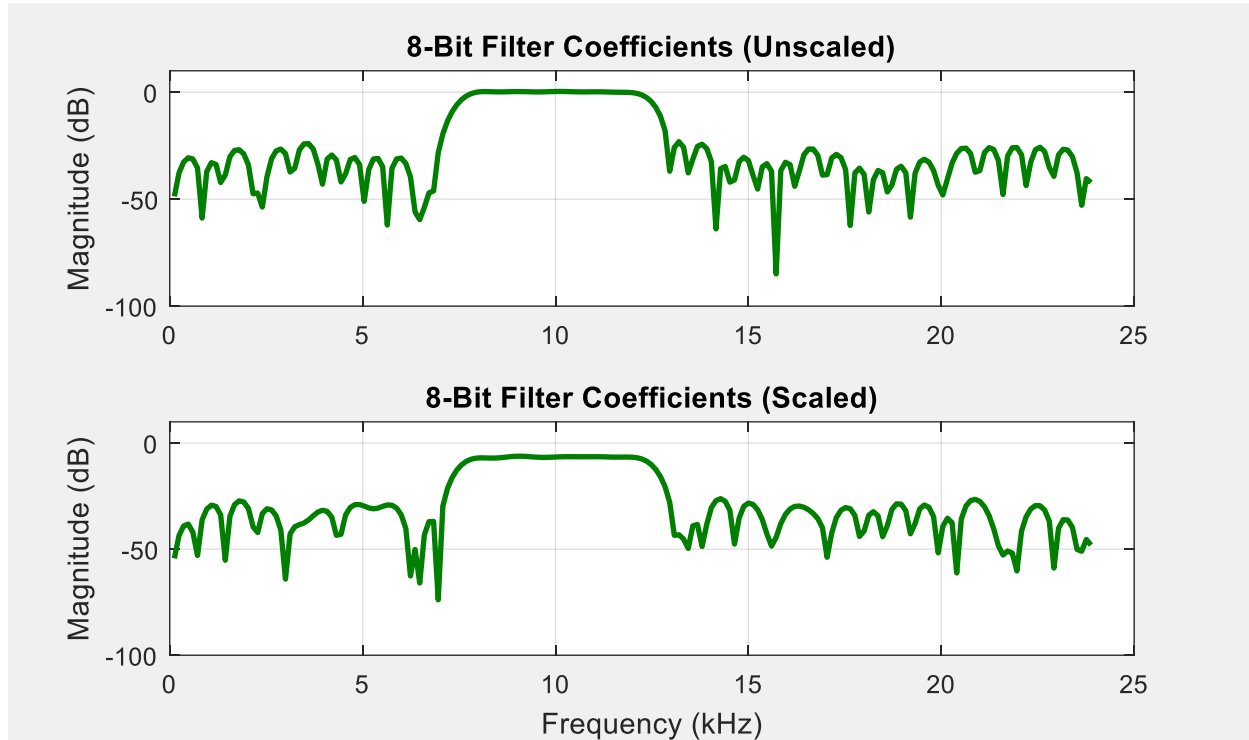


Figure 5.21: Effect of Coefficient Scaling on Filter Performance

5.6 APPLICATION: SPECIAL TYPES OF FIR FILTERS

Feed-forward Comb Filter

An FIR comb filter has the form

$$y(k) = x(k) + b_N x(k - N) \quad (5.15)$$

The output is simply the sum of the current input and a possibly attenuated past input sample (N sampling instants back). This filter attenuates a particular frequency and its harmonics. The notch frequencies of a comb filter depend on the sampling frequency, F_s , the filter order, N , and the sign of the filter coefficient, b_N . Notch frequencies can be computed as follows:

$$\text{Notch Frequencies: } \begin{cases} \frac{F_s}{N}, \frac{2F_s}{N}, \frac{3F_s}{N}, \dots & b_N < 0 \\ \frac{F_s}{2N}, \frac{3F_s}{2N}, \frac{5F_s}{2N}, \dots & b_N > 0 \end{cases} \quad (5.16)$$

Figure 5.22 shows the frequency response for a 10th order comb filter with a sampling frequency of 48 kHz and three different values for b_N . The notch frequencies, as expected, occur at 2.4, 7.2, 12.0, 16.8, and 21.6 kHz. Figure 5.22 also illustrates that the degree of attenuation depends on the filter coefficient b_N . Choosing b_N closer to 1 (or -1) provides more attenuation than choosing a value for b_N closer to zero.

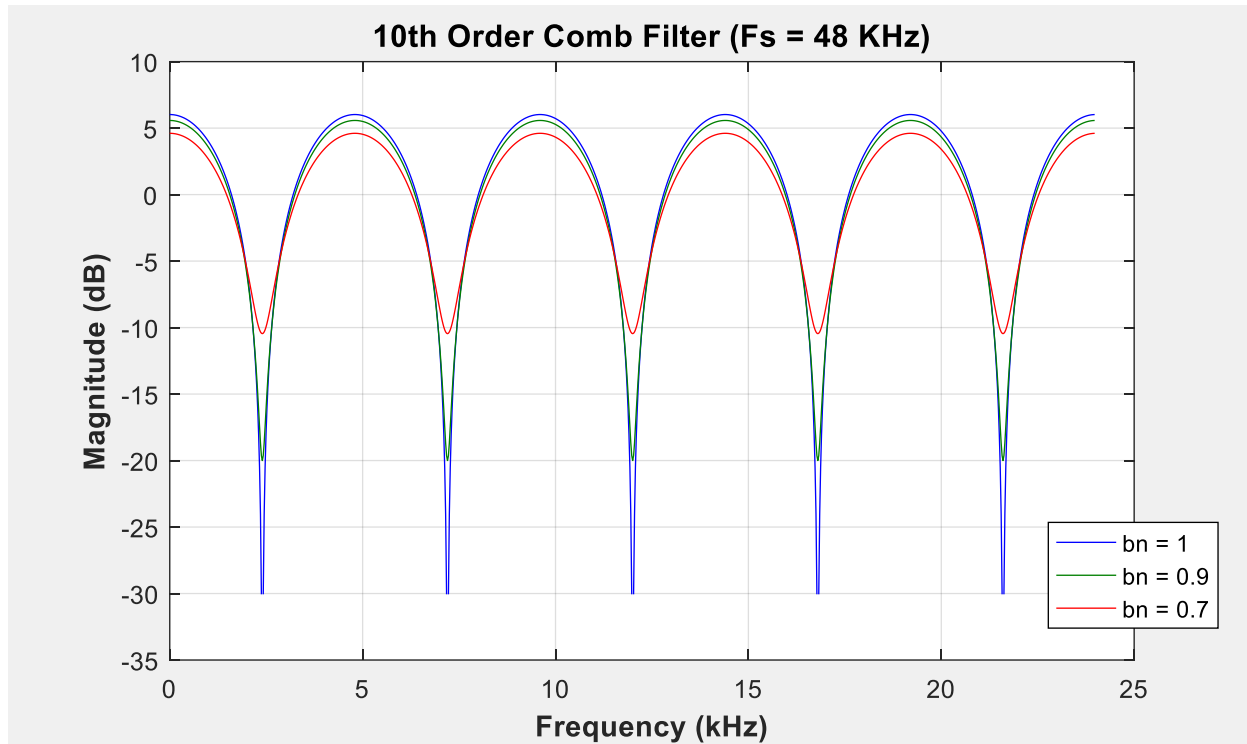


Figure 5.22: Comb Filter Frequency Response

There are several applications for comb filters. Televisions use comb filters on composite video signals to properly separate the luminance and chrominance signals. These comb filters recover finer horizontal detail and purer color information from the composite signal than simple notch or band-pass filters.

In audio applications, the comb filter can be used to produce a simple echo effect. The time delay N represents the number of sampling periods required for the signal to be reflected back; that is, distance traveled divided by $c \cdot T_s$ where c is the speed of light. The attenuating filter coefficient, b_N , models air absorption and/or spherical losses. A more sophisticated echo effect can be created by allowing the attenuating coefficient to vary with the frequency of the input signal.

Putting several comb filters in parallel as shown in Figure 5.23 produces a chorus effect, making a single instrument or voice seem like several instruments or voices. The output signal is given by:

$$y(k) = x(k) + g_1x(k - d_1) + g_2x(k - d_2) + g_3x(k - d_3) \quad (5.17)$$

The time delays typically range between 10 and 50 ms and can even be set up to vary slowly over time rather than being constant. The resulting audio signal is a richer or thicker version of the original input audio signal.

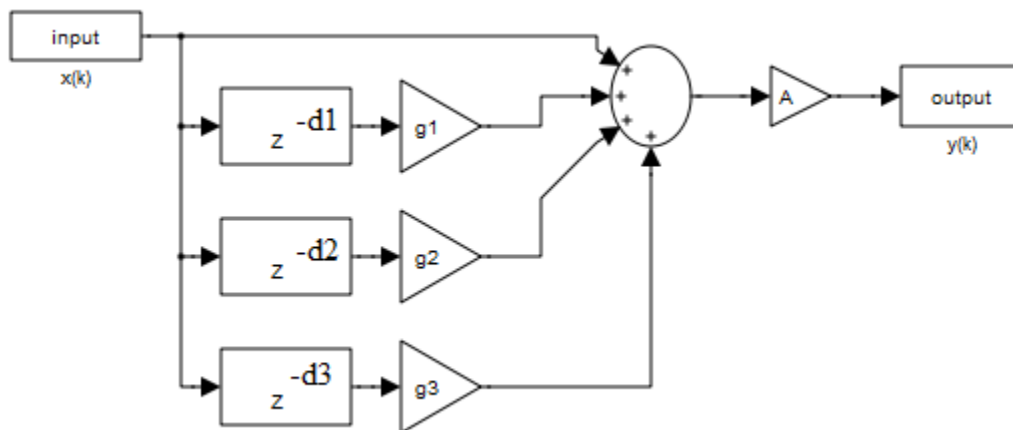


Figure 5.23: Chorus Effect Using Bank of Comb Filters

Challenge Question 5.5

If the sampling frequency in Figure 5.23 is 44100 Hz, what should the integer delay, $d1$, be to produce a time delay of 10 ms?

Flanging

Flanging adds a “whooshing” or “swishing” sound to an audio signal. As shown in Figure 5.24, the flanging effect is created by simply adding the original audio signal to a time-delayed version of the original signal. The time delay is small and varies in some manner.

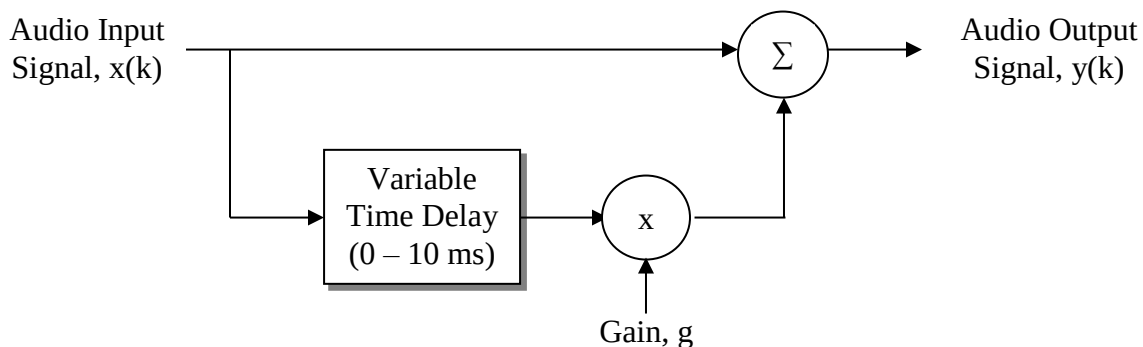


Figure 5.24: Flanging Effect Using Comb Filter with Variable Delay

Since the flanger uses a variable time delay, the notch frequencies of the comb filter are continually varying over time creating a swept comb filter. In analog systems, the flanging effect was created by playing the original audio signal on two tape decks and recording on a third tape deck. The audio engineer would press down on the rim (flange) of one of the tape reels to manually create the variable delay effect.

Example 5.12: Listening to Flanging and Chorus Effects

Open the flanging.mdl shown in Figure 5.25 from the text website. Also, copy the notes.wav file from the text website into your current MATLAB directory.

- (a) Run the model as is. With the switch in the upper position, the wave file will play without any added effects. You should hear a set of very simple piano notes.
- (b) Move the switch to the lower position to listen to the flanging effect.

Challenge Question 5.6

The amplitude in the DSP Sine Wave block in Figure 5.25 is set to 200 as is the constant DC shift. The sine wave and DC shift are added together to form the delay in sample periods. What is the range of the time delay?

- (c) The frequency in the DSP Sine Wave block is set to 0.7 Hz which is a typical choice for the flanging effect. This frequency controls the rate at which the time delay varies or equivalently, the sweep rate of the comb filter. Increase the frequency to 5 Hz and listen again. You will definitely hear that comb filter sweep and produce a pretty interesting sound effect.
- (d) This same model can be easily altered to create a simple chorus effect. Change the Constant DC shift to 1400, the amplitude of the DSP Sine Wave to 100, and the frequency of the DSP Sine Wave to 1 Hz. Now the delay will vary from 1300 to 1500 samples periods which corresponds to 29.5 to 34ms. Listen to the original wave file first by moving the switch to the upper position, and then listen to the chorus effect by moving the switch to the lower position. The delay is now long enough to produce what sounds like two instruments playing at the same time.
- (e) Change the Constant DC shift to 4000 and re-run the simulation. Now the delay is too long (88-93ms) and produces what sounds like two piano players seriously out of sync with one another.

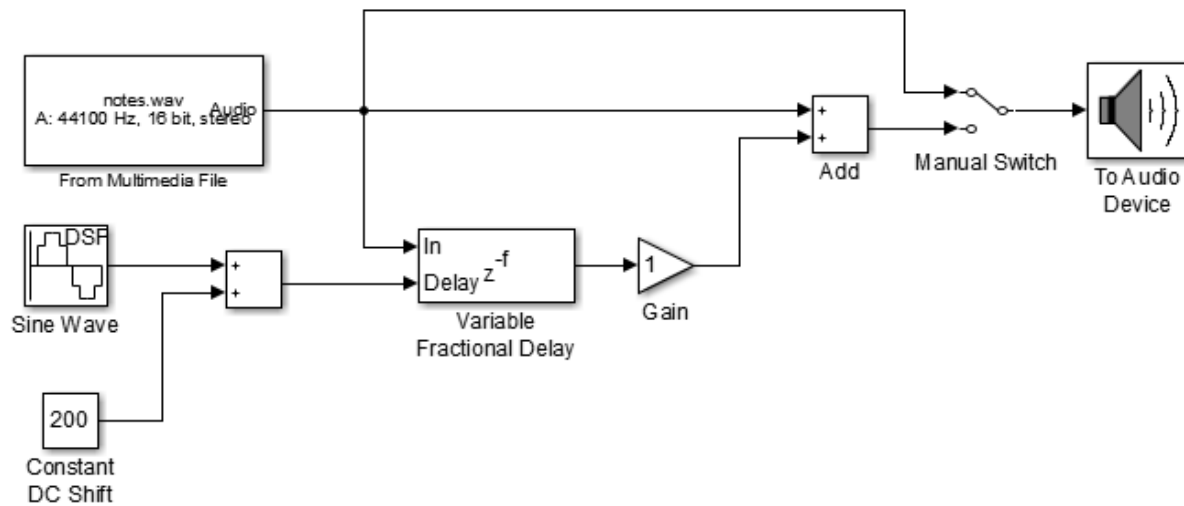


Figure 5.25: Simulink Model for Flanging and Chorus Effects

Both the flanging effect and the chorus effect involve combining the original audio signal with delayed versions of the original audio signal, but the chorus effect uses a longer delay. This longer delay allows the ear to perceive the delayed signal as a separate signal creating the “double-tracked” or richer, fuller effect. With flanging, the delay is too short for the ear to pick up as a separate signal and the “swishing” effect produced is significantly different from the chorus effect. The interested reader could replace the notes.wav file in the flanging.mdl with a favorite wave file of his or her own and experiment further with the flanging and chorus effects.

Moving Average Filters

A moving average filter is an FIR filter used to smooth out a signal and reduce the effects of random noise. It can be used to clean up noisy pulse signals or to reduce the effects of measurement noise. An N-point moving average filter has the form:

$$y(k) = \frac{1}{N}x(k) + \frac{1}{N}x(k-1) + \frac{1}{N}x(k-2) + \cdots + \frac{1}{N}x(k-N+1) \quad (5.18)$$

The output signal is simply an average of the N most recent input values. The ability of this filter to reduce the effects of random noise is illustrated in the following example.

Example 5.13: Reducing Noise using Moving Average Filters

Create a noisy pulse signal in MATLAB. Plot the output of a 3-point and an 8-point moving average filter if the input is the noisy pulse signal.

Solution: Execute the following MATLAB commands:

```
pulse = zeros(1,100); pulse(30:70)=1;
n=1:1:100;
subplot(2,2,1); plot(n,pulse);
ylim([-0.5 1.5]); title('Pulse Signal'); grid
noisy_pulse = pulse + 0.25*randn(size(pulse));
subplot(2,2,2); plot(n,noisy_pulse);
title('Noisy Pulse Signal'); grid
% 3-point MA Filter:
num=1/3*ones(1,3); den=zeros(size(num)); den(1)=1;
output = filter(num,den,noisy_pulse);
subplot(2,2,3); plot(n,output);
title('Output of 3-Point Moving Avg Filter'); grid
% 8-point MA Filter:
num=1/8*ones(1,8); den=zeros(size(num)); den(1)=1;
output = filter(num,den,noisy_pulse);
subplot(2,2,4); plot(n,output);
title('Output of 8-Point Moving Avg Filter'); grid
```

Figure 5.26 shows the clean pulse signal which is then corrupted with noise using the MATLAB function, `randn`. This noisy pulse signal is then passed through a 3-point moving average filter and through an 8-point moving average filter. The resulting reduction in noise (smoothing effect) is seen in Figure 5.26.

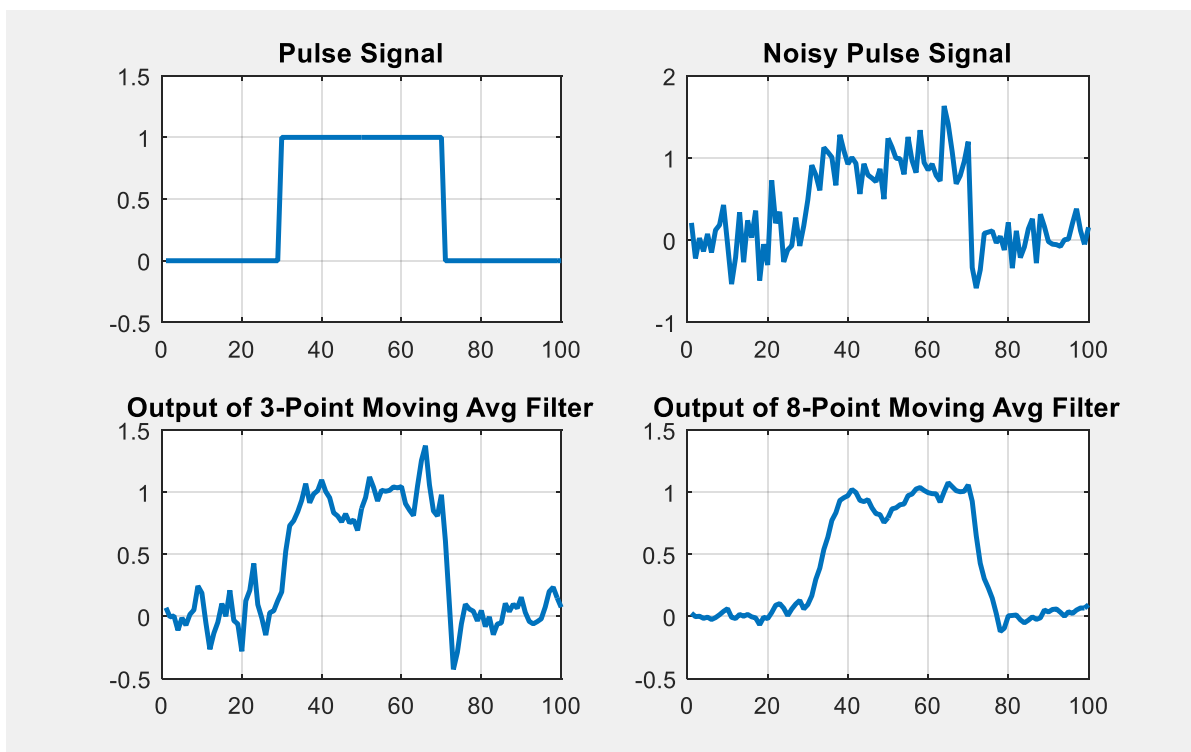


Figure 5.26: Moving Average Filters

Challenge Question 5.7

The smoothing effect (reduction in noise) appears to increase with the order of the moving average filter. Will the pulse signal continue to improve as the filter order is increased? Explain.

Gaussian Filters

Gaussian Filters are commonly used in image processing software to blur or soften an image. Images are blurred by creating a set of new pixels where each new pixel is a function of the original pixel and some of the surrounding pixels. A very simple smoothing matrix is:

$$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix} \cdot [1/3 \quad 1/3 \quad 1/3]$$

This matrix can be used to create a new smoothed image by simply averaging the original pixel (center of matrix) with the eight pixels surrounding the original pixel. This averaging function is applied to the entire original image. Filtering can be accomplished in MATLAB by using the function **conv2** to convolve a 3-point moving average filter with both the rows and the columns of the original image.

The simple smoothing matrix described above applies equal weight to the original pixel and all the surrounding pixels. A Gaussian filter puts more weight on the original pixel and smaller weighting on surrounding pixels. The weightings are determined by a Gaussian function (perhaps the term bell curve is more familiar). An example of a smoothing matrix using a Gaussian function is:

$$\left(\frac{6}{16}\right)^2 \begin{pmatrix} 0.0278 & 0.1111 & 0.1667 & 0.1111 & 0.0278 \\ 0.1111 & 0.4444 & 0.6667 & 0.4444 & 0.1111 \\ 0.1667 & 0.6667 & \mathbf{1.0000} & 0.6667 & 0.1667 \\ 0.1111 & 0.4444 & 0.6667 & 0.4444 & 0.1111 \\ 0.0278 & 0.1111 & 0.1667 & 0.1111 & 0.0278 \end{pmatrix} = \begin{pmatrix} 1/16 \\ 4/16 \\ 6/16 \\ 4/16 \\ 1/16 \end{pmatrix} \bullet \begin{bmatrix} \frac{1}{16} & \frac{4}{16} & \frac{6}{16} & \frac{4}{16} & \frac{1}{16} \end{bmatrix}$$

The original pixel (in center) is weighted the highest while the surrounding pixels receive smaller weightings. The pixels furthest from the center (original) are weighted the least. Additional uniform scaling can be added to utilize the full scale available for the image. What is the advantage of using Gaussian weighting rather than straight averaging on the pixels? Gaussian filters provide gentler smoothing and preserve the edges in an image better than similar size mean (averaging) filters.

Increasing the size of the matrix to include more of the surrounding pixels will result in an image that is more blurred. Filtering can be accomplished in MATLAB by first creating a Gaussian filter then convolving this filter with the original image using ***imfilter***. Image processing will be explored in much more detail in Chapter 9.

For now, take a look at the images in Figure 5.27. The original image was created by extracting the luminance (black and white) signal from a JPEG image file that was loaded into MATLAB using the function ***imread***. Notice that the original image appears a bit grainy particularly around the nose and the white fur on the dog's chest. The image is first smoothed using the 5x5 Gaussian matrix shown above. The result is shown in the top right window of Figure 5.27. The image is noticeably softened; graininess is reduced but not eliminated. Next the image is smoothed using a much larger (29x29) Gaussian matrix. The resulting image, shown in the bottom left window of Figure 5.27, is considerably softened eliminating the graininess in the original image but preserving the edge detail. For comparison purposes, a similar sized (28x28) mean (averaging) filter is applied to the original image. Applying a mean filter to the image results in significant loss of edge detail and a more blurry image.



Figure 5.27: Gaussian Filters for Softening Images

Linear Interpolation Filters

Interpolation filters increase the effective sampling rate by “filling in” the spaces between adjacent samples of a signal. A linear interpolation filter creates the new sample values along a straight line between adjacent existing sample values.

Sample rate is doubled by creating one new sample value between existing adjacent samples. Sample rate is tripled by creating two new sample values between existing adjacent samples. These extra sample values are created by first inserting zeros (placeholders) between the existing samples then passing the signal through a low-pass interpolation filter. The process is illustrated in Figure 5.28.

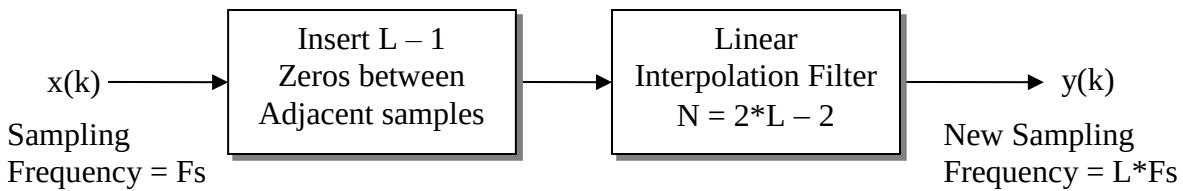


Figure 5.28: Linear Interpolation

Table 5.3 shows the filter coefficients, order, and the corresponding difference equation for several linear interpolation filters.

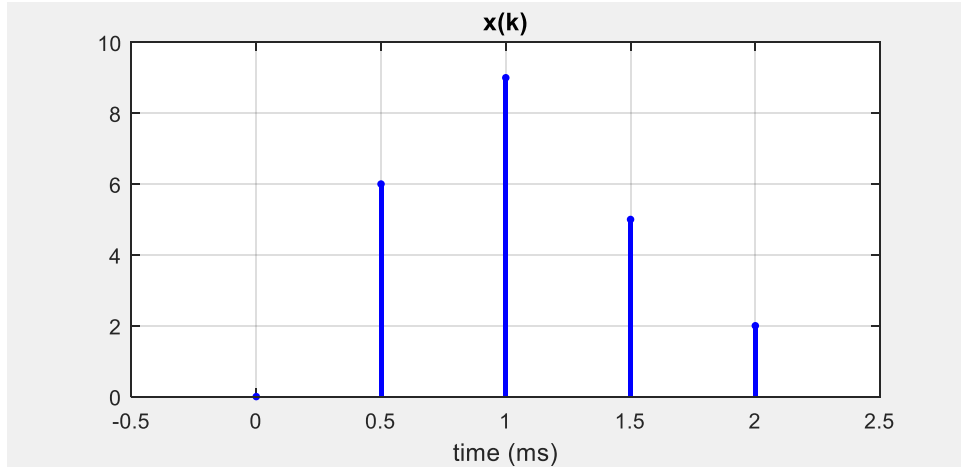
Table 5.3: Linear Interpolation Filters

Filter Order	Up-sample Rate	Filter Coefficients	Difference Equation
2	$\uparrow 2$	$[\frac{1}{2} \ 1 \ \frac{1}{2}]$	$y(k) = 0.5x(k) + x(k-1) + 0.5x(k-2)$
4	$\uparrow 3$	$[\frac{1}{3} \ \frac{2}{3} \ 1 \ \frac{2}{3} \ \frac{1}{3}]$	$y(k) = \frac{1}{3}x(k) + \frac{2}{3}x(k-1) + x(k-2) + \frac{2}{3}x(k-3) + \frac{1}{3}x(k-4)$
6	$\uparrow 4$	$[\frac{1}{4} \ \frac{1}{2} \ \frac{3}{4} \ 1 \ \frac{3}{4} \ \frac{1}{2} \ \frac{1}{4}]$	$y(k) = \frac{1}{4}x(k) + \frac{1}{2}x(k-1) + \frac{3}{4}x(k-2) + x(k-3) + \frac{3}{4}x(k-4) + \frac{1}{2}x(k-5) + \frac{1}{4}x(k-6)$

Example 5.14: Linear Interpolation

Consider the digital signal $x(k)$ sketched below.

- Triple the effective sampling rate by inserting two additional sample values (equally spaced and on a straight line) between each set of existing values.
- What are the mathematical equations used to determine the additional sample values.
- Verify that inserting two zeros between the original sample values then using the filter defined in Table 5.3 creates the new values determined in part (a) using the mathematical equations determined in part (b).



Solution:

- (a) The original signal $x(k) = [0 \ 6 \ 9 \ 5 \ 2]$ with a sampling frequency of $F_s = 1 / (0.5\text{ms}) = 2 \text{ kHz}$. To triple the effective sampling rate, two new (equally spaced) sample values must be inserted between each set of existing values. Linear interpolation places these samples along a straight line between the existing samples. Figure 5.29 shows the new, interpolated $x(k)$. The interpolated $x(k) = [0 \ 2 \ 4 \ 6 \ 7 \ 8 \ 9 \ 7.6667 \ 6.3333 \ 5 \ 4 \ 3 \ 2]$ with a sampling frequency of $F_s = 1 / (0.5/3 \text{ ms}) = 6 \text{ kHz}$ (tripled).

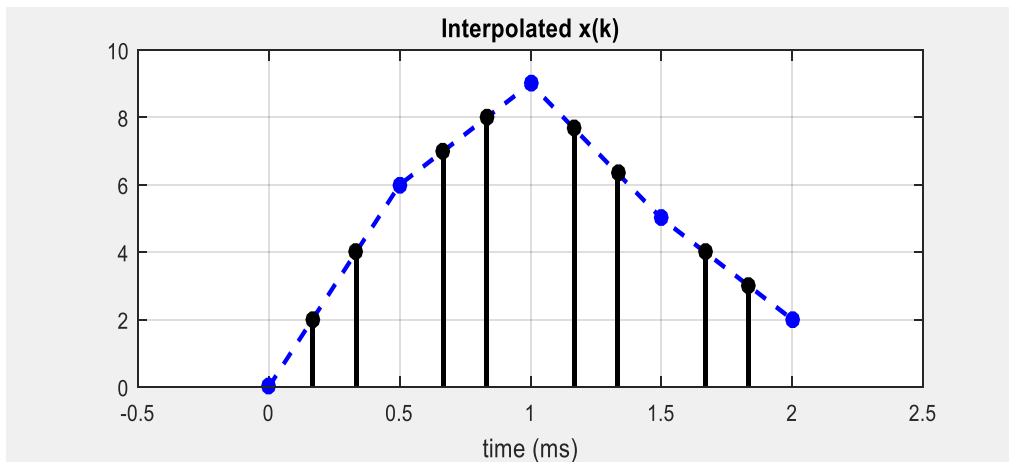


Figure 5.29: Interpolated Signal for Example 5.13

- (b) Mathematically:

$$\text{1st NewSample} = \text{PrevSample} + \frac{\text{NextSample} - \text{PrevSample}}{3} = \frac{2}{3} \text{PrevSample} + \frac{1}{3} \text{NextSample}$$

$$\text{2nd NewSample} = \text{PrevSample} + 2 \cdot \frac{[\text{NextSample} - \text{PrevSample}]}{3} = \frac{1}{3} \text{PrevSample} + \frac{2}{3} \text{NextSample}$$

PrevSample and NextSample are simply the two adjacent original signal values that the interpolated values (1st NewSample and 2nd NewSample) will be inserted between.

(c) Inserting two zeros between adjacent samples of the original x(k) gives the sequence

$$x_{\text{new}} = [0 \ 0 \ 0 \ 6 \ 0 \ 0 \ 9 \ 0 \ 0 \ 5 \ 0 \ 0 \ 2]$$

Now filter the sequence using an FIR filter with coefficients $[1/3 \ 2/3 \ 1 \ 2/3 \ 1/3]$ as defined in Table 5.3. This can be easily done in MATLAB using convolution as follows:

```
h=[1/3 2/3 1 2/3 1/3];
x_new = [0 0 0 6 0 0 9 0 0 5 0 0 2];
y= conv(h,x_new)
```

```
y =      0      0      0  2.0000  4.0000  6.0000  7.0000  8.0000  9.0000
      7.6667  6.3333  5.0000  4.0000  3.0000  2.0000  1.3333  0.6667
```

Note that the values for y are the same as those in the graph of the interpolated signal (Figure 5.29) with the exception of the two initial zeros in the sequence y. These two zeros represent the time delay through the filter; that is, the new sample values cannot be calculated until both adjacent original samples are available. So the first interpolated value between the original sample values of 0 and 6 cannot be computed until $t = 0.5$ ms (i.e., the time when the value 6 becomes available). This time delay was discussed in Chapter 3 for signal reconstruction. Linear interpolation introduces a time delay of one sampling instant while more sophisticated interpolation filters which use several adjacent sample values introduce longer time delays.

To relate to the mathematical equations from part (b), a partial convolution will be done by hand:

x_new					0	0	0	6	0	0	9	0	0	5	...	
h, flipped		1/3	2/3	1	2/3	1/3	→									y(0)
Sum of Overlap products															=	0
x_new					0	0	0	6	0	0	9	0	0	5	...	
h, flipped			1/3	2/3	1	2/3	1/3	→								y(1)
Sum of products															=	0

Two Extra Initial Zeros – Time Delay

x_new				0	0	0	6	0	0	9	0	0	5	...	
h, flipped			1/3	2/3	1	2/3	1/3	→							y(2)
Sum of Overlap Products				0										=	0

Picked up First Original Sample Value

x_new			0	0	0	6	0	0	9	0	0	5	...	
h, flipped		1/3	2/3	1	2/3	1/3	→							y(3)
Sum of Overlap products			2/3* 0			1/3* 6							=	2

*1st New Sample between 0 and 6 = 2/3*0+1/3*6*

x_new		0	0	0	6	0	0	9	0	0	5	...	
h, flipped		1/3	2/3	1	2/3	1/3	→						y(4)
Sum of Overlap products		1/3* 0			2/3* 6							=	4

*2nd New Sample between 0 and 6 = 1/3*0+2/3*6*

x_new		0	0	0	6	0	0	9	0	0	5	...	
h, flipped			1/3	2/3	1	2/3	1/3	→					y(5)
Sum of Overlap products					6							=	6

Picked up Second Original Sample Value

x_new		0	0	0	6	0	0	9	0	0	5	...	
h, flipped				1/3	2/3	1	2/3	1/3	→				y(6)
Sum of Overlap products					2/3* 6			1/3* 9				=	7

*1st New Sample between 6 and 9 = 2/3*6+1/3*9*

x_new		0	0	0	6	0	0	9	0	0	5	...	
h, flipped					1/3	2/3	1	2/3	1/3	→			y(7)
Sum of Overlap products					1/3* 6			2/3* 9				=	8

*2nd New Sample between 6 and 9 = 1/3*6+2/3*9*

x_new		0	0	0	6	0	0	9	0	0	5	...	
h, flipped						1/3	2/3	1	2/3	1/3	→		y(8)
Sum of Overlap products								9				=	9

Picked up Third Original Sample Value

It should be clear from the convolution detailed in this example how the insertion of the appropriate number of zeros between adjacent samples and the choice of filter coefficients creates two new samples along a straight line between adjacent samples of the original signal and retains the original signal samples.

5.7 MULTI-RATE DIGITAL SIGNAL PROCESSING

Multi-rate digital signal processing involves increasing or decreasing the sample rate of a signal. Why change the sample rate of a signal? There are several reasons. Interfacing two systems that operate at different sampling rates requires a change in the sample rate of any signal that is passed from one system to the next. As discussed in Chapter 3, A/D converters sometimes sample a signal at a much higher rate than necessary then reduce the sampling rate digitally in order to simplify the design of the analog anti-aliasing filter. Over-sampling in sigma-delta A/D converters spreads the quantization noise over a wider frequency range and the noise shaping filter further reduces the in-band noise. The digital signal is then down-sampled to match the operating frequency of the processor. In audio processing and image processing, lower sampling rates are often used for transmission and/or storage of signals. The signal is then interpolated (sampling rate increased) to improve the quality of the audio signal or resolution of the image.

Interpolation

Interpolation is the process of increasing the sampling rate of a signal by filling in data between existing data points. The process is illustrated in Figure 5.30. In Section 5.7, linear interpolation filters, where the extra data points are created along a straight line between existing data points, was explored. There are many other types of interpolation filters including sinc filters, raised cosine filters, and the optimal FIR filters introduced in Section 5.2. In general, sinc, raised cosine and optimal interpolation filters will produce a better quality signal than a linear interpolation filter; however, these filters introduce more time delay because they require multiple sample values of the signal rather than two adjacent signal values to create the interpolated values.

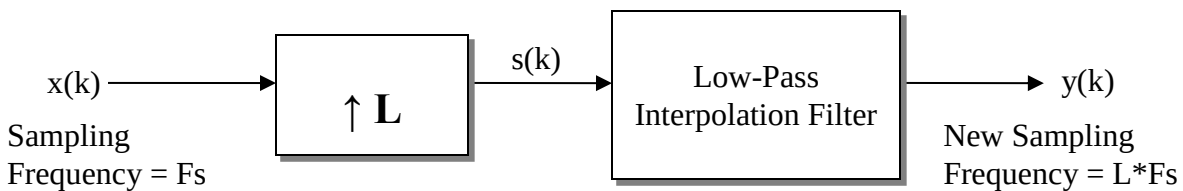


Figure 5.30: Interpolation

The specifications for the low-pass interpolation filter are:

▪ Passband	$0 \leq f \leq f_p$
▪ Maximum Passband Ripple	δ_p (dB)
▪ Stopband	$f \geq F_s/2$
▪ Minimum Stopband Attenuation	A_s (dB)
▪ Sampling Frequency	$L \cdot F_s$

The bandwidth of $x(k)$ is f_p and this bandwidth must be less than $F_s/2$.

The interpolation process (time domain) is illustrated in Figure 5.31 for $L = 4$. The original signal $x(k)$ is shown in the top plot. The center plot is of the signal $s(k)$ which is created from $x(k)$ by inserting $L - 1 = 3$ zeros between adjacent samples of $x(k)$. The up-sampled signal $s(k)$ is then filtered to produce the interpolated signal $y(k)$. The signal $y(k)$ has a sampling frequency that is four times the sampling frequency of the original signal $x(k)$.

The interpolation process (frequency domain) is illustrated in Figure 5.32 for $L=4$. The top plot shows the original spectrum for $x(k)$. The center plot shows the spectrum for the up-sampled signal, $s(k)$, which consists of the spectrum for $x(k)$ plus spectral images between $F_s/2$ and $L*F_s/2$. Increasing the sampling frequency results in an increase of system bandwidth! The low-pass interpolation filter eliminates the spectral images and the spectrum of the interpolated signal essentially matches the original signal spectrum.

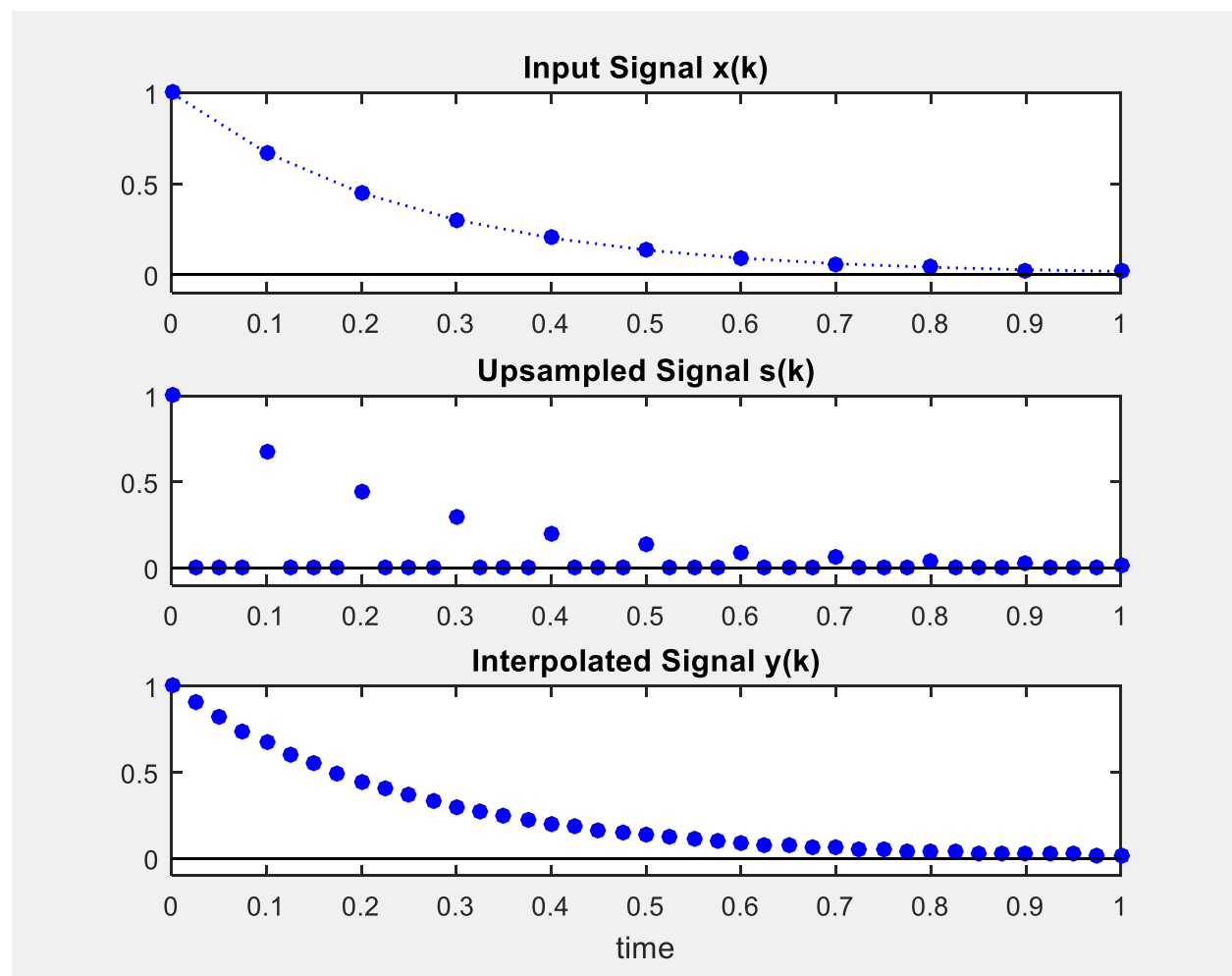


Figure 5.31: Interpolation - Time Domain ($L=4$)

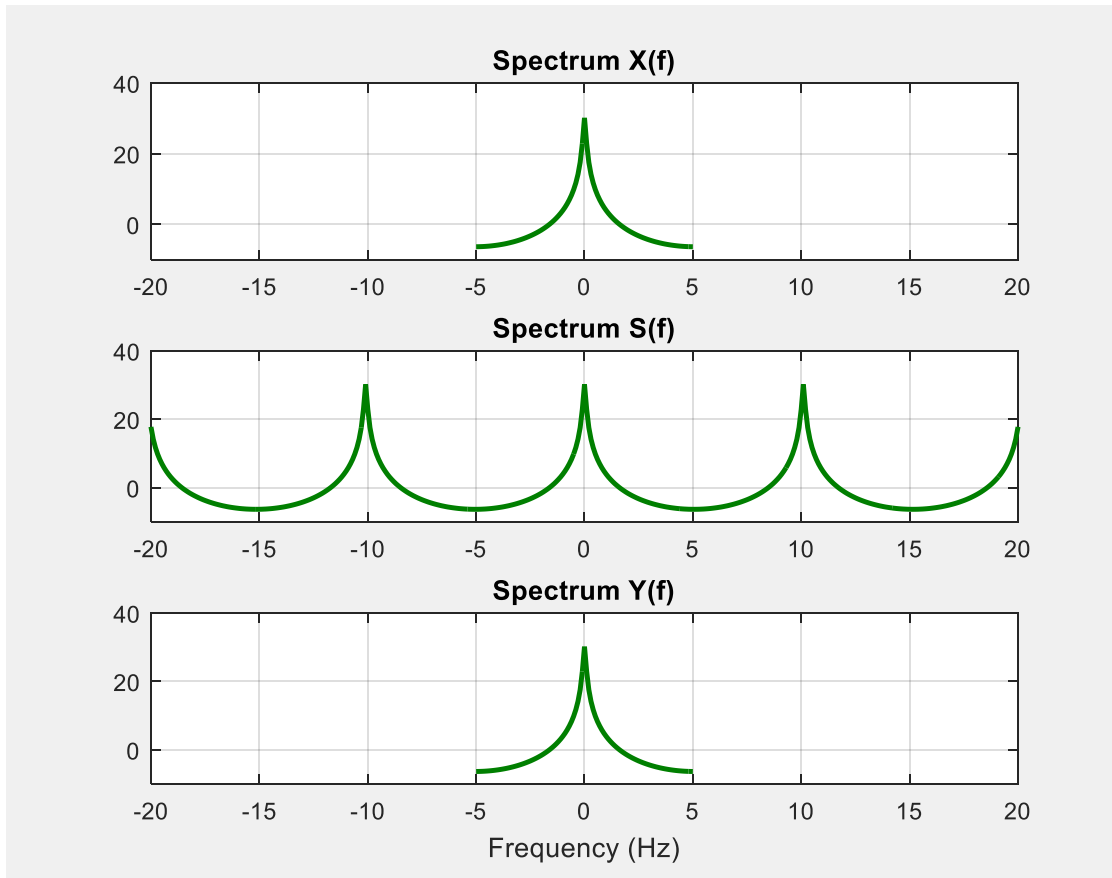


Figure 5.32: Interpolation - Frequency Domain (L=4)

Decimation

Decimation is the process of reducing the sampling rate of a signal by eliminating data points. The process is illustrated in Figure 5.33. Decimating by a factor of 2 means every other data value for the signal $x(k)$ is eliminated. Decimating by a factor of 3 requires elimination of two of every three original data points.

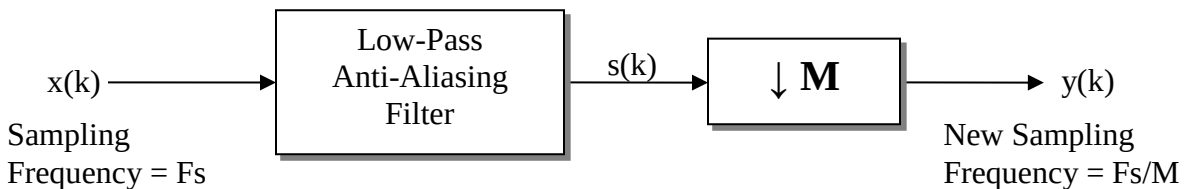


Figure 5.33: Decimation

The anti-aliasing filter must filter out any spectral components above $F_s/(2 \cdot M)$ since reducing the sampling frequency will result in a lower system bandwidth. Without this filter, any spectral components above $F_s/(2 \cdot M)$ would alias to lower frequencies causing signal distortion. It is critical that the signal $x(k)$ not have any significant spectral components above $F_s/(2 \cdot M)$ because these components will be filtered out.

The specifications for the low-pass anti-aliasing filter are:

- Passband $0 \leq f \leq f_p$
- Maximum Passband Ripple δ_p (dB)
- Stopband $f \geq F_s/(2 \cdot M)$
- Minimum Stopband Attenuation A_s (dB)
- Sampling Frequency F_s

f_p is the bandwidth of $x(k)$ and must be less than $F_s/(2 \cdot M)$

Decimation in the time domain for $M=3$ is illustrated in Figure 5.34. The top plot shows the original signal $x(k)$. Filtering this signal and retaining one of every three original sample values results in the decimated signal $y(k)$ shown in the bottom plot.

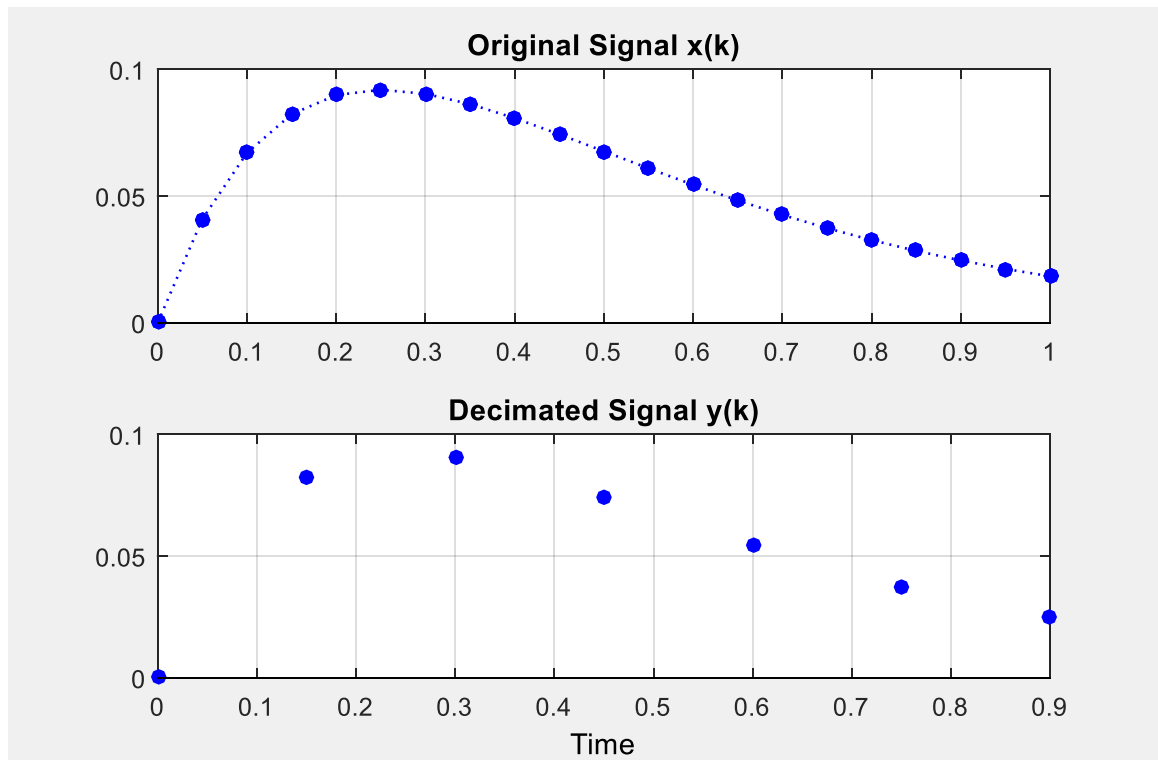


Figure 5.34: Decimation – Time Domain (M=3)

Non-Integer Rate Conversion

The interpolation and decimation processes discussed so far result in an integer change in the sampling rate. What happens if the original signal is sampled at 48 kHz and this sampling frequency must be converted to 44.1 kHz? In this case, a combination of interpolation and decimation must be used to accomplish the required rate change as follows:

$$\frac{44.1}{48} = \frac{441}{480} = \frac{147}{160}$$

The process of interpolating by 147 then decimating by 160 to achieve the desired sampling rate is illustrated in Figure 5.35. Notice that the interpolation filter and the anti-aliasing filter can be replaced with a single low-pass filter. The specifications for the filter are also listed in the figure.

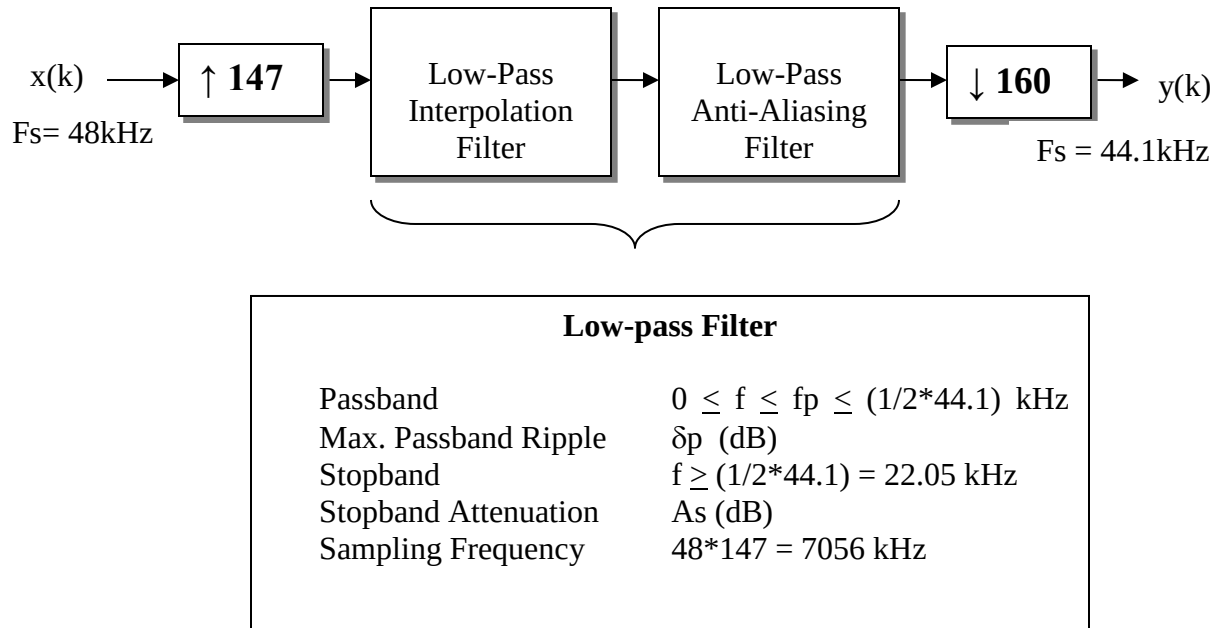


Figure 5.35: Rate Conversion by Non-Integer Factors

Challenge Question 5.8

In Figure 5.35, the signal $x(k)$ was interpolated first by a factor of 147 then decimated by a factor of 160 to get the new sample rate of 44.1 kHz. Would it work equally well to decimate $x(k)$ first by a factor of 160 then interpolate by a factor of 147 to produce the new signal at 44.1 kHz? Why or why not?

Multi-Stage Rate Conversion

If a signal must be decimated or interpolated by a very large factor, the rate conversion should be done in multiple stages rather than in a single stage. The following two examples illustrate the benefits of multi-stage rate conversion.

Example 5.15: Multi-Stage Rate Conversion using FIR Filters

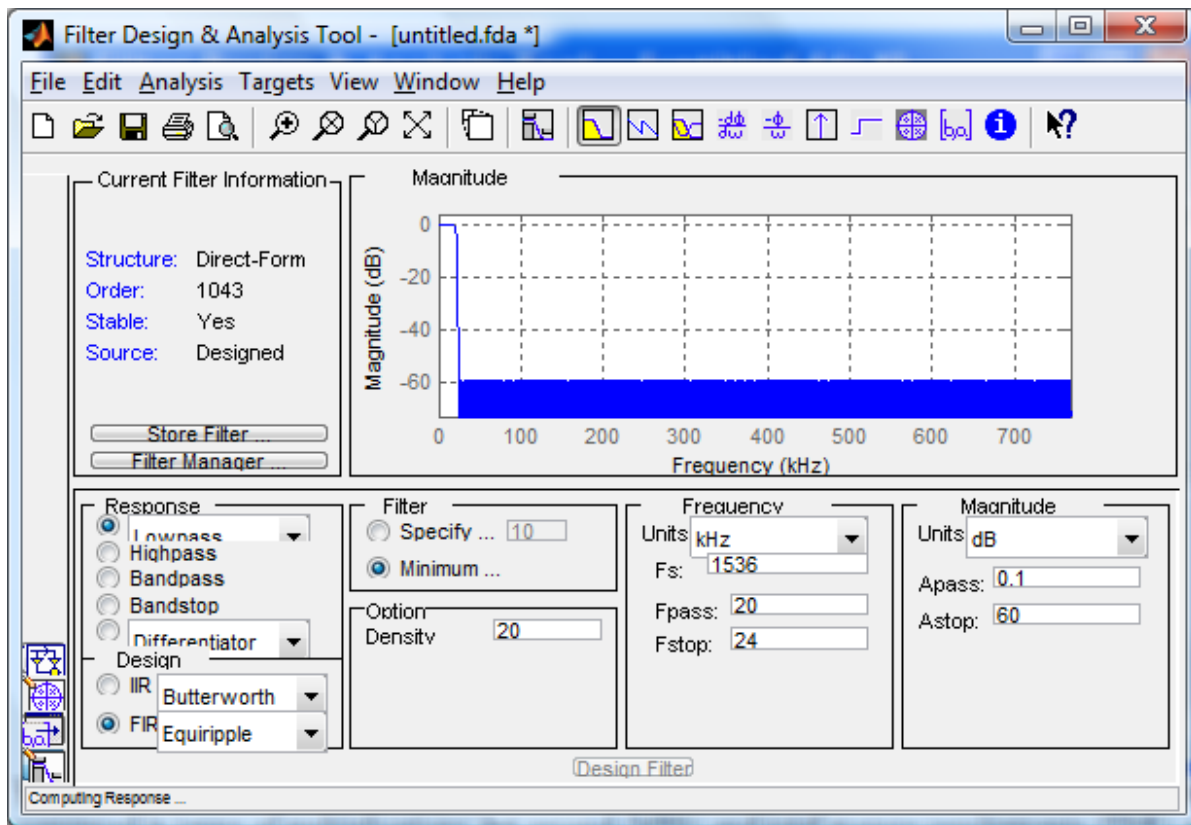
A signal of bandwidth 20 kHz is originally sampled at 1.536 MHz. Design a single stage decimator to reduce the sampling rate to 48 kHz. The filter should have no more than 0.1 dB of ripple in the passband and a stopband attenuation of at least 60 dB.

Solution:

The decimation factor is $1.536 \text{ MHz} / 48 \text{ kHz} = 32$. The anti-aliasing filter must meet the following specifications:

Signal Passband	0 to 20 kHz
Stopband	$\geq 24 \text{ kHz}$
Passband Ripple	$\leq 0.1 \text{ dB}$
Stopband Attenuation	$\geq 60 \text{ dB}$
Sampling Frequency	1.536 MHz

This filter will be designed using the FDAT in MATLAB. Selecting an FIR equiripple filter with the specifications listed above gives:



The filter order is 1043. The computational effort and storage requirements for this filter can be computed in terms of multiplications per second (MPS) and total storage requirements (TSR) as follows:

$$\text{MPS} = (N + 1) \cdot \frac{F_s}{M} = (1043 + 1) \cdot 48\text{kHz} = 50.112 \cdot 10^6 \text{ multiplications/sec}$$

$$\text{TSR} = (N + 1) = 1044 \text{ filter coefficients} \quad (5.19)$$

The computational effort and storage requirements can be significantly reduced by using several stages to complete the decimation. A 3-stage decimator is shown in Figure 5.36 and the filter specifications for each stage are listed in Table 5.4.

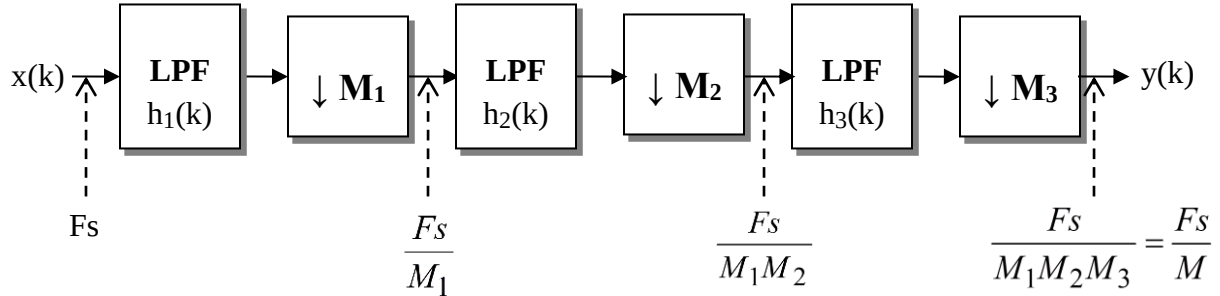


Figure 5.36: 3-Stage Decimation

Table 5.4: Filter Specifications for Three-stage Decimation

	STAGE 1 FILTER	STAGE 2 FILTER	STAGE 3 FILTER
Passband	$0 \leq f \leq f_p$	$0 \leq f \leq f_p$	$0 \leq f \leq f_p$
Max. Passband Ripple	$\frac{\delta_p}{3}$	$\frac{\delta_p}{3}$	$\frac{\delta_p}{3}$
Stopband	$f \geq \frac{F_s}{M_1} - \frac{F_s}{2M}$	$f \geq \frac{F_s}{M_1 M_2} - \frac{F_s}{2M}$	$f \geq \frac{F_s}{M_1 M_2 M_3} - \frac{F_s}{2M} = \frac{F_s}{2M}$
Min. Stopband Attenuation	A_s (dB)	A_s (dB)	A_s (dB)
Sampling Frequency	F_s	$\frac{F_s}{M_1}$	$\frac{F_s}{M_1 M_2}$

Note: Maximum Passband Ripple = $\frac{\delta_p}{\text{Number of Stages}}$

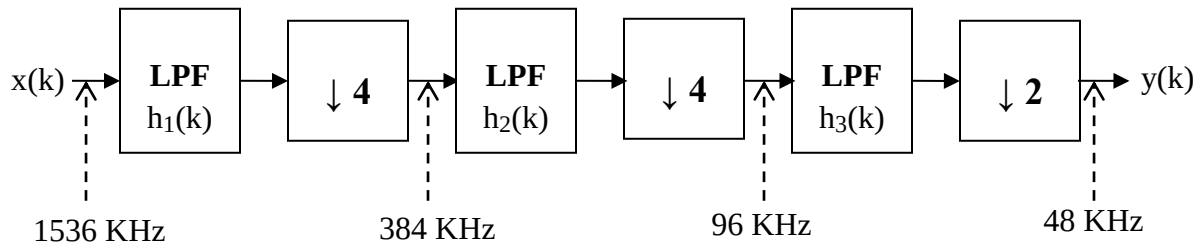
Example 5.16: Multi-Stage Rate Conversion using FIR Filters

A signal of bandwidth 20 kHz is originally sampled at 1.536 MHz. Design a 3-stage decimator to reduce the sampling rate to 48 kHz. The filter should have no more than 0.1 dB of ripple in the passband and a stopband attenuation of at least 60 dB. Compute multiplications per second (MPS) and total storage requirements (TSR) for the 3-stage decimator and compare the results to the single stage decimator designed in Example 5.15.

Solution:

As calculated in Example 5.15, the decimation factor is $1.536 \text{ MHz} / 48 \text{ kHz} = 32$. This process can be divided into 3 stages by finding 3 integers with a product of 32. There are a couple of possibilities: (4)(4)(2) and (8)(2)(2). Both will be designed in this example.

Design 1



Filter Specifications for Design 1

	STAGE 1 FILTER	STAGE 2 FILTER	STAGE 3 FILTER
Passband	$0 \leq f \leq 20 \text{ kHz}$	$0 \leq f \leq 20 \text{ kHz}$	$0 \leq f \leq 20 \text{ kHz}$
Max. Passband Ripple	$\frac{0.1}{3}$	$\frac{0.1}{3}$	$\frac{0.1}{3}$
Stopband	$f \geq 360 \text{ kHz}$	$f \geq 72 \text{ kHz}$	$f \geq 24 \text{ kHz}$
Min. Stopband Attenuation	60 dB	60 dB	60 dB
Sampling Frequency	1536 kHz	384 kHz	96 kHz

Each filter stage is designed using the FDAT in MATLAB by selecting an optimal FIR equiripple low-pass filter with the specifications listed above. The resulting filter orders and the computational and storage requirements based on Equation 5.19 are:

Stage 1: 12th order

Stage 2: 22nd order

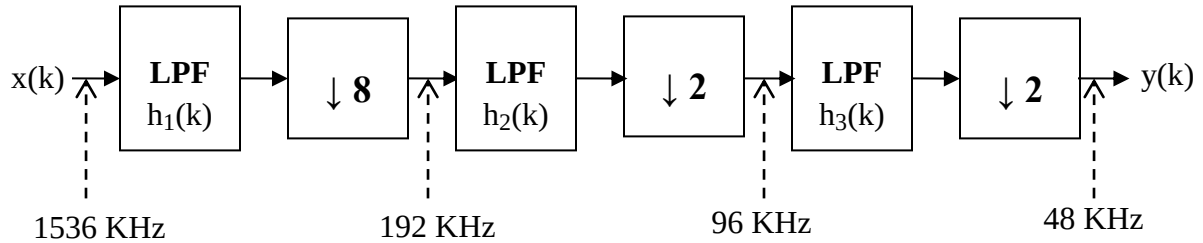
Stage 3: 73rd order

$$\text{MPS} = (12 + 1) * 384 \text{ kHz} + (22 + 1) * 96 \text{ kHz} + (73 + 1) * 48 \text{ kHz} = 10.752 * 10^6$$

$$\text{TSR} = (12 + 1) + (22 + 1) + (73 + 1) = 110$$

So, the first 3-stage design (Design 1) reduces the number of multiplies/sec by about 80% and the total storage requirements by about 90%.

Design 2



Filter Specifications for Design 2

	STAGE 1 FILTER	STAGE 2 FILTER	STAGE 3 FILTER
Passband	$0 \leq f \leq 20 \text{ kHz}$	$0 \leq f \leq 20 \text{ kHz}$	$0 \leq f \leq 20 \text{ kHz}$
Max. Passband Ripple	$\frac{0.1}{3}$	$\frac{0.1}{3}$	$\frac{0.1}{3}$
Stopband	$f \geq 168 \text{ kHz}$	$f \geq 72 \text{ kHz}$	$f \geq 24 \text{ kHz}$
Min. Stopband Attenuation	60 dB	60 dB	60 dB
Sampling Frequency	1536 kHz	192 kHz	96 kHz

Each filter stage is designed using the FDAT in MATLAB selecting an optimal FIR equiripple low-pass filter with the specifications listed above. The resulting filter orders are as follows:

Stage 1: 32nd order

Stage 2: 9th order

Stage 3: 73rd order

$$\text{MPS} = (32 + 1) * 192 \text{ kHz} + (9 + 1) * 96 \text{ kHz} + (73 + 1) * 48 \text{ kHz} = 10.848 * 10^6$$

$$\text{TSR} = (32 + 1) + (9 + 1) + (73 + 1) = 117$$

The second 3-stage design requires 96,000 more multiplies/sec than the first 3-stage design but still offers significant computational savings when compared to the single-stage design. The 2nd design also has a slightly higher storage requirement than the first 3-stage design.

Guidelines for Multi-stage Conversion:

- Typically use 2, 3, or 4 stages – rarely need more than 4 stages
- In general choose $M_1 > M_2 > M_3 \dots$ although in some cases equality might be necessary
- Experiment with the number of stages and the design of the stages in order to achieve the best computational efficiency

Challenge Question 5.9

Consider the first stage of the 3-stage decimator (Design 1) in Example 5.16. The specifications for the Stage 1 anti-aliasing filter list a stopband frequency of 360 kHz. However, since the output frequency of this stage is 384 kHz, any noise with frequencies above 192 kHz ($384/2$) will alias to a lower frequency. Why is this OK and what is the advantage of specifying the higher stopband frequency of 360 kHz rather than 192 kHz? *Hint: think about where the noise frequencies from 192 to 360 kHz will alias to.*

Answers to Chapter 5 Challenge Questions

Question 5.1 Look back at the difference equation in Example 5.1. What does this filter do numerically to an input sequence?

$$y(k) = 1/3x(k) + 1/3x(k-1) + 1/3x(k-2)$$

The difference equation can be written in a slightly different form:

$$y(k) = \frac{x(k) + x(k-1) + x(k-2)}{3}$$

The filter takes three adjacent values of the input sequence and averages the values. This type of FIR filter is referred to as a 3-point moving average filter. Moving average filters are very useful for reducing random noise in a signal.

Question 5.2 Increasing filter order clearly improves the frequency response and brings the filter closer to an ideal low-pass filter. However, increasing the filter order will require more processing (multiplications and additions) to produce the filter output. How many multiplications are required for an Nth order FIR filter?

This is a bit of a trick question. At first glance, it looks like we need N+1 multiplications to compute the output because

$$y(k) = h(0)x(k) + h(1)x(k-1) + \dots + h(N)x(k-N)$$

However, look at the graphs of the filter coefficients and the actual filter coefficients computed in Example 5.2. The filter coefficients exhibit symmetry; that is, the first coefficient is the same as the last coefficient, the second coefficient is the same as the second to last coefficients, and the pattern continues:

$$h(0) = h(N) \quad h(1) = h(N-1) \quad h(2) = h(N-2) \quad \dots$$

If we re-arrange the difference equation a bit, $y(k)$ can be computed as follows:

$$y(k) = h(0) [x(k) + x(k-N)] + h(1) [x(k-1) + x(k-N+1)] + h(2) [x(k-2) + x(k-N+2)] \dots$$

So, the number of required multiplications can be cut in half to $(N+1)/2$ (round this number up to an integer if N is even). Speeding up computations by exploiting things like symmetry is something DSP engineers do very well!

Question 5.3 Why is the output response in Figure 5.17 very small from the initial time of zero to about 0.8 ms? In other words, why does it appear that there is a time delay in the output response?

To answer this question, think about how the output of the filter is computed; that is, go back to the input/output difference equation for the FIR filter:

$$y(k) = b_0x(k) + b_1x(k-1) + b_2x(k-2) + \dots + b_{87}x(k-87)$$

The output $y(k)$ depends on the current input and, in this particular example, 87 past inputs. However, when the filter is initialized at $t = 0$, all we have is the current input. Presumably, all inputs prior to start time are zero. So, we have: $y(0) = b_0x(0)$. At the next sampling instant, we would have the current input and one past input: $y(1) = b_0x(1) + b_1x(0)$.

In other words, it takes some time to load the input buffer with values which causes a transient response in the output. The input buffer is fully loaded with values after 87 sampling instants which corresponds to a time of 1.8125 ms ($87/F_s = 87/48 \text{ kHz}$).

Question 5.4 What is a possible disadvantage of scaling down the filter coefficients to avoid arithmetic overflow?

Scaling down the filter coefficients can create increased coefficient quantization error. Smaller coefficients require more precision in terms of bits.

Question 5.5

If the sampling frequency in Figure 10.11 is 44100 Hz, what should the integer delay, d_1 , be to produce a time delay of 10 ms?

The time delay is the product of the integer delay and the sampling period, $d_1 \cdot T_s$. To produce a time delay of 10 ms, the integer delay would be:

$$d_1 = \text{round}\left(\frac{\Delta t}{T_s}\right) = \text{round}(\Delta t \cdot F_s) = \text{round}(0.010s \cdot 44100 \text{ Hz}) = 441$$

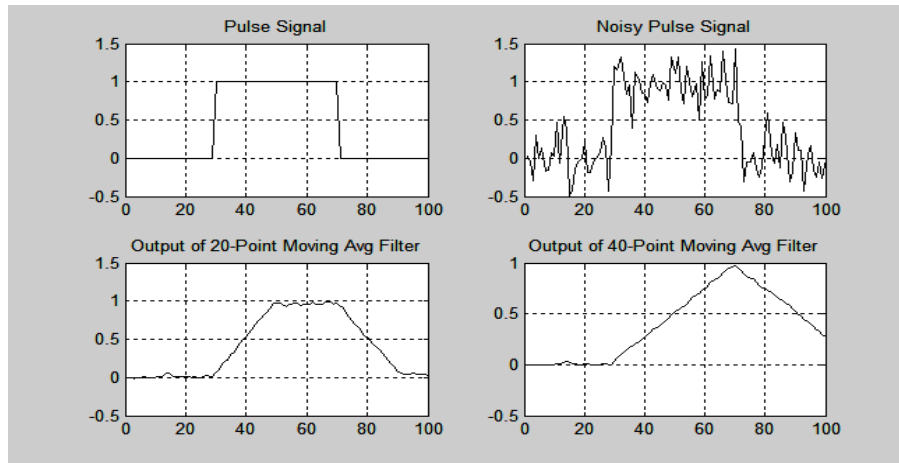
Question 5.6

The amplitude in the DSP Sine Wave block in Figure 10.14 is set to 200 as is the constant DC shift. The sine wave and DC shift are added together to form the delay in sample periods. What is the range of the time delay?

With an amplitude of 200 for the sine wave and a dc offset of 200, the delay will range from 0 to 400 sample periods. Since the sample rate is 44100 Hz, this corresponds to a time delay from 0 to 9.07ms ($400 \text{ samples} / [44100 \text{ samples/sec}] \cdot 1000 \text{ ms/sec}$).

Question 5.7 The smoothing effect (reduction in noise) appears to increase with the order of the moving average filter. Will the pulse signal continue to improve as the filter order is increased? Explain.

No, the pulse signal will not continue to improve as the filter order is increased because the variation in the signal itself will start to be averaged out. Using commands similar to those in Example 5.13 with a 20-point and a 40-point moving average filter, result in the output signals shown in the following graph. The degradation in the pulse signal as filter order becomes too high is clearly illustrated in the following set of plots.

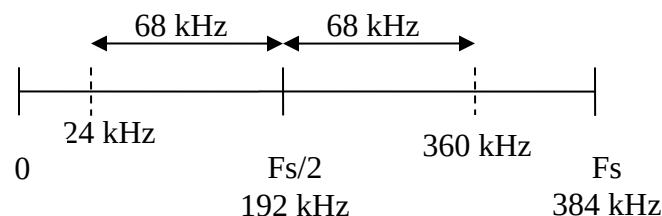


Question 5.8 In Figure 5.35, the signal $x(k)$ was interpolated first by a factor of 147 then decimated by a factor of 160 to get the new sample rate of 44.1 kHz. Would it work equally well to decimate $x(k)$ first by a factor of 160 then interpolate by a factor of 147 to produce the new signal at 44.1 kHz? Why or why not?

No. Although decimating the signal by 160 then interpolating by 147 would certainly result in a signal with the desired sampling rate of 44.1 kHz, this signal would be much lower quality than a signal created by interpolation followed by decimation. Why? Decimating the original signal by 160 means that 159 original sample values out of every 160 are being discarded. The remaining samples are then filled in with 147 interpolated values. The quality of this interpolated signal (match to original signal) would suffer because so many original sample values were discarded prior to interpolation. It is much better to create the interpolated signal using all the original sample values and then decimate (discard values) to achieve the desired sampling frequency.

Question 5.9 Consider the first stage of the 3-stage decimator (Design 1) in Example 5.16. The specifications for the Stage 1 anti-aliasing filter list a stopband frequency of 360 kHz. However, since the output frequency of this stage is 384 kHz, any noise with frequencies above 192 kHz ($384/2$) will alias to a lower frequency. Why is this OK and what is the advantage of specifying the higher stopband frequency of 360 kHz rather than 192 kHz? (Hint: think about where the noise frequencies from 192 to 360 kHz will alias to).

For a sampling frequency of 384 kHz, any frequency above 192 kHz ($384/2$) will alias to a lower frequency. Use the fold-over method discussed in Chapter 3 to determine alias frequencies. Noise frequencies from 192 to 360 kHz will fold-over into the 192 to 24 kHz range as shown in the diagram below.



Why is this OK? In the final decimation stage, anything above 24 kHz will be eliminated by the anti-aliasing filter in the final stage. So, noisy signals may alias and pass through the first couple of stages but they will be eliminated in the last stage. These signals do not alias into the final frequency range of interest (0 – 24 kHz in this example). So why push the stopband frequency to 360 kHz instead of just using 192 kHz? Pushing the stopband frequency as large as possible relaxes the filter specifications (lower quality requirement) and allows for a lower order filter which reduces the computational complexity of the design. If you check the stopband frequency for the 2nd stage in Design 1 and for the first two stages in Design 2 of Example 5.16, you will find that the stopband frequency is chosen as large as possible without allowing aliasing into the 0-24 kHz range. In other words, it is the frequency that would alias to exactly 24 kHz.

Chapter 5 Problems

Problem 5.1:

- (a) Write the difference equation for a **6-point** moving average filter.
- (b) Find the filter order (it is not 6) and the filter impulse response.
- (c) Find the filter transfer function and the filter poles.
- (d) Find the output response to the input sequence $\{1, 0, 3, 4, -3, -2, 5\}$

Problem 5.2: Using Equation 5.6, find the coefficients and write the difference equation for a 6th order low-pass FIR filter with a cutoff frequency of $f_c=6000\text{Hz}$ and a sampling frequency of $F_s = 48000\text{ Hz}$.

Problem 5.3: Using Equation 5.12, find the coefficients and write the difference equation for an 8th order bandpass FIR filter with a center frequency $f_o = 2\text{ kHz}$, a bandwidth of 1 kHz , and a sampling frequency $F_s = 8\text{ kHz}$.

Problem 5.4:

- (a) Using the Fourier series method with a Blackman window and the Filter Design and Analysis Tool (FDAT), design a digital FIR filter to satisfy the specifications listed below. Provide the magnitude frequency response and the filter coefficients for your design.
 - i. Passband $20 - 30\text{ kHz}$ $A_p \leq 1.5\text{ dB}$
 - ii. Stopband $f < 15\text{ kHz}$ and $f > 35\text{ kHz}$ $A_s \geq 50\text{ dB}$
 - iii. Sampling Frequency $f_s = 96\text{ kHz}$
- (b) Repeat (a) using a Hamming window. Comment on any differences or similarities in the two designs.
- (c) Suppose the input signal to the filter is $x(t) = \sin(2\pi(10000)t) + \sin(2\pi(25000)t)$. Plot the input signal and the output signal for each of the filter designs. Comment on results.

Problem 5.5:

- (a) Design a digital notch (bandstop) FIR filter using the optimal equiripple design option in the Filter Design and Analysis tool to meet the specifications listed below. Provide the magnitude frequency response and the filter coefficients for your design.
 - i. Notch Frequency: 18 kHz
 - ii. Attenuation at Notch Frequency $\geq 60\text{ dB}$
 - iii. Passband edge frequencies: 16 kHz and 20 kHz
 - iv. Passband Ripple $\leq 0.1\text{ dB}$
 - v. Sampling frequency $f_s = 48\text{ kHz}$
- (b) Let the input to the filter be $x(t) = \sin(2\pi(18000)t) + \sin(2\pi(22000)t)$ sampled at 48 kHz . Plot the output response of the filter and comment on the results.

Problem 5.6: Use the notch filter designed in Problem 5.5 and refer to Example 5.10.

- (a) Quantize the filter coefficients from Problem 5.5 to 16 bits and plot the frequency response of the quantized filter. Does the quantized filter still meet the filter specifications?

- (b) Repeat part (a) using 8 bits.

Problem 5.7: Use the band-pass filter designed in Problem 5.4(a) and refer to Example 5.11.

- (a) Quantize the filter coefficients of the band-pass filter to 16 bits then scale the coefficients using $\mathcal{L}_1$ scaling to reduce the possibility of arithmetic overload. Plot the frequency response for the quantized and scaled FIR filter. Does it still meet filter specifications given in Problem 5.4?
- (b) Repeat part (a) for 12 bits.

Problem 5.8: A flanging effect is created using a sinusoidal oscillator to control the time delay similar to Figure 5.25.

- (a) Assuming an audio sampling rate of 96 KHz, calculate the number of sample periods corresponding to a delay of 8 ms.
- (b) Determine the amplitude and dc offset for the sinusoidal oscillator needed to produce a time delay that varies sinusoidally from 0 to 8 ms.

Problem 5.9: Repeat Problem 5.8 for a chorus effect with a desired delay range of 25-40 ms.

Problem 5.10: Interpolation

- (a) In MATLAB, create ten cycles of a 2.5 kHz sine wave sampled at 8 kHz and plot the signal.
- (b) Use the command **upsample** to upsample the signal created in part (a) by a factor of six. Plot the new signal. Explain what **upsample** does to the signal in part (a)?
- (c) Refer to Figure 5.30 and the filter specifications following this figure. Using the Filter Design and Analysis tool, design a low-pass interpolation filter to raise the sampling rate of the signal in part (a) to 48 kHz. In your filter design, use a passband of 3.4 kHz, a maximum passband ripple of 0.1 dB, and a minimum stopband attenuation of 40 dB. Export the filter coefficients to the MATLAB workspace.
- (d) Apply the interpolation filter created in part (c) to the upsampled signal created in part (b) using the **filter** command. Plot the interpolated signal. How does the interpolated signal compare to the original signal in part (a). Why does the interpolated signal initially look so different from the original signal?

Problem 5.11:

- (a) Design a 3-stage decimator using optimal FIR filters with the following overall specifications for the decimator filter:

Voice Band (Passband)	0 to 3.4 kHz
Input Sampling Frequency	256 kHz
Output Sampling Frequency	8 kHz
Passband Ripple	< 0.01 dB
Stopband Attenuation	> 60 dB

Your solution should include a block diagram showing the decimation and sampling rate for each stage, a table of filter specifications for each stage, and screen shots from the FDAT for each design.

- (b) What would be the required filter order to do the decimation in a single stage? Turn in a screen shot of the Filter Design and Analysis Tool for this but DO NOT turn in the filter coefficients.
- (c) Compare the two designs in terms of multiplies per second (MPS) and total storage requirements (TSR).

CHAPTER 5 LAB EXERCISE

FIR Filters

Objectives

1. Implement FIR filters using MATLAB and Simulink.
2. Practice using the Filter Design and Analysis Tool for FIR filter design
3. Explore the characteristics of two commonly used FIR filters: moving average filters and linear interpolation filters.

Procedure

Exercise 1: FIR Filter Basics

Consider the FIR filter

$$y(k) = x(k) + \frac{1}{2}x(k-1) + \frac{1}{4}x(k-2)$$

(a) What is the filter order? $N =$ _____

(b) Find the impulse response, $h(k)$ for the filter.

(c) Find the transfer function, $H(z)$, of the filter. (Recall: $x(k-m) \leftrightarrow z^{-m}X(z)$)

(d) Find the poles of the FIR filter

(e) What can you conclude about the stability of an FIR filter? Why?

Exercise 2: Implementing an FIR filter in MATLAB

An N-point Moving Average filter is an FIR filter that takes N adjacent input points and averages them.

$$y(k) = \frac{1}{N}x(k) + \frac{1}{N}x(k-1) + \frac{1}{N}x(k-2) + \cdots + \frac{1}{N}x(k-(N-1))$$

The MATLAB statement **y = filter(num, den, x)** computes the output of a filter with transfer function $H(z) = \text{num}/\text{den}$ and input signal x. In this exercise, Moving Average filters of various sizes will be applied to a noisy input pulse using the MATLAB function *filter*. Type the following statements in MATLAB.

```
k = 1:1:1000;
x = zeros(size(k)); x(300:700) = 1;
plot(k,x) % Should see a nice clean pulse from n=300 to n=700.
noise = rand(1,1000) - 0.5;
x_noise = x + noise;
subplot(2,2,1); plot(k,x_noise); title('Noisy pulse');
```

The following statements will create a 5-point Moving Average Filter and compute the output of the filter when the input is the noisy pulse. The output is then plotted.

```
N=5; num = 1/N*ones(1,N);
den=zeros(size(num)); den(1)=1;
y=filter(num,den,x_noise);
subplot(2,2,2); plot(k,y); title('Order =5');
```

To complete this exercise, create a 20-point Moving Average Filter, apply the noisy pulse to this filter, and plot the output using subplot(2,2,3). Then repeat for a 50-point Moving Average Filter except plot the filter output using subplot(2,2,4).

Paste the final plot in the space below:

What does the Moving Average filter do?

Exercise 3: Creating a 20th order low-pass FIR Filter

- (a) Find and plot the filter coefficients for a low-pass filter with a sampling frequency of 48 kHz, a desired cut-off frequency of 10 kHz, and a filter order $N - 1 = 20$ as follows:

```
N = 20; % Desired filter order
k = 0:1:N; h = 2*10/48*sinc(2*10/48*(k - N/2));
subplot(2,1,1); stem(k,h); title('Filter Coefficients');
```

(b) Now plot the frequency response of the filter designed in part (a) as follows:

```
num = h; den = zeros(size(h)); den(1) = 1;
[H,f] = freqz(num,den,200,48000);
subplot(2,1,2); plot(f, 20*log10(abs(H)));
title('Magnitude Response (dB)'); grid
```

Note: type **help freqz** at the command prompt to review the function **freqz**.

Paste your plot in the space below:

Exercise 4: Creating a 40th order low-pass FIR Filter

Repeat Exercise 3 for a filter order of 40. How do the magnitude responses of the two filters compare (i.e., which filter looks closer to an ideal LPF?)

Don't clear your filter coefficients for the 40th order filter. You will need them for Exercise 5.

FIR Design Using Windowing

In Exercises 3 and 4, a FIR LPF was designed by taking ideal filter coefficients and abruptly truncating them for some desired filter order to make the filter finite and causal. This abrupt truncation creates ripple in the pass-band and stop-band of the filter. Windowing involves a gradual attenuation of filter coefficients rather than an abrupt cut. There are dozens of different windows available for FIR filter design (Hamming, Kaiser, Blackman, etc.). Abrupt truncation is referred to as a Boxcar or Rectangular window.

Exercise 5: 40th Order Low-pass FIR filter with Windowing

This exercise assumes that you still have a vector **h** with 41 coefficients for the low-pass filter from Exercise 4. Do the following in MATLAB:

```
help window
N = 40; % Filter Order
h_hamming = h .* window(@hamming,N+1)' ;
% Don't forget the transpose ' on the window vector
k = 0:1:N;
subplot(2,1,1); stem(k, h_hamming);
title('Windowed Filter Coefficients');
num = h_hamming; den = zeros(size(num)); den(1)=1;
[H,f]=freqz(num,den,200,48000);
subplot(2,1,2); plot(f, 20*log10(abs(H)));
title('Magnitude Response (dB)');grid; xlabel('Frequency (Hz)')
```

Paste your plot in the space below:

How do the frequency responses of the 40th order filter with no windowing and the 40th order filter with windowing compare in terms of pass-band ripple, stop-band ripple, and roll-off from pass-band to stop-band?

Comment: We have been focusing on low-pass filters. There are simple transformations to get filter coefficients for high-pass, notch, and band-pass filters. These transformations along with several examples are provided in Section 5.3.

FIR Design Using Optimization Techniques

The Windowing approach to FIR filter design starts with the ideal filter coefficients (time domain) then shifts and truncates them (with windowing) to make a causal finite filter. The optimization approach starts with some desired filter order and creates a set of filter coefficients that are in some sense a “best fit” design to the ideal *frequency* response. There are many algorithms available such as least squares and equiripple.

Filter Design and Analysis Tool

The filter design and analysis tool in the Signal Processing Toolbox of MATLAB automates the work done in Exercises 3, 4, and 5. To launch the Filter Design and Analysis Tool, start Simulink and open a new Model Window. In the Simulink Library Browser, look in the Signal Processing Toolbox under Filtering then Filter Designs. Drag and drop the FDAT block into your model window and double click on it.

Exercise 6: Using the Filter Design and Analysis Tool

Set up the Filter Design and Analysis Tool to design a 40th order FIR Low-Pass Filter with a Rectangular Window, a sampling frequency of 48 kHz, and a cut-off frequency of 10 kHz as shown in the figure below. After all parameters have been specified, click on Design Filter. Your FDAT window should match the one shown in Figure 1.

Compare the frequency response to the one you printed out in Exercise 4. They should be identical! Click on the icons to the right of the zoom icons to see phase response, magnitude and phase response together, group and phase delays, an impulse response (should be identical to the response printed in Exercise 4), a step response, a pole-zero distribution (all 40 poles are at the origin), and the filter coefficients.

Exercise 7: Windowing with the Filter Design and Analysis Tool

Change the Window to Hamming leaving all other specifications the same. Compare the filter coefficients to the `h_hamming` computed in Exercise 5. How do they compare?

Leave the FDAT window open with your windowed 40th order filter design for the next exercise.

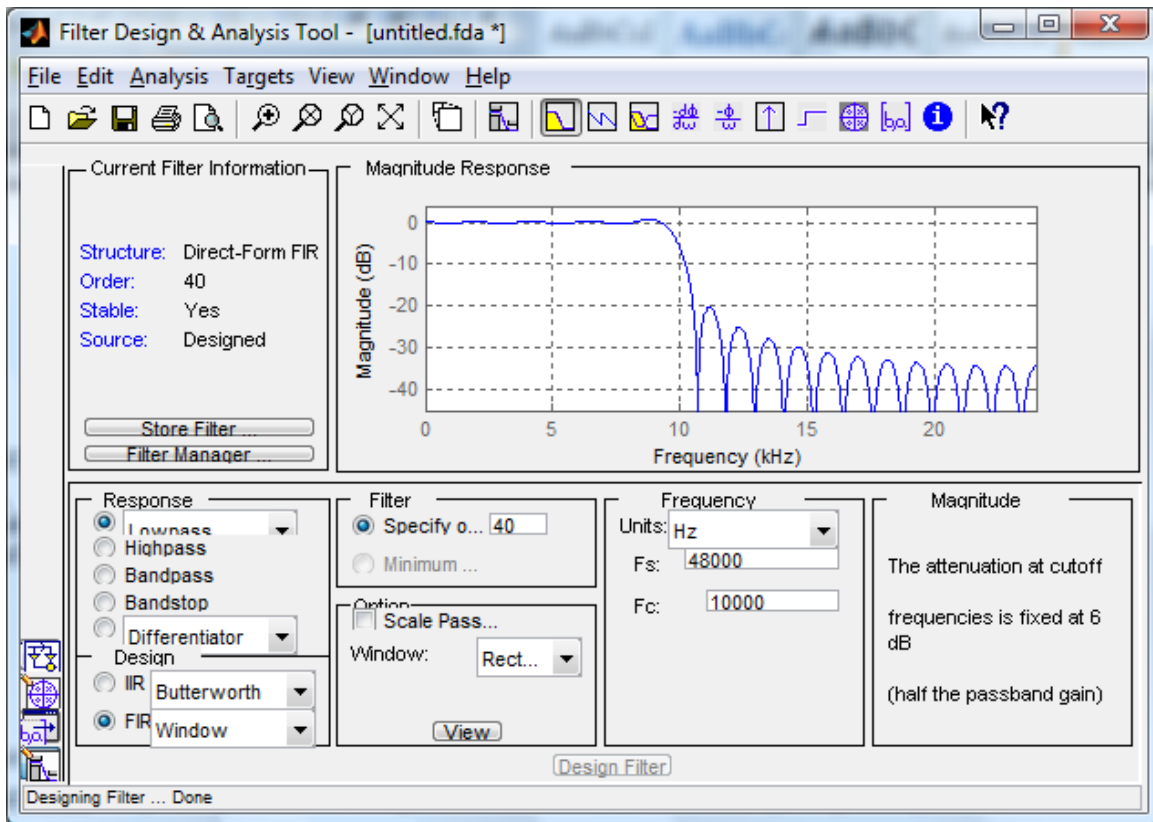


Figure 1: FDAT FIR Filter Design

Exercise 8: Exporting the Filter Coefficients to MATLAB and Computing a Response

1. Click on the magnitude response graph at 5kHz and at 15 kHz. At each point, you should see a window pop up that gives the filter magnitude at that particular frequency. Record them below.

Filter Magnitude at 5 kHz = _____ dB = _____ Straight Gain

Filter Magnitude at 15 kHz = _____ dB = _____ Straight Gain

Note: Straight Gain = $10^{(dB/20)}$

2. Now click File then Export to export the filter coefficients to the MATLAB workspace. The filter coefficients will be saved under the variable Num unless you specify something else.
3. The filter command can be used to compute an output response but we need to create the filter denominator. Recall that the denominator is $z^{\text{filter order}} = z^{40}$ in this case. The following statement works quite effectively to create the denominator:

```
Den = zeros(size(Num)); Den(1) = 1;
```

4. Next create a 5 kHz sine wave sampled at 48 kHz, then find and plot the output of the filter as follows:

```
t=0:1/48000:6/5000;  
% 6 cycles of a 5 kHz sine sampled at 48 kHz  
x = sin(2*pi*5000*t);  
y = filter(Num,Den,x);  
subplot(2,1,1); plot(t,x); grid  
subplot(2,1,2); plot(t,y); grid
```

5. Repeat step 4 for the 15 kHz sine wave but use 12 cycles instead of 6.
6. What do you observe about the output of the filter for each frequency? Do the magnitudes you recorded in step 1 of this exercise make sense? Explain.

Exercise 9: Using the Filter Design and Analysis Tool with Simulink

1. Build the model shown in Figure 2 using Simulink. The DSP sine wave is located under sources in the Signal Processing Toolbox.
2. Double Click on the FDATool and Design some type of FIR filter – your choice. List your particular specs and filter type below:
3. Put in a sine wave within your filter bandwidth. Print the output response making sure to label it with your particular input frequency.

Note: In the DSP Sine Wave, be sure to enter in your input frequency and your sample time. The sample time should be 1/Sampling Frequency used for your filter design. You may also need to adjust the stop time (under Simulation Parameters). 10 seconds may display too many cycles.

4. Repeat step 3 for a sine wave outside of your filter bandwidth.

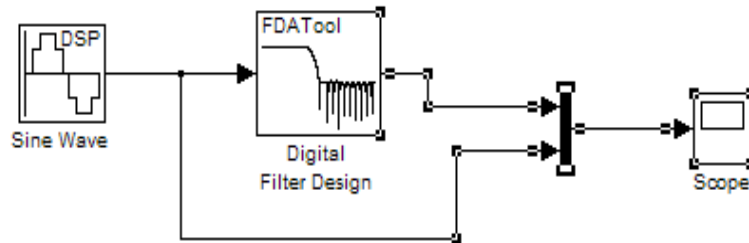


Figure 2: FDAT in Simulink®

Linear Interpolation Filter

Interpolation filters increase the effective sampling rate by “filling in” the spaces between adjacent samples of a signal. Linear interpolation inserts evenly spaced samples along a straight line between two adjacent samples of the original signal. The table below lists filter coefficients, filter order, and the corresponding difference equation for several linear interpolation filters.

Filter Order	Up-sample Rate	Filter Coefficients	Difference Equation
2	$\uparrow 2$	$[\frac{1}{2} \ 1 \ \frac{1}{2}]$	$y(k) = 0.5x(k) + x(k-1) + 0.5x(k-2)$
4	$\uparrow 3$	$[\frac{1}{3} \ \frac{2}{3} \ 1 \ \frac{2}{3} \ \frac{1}{3}]$	$y(k) = \frac{1}{3}x(k) + \frac{2}{3}x(k-1) + x(k-2) + \frac{2}{3}x(k-3) + \frac{1}{3}x(k-4)$
6	$\uparrow 4$	$[\frac{1}{4} \ \frac{1}{2} \ \frac{3}{4} \ 1 \ \frac{3}{4} \ \frac{1}{2} \ \frac{1}{4}]$	$y(k) = \frac{1}{4}x(k) + \frac{1}{2}x(k-1) + \frac{3}{4}x(k-2) + x(k-3) + \frac{3}{4}x(k-4) + \frac{1}{2}x(k-5) + \frac{1}{4}x(k-6)$

Exercise 10: Linear Interpolation by a factor of 2

1. Consider the time signal plotted below in Figure 3. Insert one sample between each pair of adjacent samples. Label the value of each new sample.
2. How does the sampling frequency of the new interpolated signal compare to the sampling frequency of the original signal?
3. How did you calculate the value for the sample being inserted? Give an equation.

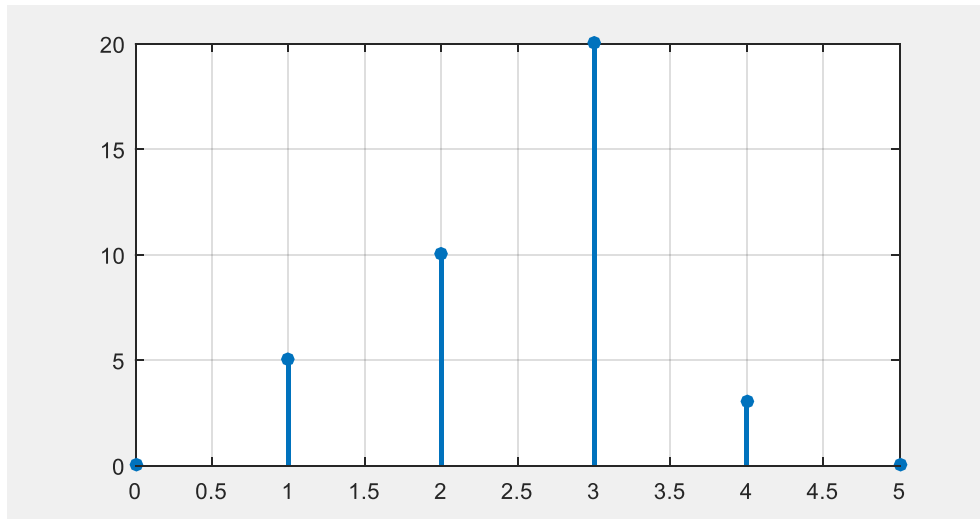


Figure 3: Discrete Signal for Exercise 10

Exercise 11: Linear Interpolation by a factor of 3

1. Consider the same time signal used in Exercise 10 and re-plotted in Figure 4. In this plot, insert two evenly spaced samples between each pair of adjacent samples.

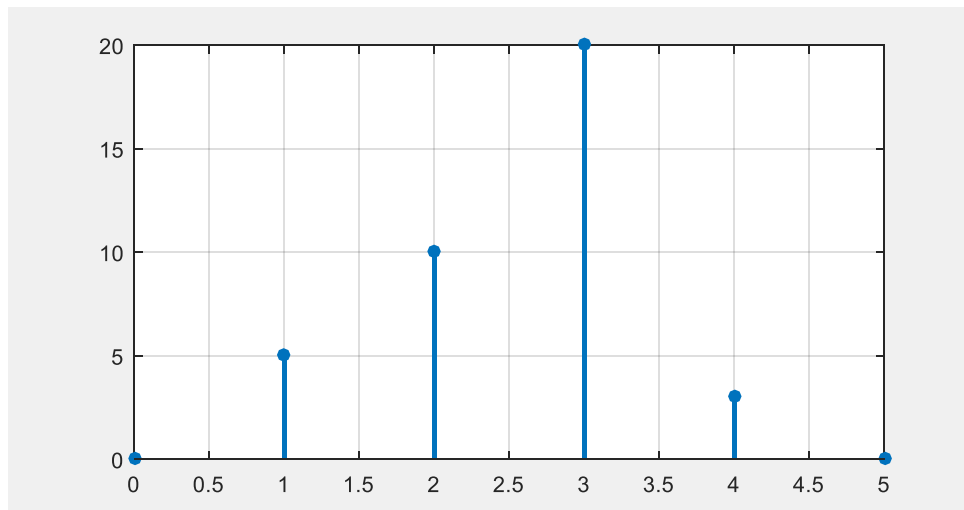


Figure 4: Discrete Signal for Exercise 11

2. How does the sampling frequency of the new signal compare with the sampling frequency of the original signal?
3. How did you calculate the values for the interpolated samples? Provide equations.

The block diagram in Figure 5 illustrates the interpolation process using DSP algorithms:

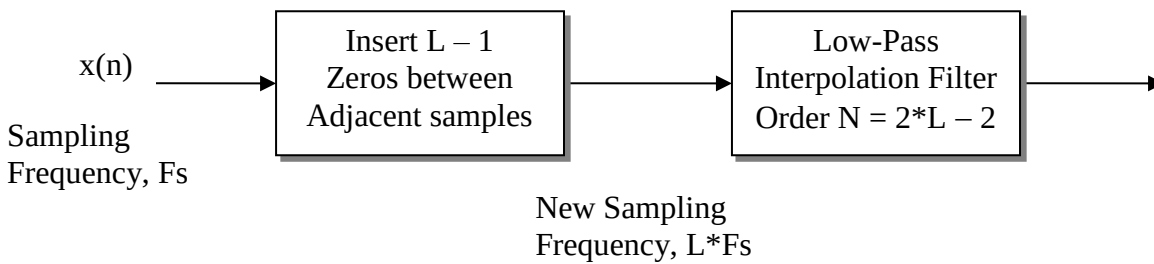


Figure 5: Interpolation in Digital Signal Processing

Exercise 12: Linear Interpolation by a factor of 2 using MATLAB

Use the following MATLAB commands to double the effective sampling rate of the signal, $x(k)$ used in Exercises 10 and 11. Assume the original signal is sampled at 10 kHz. Creating an m-file for this will make Exercise 13 go faster.

```

x = [0 5 10 20 3 0]; L=2;
Fs = 10000; %Assume an original sampling frequency of 10kHz
Fs_new = L*Fs; %Compute new sampling frequency
x_upsamp = upsample(x,L); %Insert L-1 = 1 zero between samples
%Create linear interpolation filter for factor of 2:
numf = [1/2 1 1/2]; denf = [1 0 0];
%Filter signal with inserted zeros:
x_int = filter(numf,denf,x_upsamp);
t = 0:1/Fs:(length(x)-1)*1/Fs; %Time vector for original signal
subplot(2,1,1); stem(t,x); title('Original Signal');grid
%Time vector for interpolated signal
t_int = 0:1/Fs_new:(length(x_int)-1)*1/Fs_new;
subplot(2,1,2); stem(t_int,x_int);
title('Interpolated Signal');grid
  
```

How much delay time occurs for this interpolation?

Exercise 13: Linear Interpolation by a factor of 3 using MATLAB

Triple the effective sampling rate of the signal, $x(k)$, from the previous exercises. Print your results.

How much delay time occurs for this interpolation?

Exercise 14: Linear Interpolation in Simulink

Build the model in Simulink shown in Figure 6. The DSP sine wave block, the Upsample block, and the Digital Filter Blocks are all located in the Signal Processing Blockset. The DSP sine wave block is in Signal Processing Sources, the Upsample block is in Signal Operations, and the filter is under Filtering → Filter Designs → Digital Filter (it won't look like the block in the figure until you double click on it and change it to an FIR filter). Use a 10 kHz sine wave sampled at 40 kHz and with an upsample rate of 5. Enter the appropriate filter coefficients in the filter block. Print the results.

Filter Coefficients = _____ ?

New Sampling Rate = _____ ?

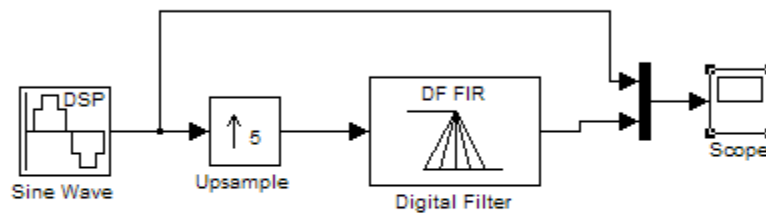


Figure 6: Simulink Model for Interpolation

CHAPTER 6 INFINITE IMPULSE RESPONSE (IIR) FILTERS

This chapter introduces infinite impulse response (IIR) digital filters. Several types of IIR filters are designed using the Filter Design and Analysis Tool (FDAT). Comparisons are made with the FIR filters introduced in the previous chapter. Practical considerations including finite wordlength effects, arithmetic rounding, arithmetic overflow, and filter realizations (cascade and parallel) are also covered.

6.1 INFINITE IMPULSE RESPONSE (IIR) DIGITAL FILTERS

IIR (Infinite Impulse Response) digital filters can be described by the input/output difference equation:

$$y(k) = \sum_{i=0}^N b_i x(k-i) - \sum_{i=1}^N a_i y(k-i) \quad (6.1)$$

- The constants $b_0, b_1, \dots, b_N$ and $a_1, \dots, a_N$ are the filter coefficients.
- N is the filter order. There are $2N+1$ filter coefficients.
- Designing an IIR filter involves finding the filter order and a set of filter coefficients that meet the given filter specifications.
- The output of the filter is a multiply and accumulate (MAC) calculation: the current input and N past inputs and outputs are multiplied by the filter coefficients. The resulting products are added together.
- To implement the filter, the filter coefficients, the current input, a finite number of past inputs, and a finite number of past outputs must be stored in memory.
- An IIR filter is recursive; that is, the output of the filter depends on past filter outputs as well as the input signal. An FIR filter depends only on the input signal and is therefore non-recursive.

Impulse Response of IIR Filters

The impulse response was defined in Chapter 4 as the output response when the input signal is a discrete unit impulse; that is $\delta(k) = 1$ for $k = 0$ and $\delta(k) = 0$ for $k \neq 0$. The impulse response of an FIR filter is simply equal to the filter coefficients and therefore is finite in length. This is not true for an IIR filter. The impulse response of an IIR filter is infinite in length; however, as long as the filter is stable then $h(k) \rightarrow 0$ as $k \rightarrow \infty$. The impulse response can be derived using the recursion method on the difference equation (although an obvious pattern doesn't always emerge), or by finding the inverse Z-Transform of the transfer function, or by using the **dimpulse** function in MATLAB®.

Transfer Function of IIR Filters

The transfer function of an IIR filter can be derived by inspection from the input/output difference equation using the time shift property of Z-transforms. The transfer function of a general IIR filter is shown in Equation 6.2.

$$H(z) = \frac{b_0 + b_1 z^{-1} + \dots + b_N z^{-N}}{1 + a_1 z^{-1} + \dots + a_N z^{-N}} = \frac{b_0 z^N + b_1 z^{N-1} + \dots + b_N}{z^N + a_1 z^{N-1} + \dots + a_N} \quad (6.2)$$

Stability of IIR Filters

Discrete-time systems are stable if the magnitude of each of the poles is strictly less than one. The poles of an IIR filter are the N roots of the denominator. Unlike an FIR filter which is always stable since all N poles are at the origin, stability can be an issue for IIR filters.

Output of an IIR Filter

The output of an IIR filter can be computed using several different methods including the following:

1. Recursively using the input/output difference equation
2. Through convolution using $y(k) = x(k) * h(k)$
3. Taking an inverse Z-Transform of $Y(z) = H(z)X(z)$
4. Through MATLAB using the command $y = \text{filter}(\text{num\_tf}, \text{den\_tf}, x)$
5. Using Fast Fourier Transforms (FFTs) to be discussed in Chapter 7

Example 6.1: Transfer Function, Impulse Response, and Output Response of an IIR Filter

An IIR filter is described by

$$y(k) = 0.3y(k-1) + 0.1y(k-2) + x(k) - x(k-1)$$

- (a) Find the filter order, the transfer function for the IIR filter, the filter poles, and determine whether or not the filter is stable.
- (b) Find the impulse response of the filter.
- (c) Find the output response for the IIR filter if the input sequence $x(k) = \{1 \ 3 \ 6 \ -1 \ 5\}$.

Solution

- (a) The filter order for the IIR filter is the largest delay on the input or output signal so $N = 2$.

The transfer function is derived by taking the Z-Transform of the difference equation and solving for $Y(z)$. Be careful about the sign of the denominator coefficients. To avoid sign errors it is a good idea to move all y terms in the difference equation to the left-hand side before determining the transfer function.

$$y(k) - 0.3y(k-1) - 0.1y(k-2) = x(k) - x(k-1)$$

$$H(z) = \frac{1 - z^{-1}}{1 - 0.3z^{-1} - 0.1z^{-2}} = \frac{z^2 - z}{z^2 - 0.3z - 0.1}$$

The filter poles are found by finding the roots of the denominator:

$$z^2 - 0.3z - 0.1 = (z - 0.5)(z + 0.2) = 0$$

The filter poles are at 0.5 and -0.2. Since both poles have magnitude smaller than one, the IIR filter is stable. The MATLAB function, **roots**, can be used for a denominator that is not easily factored by inspection.

- (b) The impulse response is the response of the system when the input is a discrete impulse. The first ten terms of the impulse response ($h(0)$ through $h(9)$) can be found using the **dimpulse** function in MATLAB as follows:

```
num_tf = [1 -1 0];
den_tf = [1 -0.3 -0.1];
h = dimpulse(num_tf,den_tf,10)
```

```
h =
    1.0000
   -0.7000
   -0.1100
   -0.1030
   -0.0419
   -0.0229
   -0.0111
   -0.0056
   -0.0028
   -0.0014
```

Since the filter is stable, the impulse response will converge to zero as $k \rightarrow \infty$. The function **dimpulse** will provide any specified number of values for $h(k)$ but does not provide a mathematical expression for $h(k)$. Solving for $h(k)$ recursively using the difference equation will provide the same finite set of values although it takes a lot longer. If a mathematical expression is required for $h(k)$, an inverse Z-Transform of the transfer function, $H(z)$, will do the trick.

$$\frac{H(z)}{z} = \frac{z - 1}{z^2 - 0.3z - 0.1}$$

The partial fraction expansion can be easily computed using the MATLAB command:

```
[coeff poles] = residue([1 -1], [1 -0.3 -0.1]);
[coeff poles]

ans =   -0.7143    0.5000
        1.7143   -0.2000
```

$$H(z) = \frac{-0.7143z}{(z-0.5)} + \frac{1.7143z}{(z+0.2)}$$

$$h(k) = -0.7143(0.5)^k + 1.7413(-0.2)^k$$

(c) There are many methods for computing the output response. Using the ***filter*** function in MATLAB gives:

```
num_tf = [1 -1 0];
den_tf = [1 -0.3 -0.1];
x = [1 3 6 -1 5];
y = filter(num_tf, den_tf, x)

y = 1.0000 2.3000 3.7900 -5.6330 4.6891
```

When using the filter function, the length of the output signal is always the same size as the length of the input signal. If more output values are desired, simply lengthen the input signal by padding with zeros as follows:

```
x = [1 3 6 -1 5 zeros(1,5)];
y = filter(num_tf, den_tf, x)

y = 1.0000 2.3000 3.7900 -5.6330 4.6891 -4.1566
    -0.7781 -0.6491 -0.2725 -0.1467
```

Comparison of IIR Filters and FIR Filters

- FIR filters are always stable even when the filter coefficients are quantized. IIR filters may not be stable particularly when filter coefficients are quantized.
- IIR filters are generally of lower order than FIR filters.
- FIR filters have a linear phase response – IIR filters do not have a linear phase response.
- IIR filters are much more sensitive to coefficient word-length effects so choosing how to implement the filter (filter realization) is critical.
- The recursive property (feedback) of IIR filters creates an increased susceptibility to arithmetic overflow.

6.2 IIR FILTER DESIGN

There are two general approaches to designing digital IIR filters: indirect and direct. Both design methods are covered in this section.

Indirect Design of IIR Filters

The indirect design method involves first designing an analog filter to meet the filter specifications then transforming the analog filter to an “equivalent” digital filter as outlined in the following steps:

1. Convert the digital filter specifications to analog specifications.
2. Convert the analog specifications to Low-Pass (LP) prototype specifications.
3. Design the LP prototype analog filter (Butterworth, Chebychev I, Chebychev II, or Elliptic).
4. Transform the LP prototype filter to the desired analog filter.
5. Transform the analog filter to an “equivalent” digital filter.

Specific details on each step will not be provided because there are several excellent software tools including FDAT which complete these design steps automatically. However, a little background on this process, which appears somewhat convoluted, will be provided. In the early stages of digital filter design, there was a wealth of information about analog filter design: equations describing the frequency response of low-pass Butterworth, Chebychev, and Elliptic filters and equations for transforming among filter types (low-pass to high-pass or band-pass or notch). So, it made sense to design a good analog filter to meet the required specifications then transform that analog filter to an “equivalent” digital filter. The transform of choice is a Bilinear transform with pre-warping shown in Equation 6.3.

$$H(z) = H(s) \Big|_s = \frac{2}{T_s \tan\left(\frac{\omega_n}{T_s}\right)} \left(\frac{z-1}{z+1} \right)$$

$T_s = 1/F_s = \text{sampling interval}$
 $\omega_n = \text{desired frequency for exact match to analog filter} \quad (6.3)$

The analog filter transfer function, $H(s)$, is converted to an equivalent digital filter transfer function, $H(z)$, using the bilinear substitution for s . Pre-warping ensures that the digital filter matches the analog filter at some specified desired matching frequency, ω_n . For low-pass or high-pass filters, the matching frequency would be the 3 dB cutoff frequency. For band-pass or notch filters, the matching frequency would be the center frequency.

The indirect method of design works very well for designing IIR digital filters. There are several excellent software tools including the Filter Design and Analysis Tool in MATLAB that utilize the indirect design method.

Direct Design of IIR Filters

The direct design method involves designing the digital filter directly (no analog intermediate design stage) to meet the filter specifications. There are two direct design algorithms available in MATLAB: **maxflat** and **yulewalk**. **Maxflat** directly designs a maximally flat digital Butterworth filter. The design algorithm can be run in the MATLAB workspace or in the Filter Design and Analysis Tool. The user must provide the order for the numerator, the order for the denominator, and the desired cut-off frequency. **Yulewalk** is a recursive filter design algorithm that uses the least squares method to match a digital filter of specified order N to a magnitude frequency response. The user provides a filter order, a vector of frequencies, and a vector of desired magnitudes at the specified frequencies. The **yulewalk** function then generates an IIR filter with the order specified that best matches the provided magnitude frequency response in a least squares sense. Both of these methods require the user to provide the filter order and will therefore require some experimentation on the part of the user to determine a reasonable filter order that successfully meets the filter specifications.

6.3 DESIGNING IIR FILTERS USING THE FDAT

The Filter Design and Analysis Tool was introduced in Chapter 5 for designing FIR filters. FDAT can also be used to design IIR filters: both indirect and direct designs. The filter types available are Butterworth, Chebychev I, Chebychev II, Elliptic, and Maximally flat. The filter type chosen will depend on the performance specifications for the filter.

Butterworth filters have a flat magnitude response in the passband and stopband but the transition region is wider than the transition region of Chebychev and Elliptic filters. Chebychev filters have a much sharper roll-off than Butterworth filters but exhibit ripple in either the passband (Chebychev I) or in the stopband (Chebychev II). Elliptic filters have the sharpest roll-off characteristics but exhibit equiripple in both the passband and stopband. Chebychev and Elliptic filters are generally of lower order than Butterworth filters and represent a better choice unless ripple in the passband and stopband are unacceptable. None of the IIR filters exhibit the ideal linear phase characteristics inherent in FIR filters, but Butterworth filters are the closest to a linear phase response.

Example 6.2: Design of Low-Pass IIR Filters using FDAT

- (a) Design an IIR low-pass filter for each of the following filter types: Butterworth, Chebychev I, Chebychev II, and Elliptic to meet the following specifications:
- Passband: 0 – 10 kHz
 - Maximum Passband Ripple: 1 dB
 - Stopband: ≥ 12 kHz
 - Minimum Stopband Attenuation: 60 dB
 - Sampling Frequency: 44.1 kHz

Make note of the filter order for each filter type. Plot the magnitude responses for all four filter types and compare them.

- (b) Plot the phase response for the Butterworth and Elliptic filters. Which filter is closer to the ideal linear response?
- (c) Find the difference equation, the transfer function, and the poles for the Elliptic filter.

Solution

(a) All four filter types are designed using FDAT in MATLAB. The screen shot for the Butterworth Design is shown in Figure 6.1. The remaining figures in this example are generated by clicking on the Full-View Analysis icon highlighted in Figure 6.1. This tool exports the data to a figure window in MATLAB that can be easily edited using plotting tools.

The minimum filter orders required to meet the given specifications are:

Butterworth:	27 th order
Chebychev I and II:	11 th order
Elliptic:	7 th order

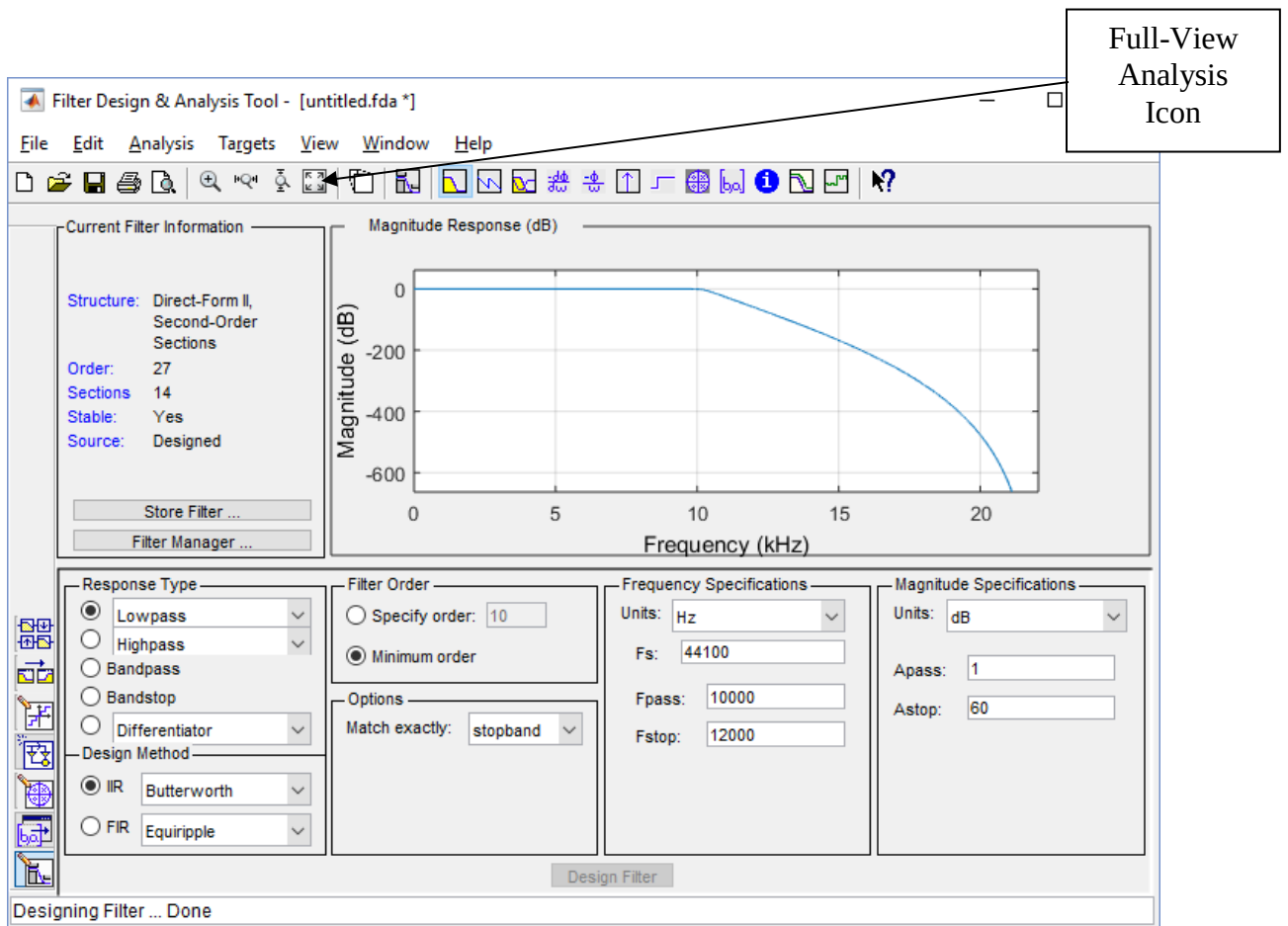


Figure 6.1: Butterworth Filter Design Using FDAT

The magnitude responses for each filter type are shown in Figure 6.2. The characteristics of each filter type are clearly illustrated. The Butterworth filter has a flat passband response with a gradual roll-off between passband and stopband. The Chebychev I filter exhibits some ripple in the passband but has a sharper roll-off than the Butterworth filter. The Chebychev II filter is flat in the passband but has ripple in the stopband. Roll-off is sharper than the Butterworth filter but not as steep as the Chebychev I filter. The Elliptic filter has the sharpest roll-off of the four filter types but exhibits ripple in both the passband and the stopband. All four filters meet the given filter specifications.

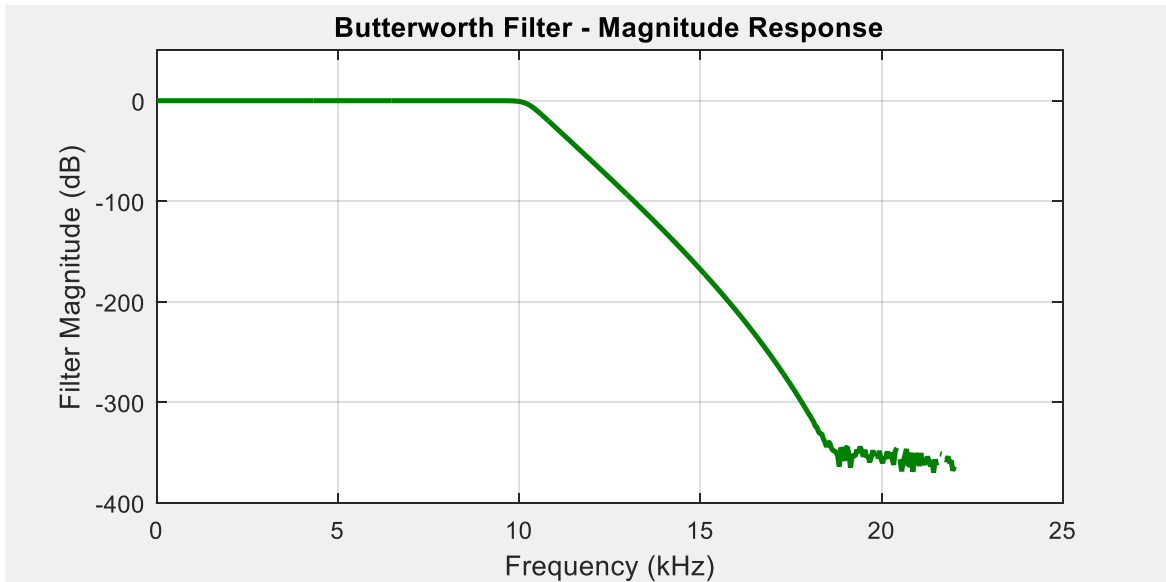


Figure 6.2a: Butterworth IIR Filter

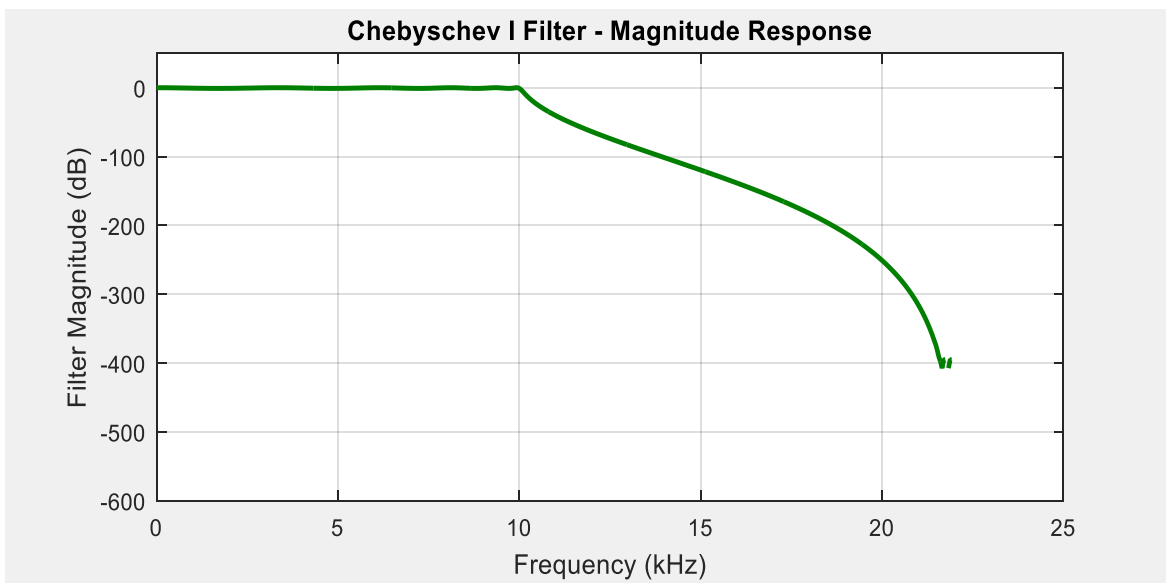


Figure 6.2b: Chebyshev I IIR Filter

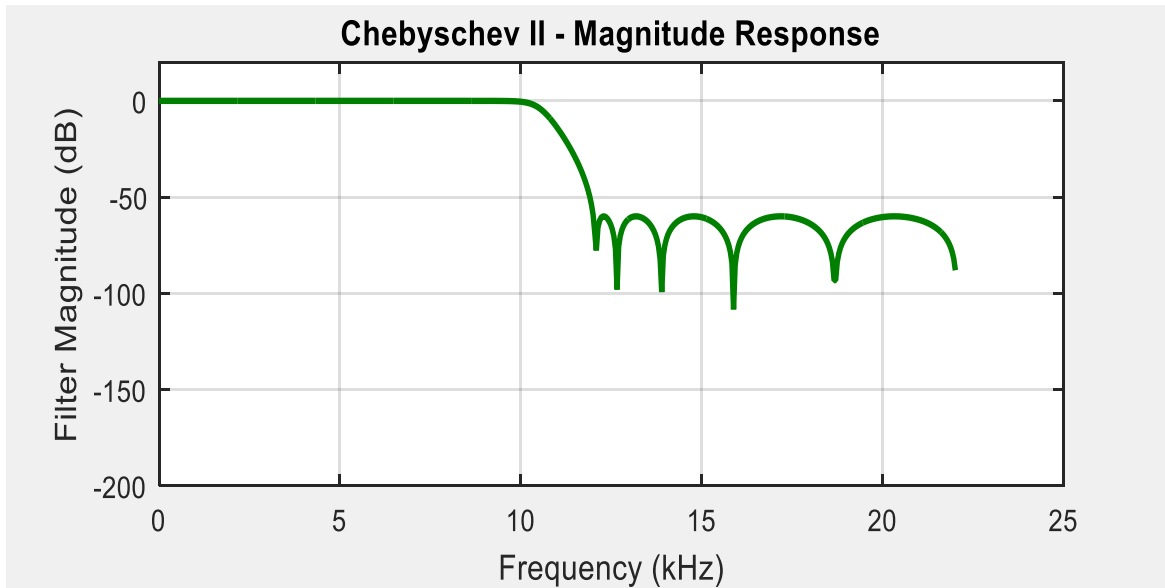


Figure 6.2c: Chebyshev II IIR Filter

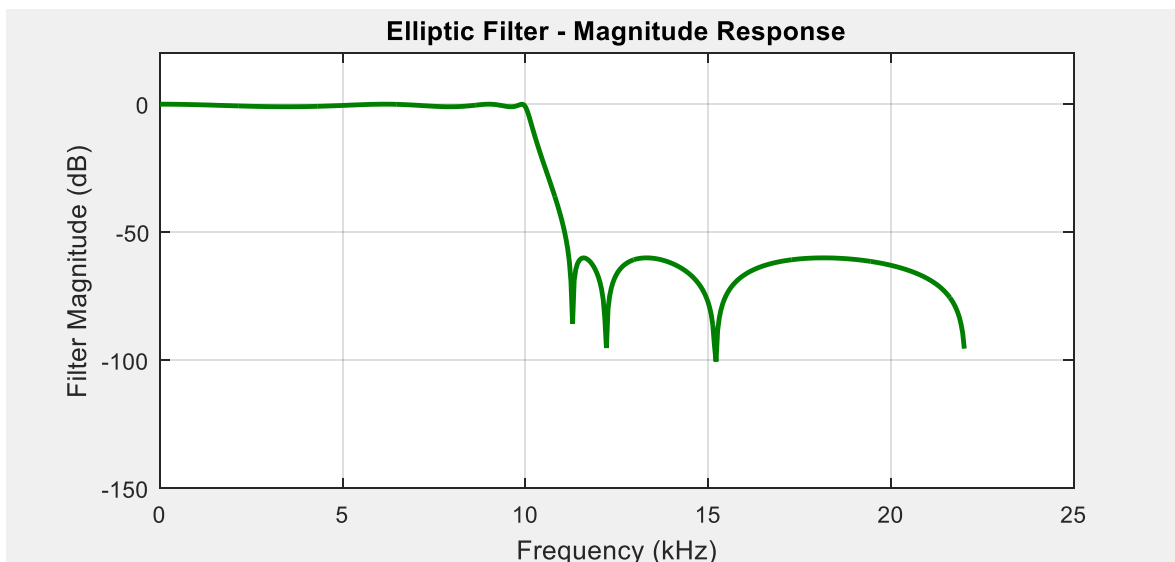


Figure 6.2d: Elliptic IIR Filter

Figure 6.2: Magnitude Responses for IIR Filters (Example 6.2)

(b) The phase responses for the Butterworth and Elliptic Filters are shown in Figure 6.3. As expected, although neither filter has an ideal linear phase response, the Butterworth filter is closest to ideal.

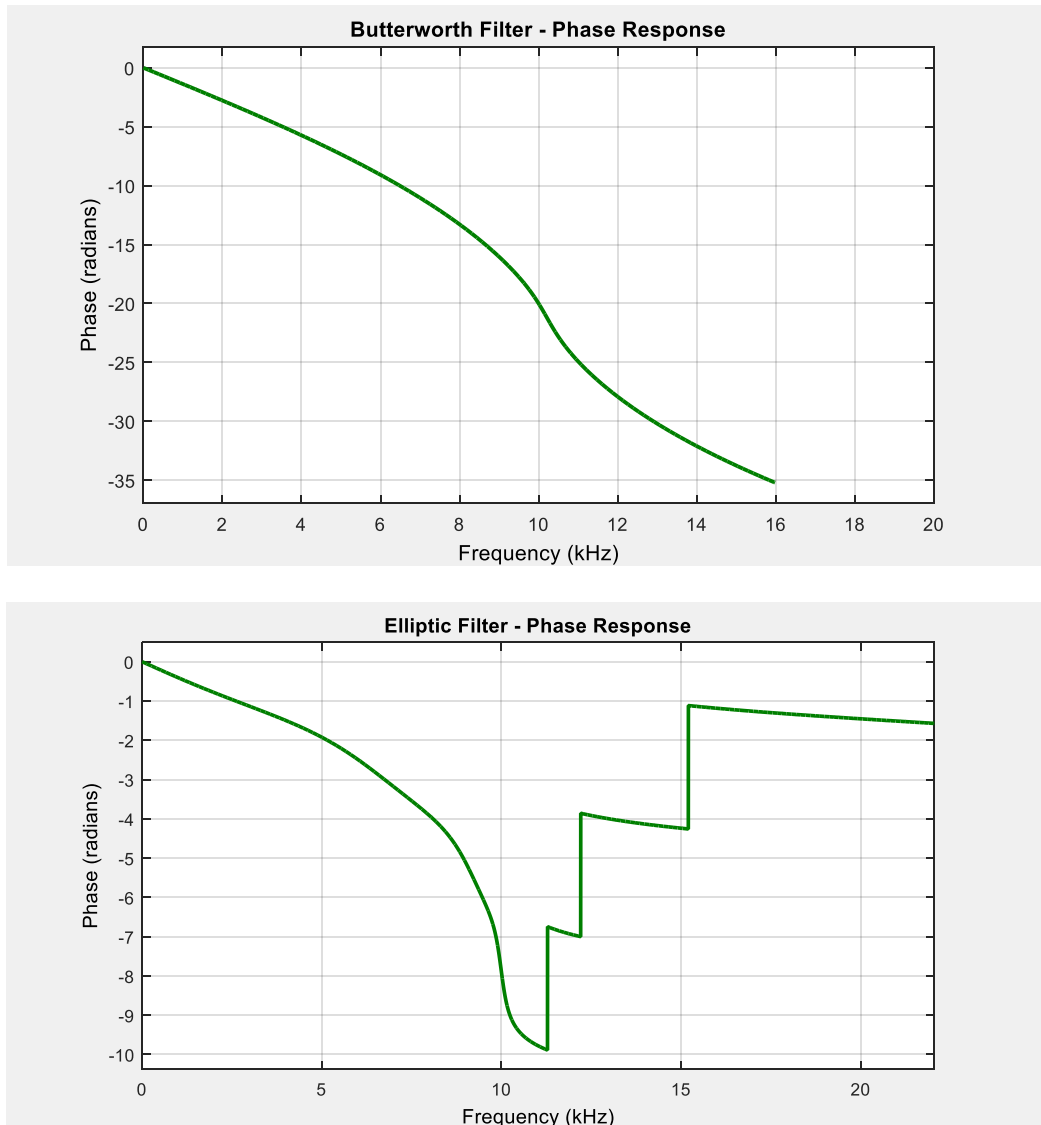


Figure 6.3: Phase Responses (Example 6.2)

(c) The filter coefficients for the Elliptic filter can be exported to the MATLAB workspace using the same Export feature used in Chapter 5 for FIR filters. Before exporting the coefficients, choose Edit → Convert to Single Section. The FDAT defaults to cascaded 2nd order sections which will be discussed later in the chapter. For this example, a single 7th order transfer function is required.

Once the design has been converted to a single section, select File → Export to export the filter coefficients to the MATLAB workspace. The default variables of Num and Den are fine for this example. Recall that the FIR filter only required a numerator because all of its poles are at the origin. This is not true for IIR filters so both a numerator and a denominator must be exported. The following MATLAB command can be used to generate a transfer function:

```
tf(Num, Den, 1/44100)
```

Transfer function:

$$\frac{0.0194 z^7 + 0.04907 z^6 + 0.09724 z^5 + 0.1274 z^4 + 0.1274 z^3 + 0.09724 z^2 + 0.04907 z + 0.0194}{z^7 - 2.262 z^6 + 4.159 z^5 - 4.745 z^4 + 4.197 z^3 - 2.6 z^2 + 1.108 z - 0.2716}$$

Sampling time: 2.2676e-005

The difference equation can be determined from the transfer function or simply from the exported numerator and denominator:

$$\begin{aligned} y(k) = & 0.0194x(k) + 0.04907x(k-1) + 0.09724x(k-2) + 0.1274x(k-3) + 0.1274x(k-4) \\ & + 0.09724x(k-5) + 0.04907x(k-6) + 0.0194x(k-7) + 2.262y(k-1) - 4.159y(k-2) \\ & + 4.745y(k-3) - 4.197y(k-4) + 2.6y(k-5) - 1.108y(k-6) + 0.2716y(k-7) \end{aligned}$$

Notice the denominator coefficients are negated because the past output terms are being moved to the right side of the difference equation.

The poles are the roots of the denominator of the filter and are determined using **roots** in MATLAB:

```
poles = roots(Den)
poles =
    0.1433 + 0.9663i
    0.1433 - 0.9663i
    0.2339 + 0.8741i
    0.2339 - 0.8741i
    0.6132
    0.4470 + 0.6058i
    0.4470 - 0.6058i

mag_poles = abs(poles)
mag_poles =
    0.9769
    0.9769
    0.9049
    0.9049
    0.6132
    0.7528
    0.7528
```

Notice that while the filter is stable (all poles have magnitude less than one) some of the poles are quite close to the edge of the unit circle. Any rounding of the filter coefficients will cause these poles to drift – potentially outside the unit circle which would make the filter unstable.

Example 6.3: Design of a Band-Pass IIR Filter using FDAT

- (a) Design an IIR elliptic band-pass filter to meet the following specifications. Determine the required filter order and plot the frequency magnitude response.
- Passband: 8 – 12 kHz
 - Maximum Passband Ripple: 0.1 dB
 - Stopband: ≤ 7 kHz and ≥ 13 kHz
 - Minimum Stopband Attenuation: 30 dB
 - Sampling Frequency: 48 kHz
- (b) Compare this design to the optimal FIR elliptic filter designed in Chapter 5, Example 5.8.

Solution

- (a) The band-pass IIR elliptic filter is designed using FDAT. The minimum filter order to meet the given specifications is 10th order. The magnitude and phase responses are shown in Figure 6.4.

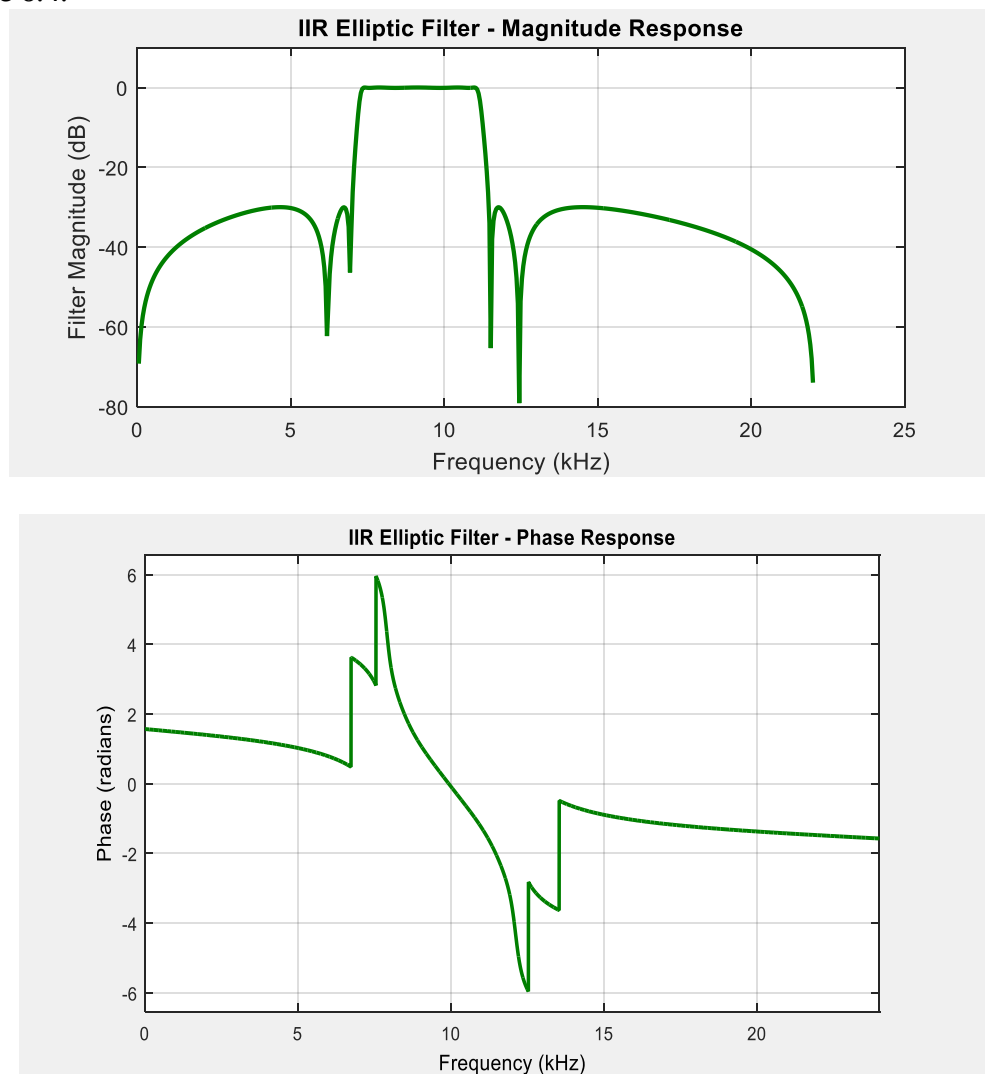


Figure 6.4: 10th Order IIR Elliptic BPF: Magnitude and Phase Response

(b) An FIR Equiripple filter with the same specifications was designed in Example 5.8. The minimum filter order to meet the given specifications was 87th order. The magnitude and phase responses are shown in Figure 6.5.

The IIR filter is able to meet the given specifications with a much lower filter order and therefore less computational complexity than the FIR filter. Both filters exhibit a small amount of ripple in the passband (within specs) and ripple in the stopband. The phase response of the FIR Equiripple filter is linear (ideal); whereas, the phase response of the IIR Elliptic filter is very nonlinear. The IIR filter would therefore be an inappropriate choice in an application such as video where signal phase information is important.

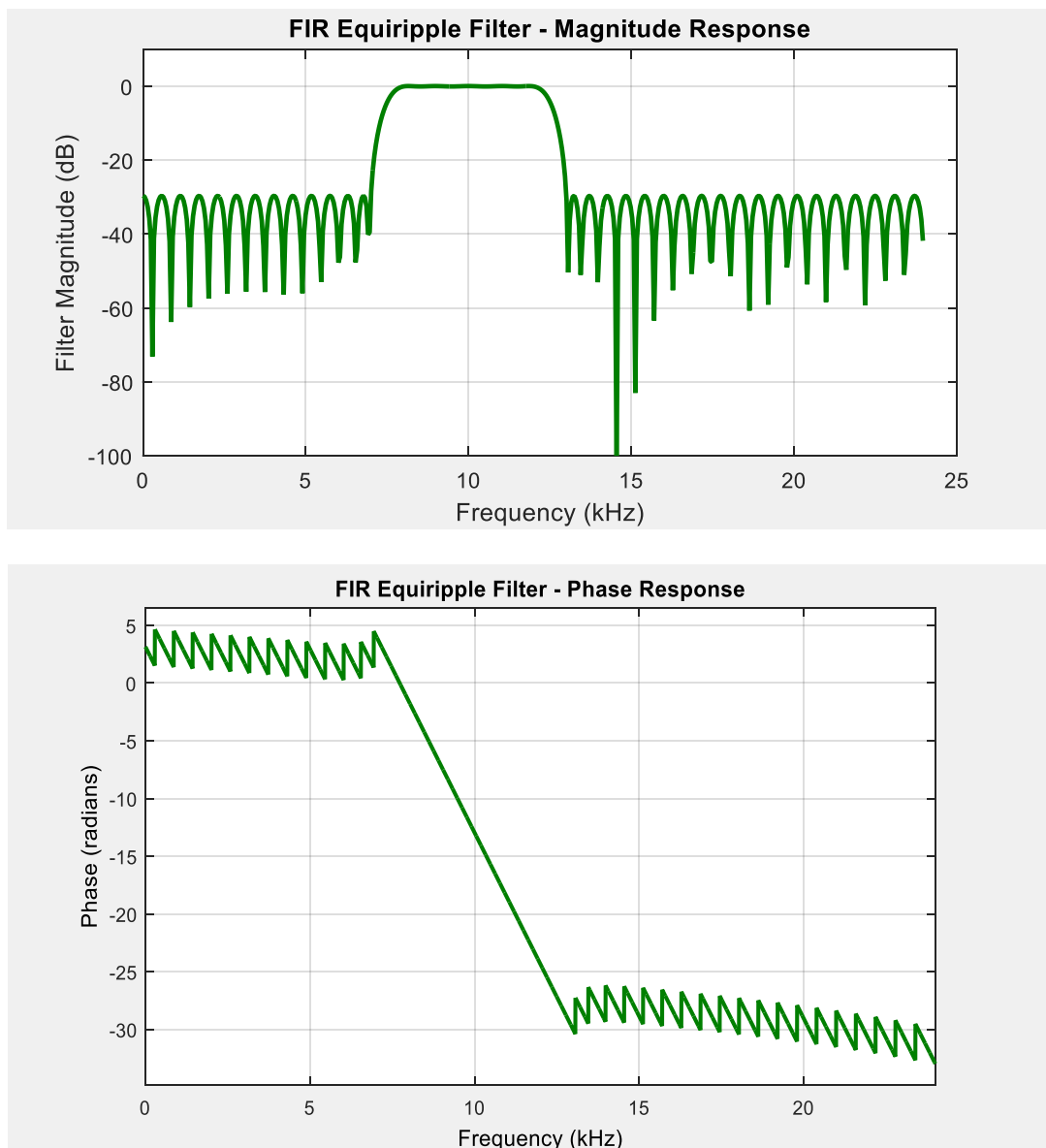


Figure 6.5: 87th Order FIR Equiripple BPF: Magnitude and Phase Response

Challenge Question 6.1

How many multiplications per sampling instant must be performed with the FIR and IIR filters designed in Example 6.3? How many multiplications per second (MPS) must be performed for each of these filters?

6.4 FILTER REALIZATIONS

Three different filter realizations were described in Chapter 5 for FIR filters: Direct, Direct-Transposed, and Direct-Symmetric. The Direct and Direct-Transposed require $N+1$ multiply and accumulate taps for an N^{th} order filter and differ only in how the filter is implemented on a DSP processor or an FPGA. The Direct-Symmetric realization cuts the number of multiplies in half by taking advantage of the symmetry of the filter coefficients. Scaling can be added to eliminate or minimize arithmetic overflow. There are several options for implementing IIR filters also.

Single Stage (Direct)

IIR filters can be implemented directly as shown in Figure 6.6 for a 4th order filter. A higher order filter would simply require additional feedforward and feedback loops with multipliers and adders. *Direct realizations of IIR filters are extremely sensitive to wordlength effects (coefficient quantization)!* Therefore, IIR filters above 3rd order are not usually implemented in direct form (one single section), but are split into 2nd order sections instead.

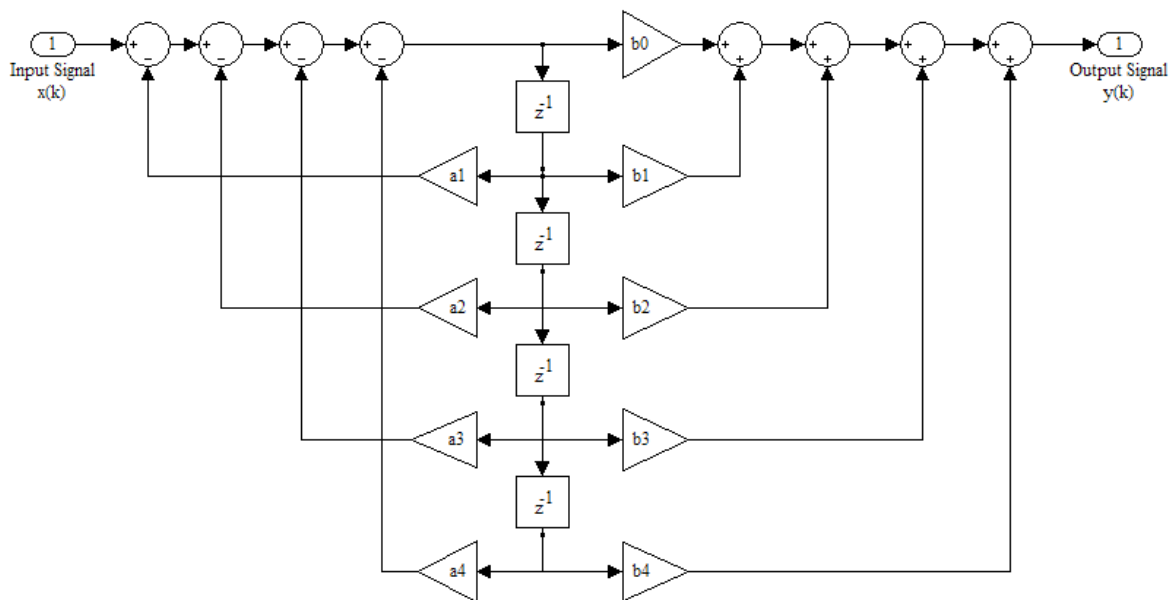


Figure 6.6: Direct Realization of IIR Filter

Cascade with 2nd Order Sections

A cascade realization breaks the IIR filter into the product of several 2nd order sections as follows:

$$H(z) = \frac{Num(z)}{Den(z)} = G \left(\frac{N_1(z)}{D_1(z)} \right) \left(\frac{N_2(z)}{D_2(z)} \right) \dots \left(\frac{N_M(z)}{D_M(z)} \right)$$

$$H(z) = G \left(\frac{1 + b_{11}z^{-1} + b_{21}z^{-2}}{1 + a_{11}z^{-1} + a_{21}z^{-2}} \right) \left(\frac{1 + b_{12}z^{-1} + b_{22}z^{-2}}{1 + a_{12}z^{-1} + a_{22}z^{-2}} \right) \dots \left(\frac{1 + b_{1M}z^{-1} + b_{2M}z^{-2}}{1 + a_{1M}z^{-1} + a_{2M}z^{-2}} \right) \quad (6.4)$$

A block diagram realization of the cascade structure is shown in Figure 6.7 for a 4th order filter which of course requires only two 2nd order sections. Additional 2nd order sections are easily added for higher order filters.

There is no unique way to break a transfer function into the product of 2nd order sections. Nth order IIR filters have N poles and N zeros. 2nd order sections are created by picking two of the poles and two of the zeros and pairing them together. There are $[(N/2)!]^2$ different ways to break an Nth order transfer function into 2nd order sections. Although mathematically equivalent, the various options are not equivalent in terms of filter performance. Some general guidelines for choosing the 2nd order sections to maximize filter output signal to noise ratio are:

1. Pair the poles and zeros that are closest to one another together. This will help minimize the amplitude gain of the filter at frequencies close to the section poles.
2. Put the most stable 2nd order section first and end with the least stable 2nd order section (i.e., the section with poles closest to the edge of the unit circle).

Fortunately, there are several software algorithms (including the Filter Design and Analysis Tool) that create an optimal arrangement of 2nd order sections based on maximizing the filter signal-to-noise ratio (SNR). Examples are included later in the chapter.

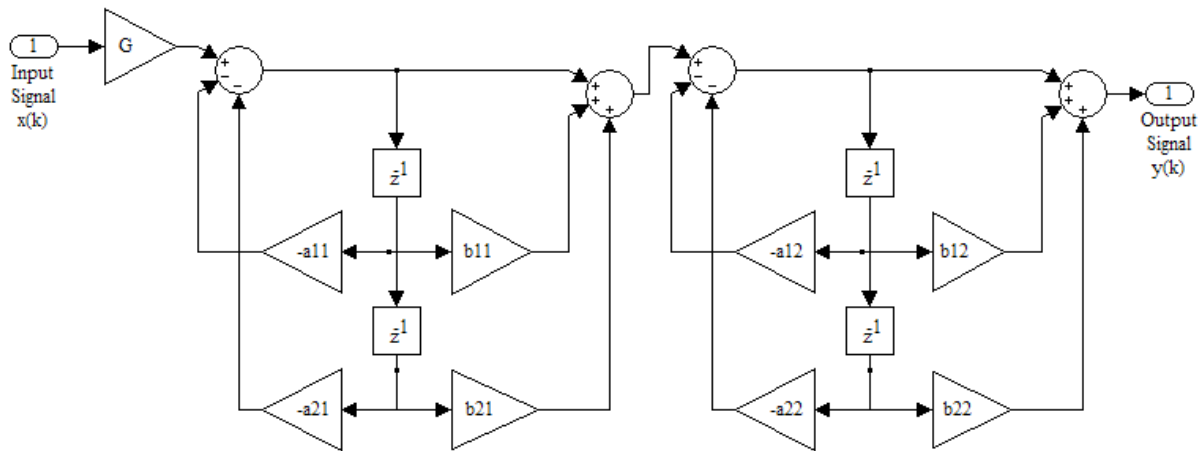


Figure 6.7: Cascade Realization of IIR Filter

Parallel with 2nd Order Sections

A parallel realization breaks the IIR filter into the sum of 2nd order sections using partial fraction expansion as shown in Equation 6.5. A block diagram realization of the parallel structure is shown in Figure 6.8 for a 4th order filter. Additional 2nd order sections are easily added for higher order filters.

$$H(z) = \frac{Num(z)}{Den(z)} = G + \left(\frac{N_1(z)}{D_1(z)} \right) + \left(\frac{N_2(z)}{D_2(z)} \right) + \dots + \left(\frac{N_M(z)}{D_M(z)} \right)$$

$$H(z) = G + \left(\frac{b_{01} + b_{11}z^{-1}}{1 + a_{11}z^{-1} + a_{21}z^{-2}} \right) + \left(\frac{b_{02} + b_{12}z^{-1}}{1 + a_{12}z^{-1} + a_{22}z^{-2}} \right) + \dots + \left(\frac{b_{0M} + b_{1M}z^{-1}}{1 + a_{1M}z^{-1} + a_{2M}z^{-2}} \right) \quad (6.5)$$

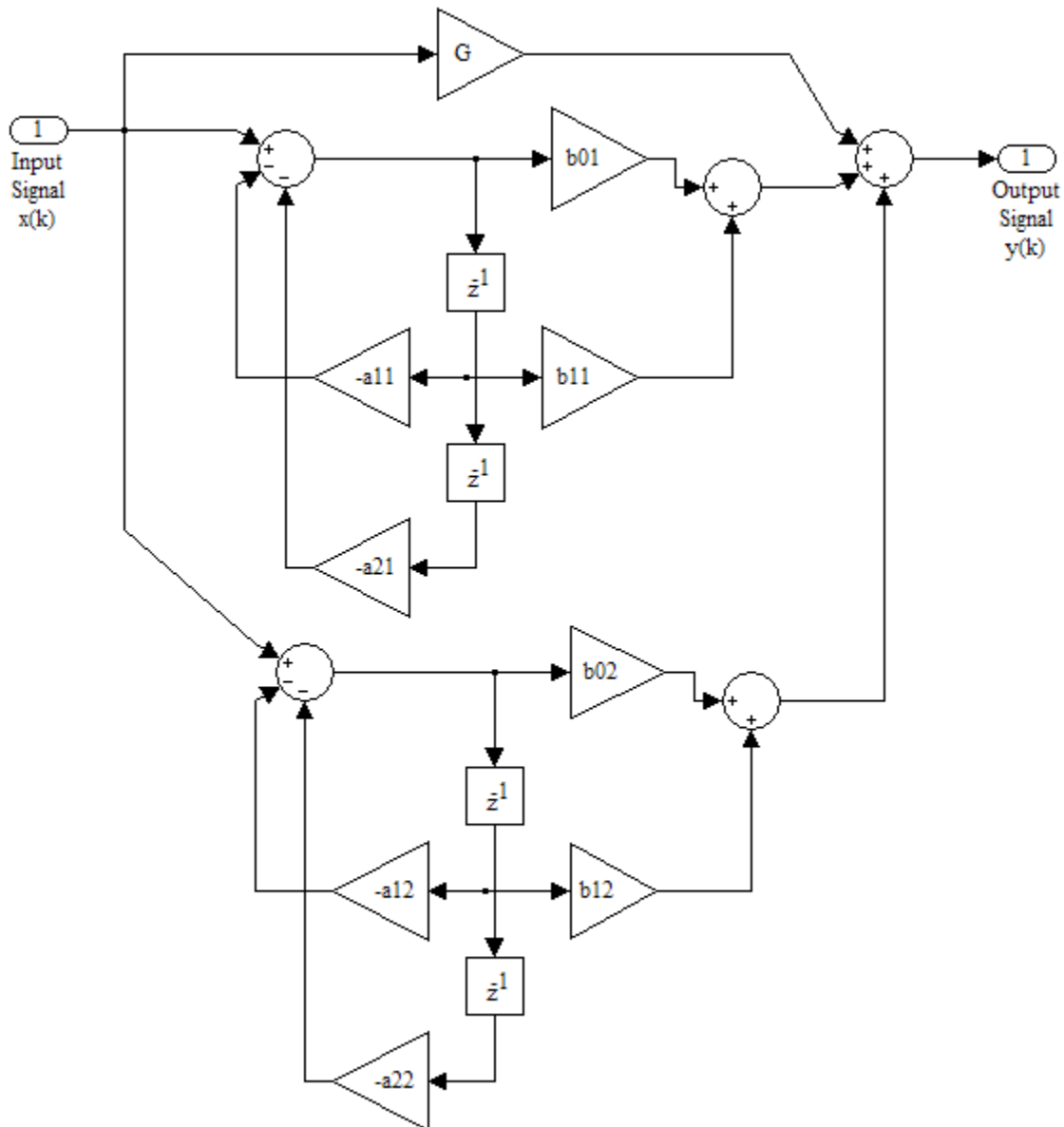


Figure 6.8: Parallel Realization of IIR Filter

How does the parallel realization compare to a cascade realization?

- In cascade structures, the poles are paired with the closest zeros to minimize the section gain at frequencies close to the pole locations.
- Ordering of the 2nd order sections is not important for a parallel realization because each section feeds directly to the output. In cascade structures, where each section feeds into the next section, it is important to put the least stable section last.
- The signal-to-noise ratio (SNR) of a parallel structure is comparable to the best (optimized) cascade structure.
- The zero locations in a parallel structure are very sensitive to coefficient quantization errors unless the wordlength is 12 or more bits.
- In a cascade structure, 25-50% of the filter coefficients are simple integers when the filter is designed using a bilinear transformation of an analog filter.
- The Filter Design and Analysis Tool will create a cascade structure for IIR filters but does not provide a parallel realization.

6.5 PRACTICAL CONSIDERATIONS FOR IIR FILTERS

Word-length Effects (Filter Coefficient Quantization Errors)

In order to implement a filter on a DSP processor, the filter coefficients must be stored in memory, either in floating-point or fixed-point format depending on the processor. In Chapter 4, it was shown through several examples that quantization error often occurs when representing numbers in a specified format. The filter coefficients derived using the FDAT (or some other design tool) will very likely be quantized when stored in the memory of a DSP processor. In Chapter 5, the effects of coefficient quantization errors on filter performance were illustrated through several examples for FIR filters. IIR filters are considerably more sensitive to wordlength effects than FIR filters due to the recursive nature (feedback) of IIR filters. *In digital filter design, it is absolutely necessary to check the effect of filter coefficient quantization on the performance of a filter!* The next example illustrates how quantization of the filter coefficients can impact filter performance.

Example 6.4: Coefficient Word-Length Effects on IIR Filter Performance

Consider the 10th order IIR elliptic band-pass filter designed in Example 6.3 to meet the specifications:

- i. Passband: 8 – 12 kHz
 - ii. Maximum Passband Ripple: 0.1 dB
 - iii. Stopband: ≤ 7 kHz and ≥ 13 kHz
 - iv. Minimum Stopband Attenuation: 30 dB
 - v. Sampling Frequency: 48 kHz
- (a) Using a direct realization, determine the minimum number of bits required for the filter coefficients in order to keep the filter stable.
 - (b) Plot the filter magnitude response when the filter coefficients are quantized to the number of bits determined in part (a).
 - (c) Determine the minimum number of bits required to meet the given filter specifications and plot the corresponding magnitude response.

Solution

(a) Enter the specifications for the elliptic band-pass filter into the FDAT, select minimum order, and then click Design Filter to create a 10th order elliptic band-pass filter. Since the FDAT defaults to a cascade structure, choose Edit → Convert to Single Section then File → Export to create a direct realization of the filter with the numerator (Num) and denominator (Den) exported to the MATLAB workspace. The filter coefficients are:

Num '

```
0.03203191821694
-0.05878107521081
0.10186869443828
-0.09766972701608
0.07231401919473
0.00000000000000
-0.07231401919473
0.09766972701608
-0.10186869443828
0.05878107521081
-0.03203191821694
```

Den '

```
1.00000000000000
-2.35045087132525
6.00883384914710
-8.37048075001733
11.66918777878540
-10.76965153277673
9.81514391724758
-5.90678320013480
3.56194382465704
-1.15506123192936
0.41170437266444
```

The filter poles are the roots of the denominator and the magnitudes of the poles are:

abs (roots (Den))

```
0.97875292352670
0.97875292352670
0.90061863412144
0.90061863412144
0.98170470724316
0.98170470724316
0.91188762957079
0.91188762957079
0.81312355369549
0.81312355369549
```

The filter is stable since all poles have magnitude strictly less than one; however, many of the filter poles are very close to the edge of the unit circle.

In order to determine the minimum number bits required to maintain filter stability, the denominator coefficients must be quantized to a specified number of bits and the magnitude of the roots of the quantized denominator must be checked. Since this will require some experimentation with the number of bits, it is worthwhile to write a script m-file for this part with the following code:

```
B=9;
Fs=48000;
Denq=round(Den*2^(B-1))/(2^(B-1));
abs(roots(Denq))
Numq=round(Num*2^(B-1))/(2^(B-1));
[Hq,fq]=freqz(Numq,Denq,500,Fs);
[H,f]=freqz(Num,Den,500,Fs);
plot(f/1000,20*log10(abs(H)),fq/1000,20*log10(abs(Hq)));
legend('Unquantized Response','Quantized Response');
xlabel('Frequency (kHz)'); ylabel('Filter Magnitude (dB)');
title(['IIR Filter Magnitude Response (' num2str(B) '
bits)']); grid
```

This m-file allows the user to specify a number of bits (B), quantizes the denominator to B bits, displays the magnitude of the poles of the quantized denominator, quantizes the numerator of the filter to B bits, and then plots the frequency response of the quantized filter and the original (unquantized) filter. Changing the number of bits, B, in Line 1 and then clicking on Run will re-run the code for any specified number of bits.

The magnitudes of the poles for B = 9 bits are:

```
0.99244484746493
0.99244484746493
0.98946521292787
0.98946521292787
0.88479603523624
0.88479603523624
0.90496389777822
0.90496389777822
0.81450421020215
0.81450421020215
```

Since all poles are within the unit circle, the IIR filter is stable when quantized to 9 bits. The magnitudes of the poles for B = 8 bits are:

```
1.01909858840302
1.01909858840302
```

```

1.03587538594865
1.03587538594865
0.89398395056256
0.89398395056256
0.85986520978545
0.85986520978545
0.79295608556608
0.79295608556608

```

The IIR filter is unstable when quantized to 8 bits. So, the minimum number of bits for stability is 9 bits.

(b) The minimum number of bits for stability was determined to be 9 bits. The m-file created for part (a) also includes the code to plot the frequency response (magnitude) of the quantized filter. Setting $B = 9$ and running the m-file creates the frequency response plot shown in Figure 6.9. Notice that even though the IIR filter is stable when the coefficients are quantized to 9 bits, the performance of the quantized filter does not meet the specifications and is therefore unacceptable!

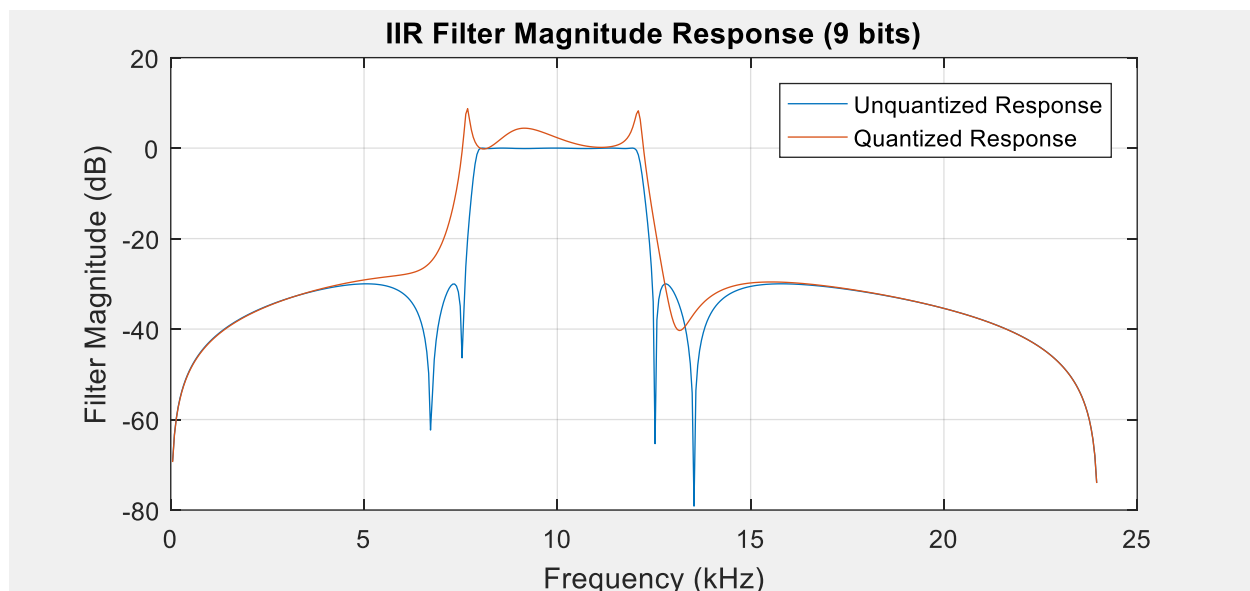


Figure 6.9: Frequency Response Quantized to 9 bits (Direct Realization)

(c) The number of bits required to meet the specifications can be determined by experimenting with the number of bits, B , in the m-file from part (a) till the performance of the quantized filter matches that of the un-quantized filter. After some experimentation, 16 bits for the filter coefficients results in a filter which meets the given specification as shown in Figure 6.10.

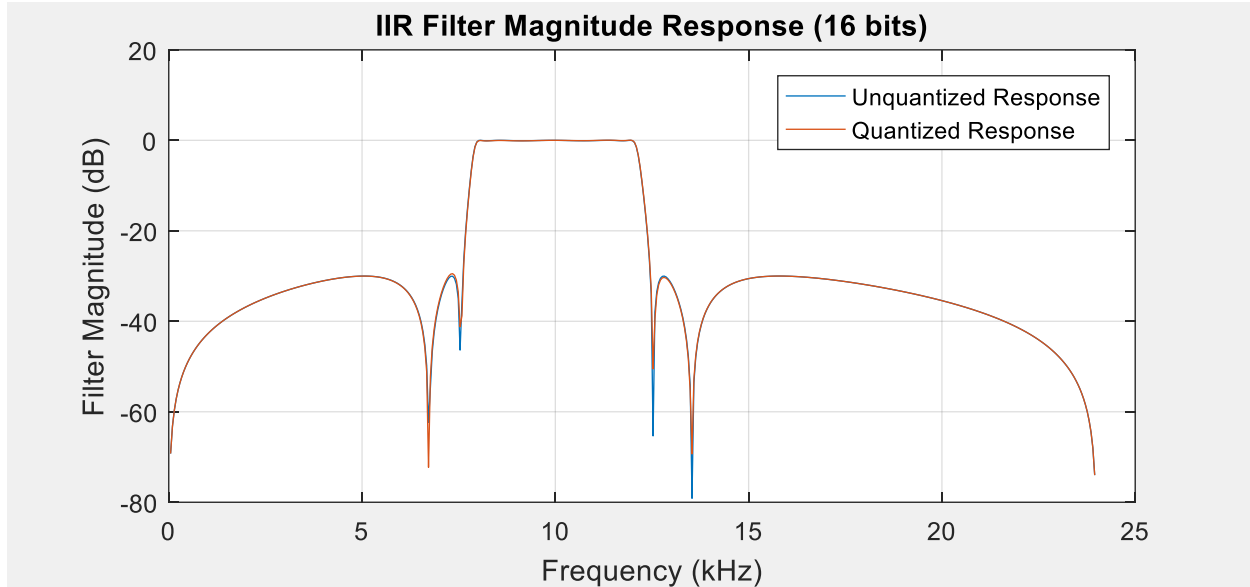


Figure 6.10: Frequency Response Quantized to 16 bits (Direct Realization)

Challenge Question 6.2

In example 6.4, the poles of the IIR filter shift when the coefficients are quantized to B bits and it is even possible to make the filter unstable. Will an FIR filter exhibit the same instability problem if the number of bits for the filter coefficients is continually reduced? Explain.

In the previous section on filter realizations, it was mentioned that breaking IIR filters into 2nd order sections (either cascade or parallel) makes the filter less sensitive to word-length effects. The FDAT has the capability to create a cascade realization for IIR filters, in fact, it defaults to this structure. When File → Export is selected, the FDAT exports an SOS matrix and a G vector to the MATLAB workspace. The G (Gain) vector is the gain for each section. The SOS (Second Order Sections) matrix contains the numerator and denominator coefficients of each 2nd order section and is structured as follows:

$$SOS = \begin{bmatrix} Num1 & Den1 \\ Num2 & Den2 \\ Num3 & Den3 \\ Num4 & Den4 \end{bmatrix} = \begin{bmatrix} 1 & b_{11} & b_{21} & 1 & a_{11} & a_{21} \\ 1 & b_{12} & b_{22} & 1 & a_{12} & a_{22} \\ 1 & b_{13} & b_{23} & 1 & a_{13} & a_{23} \\ 1 & b_{14} & b_{24} & 1 & a_{14} & a_{24} \end{bmatrix}$$

Row 1 of the SOS matrix contains the numerator and denominator coefficients for the first 2nd order section. The next 2nd order section coefficients are located in Row 2 and the pattern continues. This particular SOS matrix has only 4 rows and therefore describes an 8th order filter (4 sections). Higher order filters would simply have additional rows in the SOS matrix. In MATLAB, specific sections can be referenced as follows:

```
Den3 = SOS(3, 4:6); % Row 3, Columns 4-6 of the SOS Matrix
```

```
Num2 = SOS(2, 1:3); % Row 2, Columns 1-3 of the SOS Matrix
```

The next example illustrates the advantage of implementing IIR filters as 2nd order sections rather than using a direct (single stage) realization.

Example 6.5: Using Cascade Realization to Reduce Effects of Coefficient Word-Length

Repeat Example 6.4 using a cascade realization for the IIR elliptic band-pass filter and compare the results to those using a direct realization for the filter.

Solution

Enter the specifications for the elliptic band-pass filter into the FDAT, select minimum order, and then click Design Filter to create the 10th order elliptic band-pass filter. Convert the structure to Direct Form II Second Order Sections. The structure for the filter is now five 2nd order cascaded sections. Select File → Export to export the SOS matrix and G vector to the MATLAB workspace. The resulting SOS matrix and G vector can be viewed by typing SOS and G at the MATLAB prompt:

SOS =

1.0000	-0.0000	-1.0000	1.0000	-0.4451	0.6612
1.0000	0.3987	1.0000	1.0000	-0.0653	0.8111
1.0000	-1.2720	1.0000	1.0000	-0.8654	0.8315
1.0000	0.1385	1.0000	1.0000	0.0283	0.9580
1.0000	-1.1004	1.0000	1.0000	-1.0030	0.9637

G =

0.0320
1.0000
1.0000
1.0000
1.0000
1.0000

To determine the minimum number of bits for stability, the denominator coefficients of each 2nd order section must be quantized and the poles must be checked to ensure the magnitude of each pole is less than one. This will require some experimentation on the number of bits so it is worthwhile to write an m-file for this example. Figure 6.11 shows the code for the script file.

This script first quantizes the SOS matrix and the G vector to B bits as specified by the user. The magnitudes of the poles of each quantized section are displayed in a matrix called mag_poles. In order to plot the frequency response, a single numerator and denominator are created by multiplying the numerators and denominators of each section together. For comparison purposes, this is done for both the quantized and un-quantized filter. The frequency response (magnitude) is then calculated and plotted.

```

B = 4;
Fs=48000;
% Quantize the SOS Matrix and the Gain, G to B bits
SOS_q=round(SOS*2^(B-1))/(2^(B-1));
G_q=round(G*2^(B-1))/(2^(B-1));
% Check stability of the Quantized Denominator of each section
rows=size(SOS,1);
for m=1:rows,
    mag_poles(m,1:2)=abs(roots(SOS_q(m,4:6)));
end
% Stability Check:
mag_poles
% Check filter performance for quantized filter coefficients
% Create a single numerator and denominator by convolving the
% numerators and denominators of each section together.
Num=prod(G); Den=1;
Numq=prod(G_q); Denq=1;
for m=1:rows,
    Num=conv(Num,SOS(m,1:3));
    Den=conv(Den,SOS(m,4:6));
    Numq=conv(Numq,SOS_q(m,1:3));
    Denq=conv(Denq,SOS_q(m,4:6));
end
[H f] = freqz(Num,Den,500,Fs);
[Hq fq]= freqz(Numq,Denq,500,Fs);
plot(f/1000, 20*log10(abs(H)),fq/1000, 20*log10(abs(Hq)));
legend('Unquantized Response','Quantized Response');
xlabel('Frequency (kHz)'); ylabel('Filter Magnitude (dB)');
title(['IIR Filter Magnitude Response (' num2str(B) ' bits)']);
grid

```

Figure 6.11: m-file (IIR_Quant.m) for Example 6.5

When the code in Figure 6.11 is executed for $B = 4$ bits, the magnitude of the sections poles are:

```

mag_poles =
    0.79056941504209    0.79056941504209
    0.86602540378444    0.86602540378444
    0.93541434669349    0.93541434669349
    1.00000000000000    1.00000000000000
    1.00000000000000    1.00000000000000

```

The last two sections of the filter are unstable since the poles have a magnitude of one. The IIR filter is therefore unstable when the coefficients are quantized to 4 bits.

Changing B to 5 bits and re-running the code yields section poles with the following magnitudes:

```
mag_poles =
    0.82915619758885  0.82915619758885
    0.90138781886600  0.90138781886600
    0.90138781886600  0.90138781886600
    0.96824583655185  0.96824583655185
    0.96824583655185  0.96824583655185
```

Since all of the poles have magnitude less than one, the IIR filter is stable when the coefficients are quantized to 5 bits. For a direct realization (Example 6.4), 9 bits were required to ensure stability of the IIR filter. Notice also that the most stable section is first (row 1) and the least stable section is last (row 5).

Setting B = 5 and running the code in Figure 6.11 creates the frequency response plot shown in Figure 6.12. Notice that even though the IIR filter is stable when the coefficients are quantized to 5 bits, the performance of the quantized filter does not meet the specifications and is therefore unacceptable.

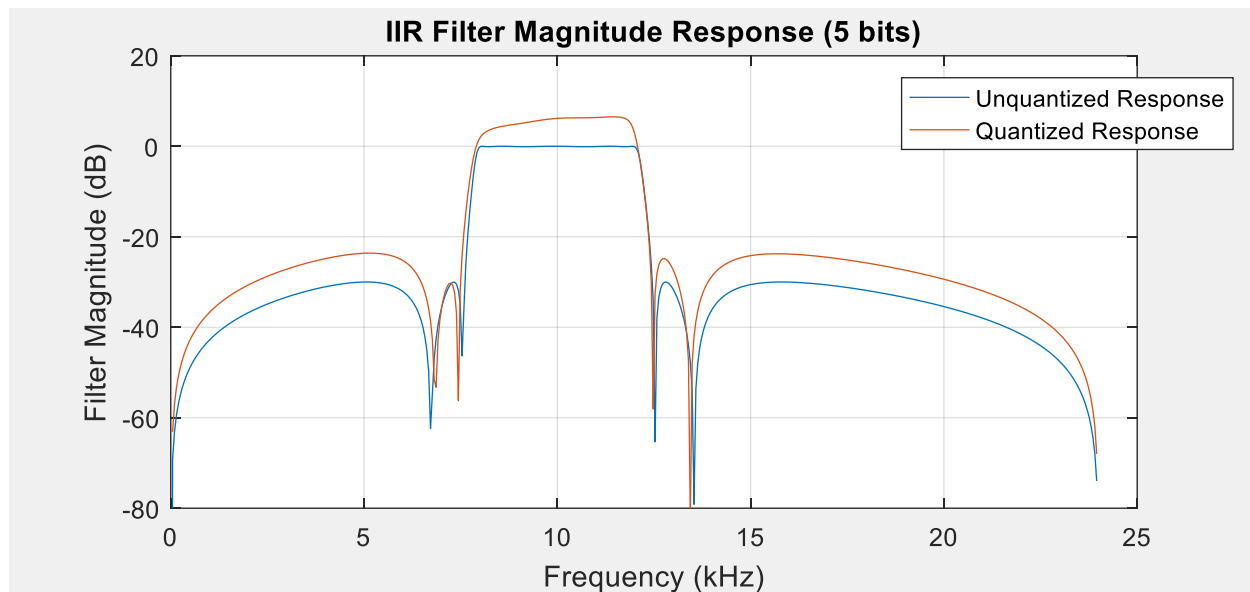


Figure 6.12: Frequency Response Quantized IIR Filters (Cascade Realization)

The number of bits required to meet the specifications can be determined by experimenting with the number of bits, B, and re-running the code until the performance of the quantized filter matches that of the un-quantized filter. After some experimentation, 9 bits for the filter coefficients results in a filter which meets the given specification as shown in Figure 6.13. The direct realization required 16 bits to meet the specifications. Clearly, a cascade realization is significantly less sensitive to word-length effects than a direct realization!

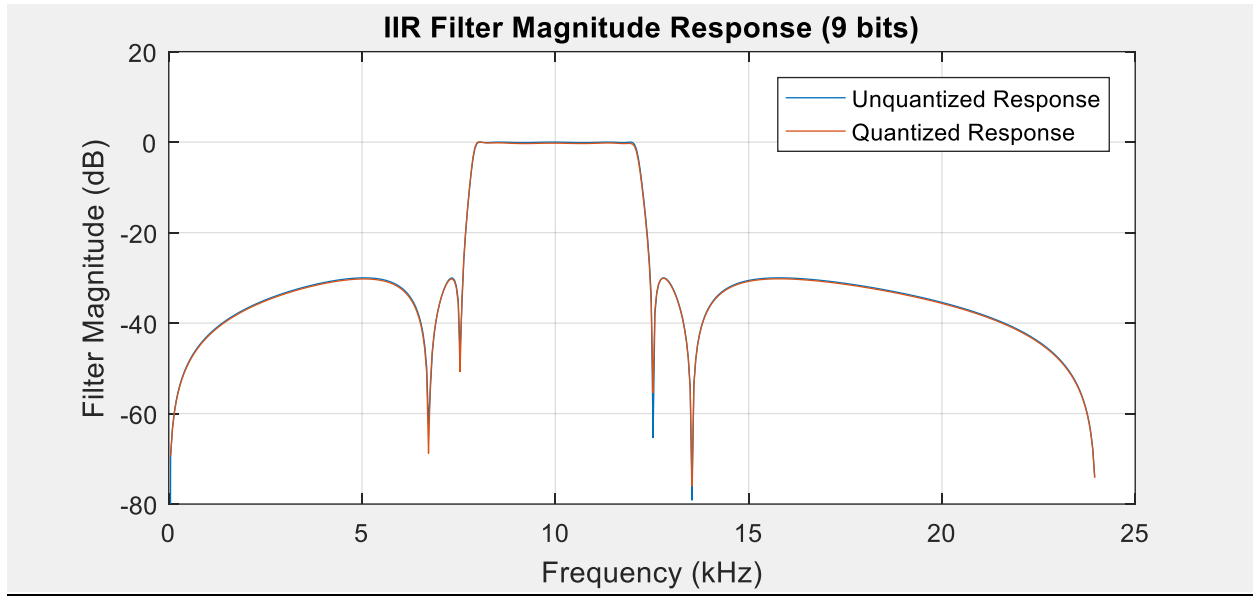


Figure 6.13: Frequency Response Quantized IIR Filters (Cascade Realization)

Roundoff Errors in Arithmetic Operations

The output signal of an IIR filter is computed by:

$$y(k) = \sum_{i=1}^N -a_i y(k-i) + \sum_{i=0}^N b_i x(k-i)$$

$$y(k) = -a_1 y(k-1) - a_2 y(k-2) - \dots - a_N y(k-N) + b_0 x(k) + b_1 x(k-1) \dots + b_N x(k-n) \quad (6.6)$$

Suppose the filter coefficients are 16 bits and the processor utilizes a 16-bit ADC. Each multiplication will be 32 bits. Eventually, this product must be truncated or rounded to 16-bits. Rounding the result will result in a reduction of the Signal-Noise Ratio (SNR). To minimize the effect of rounding, it is much better to store each product in a double precision register and round after the final summation instead of rounding to 16 bits after each individual multiplication.

Arithmetic Overflow in Fixed Point Processors

As discussed in Chapter 4, fixed point digital signal processors typically use a Q15 format in which all numbers are scaled to be between -1 and $+1$. Any multiplication operations on numbers in Q15 format result in a number that is also between -1 and $+1$ (no overflow). However, addition operations on numbers in Q15 format can result in overflow. Arithmetic overflow will cause either a sign change in the result (undesirable) or clipping (saturating at the maximum or minimum value). Arithmetic overflow can be a messier problem for IIR filters because the feedback of IIR filters can cause an overflow limit cycle which can be difficult to

stop. Arithmetic overflow can be eliminated (or significantly reduced) by scaling the filter coefficients and the input data.

In Figure 6.14, scaling for one 2nd order section of an IIR filter is illustrated. The following observations on the effectiveness of scaling can be made:

- Dividing the input signal by the scale factor, s , reduces the input in order to either eliminate or minimize overflow through the feedback terms.
- Multiplying the numerator coefficients by the scale factor, s , ensures that the overall filter gain is not changed.
- Scaling will reduce the signal to noise ratio (SNR).

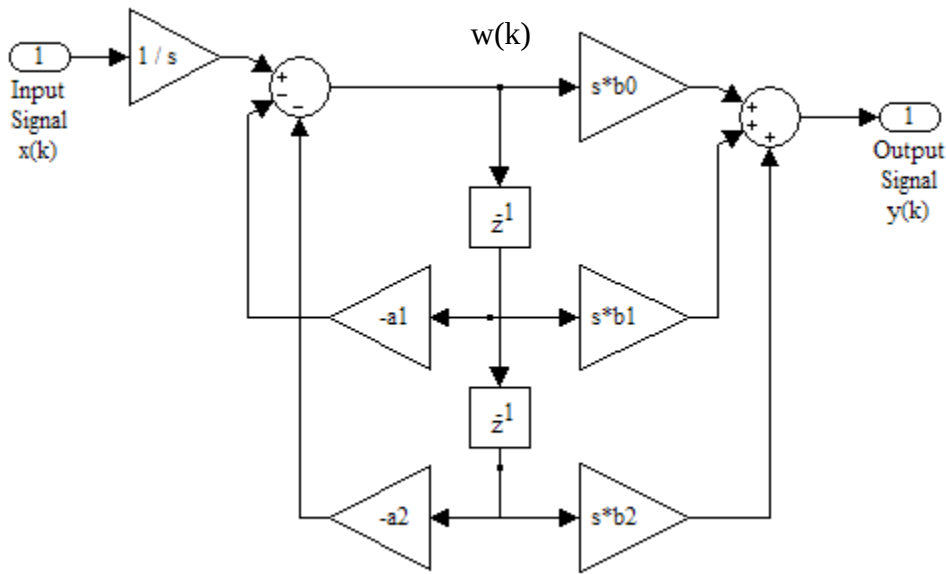


Figure 6.14: Scaling a 2nd Order Section of an IIR Filter

How is the scale factor chosen? In Chapter 5, only $\mathcal{L}_1$ scaling was explored for FIR filters. There are actually three common scaling schemes used for FIR and IIR filters: $\mathcal{L}_1$, $\mathcal{L}_2$, and $\mathcal{L}_\infty$ scaling. Referring to Figure 6.14, the transfer function from the input $x(k)$ to the intermediate node signal $w(k)$ is:

$$H(z) = \frac{W(z)}{X(z)} = \frac{1}{1 + a_1 z^{-1} + a_2 z^{-2}} \quad (6.7)$$

Letting $h(k)$ denote the impulse response (that is, $h(k) = Z^{-1}[H(z)]$), the three different scale factors are determined as follows:

$$\begin{aligned}
\mathcal{L}_1 \text{ Scaling:} \quad s &= \sum_{k=0}^{\infty} |h(k)| = \|H(z)\|_1 \\
\mathcal{L}_2 \text{ Scaling:} \quad s &= \sqrt{\sum_{k=0}^{\infty} h^2(k)} = \|H(z)\|_2 \\
\mathcal{L}_{\infty} \text{ Scaling:} \quad s &= \max |H(e^{j\omega})| = \|H(z)\|_{\infty}
\end{aligned} \tag{6.8}$$

How do these scale factors compare in terms of filter performance?

- $\mathcal{L}_1$ scaling ensures that there will never be arithmetic overflow because this scale factor sets the overall filter gain from $x(k)$ to $w(k)$ equal to one. $\mathcal{L}_1$ scaling results in the largest scale factor of the three schemes and therefore has the worst SNR performance. $\mathcal{L}_1$ scaling is generally more drastic than what is required to keep arithmetic overflow in check.
- $\mathcal{L}_2$ scaling places an energy constraint on the input signal and the transfer function. Arithmetic overflow is reduced but will occur sometimes. $\mathcal{L}_2$ scaling results in the smallest scale factor of the three schemes and offers better SNR performance than $\mathcal{L}_1$ scaling.
- $\mathcal{L}_{\infty}$ scaling ensures that arithmetic overflow will not occur when the input signal is a sine wave. The scale factor using $\mathcal{L}_{\infty}$ scaling lies between the $\mathcal{L}_2$ scale factor and $\mathcal{L}_1$ scale factor and is typically the best choice, offering a compromise between the other two schemes. Arithmetic overflow is not completely eliminated but is significantly reduced and this scaling scheme results in better SNR performance than $\mathcal{L}_1$ scaling.
- The FDAT tool has the capability of creating filter realizations with either $\mathcal{L}_{\infty}$ scaling or $\mathcal{L}_2$ scaling. It does not offer an option for $\mathcal{L}_1$ scaling. Choose Edit→Re-order and Scale Second Order Sections to create a scaled, cascade filter realization.

Figure 6.15 shows how scaling is implemented in a cascade realization for an IIR filter. The scale factor for the i^{th} section, s_i , is determined by using the $\mathcal{L}_1$, $\mathcal{L}_2$, or $\mathcal{L}_{\infty}$ norm as follows:

$$s_i = \|H_i(z)\|_p \quad p = 1, 2, \text{ or } \infty$$

$$H_i(z) = \frac{\prod_{m=1}^{i-1} T_m(z)}{1 + a_{1i}z^{-1} + a_{12}z^{-2}} \quad T_m(z) = \frac{b_{0m} + b_{1m}z^{-1} + b_{2m}z^{-2}}{1 + a_{1m}z^{-1} + a_{2m}z^{-2}}$$

(6.9)

In equation 6.9, $H_i(z)$ is the transfer function from the input, $x(k)$, to the intermediate node signal $w_i(k)$ for section i . So, the scaling factor for a particular section depends not only on the section of interest but also on all prior sections which certainly complicates matters.

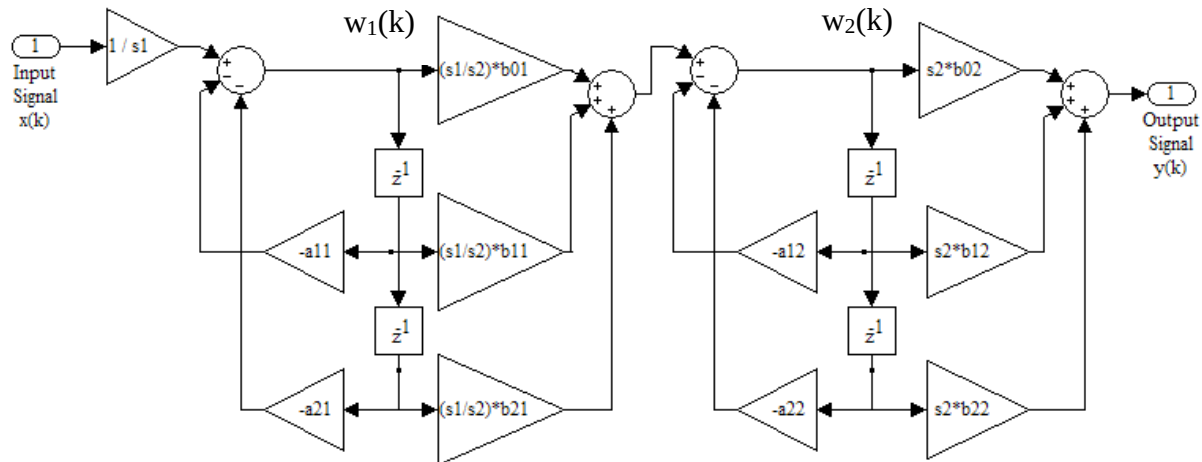


Figure 6.15: Scaling in a Cascade Structure

In Figure 6.16, scaling is added to a parallel realization for an IIR filter. Here, the scale factor is determined independently for each 2nd order section using an $\mathcal{L}_1$, $\mathcal{L}_2$, or $\mathcal{L}_\infty$ norm. Unlike the cascade structure, it is not necessary to take any of the other sections into account because each section feeds directly to the output signal from the input signal. Therefore, the scaling factors are much simpler to calculate for a parallel realization.

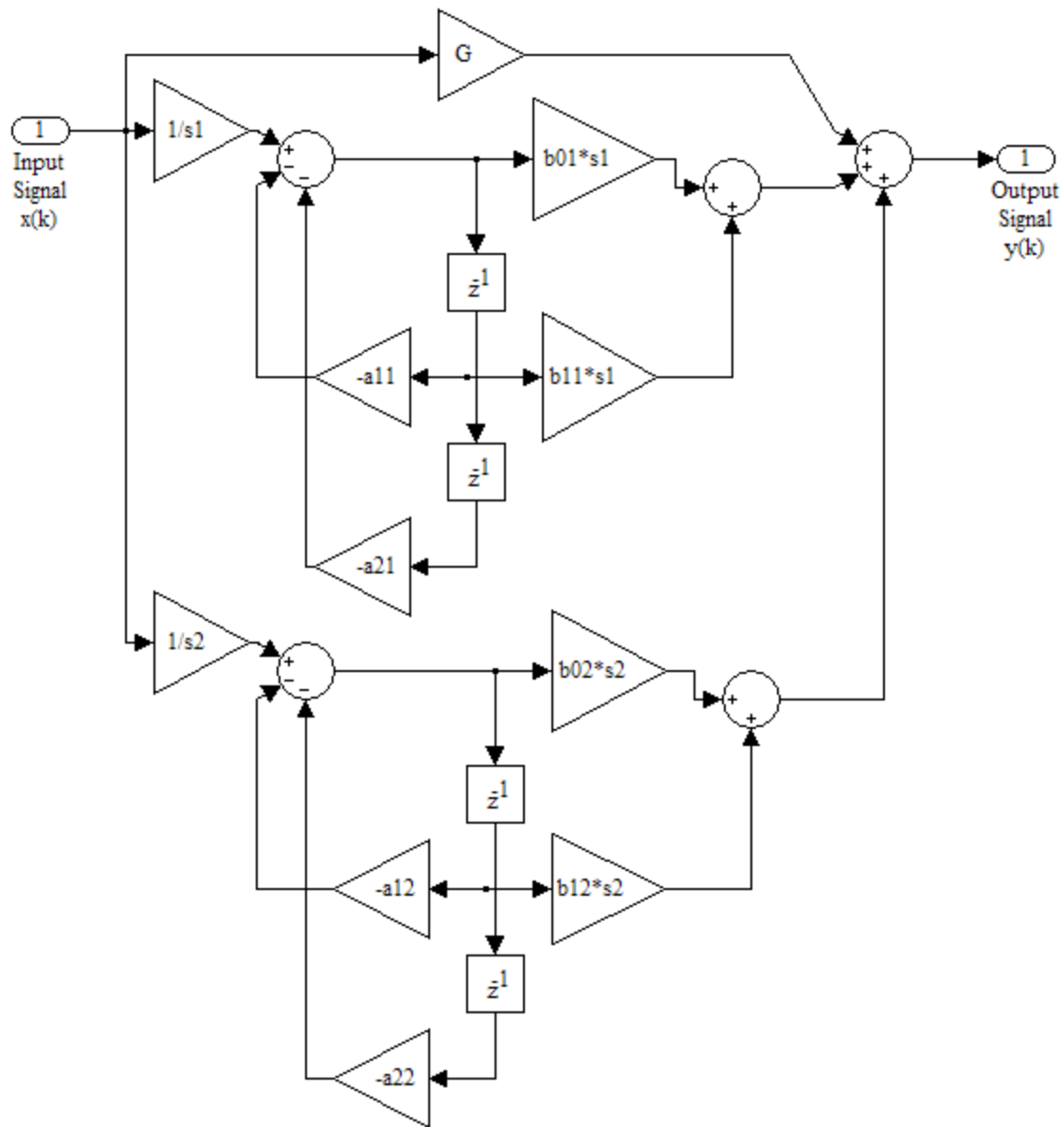


Figure 6.16: Scaling in a Parallel Structure

6.6 EXAMPLES OF IIR FILTERS

Fading Echo (Feedback Comb Filter)

An echo filter feeds back an attenuated output signal to produce a fading echo effect as follows:

$$y(k) = x(k - N) + \alpha_N y(k - N) \quad 0 < \alpha_N < 1 \quad (6.10)$$

The difference equation models a sound wave bouncing off a wall or surface and returning to the ear. The constant α is the round-trip attenuation factor and $N \cdot T_s$ is the round-trip time delay. The transfer function of the echo filter is

$$H(z) = \frac{1}{1 - \alpha_N z^{-N}} = \frac{z^N}{z^N - \alpha_N} \quad (6.11)$$

Challenge Question 6.3

What is the impulse response of the echo filter?

The frequency response of the echo filter is shown in Figure 6.17 for several values of gain, α . It is interesting to note that the frequency response of this echo filter is the inverse of the feedforward comb filter discussed in Chapter 5. The feedforward comb filter rejects a specific frequency and its harmonics; whereas, the feedback comb filter amplifies a specific frequency and its harmonics.

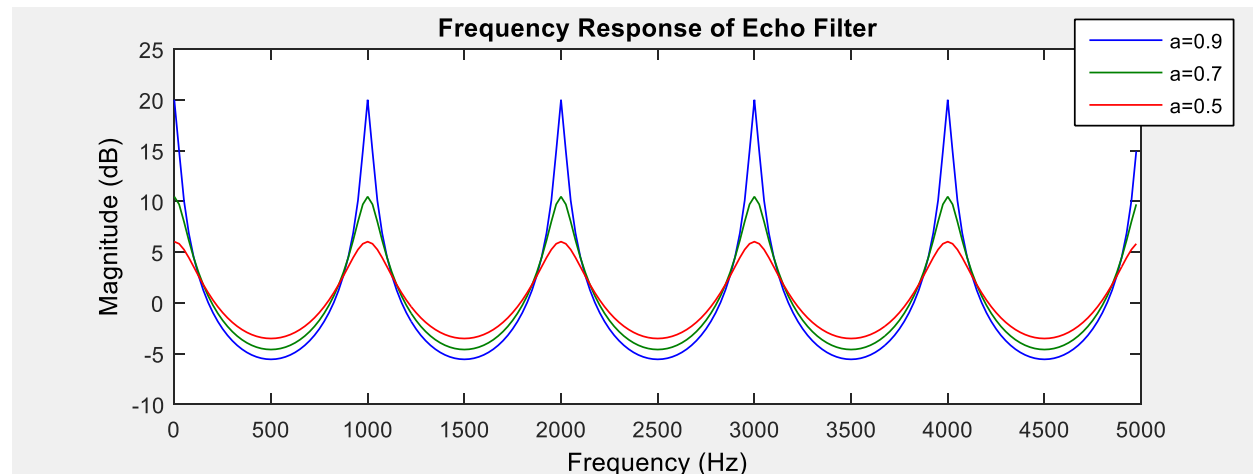


Figure 6.17: Frequency Response of Echo Filter

Challenge Question 6.4

What will happen if the feedback factor, α_N , is chosen to be greater than one for the echo filter? Why?

All-Pass Filter

The block diagram for an all-pass IIR filter is shown in Figure 6.18. All-pass filters are useful for creating digital audio effects such as reverb and phasing. All-pass filters can also be used to re-shape the phase of an IIR filter to reduce phase distortion.

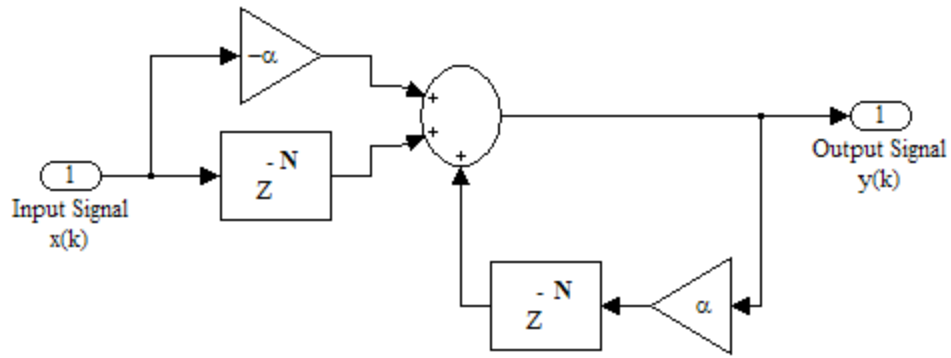


Figure 6.18: All-Pass IIR Filter

The difference equation for the filter is

$$y(k) = -\alpha x(k) + x(k - N) + \alpha y(k - N) \quad \text{where } 0 < \alpha < 1 \quad (6.12)$$

The impulse response of the all-pass filter is sketched in Figure 6.19.

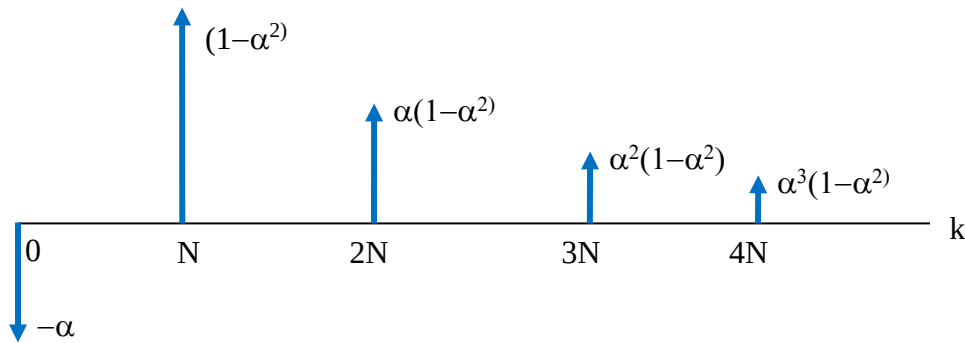


Figure 6.19: Impulse Response of All-Pass Filter

The transfer function of this filter is

$$H(z) = \frac{-\alpha + z^{-N}}{1 - \alpha z^{-N}} = \frac{-\alpha z^N + 1}{z^N - \alpha} \quad (6.13)$$

The frequency response of a 5th order all-pass filter with $\alpha = 0.3$ can be computed then plotted in MATLAB using the following commands:

```
alpha = 0.3;
N = 5;
```



```

num = zeros(1,N+1); num(1) = -alpha; num(N+1) = 1;
den = zeros(1,N+1); den(1) = 1; den(N+1) = -alpha;
freqz(num,den,200,1000);
% 3rd argument in freqz is number of data points to
generate and the 4th argument in freqz is the sampling
frequency

```

The resulting frequency response is plotted in Figure 6.20. Not surprisingly, the gain of the filter is 0 dB or 1.0 over the entire frequency range for the filter. The phase response is non-ideal and introduces some distortion.

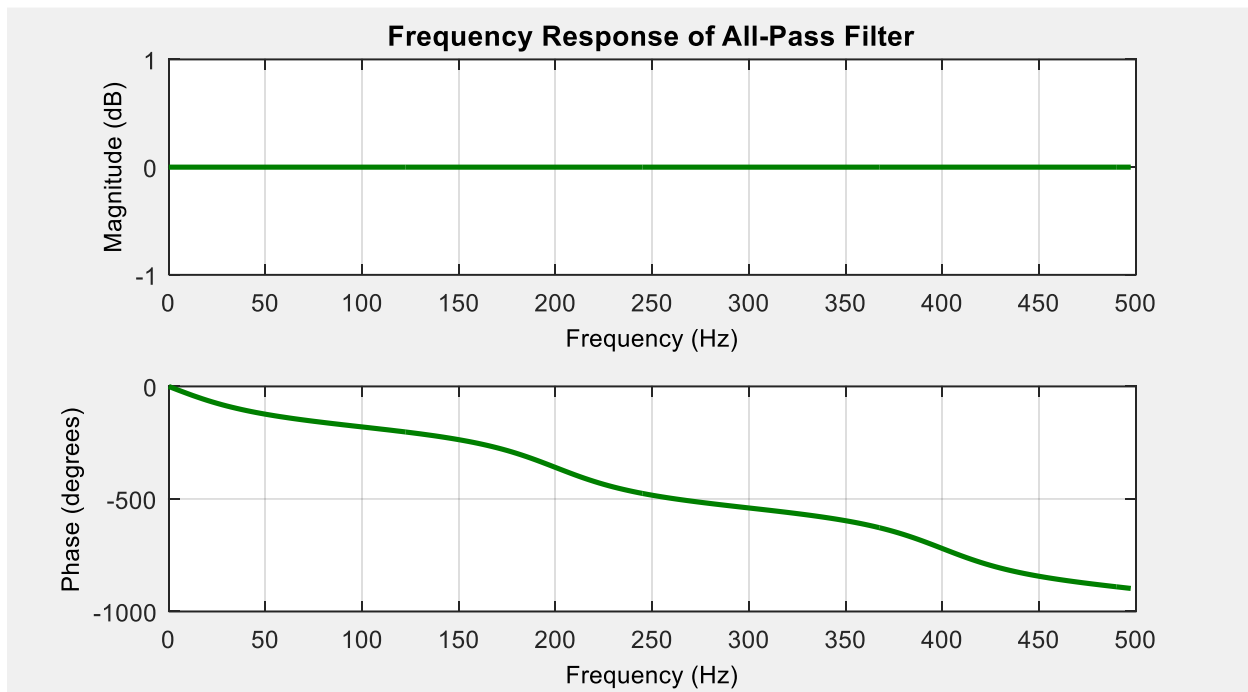


Figure 6.20: Frequency Response of All-Pass Filter

6.7 APPLICATION: AUDIO EFFECTS USING IIR FILTERS

Phasing

A phase shifter or phaser notches one or more frequencies in the original audio signal by filtering the audio signal through a chain (cascade) of first order all-pass filters, then mixing the filtered signal with the original audio signal as shown in Figure 6.21.

So where do the notches occur in the phaser? A notch will occur at all the frequencies at which the all-pass filter chain produces a phase shift of 180° or an odd multiple of 180° . The 180° phase shift inverts the signal frequency which is then re-combined with the original audio signal through the mixer resulting in complete cancellation at the frequency if the depth (gain) is chosen to be one, or partial cancellation for smaller gains. The all-pass filter chain can be

designed to control which frequencies are notched, the width of the notch, and how the notches vary or sweep over time.

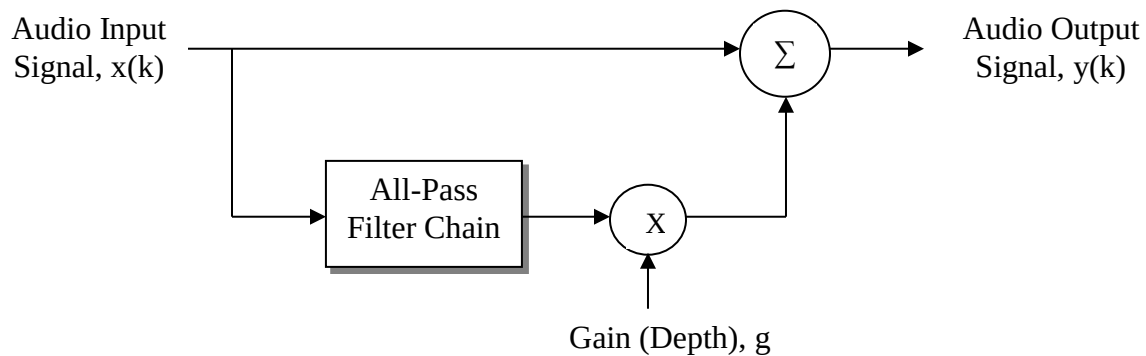


Figure 6.21: Phasing Effect

A 1st order all-pass filter is shown in Figure 6.22. Figure 6.23 shows the frequency response of a chain of four 1st order all-pass filters, and the frequency response of the resulting 4-chain phaser with the depth set to one. In Figure 6.23(a), the gains of all four all-pass filters are set to 0.2; while in Figure 6.23(b), the gains of all four filters are set to 0.8.

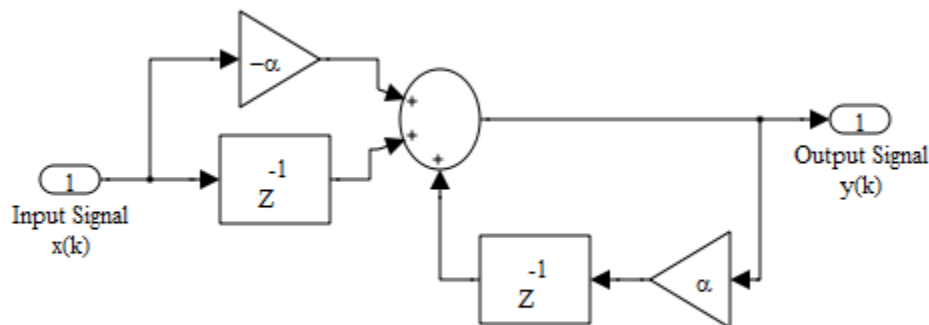


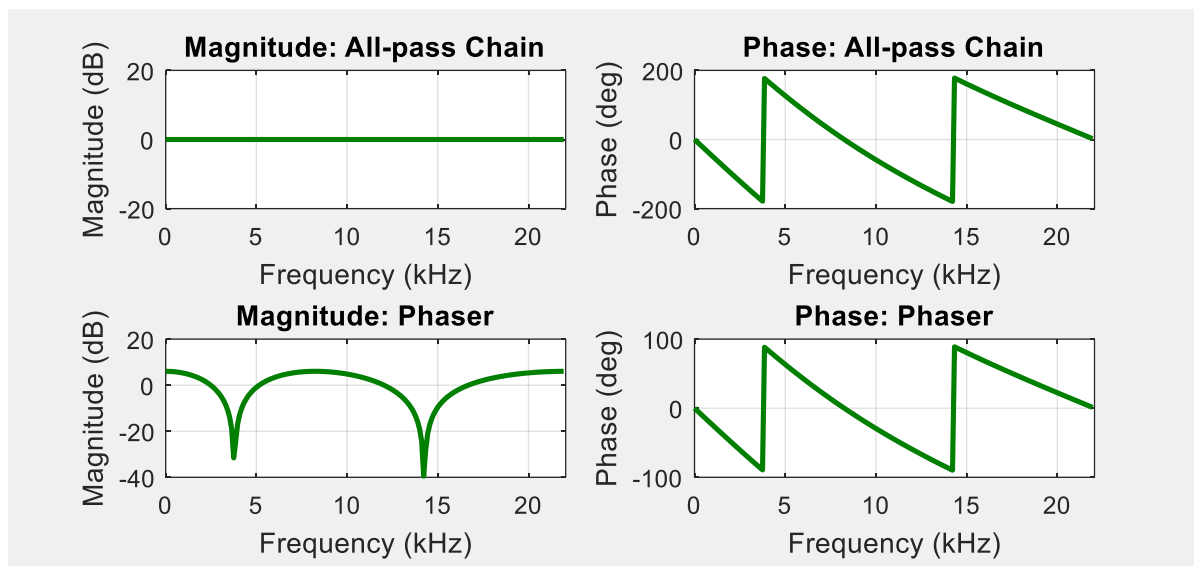
Figure 6.22: First-Order All-Pass Filter

As indicated in Figure 6.23, a chain of four 1st order all-pass filters will have a phase of 180° at two frequencies, and will cause inversion of the audio signal at these two frequencies. The resulting notch in the output audio signal at these two frequencies is evident in the magnitude plot for the phaser in Figure 6.23. The phaser will boost signal level at frequencies that are not in the vicinity of the notch frequencies. Smaller values for alpha spread and increase the notch frequencies, while larger values of alpha compress and reduce the notch frequencies.

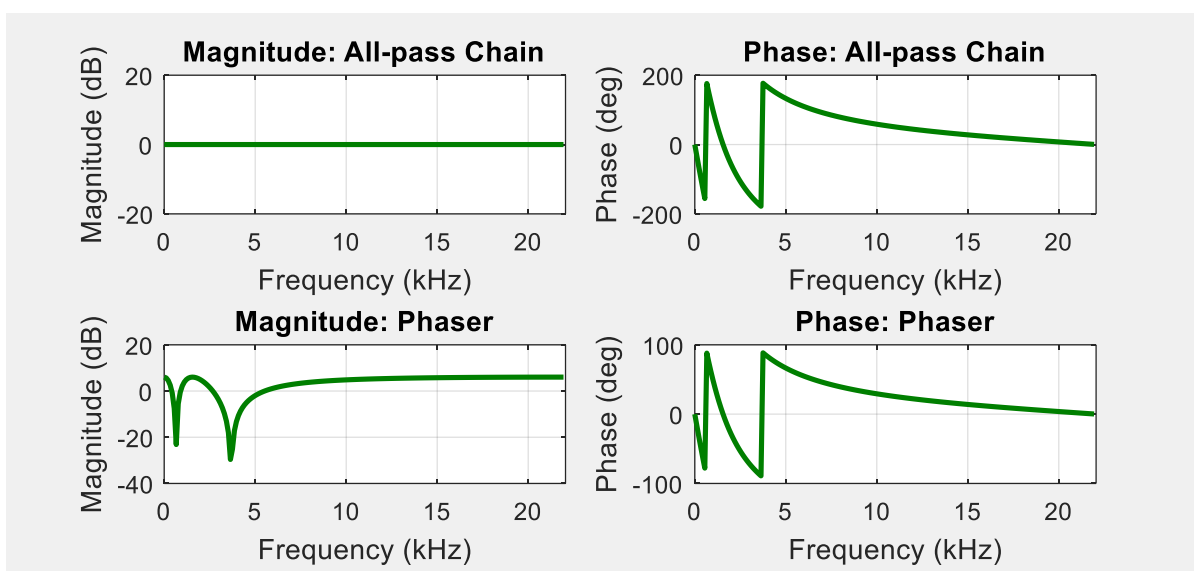
Phasing is a more sophisticated version of the flanging effect introduced in Chapter 5. With flanging, the notch frequencies are all evenly spaced, and are swept or varied using a low frequency oscillator to vary the time delay. With phasing, the notch frequencies are controlled by varying the gain, α , in each of the all-pass filters in the chain. The notch frequencies are not evenly spaced, and the frequencies are often swept exponentially resulting in small variations

initially, followed by faster changes in notch frequencies as time progresses. Another interesting effect can be achieved by stepping the notch frequencies up and down rather than varying them continually.

Commercial analog phaser pedals for guitars typically allow the musician to select either a four chain or an eight chain phaser, and have control knobs for speed of sweep and depth. Some also offer control knobs for resonance and a step option for the notch frequencies. Some commercial digital phasers offer four, eight, ten, or twelve chain options.



(a) $\alpha = 0.2$



(b) $\alpha = 0.8$

Figure 6.23: Frequency Response of All-Pass Filter Chain and Phase

Reverb

Reverb is a series of tightly spaced echoes that occur when sound travels in an acoustic environment (room, auditorium, or concert hall) to a listener. Sound waves travel not only directly to the listener but also as reflections off of the walls and ceiling. Reverb is affected by the dimensions of a room, by the materials the walls are made of, whether or not the room has carpet, and even by the number of people in the room. Figure 6.23 illustrates what the impulse response of a room might look like.

As illustrated in Figure 6.24, reverb is characterized by early reflections representing single reflections off of the walls followed by diffuse reverberation (very closely packed reflections coming to the listener from multiple directions). Because there are so many reflections very tightly packed, individual reflection or echo signals are not distinguishable by the ear, but are instead mixed together forming the overall reverb effect.

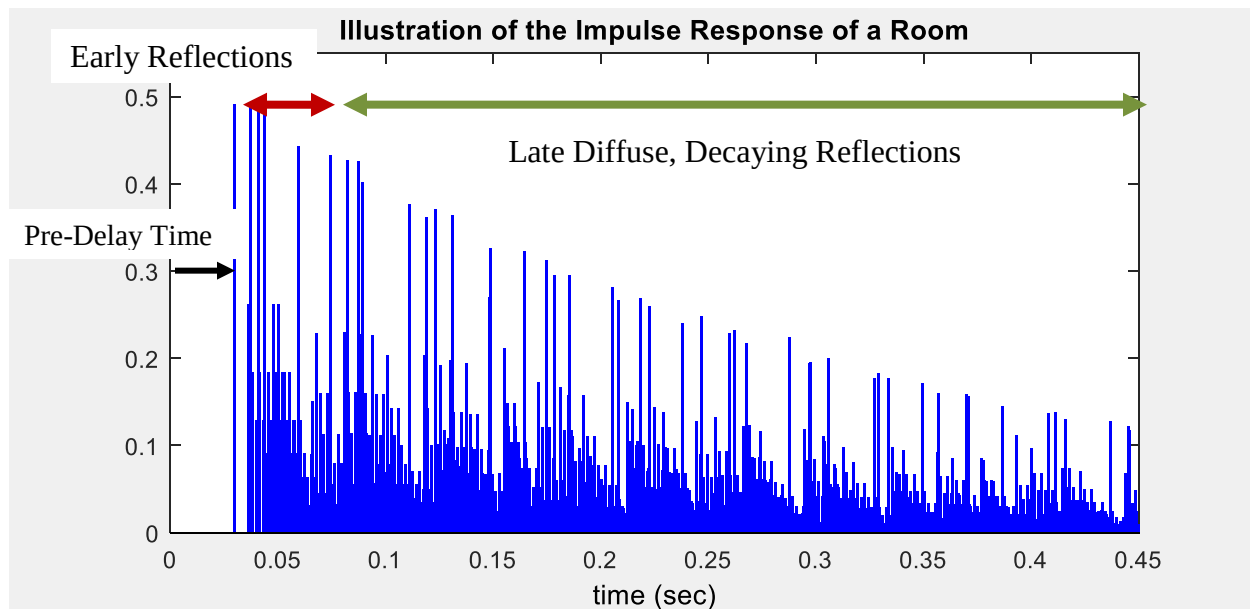


Figure 6.24: Impulse Response of a Room (Reverb)

The key parameters for reverb are the reverb time, the early reflection pattern, the pre-delay time, and the high frequency damping. The reverb time, also known as overall decay time, is the amount of time it takes for the echo signals to drop 60 db below or 1/10,000 of the original sound level. Reverb time is dependent both on the size of the space and the surfaces within the space. Materials such as glass, brick or marble increase reverb time, while drapes, carpet, and people reduce reverb time. Larger rooms tend to have longer reverb times. For a large concert hall, reverb time would typically be between 1.5 and 2 seconds. The early reflection pattern is the first reflections of the original sound off the walls. The pre-delay time is the amount of time before these first reflections reach the listener. Longer pre-delay time and larger spacing between early reflections are indicative of a larger room or space. High frequency damping models the frequency-dependent effect of humidity and surfaces on signal absorption. Higher frequencies attenuate more rapidly than lower frequencies.

There are many audio effects products on the market that reproduce the natural reverb in an actual physical space by digitally adding reverb to music. A block diagram of a generic digital reverb filter, first developed and refined by Manfred R. Schroeder in the 1960s and 1970s, is shown in Figure 6.25. Schroeder's work still forms the basis for reverb filter design. The parallel bank of comb filters produces the early reflection effect. The delays and gains are chosen to simulate the physical dimensions and materials of a specific room. Each filter has a different gain and the delays are chosen to be mutually prime. The cascaded all-pass filters form the diffuser.

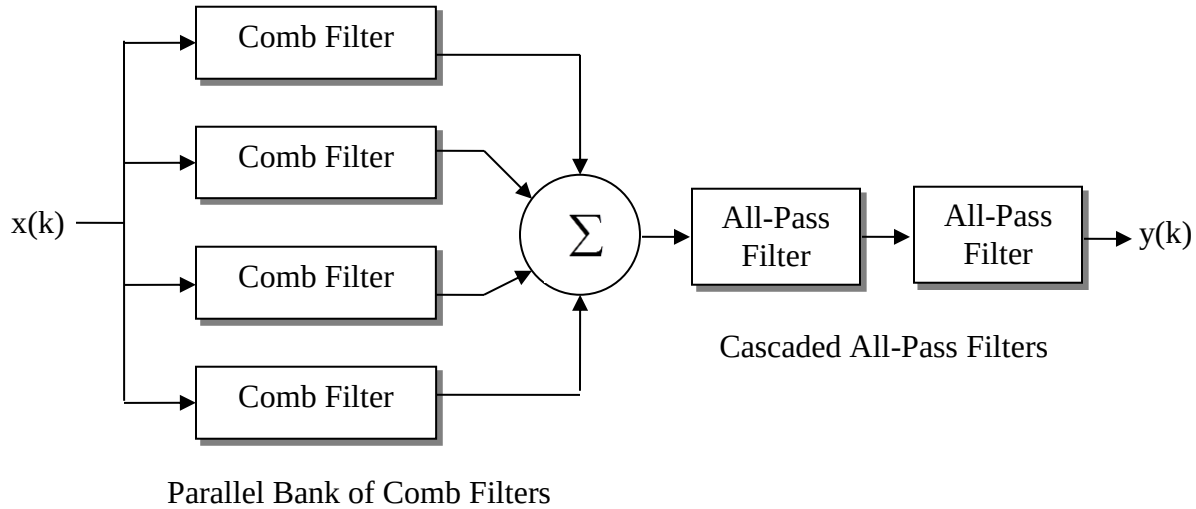


Figure 6.25: Schroeder's Digital Reverb Filter

Comb (Echo) filters and all-pass filters were introduced in the previous section as special types of IIR filters. The difference equation for a feedback comb filter (repeated from equation 6.10) is given by:

$$y(k) = x(k - N) + \alpha_N y(k - N) \quad 0 < \alpha_N < 1 \quad (6.14)$$

The difference equation models a sound wave bouncing off a wall or surface and returning to the ear. The constant α_N is the round-trip attenuation factor and $N \cdot T_s$ is the round-trip time delay. In order to maintain stability the attenuation constant, α_N , must be chosen less than one; otherwise, the output signal will continue to grow in magnitude eventually causing the sound system to saturate.

The impulse response for a comb filter is shown in Figure 6.26. The figure indicates that the comb filter produces a series of decaying echoes spaced apart by the time delay $\tau = N \cdot T_s$. This time delay is also referred to as the comb filter loop time. Loop times in the vicinity of 50 ms are used to create the reverb effect in a large concert hall, while shorter loop times (around 10 ms) would simulate the reverb heard in a small tiled shower. It is important that the loop times for the comb filters are all mutually prime; otherwise, echoes from two or more of the comb filters will occur at the same time, get added at the summer, and produce an undesirable echo spike.

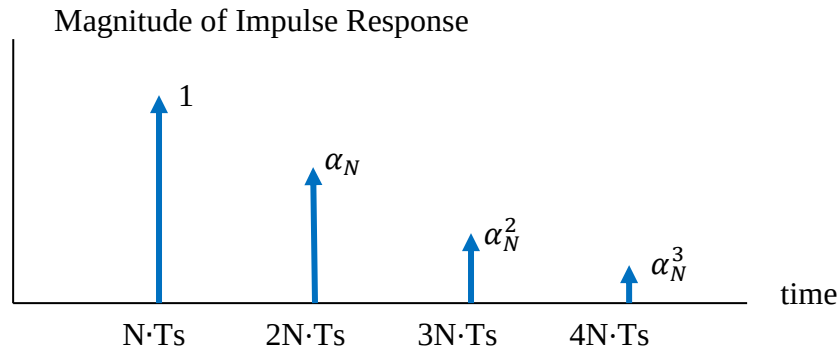


Figure 6.26: Impulse Response of Feedback Comb Filter

As illustrated in Figure 6.26, choosing the attenuation constant, α_N , close to one will produce a series of echoes that persist for a long period of time; whereas, choosing α_N close to zero will produce a quickly diminishing series of echoes. Given the comb filter loop time, τ , and the total desired reverb time, T_R , the attenuation constant is calculated as:

$$\alpha_N = 0.001^{\tau/T_R} \quad (6.15)$$

The difference equation for an all-pass filter (repeated from equation 6.12) is:

$$y(k) = -\alpha x(k) + x(k - N) + \alpha y(k - N) \quad 0 < \alpha < 1 \quad (6.16)$$

As discussed previously, an all-pass filter has a magnitude of one at all frequencies (hence the name all-pass) but does introduce phase distortion. As mentioned earlier in this section, one application for all-pass filters is to create the phasing effect. In a reverb filter, the all-pass filters are the diffusers creating the closely spaced indistinguishable echoes that enrich the sound. Typically, there would be 1000-3000 reflections per second, and the spacing between these reflections would be random. Fewer reflections would make echoes distinguishable, while non-random spacing between reflections creates an undesirable metallic ringing effect. The loop times (or delay) for the all-pass filters is considerably shorter than the loop times for the comb filters.

Challenge Question 6.5

It is possible to create a reverb filter using a digital FIR filter instead of the IIR echo and all-pass filters. Explain how and discuss the obvious disadvantage of an FIR filter.

Modern reverb filters are based on Schroeder's initial design ideas but are considerably more sophisticated. Impulse responses of spaces can be measured and matched. Designs and algorithms vary from manufacturer to manufacturer and are typically proprietary. Figure 6.27 illustrates a more sophisticated reverb design that uses an FIR filter to produce the early reflections, banks of comb and all-pass filters for diffusers, and low-pass or equalization filters to model frequency dependent absorption effects.

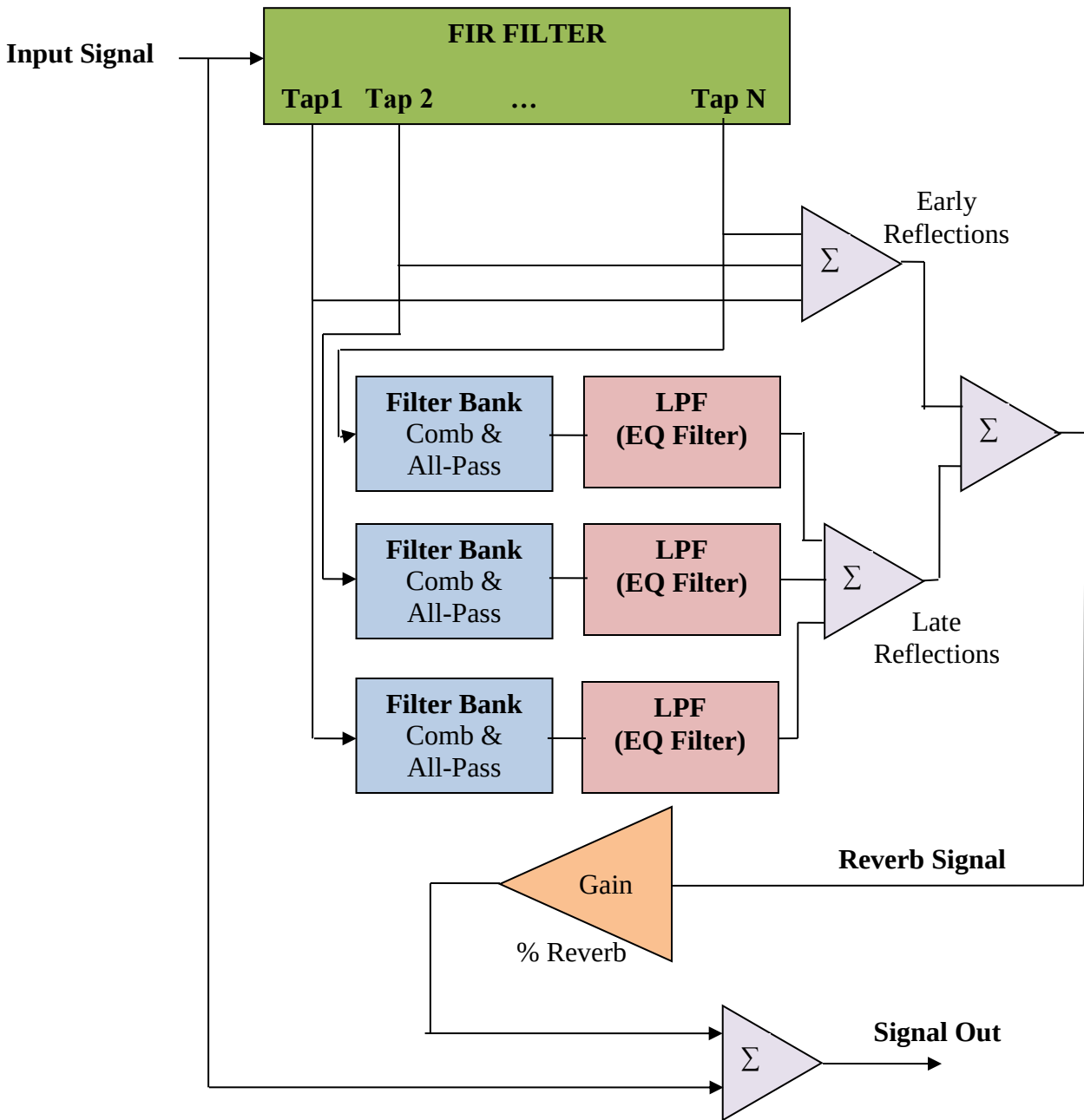


Figure 6.27: Reverb Filter Block Diagram

Example 6.6: Design of a Reverb Filter

This example utilizes the parameters of the reverb filter described in Dodge and Jerse; namely, the filter loop times and total desired reverb time suggested for each of the filters to simulate a medium-sized concert hall as shown in Table 6.1. Also shown in Table 6.1 are the calculated filter attenuation constants using equation 6.15, the calculated delay (in samples) based on a

sampling frequency of 44,100 Hz, and the adjusted delay (in samples) to ensure that all of the comb filter loop times are indeed mutually prime.

Table 6.1: Filter Parameters for Reverb Filter

Filter	Loop Time (Delay)	Total Desired Reverb Time	Attenuation Constant	Delay (Samples)	Delay (Factored)	Delay (Samples) for Mutually Prime
Comb 1	29.7 ms	2 sec	0.9025	1310	$2*5*131$	1311
Comb 2	37.1 ms	2 sec	0.8797	1636	$2*2*409$	1636
Comb 3	41.1 ms	2 sec	0.8677	1813	$7*7*37$	1813
Comb 4	43.7 ms	2 sec	0.8599	1927	$41*47$	1927
Allpass1	5.0 ms	96.83 ms	0.7	221		
Allpass2	1.7 ms	32.92 ms	0.7	75		

Since 1310 and 1636 share a common factor of 2, the first delay was increased to 1311 sample periods which factors as $3*19*23$ and is therefore mutually prime with the delays of the other three comb filters.

Download the Reverb.mdl shown in Figure 6.28 and the BACH PRELUDE.wav files from the text website. Run the simulation and use the Manual Switch to turn the reverb effect on and off. Notice that output signal from the reverb filter is combined with the original signal, and a gain of 0.1 is applied to the reverb signal to adjust for % of reverb. The interested reader can easily adjust any of the parameters in the simulation to explore the effects on the overall sound.

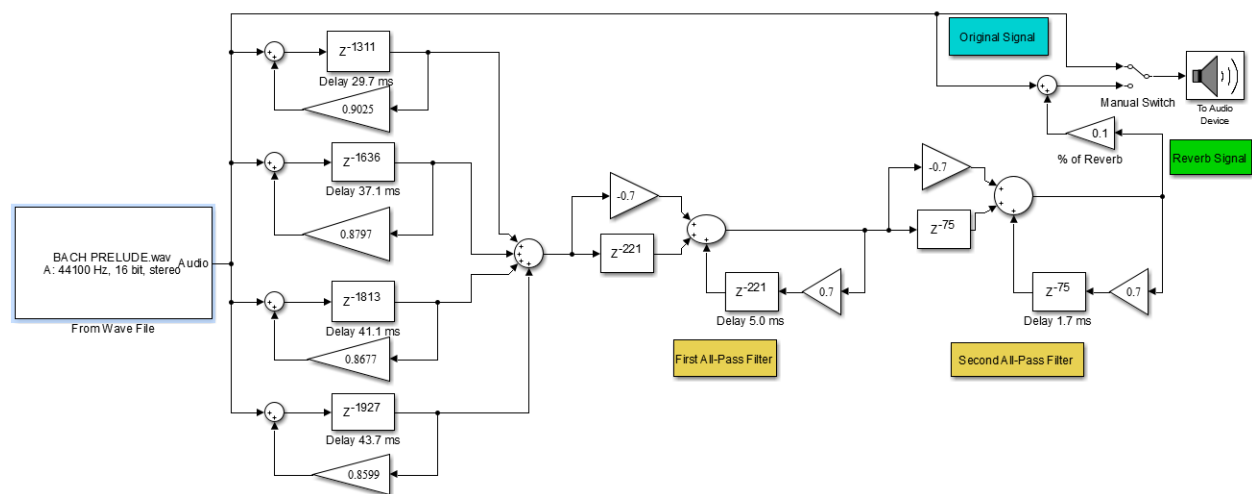


Figure 6.28: Simulink® Model of Reverb Filter

Figure 6.29 shows the impulse response of each comb filter and the overall impulse response of the parallel bank of comb filters. As expected, each comb filter produces a distinct set of decaying echoes at integer multiples of the time delay for that filter. The 1st comb filter produces echoes at 1311, $2*1311$, $3*1311$, ... The impulse response of the parallel bank of comb filters is

simply the four individual impulse responses added together. Because the delays are mutually prime, none of the filters produces an echo at the same time as another filter within the time-span of the total reverb time, so undesirable spikes are not a problem.

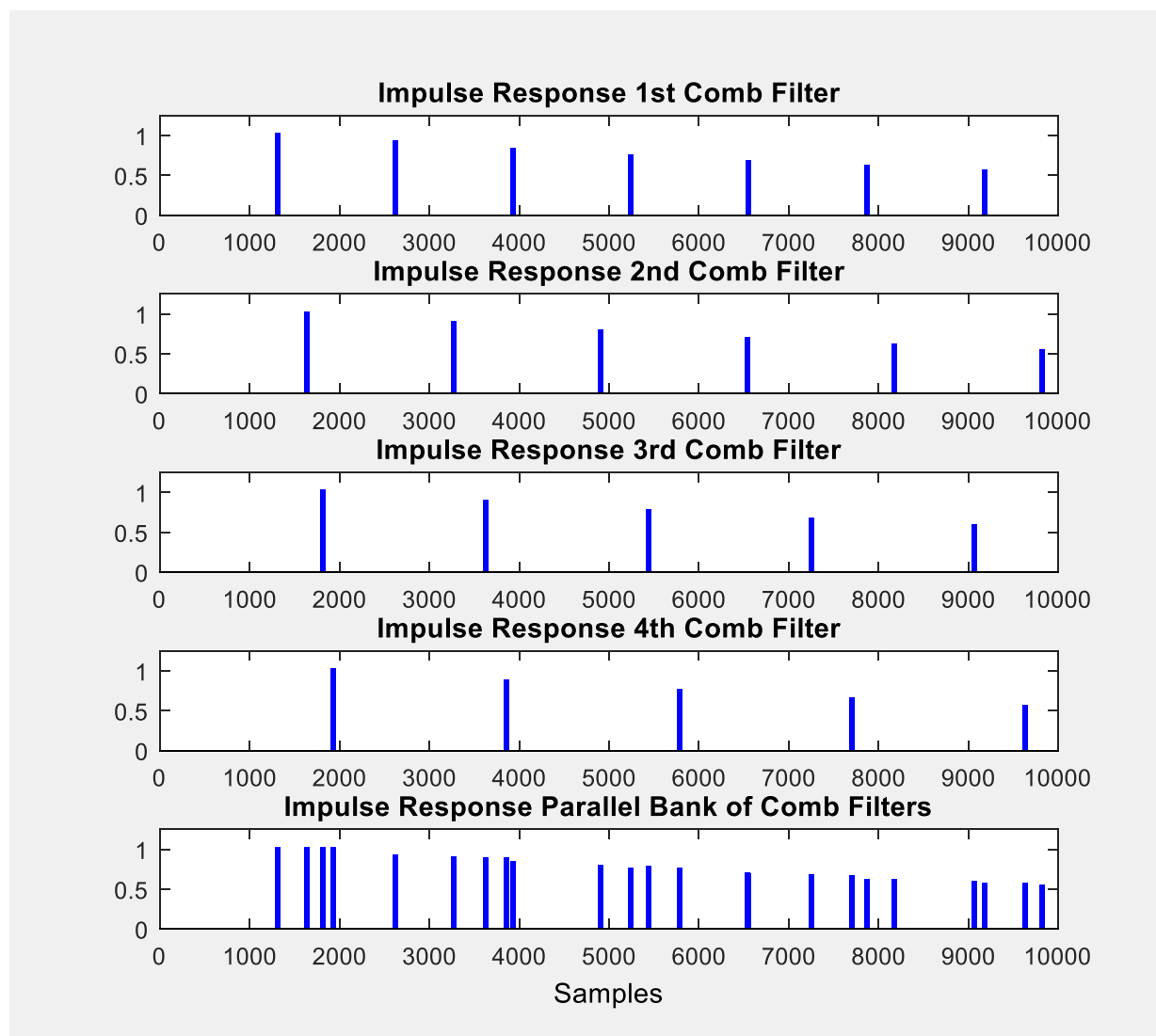


Figure 6.29: Impulse Response of the Comb Filters

Figure 6.30 shows the impulse response for each of the all-pass filters, and for the cascaded combination of the two all-pass filters. Since these filters are cascaded, the overall impulse response is the convolution of the two individual impulse responses.

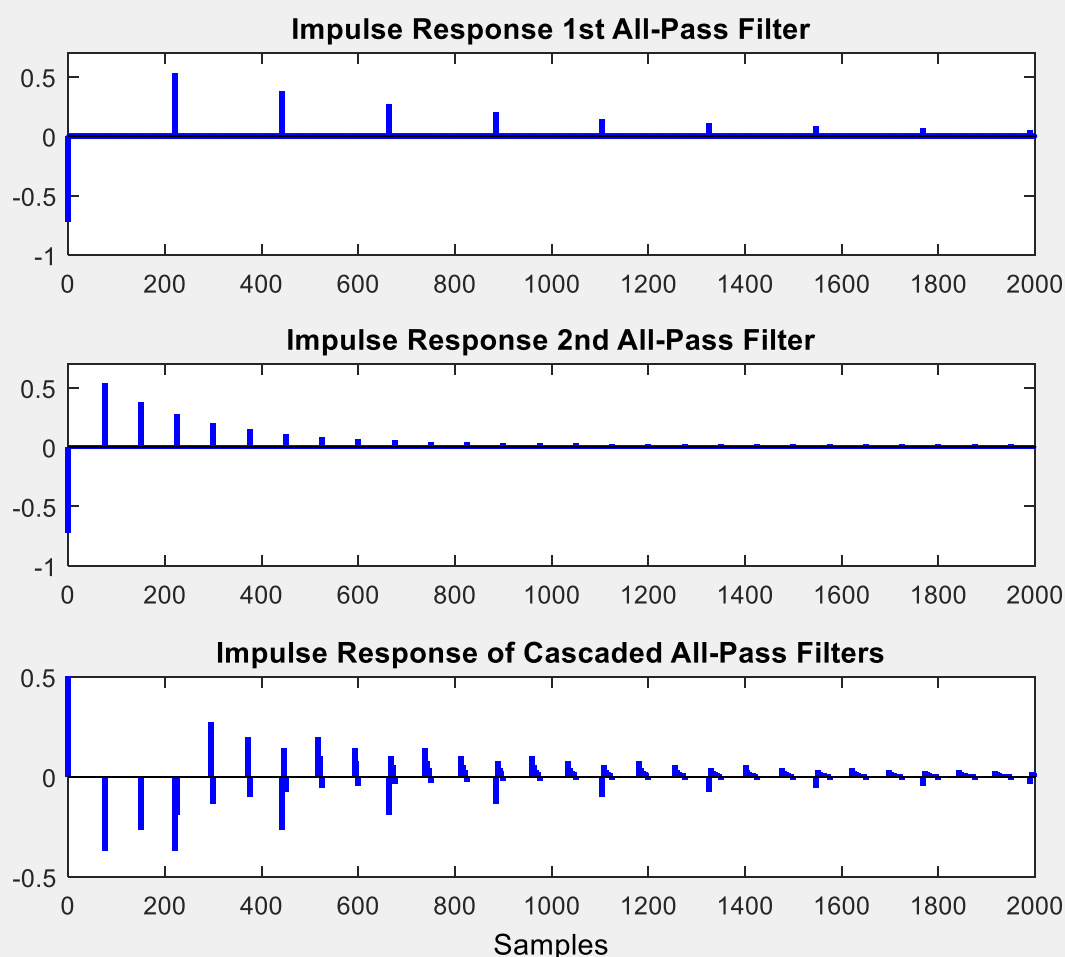


Figure 6.30: Impulse Response the All-Pass Filters

Finally, Figure 6.31 shows the impulse response of the entire reverb filter. The upper plot shows the impulse response for the first 10000 samples (about $\frac{1}{4}$ of a second), while the lower plot extends the impulse response to a time-span of 1 second.

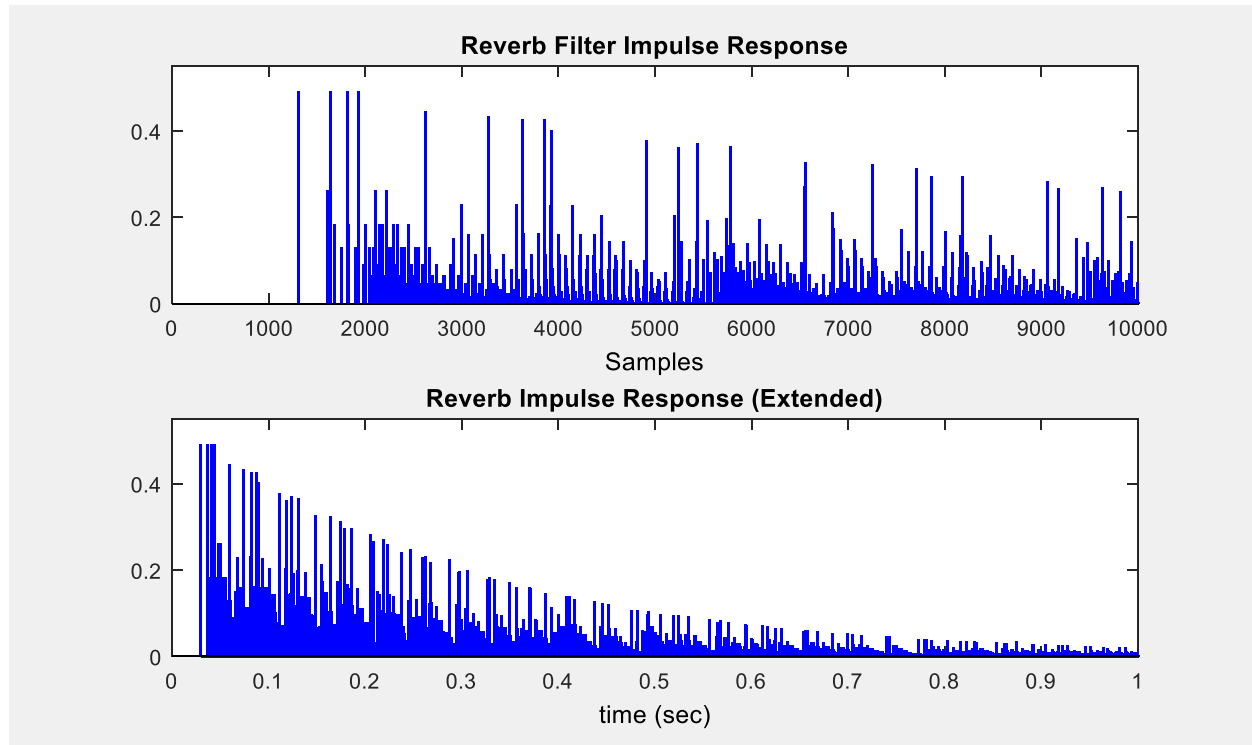


Figure 6.31: Impulse Response of Reverb Filter

Digital Graphic Equalizer

A graphic equalizer is used to alter an audio signal by boosting and/or attenuating selected frequency bands of the signal. Graphic equalizers are used to tune an audio signal to match a listener's preference; that is, boost the bass portion of the signal or enhance/re-shape the instrumental or vocal portion of the signal. Graphic equalizers are also used to eliminate noise and to correct losses incurred in an audio signal perhaps due to the type of room in which the signal was played/recorded. Some listening environments provide lots of signal reflection resulting in lively music; whereas, other listening environments deaden the higher frequencies due to poor reflection characteristics.

A digital graphic equalizer consists of a parallel bank of FIR or IIR low-pass, band-pass and high-pass filters in the audio band. One of the most difficult aspects of the design is how to handle the overlap effect between adjacent filters where the two frequency responses are summed together. Figure 6.32 illustrates the unwanted effects (extra boost and/or ripple) that can occur between adjacent filters, as well as the desirable smooth response that can be achieved using digital filters with additional digital processing algorithms.

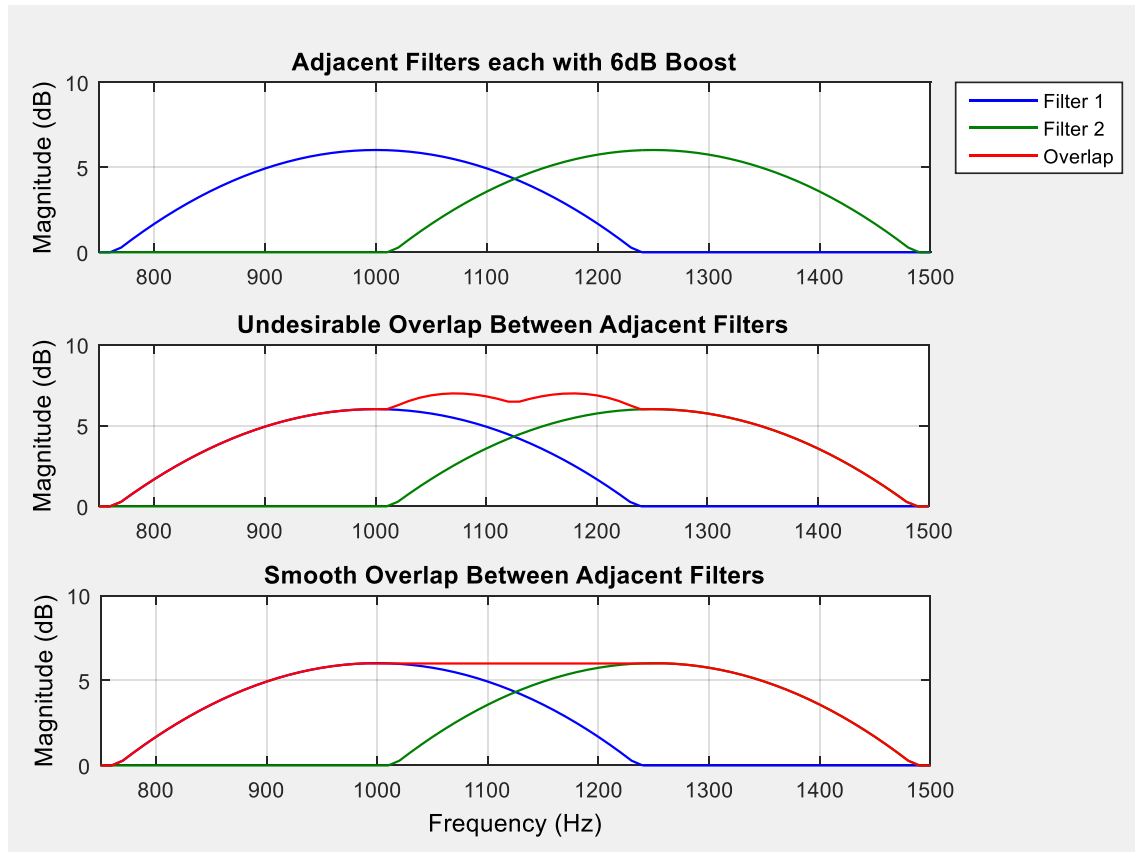


Figure 6.32: Graphic Equalizer – Adjacent Filter Effects

Example 6.7: Digital Graphic Equalizer

Download the GraphicEq model shown in Figure 6.33 and the BACH PRELUDE wav file from the text website. GraphicEq.mdl consists of one low-pass filter, four bandpass filters, and a high-pass filter connected in parallel. The filters were designed using the Filter Design and Analysis tool. Specific filter parameters are shown in Table 6.2. Play the simulation and experiment with the slider gains to boost or attenuate various frequencies.

Table 6.2: Filter Properties for Graphic Equalizer

Filter Type	Design Method	Filter Order	Sampling Frequency (Hz)	Cutoff Frequencies (Hz)
Lowpass	FIR Hamming	256	44100	750
Bandpass	FIR Hamming	256	44100	750; 1500
Bandpass	FIR Hamming	256	44100	1500; 3000
Bandpass	FIR Hamming	256	44100	3000; 6000
Bandpass	FIR Hamming	256	44100	3000; 11500
Highpass	FIR Hamming	256	44100	11500

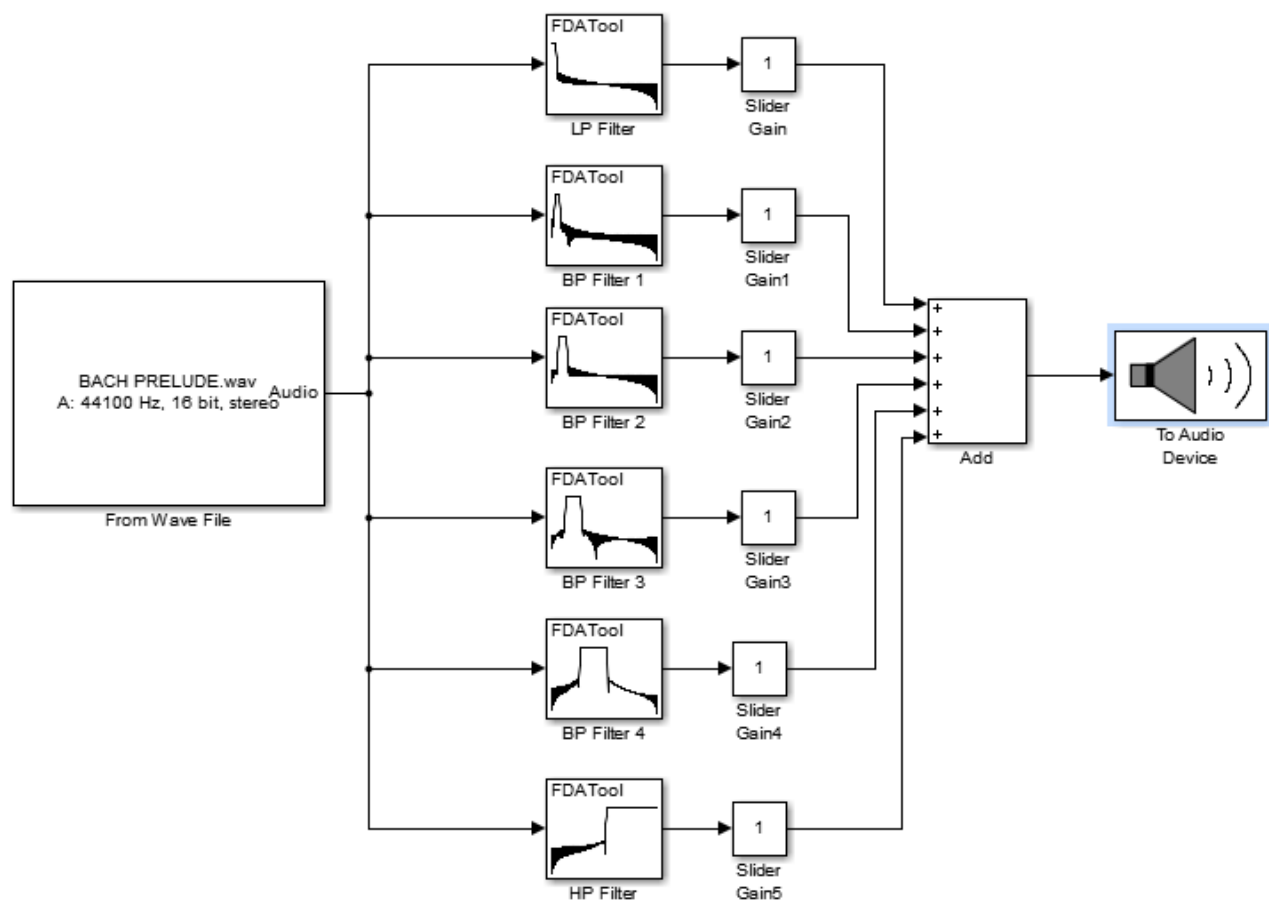


Figure 6.33: Graphic Equalizer Simulink Model

Figure 6.34 illustrates the frequency response for each individual filter as well as the overall frequency response with all six filters connected in parallel and all slider gains set to 1. The overall frequency response is not perfect; ideally, the response would be a flat line at 0 dB over the entire frequency range. This particular design does exhibit some ripple which would require some alteration to the design to flatten out.

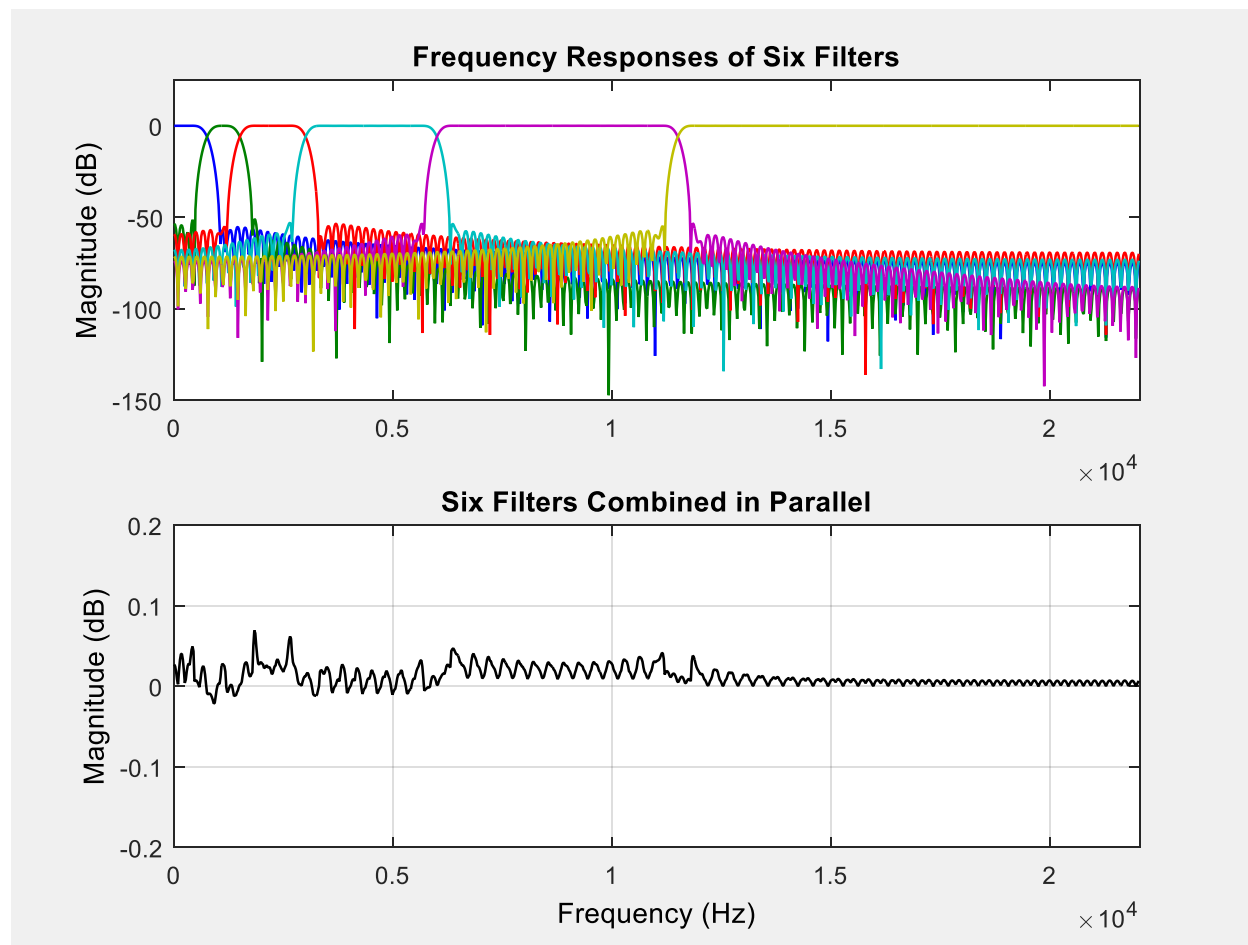


Figure 6.34: Frequency Response of Graphic Equalizer

Answers to Challenge Questions

Question 6.1 How many multiplications per sampling instant must be performed with the FIR and IIR filters designed in Example 6.3? How many multiplications per second (MPS) must be performed for each of these filters?

Assuming each of the filters is implemented in the time domain, we have:

$$\text{FIR Filter } y(k) = \sum_{i=0}^{87} b_i x(k-i)$$

$$\text{IIR Filter } y(k) = \sum_{i=0}^{10} b_i x(k-i) - \sum_{i=1}^{10} a_i y(k-i)$$

The FIR filter would require 88 (N+1) multiplications per sampling instant or 4,224,000 (88*48K) multiplications per second (MPS). The IIR filter would require 21 (2*N+1) multiplications per sampling instant or 1,008,000 multiplications per second.

Computational complexity can be reduced by taking advantage of the symmetry of the filters. Recall from Chapter 5 that for an FIR filter, $b_0=b_N$, $b_1=b_{N-1}$, ... so the number of multiplications could be cut in half to 44 multiplications per sampling instant or 2,112,000 MPS. The IIR filter also has a symmetric numerator (with a sign change):

b0 = 0.03203191821694
b1 = -0.05878107521081
b2 = 0.10186869443828
b3 = -0.09766972701608
b4 = 0.07231401919473
b5 = 0.00000000000000
b6 = -0.07231401919473
b7 = 0.09766972701608
b8 = -0.10186869443828
b9 = 0.05878107521081
b10 = -0.03203191821694

So the number of multiplies can be reduced to 15 per sampling instant or 720,000 MPS.

Question 6.2 In example 6.4, the poles of the IIR filter shift when the coefficients are quantized to B bits and it is even possible to make the filter unstable. Will an FIR filter exhibit the same instability problem if the number of bits for the filter coefficients is continually reduced? Explain.

No, the FIR filter will not exhibit stability problems when the filter coefficients are quantized. The FIR filter has all of its poles at the origin (denominator = z^N) and these poles will not shift.

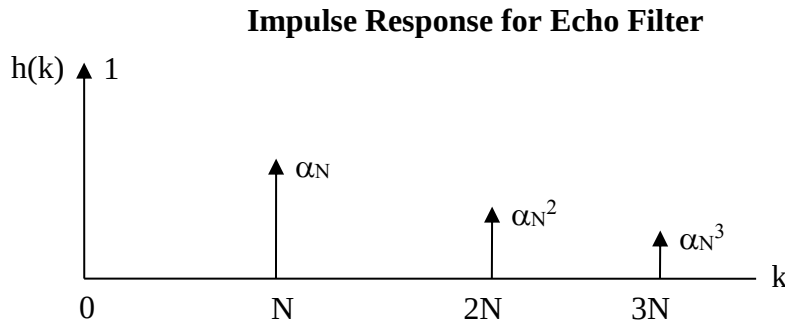
Another way to look at this is that the FIR filter is non-recursive (no feedback) and will not exhibit stability problems. However, as shown in Chapter 5, quantizing the FIR coefficients can result in poor filter performance (failure to meet design specifications).

Question 6.3 What is the impulse response of the echo filter?

The impulse response of the echo filter can be found by setting the input $x(k) = \delta(k)$ and solving the difference equation recursively until a pattern emerges. For simplicity, assume a value for N , say 5.

$$\begin{aligned} y(k) &= x(k) + \alpha_N y(k-5) \\ y(0) &= \delta(0) + \alpha_N y(-5) = 1 \\ y(k) &= \alpha_N y(k-5) = 0 \quad \text{for } k = 1 \dots 4 \\ y(5) &= \alpha_N y(0) = \alpha_N \\ y(k) &= \alpha_N y(k-5) = 0 \quad \text{for } k = 6 \dots 9 \\ y(10) &= \alpha_N y(5) = \alpha_N^2 \\ y(k) &= \alpha_N y(k-5) = 0 \quad \text{for } k = 11 \dots 14 \\ y(15) &= \alpha_N y(10) = \alpha_N^3 \end{aligned}$$

The graph of the impulse response for an echo filter with a time delay of N is in the following graph.



Each of the impulses represents another reflection of the original sound wave off the two parallel walls.

Question 6.4 What will happen if the feedback factor, α_N , is chosen to be greater than one for the echo filter? Why?

If the feedback gain, α_N , is chosen to be greater than one, the output signal will grow continually louder until the signal finally saturates the audio system. This can be seen from the difference equation (amplification of the feedback signal) or from the transfer function. The echo filter has N poles equally spaced around a circle of radius α_N (i.e., $z^N = \alpha_N$). If α_N is greater than one, then the filter poles all lie outside the unit circle making the filter unstable.

Question 6.5

It is possible to create a reverb filter using a digital FIR filter instead of the IIR echo and all-pass filters. Explain how and discuss the obvious disadvantage of an FIR filter.

Figure 6.24 shows what the impulse response of a room might look like. If we measured the impulse response in some actual desired space, this impulse response would then be the coefficients of the corresponding digital FIR filter. The problem is that the filter order would be huge since there are thousands of reflections per second.

Chapter 6 Problems

Problem 6.1: The transfer function for an IIR filter is:

$$\frac{0.07083 z^5 + 0.1611 z^4 + 0.2523 z^3 + 0.2523 z^2 + 0.1611 z + 0.07083}{z^5 - 1.086 z^4 + 1.6 z^3 - 0.9263 z^2 + 0.4804 z - 0.09993}$$

- (a) Find the filter order and the filter poles. Is the filter stable?
- (b) Write the difference equation for the filter.
- (c) Using the MATLAB command ***dimpulse***, plot the first twenty terms of the impulse response.
- (d) Using the MATLAB command ***freqz***, plot the frequency response for this filter (assume a sampling frequency of 16 kHz). What type of filter is this?

Problem 6.2:

- (a) Using the Filter Design and Analysis tool, design a Butterworth digital IIR LPF to meet the following specifications:

- i. Passband: 0 - 22.5kHz (with maximum 1 dB ripple)
- ii. Stopband: $f \geq 28$ kHz with $A_s \geq 60$ dB
- iii. Sampling Freq: 96 kHz

- (b) Repeat part (a) using a Chebyshev II design.
- (c) Repeat part (a) using an Elliptic design.
- (d) Compare the frequency responses of the three filters. Also, compute the number of multiplications per second required for each filter.

Problem 6.3: (IIR Filter Design)

- (a) Design a Chebyshev II digital IIR BPF to meet the following specifications:

- i. Passband: 5 - 10kHz (with maximum 0.1 dB ripple)
- ii. Stopband: $f \leq 3$ kHz and $f \geq 12$ kHz with $A_s \geq 40$ dB
- iii. Sampling Frequency: 30 kHz

Include the transfer function of your filter as well as **magnitude** and **phase** plots.

- (b) Suppose the filter input is $x = \sin(2\pi \cdot 2000 \cdot t) + \sin(2\pi \cdot 8500 \cdot t)$. Plot the filter input and the filter output. Also, compute the time delay through the filter for the two input frequencies.
- (c) Repeat parts (a) and (b) for an Optimal Equiripple FIR filter.
- (d) Compare your two filter designs in terms of phase distortion, number of multiplies required to implement the difference equation, etc.

Problem 6.4: (Effect of Wordlength on IIR Filter Stability and Performance)

- (a) Design an IIR bandstop filter (with 2nd order sections) to meet the following specifications:
- Notch Frequency: 2 kHz
 - Attenuation at Notch Frequency ≥ 60 dB
 - Passband edge frequencies: 1.8 kHz and 2.2 kHz
 - Passband Ripple ≤ 0.1 dB
 - Sampling frequency $f_s = 8$ kHz

Provide the magnitude frequency response plot from the Filter Design and Analysis Tool. DO NOT turn in the filter coefficients.

- (b) Determine the minimum number of bits required to maintain stability of the filter. Turn in the MATLAB code to document your answer.
- (c) Quantize the filter coefficients to the number of bits determined in (b) and plot the frequency response of the quantized filter vs. the frequency response of the original filter. How do they compare? (Turn in your MATLAB code for this part)
- (d) Determine the number of bits required for good filter performance. Again, plot the frequency response of the quantized filter vs. the frequency response of the original filter to verify good filter performance. (Turn in your MATLAB code for this part)

Problem 6.5:

- (a) Write the difference equation for a 4th order all-pass filter assuming $\alpha = 0.2$.
- (b) Calculate and plot the first 10 terms of the impulse response.
- (c) Find the poles of the all-pass filter.
- (d) Plot the frequency response for the filter assuming a sampling frequency of 48 kHz.
- (e) Repeat parts (a)-(d) assuming $\alpha = 0.8$. How does α affect the impulse response and the frequency response?

Problem 6.6:

- (a) Write the difference equation for a 6th order echo filter assuming $\alpha = 0.2$.
- (b) Calculate and plot the first 20 terms of the impulse response for the echo filter.
- (c) Find the poles of the echo filter.
- (d) Plot the frequency response for the filter assuming a sampling frequency of 48 kHz. Identify the frequencies that are amplified by the echo filter.
- (e) Repeat parts (a)-(d) assuming $\alpha = 0.8$. How does α affect the impulse response and the frequency response?
- (f) Plot the frequency response for a 10th order echo filter assuming $\alpha = 0.8$ and the sampling frequency is 48 kHz. How does the increase in order affect the frequency response?

Problem 6.7:

- (a) Referring to Figure 6.14, derive Equation 6.7.
- (b) Also using Figure 6.14, derive the transfer function from the intermediate signal, $w(k)$, to the output signal, $y(k)$.

CHAPTER 6 LAB EXERCISE

IIR Filters

Objectives

1. Design two IIR filters using the Filter Design and Analysis Tool in MATLAB.
2. Investigate the effect of coefficient wordlength on the filter performance.
3. Explore the effect of phasing on music.

Procedure

A. IIR FILTER DESIGN

1. Choose two *different* filter types (low-pass, high-pass, band-pass, or notch). Enter your choices in the IIR Design Table.
2. Choose a sampling rate for each of the filters. Enter your choices in the IIR Design Table.
3. Specify passband and stopband for your filters. Remember, you are constrained to frequencies which are less than half the sampling frequency. Enter your choices in the IIR Design Table.

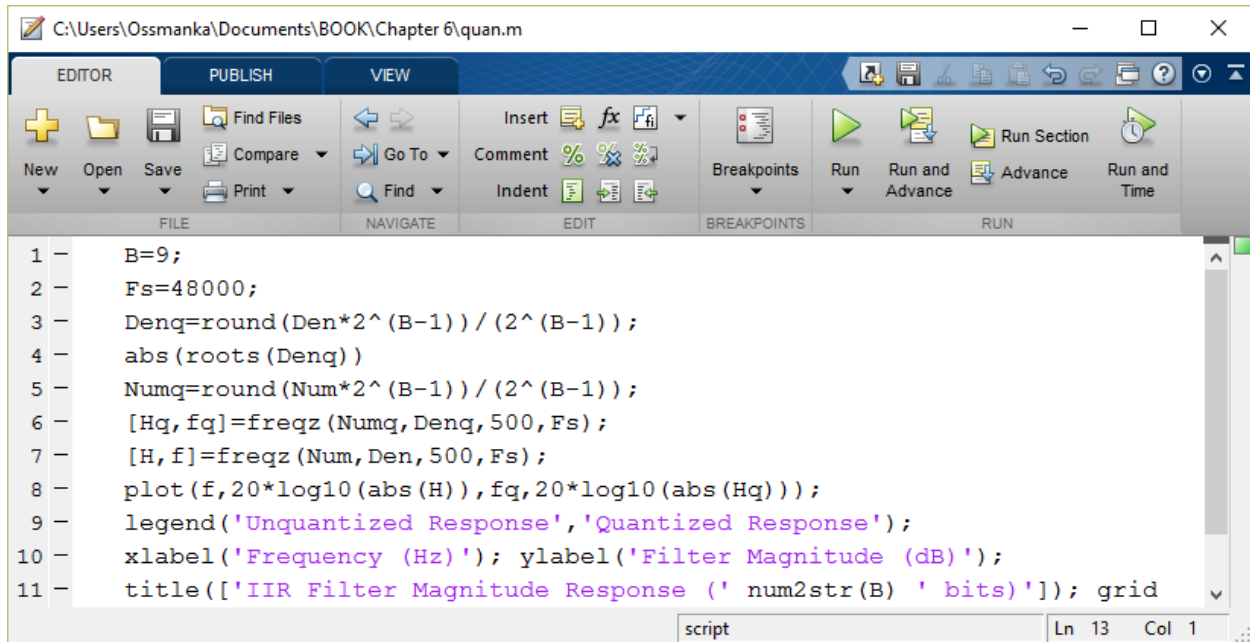
Table 1: IIR DESIGN SPECIFICATIONS

Filter Type	Fs (kHz)	Passband	Stopband	Method	Order

4. Start MATLAB.
5. Launch the Filter Design and Analysis Tool (`>> fdatool`).
6. Design an IIR filter using your first set of specifications in the IIR Design table. Enter your filter method (Butterworth, Chebychev I, Chebychev II, or Elliptic) and the resulting filter order in the IIR Design Table.
7. Print the resulting magnitude response and phase response. Notice that an IIR filter does not have linear phase response!
8. Convert the IIR filter to a single section by selecting Edit→Convert to Single Section.
9. Export the filter coefficients by selecting File → Export. Choose some unique variable names for your filter numerator and denominator.
10. Repeat steps 6 - 9 for your second set of filter specifications in the IIR Design Table. ***Be sure to save the filter numerator and denominator under different variable names than those used for the first filter!***

B. WORDLENGTH EFFECTS

The following m-file of MATLAB commands will quantize the filter denominator coefficients (Den) to a specified number of bits then calculate the magnitude of the poles of the quantized denominator. The m-file also quantizes the filter numerator (Num) and plots the frequency response of the quantized filter versus the original filter. ***You will need to modify this code for your filter; that is, enter your sampling frequency, F_s , and your variable names for Num and Den.***



```

1 - B=9;
2 - Fs=48000;
3 - Denq=round(Den*2^(B-1))/(2^(B-1));
4 - abs(roots(Denq))
5 - Numq=round(Num*2^(B-1))/(2^(B-1));
6 - [Hq,fq]=freqz(Numq,Denq,500,Fs);
7 - [H,f]=freqz(Num,Den,500,Fs);
8 - plot(f,20*log10(abs(H)),fq,20*log10(abs(Hq)));
9 - legend('Unquantized Response','Quantized Response');
10 - xlabel('Frequency (Hz)'); ylabel('Filter Magnitude (dB)');
11 - title(['IIR Filter Magnitude Response (' num2str(B) ' bits)']); grid

```

1. Vary the number of bits, B , in the m-file and find the minimum number of bits needed for stability for your first IIR filters. Record your answer in Table 2. Recall, a discrete system is stable if and only if the magnitude of the poles is strictly less than 1.

Table 2

IIR FILTER	Minimum Bits for Stability (Direct Realization)

2. Using the minimum number of bits required for stability, plot the frequency response of the quantized system and the frequency response of the original IIR filter on the same plot. How does the quantized response compare to the original frequency response?

- Determine the number of bits required to achieve a good match between the quantized filter and the original IIR filter. Fill in the answer in the Table 3.

Table 3

IIR FILTER	Minimum Bits for Good Performance (Direct Realization)

- Repeat steps 1-3 for the 2nd IIR filter making sure to modify the m-file appropriately.

C. CASCADE REALIZATION AND WORDLENGTH EFFECTS

- Open the Filter Design and Analysis Tool in MATLAB.
- Enter the design parameters for your first IIR filter designed in Part A.
- Click on Edit→Convert Structure. Choose 2nd Order Sections.
- Export the SOS matrix and the G matrix (it is not necessary to change these names).
- Return to the MATLAB workspace and execute the statement: `format long; SOS`

Row 1 is the transfer function for the 1st biquad. The first three numbers (columns 1-3) are the numerator and the last three coefficients (columns 4-6) are the denominator. Row 2 contains the 2nd biquad transfer function and the pattern continues.

The m-file, IIR-quant.m, (code provided in Figure 6.11) will quantize the denominator of each 2nd order section and find the magnitude of the poles for each quantized section. It will also quantize the numerator of each section and plot the frequency response of the quantized cascaded IIR filter versus the un-quantized filter. ***You will need to modify this m-file by entering your sampling frequency, Fs. As long as you exported your cascaded filter under the default variables of SOS (Second Order Sections) and G (Gain), the rest of the code will execute fine.***

- Vary the number of bits, B, in the m-file to determine the minimum number of bits required for stability in the cascade realization for your first filter. Enter your answer in Table 4.

Table 4

IIR Filter	Minimum Bits for Stability In Cascade Realization	Minimum Bits for Stability In Direct (Full) Realization

- Determine the number of bits required for good filter performance (i.e. a good match to the original filter) for each of your IIR filters. Fill in the answer in Table 5.

Table 5

IIR Filter	Minimum Bits for Good Performance In Cascade Realization	Minimum Bits for Good Performance In Direct (Full) Realization

- Repeat Steps 1-7 for your 2nd IIR filter using a cascade implementation.

How does the cascade implementation differ from the single stage implementation in terms of the number of bits needed to maintain stability and the number of bits required to meet the filter design specifications?

D. Phasing

A block diagram for a phaser is shown in Figure 1. As discussed in the text, the all-pass filter chain creates a set of notch frequencies in the phaser and the spectrum of incoming audio signal will be attenuated or even eliminated at these frequencies. The notch frequencies are swept or varied by changing the gains, α , in the all-pass filters. This portion of the lab will begin with an exploration of the all-pass filter chain to determine how the various parameters affect the frequency response of the phaser.

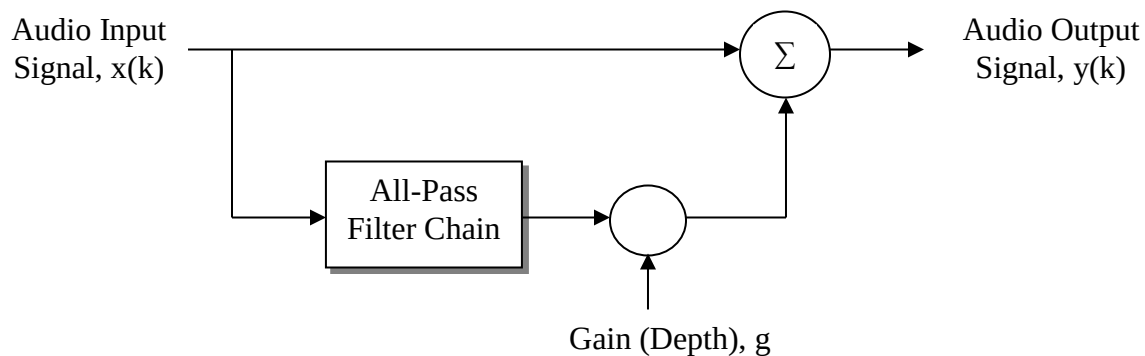


Figure 1: Phaser

- Copy the m-file ***allpass.m*** from the text website into the current MATLAB directory.

2. In the MATLAB workspace, type the command

```
allpass( [0.2 0.2 0.2 0.2] , 1, 44100)
```

You should see a plot exactly like Figure 6.23(a). The first argument in **allpass.m** specifies the gain for each 1st order all-pass filter. The gains do not need to be the same but they must be between 0 and 1. The size of the vector determines how many all-pass filters are in the chain. There would be four all-pass filters for the command just executed. The second argument in **allpass.m** is the depth, g , and may range from 0 up to 1. The 3rd argument is the sampling frequency of the incoming audio signal.

3. Experiment with **allpass.m** in order to answer the following questions:

What is the relationship between the number of notch frequencies and the number of all-pass filters included in the chain?

How do the filter gains, α , affect the location of the notch frequencies?

What effect does the variable depth have on the frequency response of the phaser?

4. Copy the files **phaser_env.m**, **phase.mdl** and **G_Maj.wav** from the text website into the current MATLAB directory.

Note: The file **G_maj.wav** was originally downloaded from the freesound website at <http://www.freesound.org>.

5. Open the Simulink file, **phaser.mdl**. The model should look like Figure 2.

The model consists of a chain of four 1st order all-pass filters. The first two filters use gains from **alpha1.signals.values** while the second two filters use gains from **alpha2.signals.values**. Variations in the gain values over time will cause the notch frequencies to sweep. These gains will be generated using the m-file **phaser_env.m**. The model includes a slider gain to control depth. Setting depth to 0 will eliminate the output from the all-pass filter chain and simply play the original audio signal. Setting depth equal to 1 will cause 100% cancellation of the audio input spectrum at the notch frequencies.

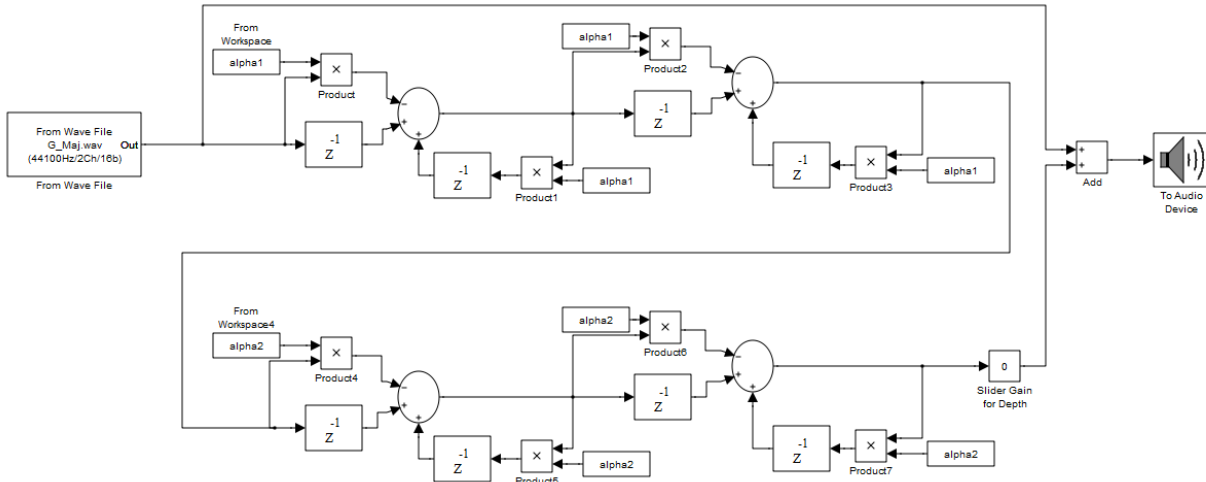


Figure 2: phaser.mdl

6. In the MATLAB workspace, run the command

```
[alpha1 alpha2] = phaser_env([0.1 0.9 0.3 0.9], 1, 3);
```

The **phaser_env.m** file will compute and plot the α_1 and α_2 signal values that are needed to run the **phaser.mdl**. The first argument (a vector of length 4) specifies min and max values for α_1 and min and max values for α_2 respectively. The second argument, T , specifies the period of time to complete one cycle over which the gains α_1 and α_2 will sweep from max to min and back to max. Equivalently, T controls the sweep time of the notch frequencies. The third argument, K , controls the steepness of the gain transition from min to max.

7. Run the **phaser.mdl** and listen to the audio through headphones. The slider gain is initially set at 0 (original audio out). Vary this gain to hear the phasing effect.
8. Experiment with the input variables for **phaser_env**, particularly cycle time, T , and steepness of transition, K . Explain the effect of the variables on the sound that you hear.

Comments:

- The **phaser.mdl** could be easily modified to an eight chain all-pass filter.
- The **phaser_env.m** file could be modified for step changes in α which would cause step changes in the notch frequencies rather than continuous sweeping.

CHAPTER 7 DISCRETE FOURIER TRANSFORMS

Discrete Fourier transforms (DFTs) form the basis for many digital signal processing applications including efficient implementation of high order digital filters, fast convolution and correlation, and radar signal processing. The chapter begins with an introduction to Discrete Fourier transforms which represent a sampled frequency spectrum of a signal. A fast Fourier transform (FFT) is simply an efficient algorithm for computing a discrete Fourier transform. The Radix-2 Decimation in Time FFT is presented along with the computational benefits of the algorithm. Specialized algorithms for focused spectral analysis, Zoom FFTs and the Goertzel Algorithm, are then introduced. Section 7.6 illustrates how FFTs can be efficiently used for fast convolution and correlation calculations. Matched filter design is explored in the final section.

7.1 DISCRETE FOURIER TRANSFORMS

The spectrum (frequency content) of a signal was introduced in Sections 2.5 and 2.6 for periodic and non-periodic analog signals. Discrete Fourier Transforms (DFTs) essentially digitize the spectrum making it useable for DSP applications. The key points to understand about a Discrete Fourier Transform are:

- The Discrete Fourier Transform of a time signal, if done “correctly,” is simply a sampled version of the spectrum of the signal over some finite frequency range.
- The Discrete Fourier Transform is very useful in computationally intensive filtering, convolution, or correlation applications where it is often more efficient to do calculations in the frequency domain instead of the time domain.
- The Discrete Fourier Transform can be efficiently computed using a technique called a Fast Fourier Transform.

We begin the mathematical development of the DFT with the Fourier Transform:

$$\begin{aligned}\text{Fourier Transform :} \quad X(f) &= \int_{-\infty}^{\infty} x(t)e^{-j2\pi ft} dt \\ \text{Inverse Fourier Transform :} \quad x(t) &= \int_{-\infty}^{\infty} X(f)e^{j2\pi ft} df\end{aligned}\tag{7.1}$$

The signal $x(t)$ is some general analog time signal and $X(f)$ is the spectrum of the time signal. Both are complete representations of the signal, x . Obviously, neither one of these representations are useful for DSP since both are analog. In Chapter 3, sampling analog signals in the time domain and reconstructing signals from sampled values was covered in some detail. It is logical to assume that the same thing could be done in the frequency domain using sampled values of the spectrum rather than sampled values of the time signal. The obvious question is how to compute sampled values of the spectrum of a signal.

Sampling the analog signal in time leads to the Discrete-Time Fourier Transform (DTFT):

$$\begin{aligned} \text{DTFT :} \quad X_p(f) &= \sum_{k=0}^{\infty} x(kT_s) e^{-j2\pi k f} \\ T_s &= \text{sampling interval} \end{aligned} \quad (7.2)$$

The time signal, $x(t)$, must be sampled at a sufficient rate to avoid aliasing (at least twice the signal bandwidth).

As discussed in Chapter 3, the spectrum of a sampled signal consists of the original signal spectrum and spectral images spaced apart by F_s . Therefore, $X_p(f)$ is periodic (with period = F_s) but still continuous over frequency which is unsuitable for DSP. In addition, an infinite number of time samples of the signal $x(t)$ are required to compute the DTFT which is also unsuitable for DSP. Taking only a finite number, N , of time samples of $x(t)$, and computing the same finite number of frequency components of $X_p(f)$ evenly spaced over a single period leads to the Discrete Fourier Transform (DFT):

$$\text{DFT :} \quad X_{DFT}(nf_o) = \sum_{k=0}^{N-1} x(kT_s) e^{-j2\pi mk / N} \quad n = 0 \dots N-1$$

$$\text{IDFT :} \quad x(kT_s) = \frac{1}{N} \sum_{n=0}^{N-1} X_{DFT}(nf_o) e^{j2\pi mk / N} \quad k = 0 \dots N-1$$

T_s = sampling interval in time domain

$$f_o = \frac{F_s}{N} = \text{sampling interval in frequency domain} \quad (7.3)$$

The signal $x(kT_s)$ is created by taking N samples of the original signal $x(t)$ over some finite duration of time, D . The signal $X_{DFT}(nf_o)$ is the DFT of the sampled time signal, $x(kT_s)$. Equation 7.3 shows the DFT can be computed by taking the N samples of $x(t)$, multiplying each sample by a weighted exponential function, and then summing the results. The DFT will have N values spaced apart in the frequency domain by $f_o = F_s/N$. The following important result will be simply stated here without proof.

If $x(t)$ is sampled sufficiently fast for a long enough duration of time, D , then

$$X_{DFT}(nf_o) \approx F_s X(f) \big|_{f=nf_o} \quad (7.4)$$

where

- D = the duration of time over which $x(t)$ is sampled
- N = the number of samples of $x(t)$
- T_s = the sampling interval
- $F_s = 1/T_s$ = the sampling rate or frequency
- $f_o = F_s/N = 1/D$ = the resolution or spectral spacing of $X_{DFT}(nf_o)$

DFT Frequencies: N frequencies symmetric about 0 Hz spaced by f_o

This is an amazing result: an estimate of the spectrum of the signal can be generated from a finite number of samples of the time signal! N samples of the time signal result in N samples of the signal spectrum spaced apart by $f_o = F_s/N$. The accuracy of the DFT depends on two factors: the sampling rate chosen for $x(t)$, and the duration of time over which $x(t)$ is sampled. The important factors to stress are:

- If $x(t)$ is not sampled sufficiently fast, aliasing will occur causing the DFT to be a poor match to the original signal spectrum.
- If $x(t)$ is not sampled for a long enough duration of time, the resolution of the DFT will be poor resulting in a poor estimate of the frequency content of the signal.
- An increase in either sampling frequency or duration will increase the size of the DFT which in turn increases computational complexity and cost.

Example 7.1: Duration, D , Size, N , and Discrete Frequencies for DFT

The spectrum of a signal with a bandwidth of 4 kHz is to be estimated using a DFT. The desired resolution for the spectrum is 10 Hz. Find the duration of time, D , over which the signal must be sampled and the minimum size, N , required for the DFT. Using the minimum size, N , for the DFT, determine the discrete frequencies for the DFT.

Solution

The desired resolution for the DFT is $f_o = 10\text{Hz}$. Since $f_o = 1/D$, it follows that $D = 0.1$ seconds. Therefore, the input signal must be sampled for a duration of 0.1 seconds to achieve the specified spectral resolution of 10 Hz.

Since the input signal has a bandwidth of 4 kHz, the minimum sampling rate for the signal to avoid aliasing is $F_s = 8$ kHz. The minimum size of the DFT would be

$$N = F_s * D = 8000 \frac{\text{samples}}{\text{second}} \times 0.1 \text{seconds} = 800 \text{samples}$$

Using this minimum size of $N = 800$, the DFT will produce a spectral estimate at 800 discrete frequencies approximately centered at 0 Hz and spaced apart by $f_0 = 10\text{Hz}$. The discrete frequencies (or frequency bins) would be $-400, -390, -380, \dots, 0, 10, 20, \dots, 390\text{ Hz}$.

The following set of examples illustrate the effect of the sampling rate and the duration of time over which $x(t)$ is sampled on the accuracy of the DFT.

Example 7.2: Effect of Sampling Rate and Duration on Resolution and DFT Frequencies

The time signal for this example is a 2.5 kHz sine wave with an amplitude of one. The spectrum of this signal consists of only two frequency components located at $\pm 2.5\text{ kHz}$. Several choices are explored for the sampling frequency, F_s , and time duration, D . These choices are summarized in Table 7.1 along with the resulting number of samples, N , of the signal, the resolution of the DFT, and the DFT frequencies.

Table 7.1: Parameters for Example 7.2

Sampling Frequency, F_s	Duration, D	Number of Samples, N	Resolution, f_0	DFT Frequencies kHz
10 kHz	0.4 ms	4	2.5 kHz	$-5, -2.5, 0, 2.5$
10 kHz	2.0 ms	20	0.5 kHz	$-5, -4.5, \dots, 0, 0.5, \dots, 4.5$
4 kHz	2.0 ms	8	0.5 kHz	$-2, -1.5, \dots, 0, 0.5, 1, 1.5$
10 kHz	0.5 ms	5	2.0 kHz	$-4, -2, 0, 2, 4$
10 kHz	2.5 ms	25	0.4 kHz	$-4.8, -4.4, \dots, -4, 0, 0.4, \dots, 4.8$

A portion of an m-file created for this example is shown in Figure 7.1.

```

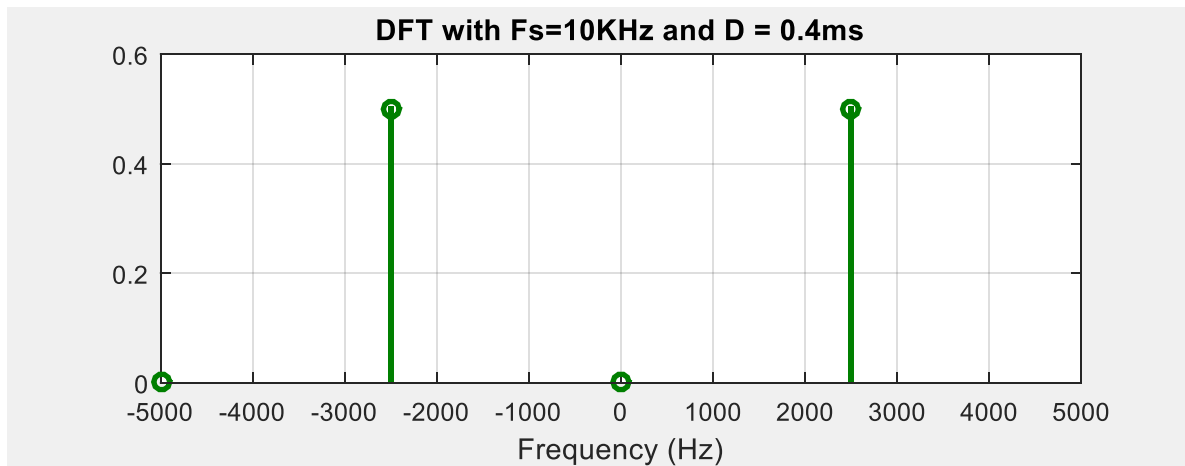
Fs=10000; D=0.4e-3; N=Fs*D;
t=0:1/Fs:(N-1)*1/Fs;           % Set up Sampling Times
x=sin(2*pi*2500*t);             % Sample the 2.5 kHz sine wave
Xf=fft(x);Xf=fftshift(Xf);      % Compute then center the DFT
fo=1/D;                         % Compute resolution
f=-floor(N/2)*fo:fo:(ceil(N/2)-1)*fo; % Set up DFT Frequencies
stem(f,1/N*abs(Xf));
title('DFT with Fs=10kHz and D = 0.4ms');grid

Fs=10000; D=2e-3; N=Fs*D;
t=0:1/Fs:(N-1)*1/Fs;           % Set up Sampling Times
x=sin(2*pi*2500*t);             % Sample the 2.5 kHz sine wave
Xf=fft(x);Xf=fftshift(Xf);      % Compute then center the DFT
fo=1/D;                         % Compute resolution
f=-floor(N/2)*fo:fo:(ceil(N/2)-1)*fo; % Set up DFT Frequencies
figure
stem(f,1/N*abs(Xf));
title('DFT with Fs=10kHz and D = 2ms');grid

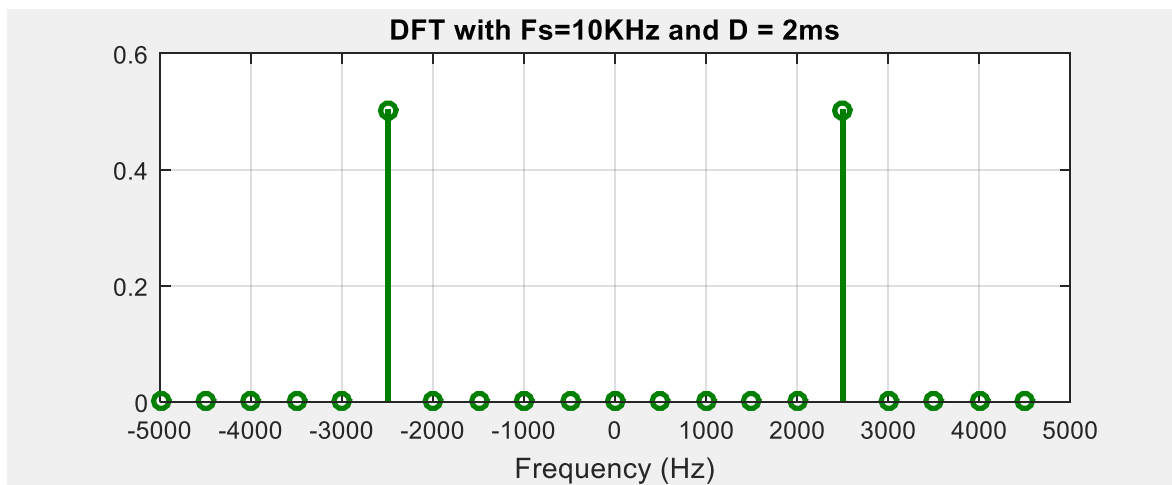
```

Figure 7.1: Sample of MATLAB® Code for Example 7.2

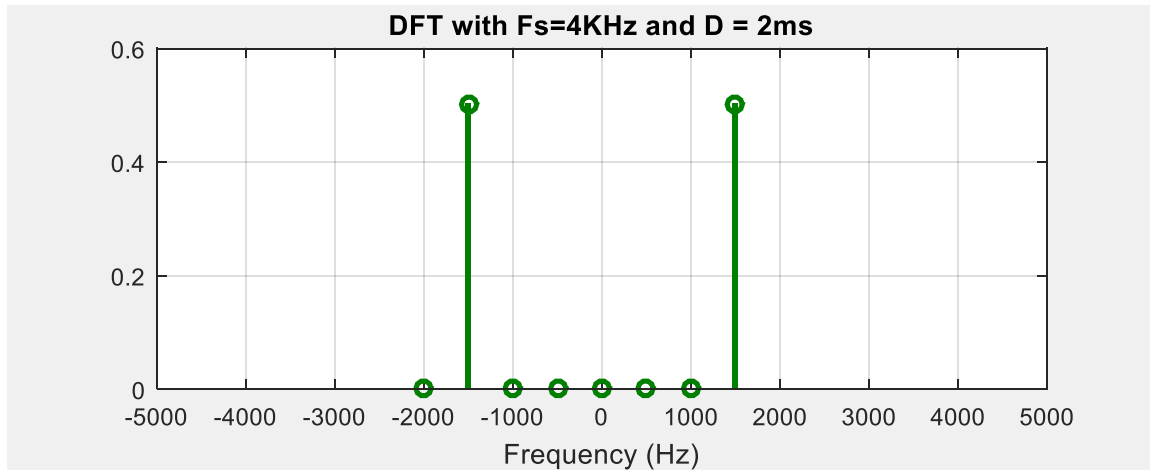
The results are shown in Figure 7.2. The first two plots (Figure 7.2 (a), (b)) show the expected spectrum for a 2.5 kHz sine wave; that is, frequency components at ± 2500 Hz. The second DFT has more frequency bins than the first DFT but all the frequency bins except ± 2500 Hz are empty. Recall that the number of discrete frequencies or frequency bins is exactly equal to the number of time samples, N , taken of the signal $x(t)$.



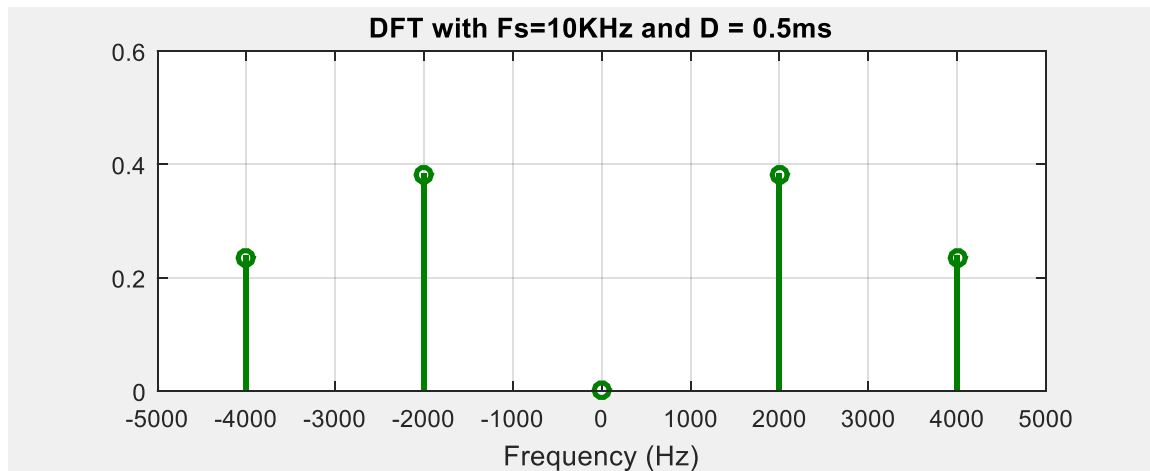
(a) Sufficient Sampling Rate and Duration



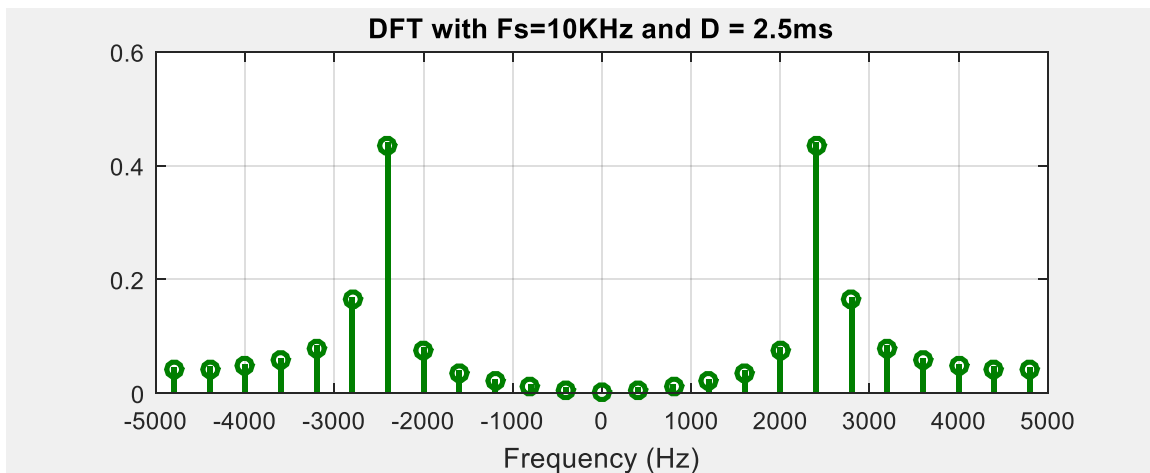
(b) Sufficient Sampling Rate with Longer Duration (more bins)



(c) Aliasing Because Sampling Rate is Too Low



(d) No Bin at 2500 Hz so Spectrum Spreads into Closest Bins (Leakage)



(e) No Bin at 2500 Hz – Longer Duration Improves Resolution and Reduces Leakage

Figure 7.2: DFT for Example 7.2

Challenge Question 7.1

The third DFT in Figure 7.2c shows two frequency components at ± 1500 Hz. Explain this.
Hint: consider the sampling frequency chosen for this case.

The last two DFTs in Figure 7.2 (d), (e) illustrate a problem that occurs with the choice of time duration, D . With periodic functions, ideally the duration, D , should be an integer multiple of the period of the waveform. A 2.5 kHz sine wave has a period of 0.4 ms. In the first two cases for this example, the sampling frequency exceeded the Nyquist rate and the time duration, D , was an integer multiple of the period (0.4 ms and 2.0 ms). Therefore, the DFT in these cases accurately reflected the spectrum of the waveform. In the fourth and fifth cases, the time durations were chosen to be 0.5 ms and 2.5 ms respectively. Neither of these is an integer multiple of the period, 0.4 ms. Go back to Table 7.1 and examine the discrete frequencies (bins) for the last two cases. Notice that ± 2.5 kHz is not an available frequency in either case. Therefore, the DFT results in a split among available bins surrounding ± 2.5 kHz. The term for this phenomenon is **leakage**. Notice that the spectrum in the fifth case is closer to ideal than the in the fourth case. Increasing the time duration improved the resolution so there are frequency bins closer to ± 2.5 kHz. For periodic waveforms, sampling over just a single period works just fine if one knows the period exactly. Otherwise, it is better to sample over a much longer period of time to tighten resolution and improve accuracy of the spectrum.

Example 7.3: Effect of Sampling Rate on Accuracy of Sampled Spectrum (DFT)

In this example, the effect of sampling rate on the DFT is explored. The time signal is chosen to be a decaying exponential. The signal and its spectrum are given by:

$$x(t) = te^{-t} \quad \Leftrightarrow \quad X(f) = \frac{1}{(1 + j2\pi f)^2}$$

The signal and its spectrum are shown in Figure 7.3.

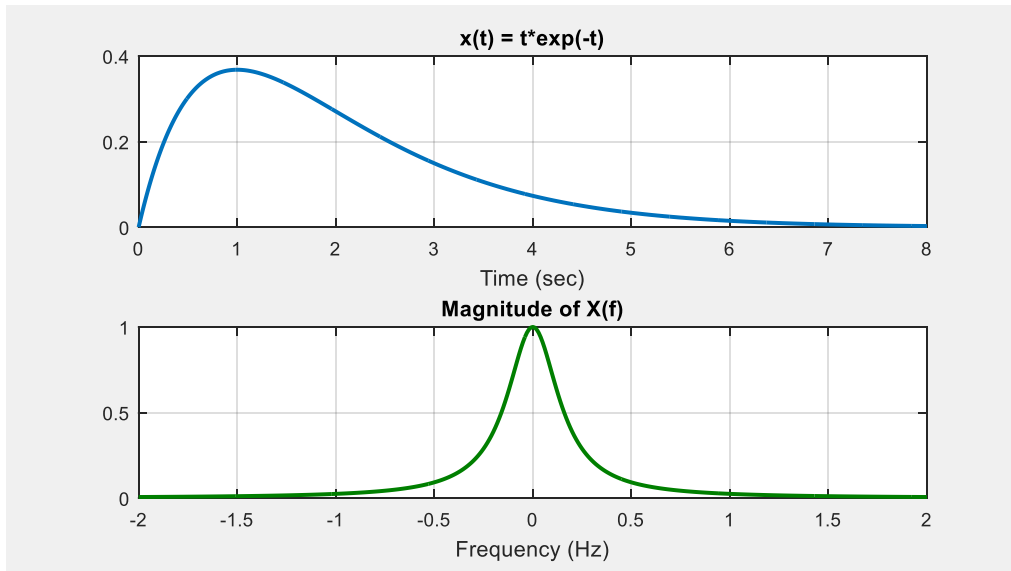


Figure 7.3: Signal and Signal Spectrum for Example 7.3

In order to compute a DFT, the time signal $x(t)$ must be sampled over some duration of time. According to the Nyquist Theorem, the sampling frequency should be at least twice the bandwidth of the signal. As seen from the plot of the spectrum in Figure 7.3, the spectral components above 1.0 Hz are relatively small. In order to see the effects of sampling rate, two different sampling frequencies are chosen: 2 Hz and 16 Hz. The duration of time over which the signal is sampled is chosen to be 8 seconds. Table 7.2 shows some of the key parameters for this example.

Table 7.2: Parameters for Example 7.3

Sampling Frequency F_s	Duration D	Number of Samples N	Resolution f_0	Frequency Range
2 Hz	8 sec.	16	1/8 Hz	–2 Hz to 1.875 Hz
16 Hz	8 sec.	128	1/8 Hz	–8 Hz to 7.875 Hz

Figure 7.4 illustrates the effect of increasing the sampling rate on the accuracy of the DFT. At the lower sampling rate of 2 Hz, the accuracy of the DFT begins to degrade around 0.25 Hz due to aliasing. A sampling rate of 16 Hz results in a very accurate sampled spectrum. The DFT in the second case actually ranges from –8 to 7.875 Hz but was truncated to ± 2 Hz to allow a nice comparison with the lower sampling rate over the frequency range of interest.

A general rule of thumb for choosing the sampling frequency is to choose F_s to be eight times the highest frequency of interest in the signal. In Example 7.3, a sampling frequency of 2 Hz results in an accurate (within 5%) representation of the signal spectrum up to 0.25 Hz ($F_s/8$) whereas the sampling frequency of 16 Hz results in an accurate representation of the signal spectrum up to 2 Hz ($F_s/8$). The signal should also be filtered prior to sampling so that it is indeed band-limited to the highest frequency of interest; otherwise, aliasing will occur.

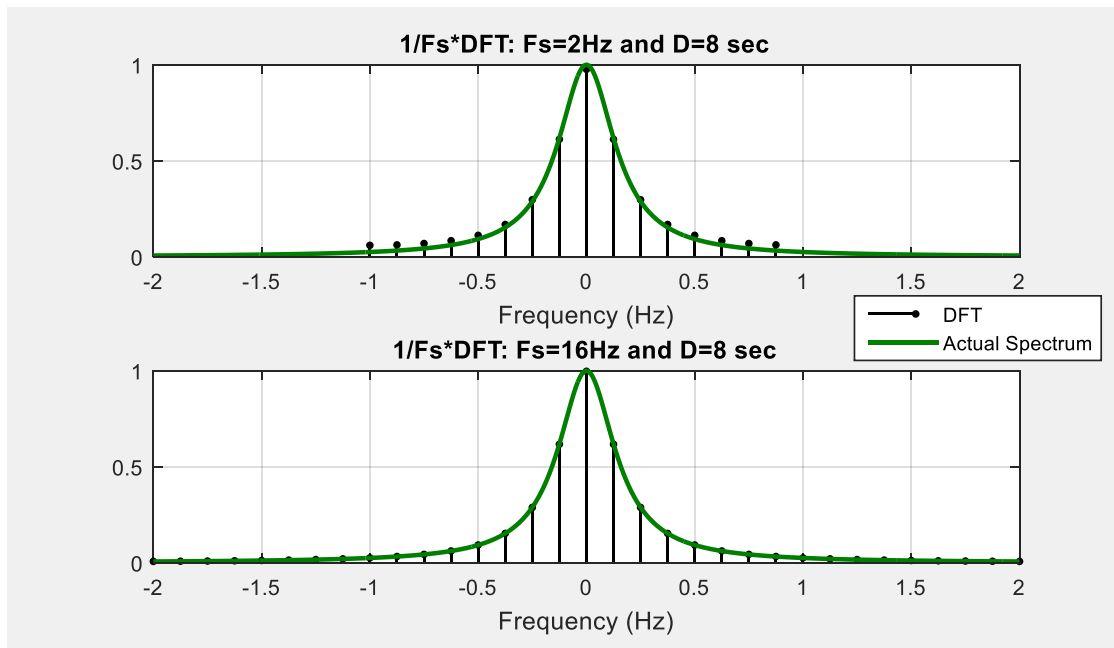


Figure 7.4: DFT for Example 7.3

MATLAB Code for Figure 7.4:

```
% Fs = 2 Hz, D=8 sec, N=16 samples
Fs=2; D=8; N=16;
% Sample the analog signal
t=0:1/Fs:(N-1)*1/Fs; x=t.*exp(-t);
Xdft=fft(x,N); Xdft=fftshift(Xdft);
fo=1/D; %Spectral Resolution
fd= -floor(N/2)*fo:fo:(ceil(N/2)-1)*fo; % Frequency Range
subplot(2,1,1); stem(fd,1/Fs*abs(Xdft));
title('1/Fs*DFT: Fs=2Hz and D=8 sec'); grid; hold on;
plot(f,abs(Xf)); hold off;

% Fs=16 Hz; D= 8 sec; N=128 samples
Fs=16; D=8; N=128;
t=0:1/Fs:(N-1)*1/Fs; x=t.*exp(-t);
Xdft=fft(x,N); Xdft=fftshift(Xdft);
fo=1/D;
fd= -floor(N/2)*fo:fo:(ceil(N/2)-1)*fo;
subplot(2,1,2); stem(fd,1/Fs*abs(Xdft));
title('1/Fs*DFT: Fs=16Hz and D=8 sec'); grid;
hold on; plot(f,abs(Xf)); hold off;
```

Example 7.4: Effect of Time Duration, D, on Accuracy of Sampled Spectrum (DFT)

In this example, the effect of duration of time over which the signal is sampled is explored. The time signal, $x(t)$, is chosen to be a pulse of amplitude one with a duration of 1 second. The spectrum of this signal is a weighted sinc function.

$$x(t) = \text{rect}\left(t - \frac{1}{2}\right) \Leftrightarrow X(f) = e^{-j\pi f} \text{sinc}(f)$$

The pulse signal and the signal spectrum are shown in Figure 7.5. The MATLAB Code for Figure 7.5 is:

```
t1=0:0.01:1; t2=1.01:0.01:2; t=[t1 t2];
x1=ones(1,length(t1)); x2=zeros(1,length(t2)); x=[x1 x2];
f=-5:0.01:5; Xf=abs(exp(-j*pi*f).*sinc(f));
subplot(2,1,1); plot(t,x);
title('Pulse Signal'); xlabel('time (sec)'); grid
subplot(2,1,2); plot(f,Xf);
title('Magnitude Spectrum of Pulse Signal');
xlabel('Frequency (Hz)'); grid
```

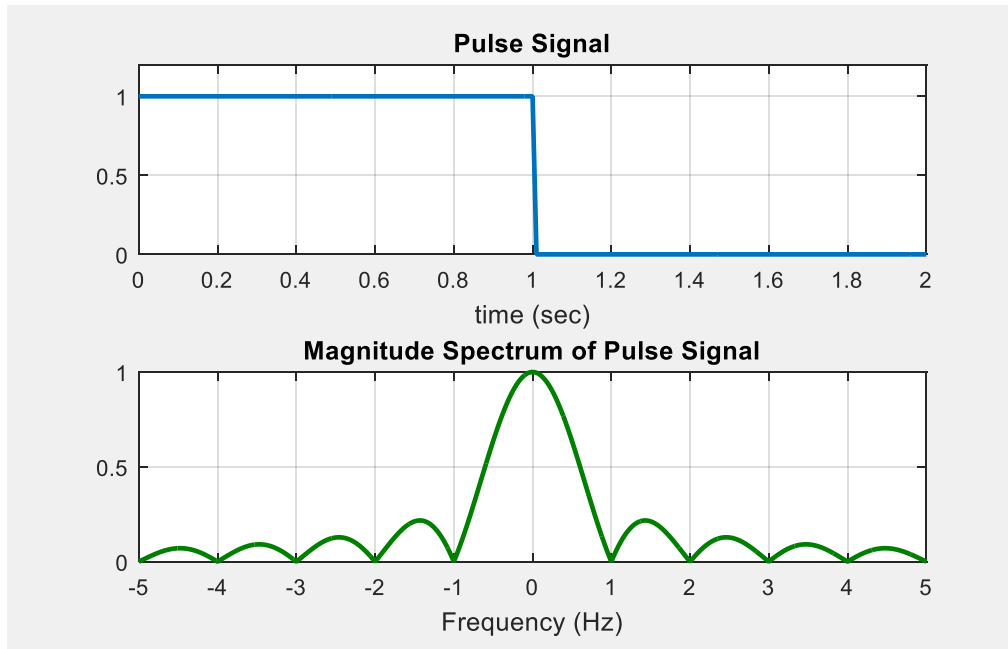


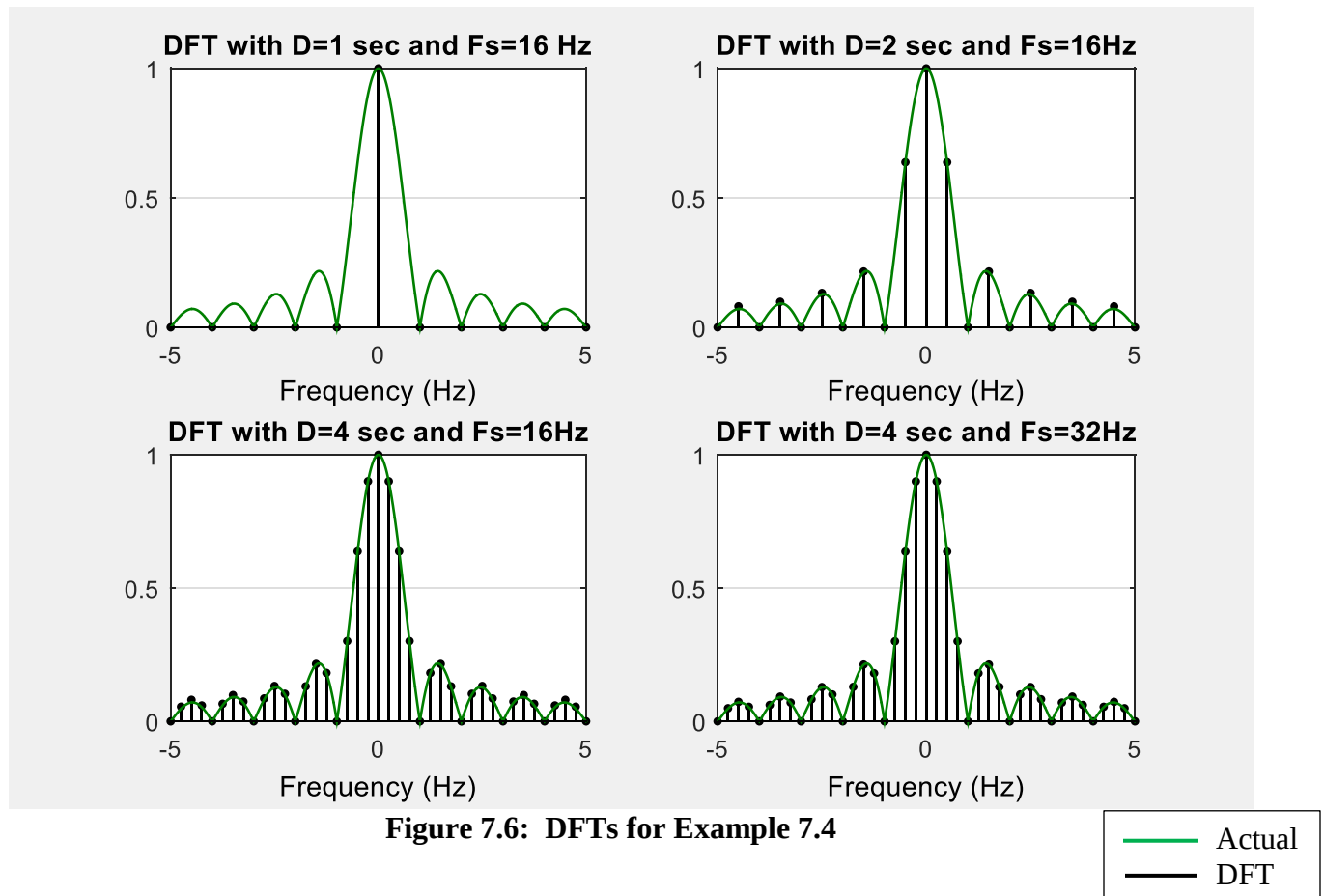
Figure 7.5: Signal and Spectrum for Example 7.4

In order to compute a DFT, the time signal $x(t)$ must be sampled over some duration of time. It is necessary to choose a sampling rate, F_s , and a duration of time, D . Several choices are explored in this example. To accurately represent the spectrum up to 2 Hz, the sampling frequency should be chosen at 16 Hz (8×2 Hz). An accurate representation of the spectrum up to 4 Hz would require a sampling frequency of 32 Hz (8×4 Hz). Since the signal will not be pre-filtered in this example, some aliasing will occur. The duration of time, D , is initially chosen to be 1 sec then increased to 2 sec and then to 4 sec to illustrate the effect on resolution. The four cases to be investigated are summarized in Table 7.3.

Table 7.3: Parameters for Example 7.4

Sampling Frequency, F_s	Duration, D	Number of Samples, N	Resolution, f_0	DFT Frequencies kHz
16 Hz	1 sec	16	1 Hz	$-8, -7, \dots, 0, 1, 2, \dots, 7$
16 Hz	2 sec	32	0.5 Hz	$-8, -7.5, \dots, 0, 0.5, 1, \dots, 7.5$
16 Hz	4 sec	64	0.25 Hz	$-8, -7.75, \dots, 0, 0.25, 0.5, \dots, 7.75$
32 Hz	4 sec	128	0.25 Hz	$-16, -15.75, \dots, 0, 0.25, 0.5, \dots, 15.75$

Figure 7.6 clearly illustrates the effect of duration, D , on the resolution or spectral spacing of the DFT. A duration of 1 second provides a resolution of 1 Hz and does not yield a good sampled version of the spectrum of $x(t)$. As D is increased, the resolution improves; that is, the spectrum samples are more closely spaced. The discrepancies between the DFT and the original signal spectrum illustrate the effect of aliasing. A higher sampling rate reduces the aliasing effect and produces a closer match between the DFT and the signal spectrum. Increasing the sampling frequency to 32 Hz improves the accuracy of the DFT between 2 Hz and 4 Hz. Note that although the DFT really extends to $\pm F_s/2$, the spectra in Figure 7.6 were displayed only to ± 5 Hz to avoid crowded plots.



The MATLAB Code (partial) to compute the DFTs and generate Figure 7.6 is:

```

Fs = 16; D = 1; N = Fs*D;
t=0:1/Fs:(N-1)*Fs; x = ones(1,length(t));
Xdft=fft(x,N); Xdft = fftshift(Xdft);
fo = 1/D;
fd = -floor(N/2)*fo:fo:(ceil(N/2) - 1)*fo;
subplot(2,2,1); stem(fd,1/Fs*abs(Xdft));
hold on; plot(f,Xf); hold off;
title('DFT with D=1 sec and Fs=16 Hz');grid

Fs=16; D = 2; N = Fs*D;
t1=0:1/Fs:1-1/Fs; t2=1:1/Fs:(N-1)*Fs;
x = [ones(1,length(t1)) zeros(1,length(t2))];
Xdft=fft(x,N); Xdft = fftshift(Xdft);
fo = 1/D;
fd = -floor(N/2)*fo:fo:(ceil(N/2)- 1)*fo;
subplot(2,2,2);stem(fd,1/Fs*abs(Xdft));
hold on; plot(f,Xf); hold off;
title('DFT with D=2 sec and Fs=16Hz'); grid

```

Summary of Key Points about Discrete Fourier Transforms

1. The DFT is calculated by taking N samples of an analog signal $x(t)$ at a sampling frequency of F_s . These samples are then weighted by exponentials and summed together (more on this later) to calculate the DFT.
2. The DFT has N frequency components (bins) centered at 0 Hz and spaced apart by $f_0 = 1/D = F_s/N$.
3. The sampling frequency, F_s , should be chosen to be at least eight times the highest frequency component of interest in the signal, $x(t)$.
4. The signal $x(t)$ should be filtered to band-limit the signal prior to sampling and computing a DFT.
5. The time duration, D , over which the signal is sampled directly affects the resolution of the DFT. Sampling over a longer period of time tightens the resolution.
6. Periodic signals can be sampled over a single period if the period is known exactly but should be sampled over several periods if the period is not known precisely.
7. If the signal has frequency components that don't coincide with one of the discrete frequencies (bins) for the DFT, then the DFT will show components in the bins closest to the signal's frequencies. This is referred to as leakage, the actual frequency component spreads among adjacent frequency bins. Improved resolution decreases leakage.

Challenge Question 7.2

Figure 7.7 plots the DFT of the signal $x(t) = \sin(2\pi(2000)t) + 2\sin(2\pi(2500)t)$ for several different sampling frequencies and time durations. Which of the DFTs in the plot provide accurate estimates of the spectrum of $x(t)$? For the DFTs which do not accurately reflect the signal spectrum, explain the problem.

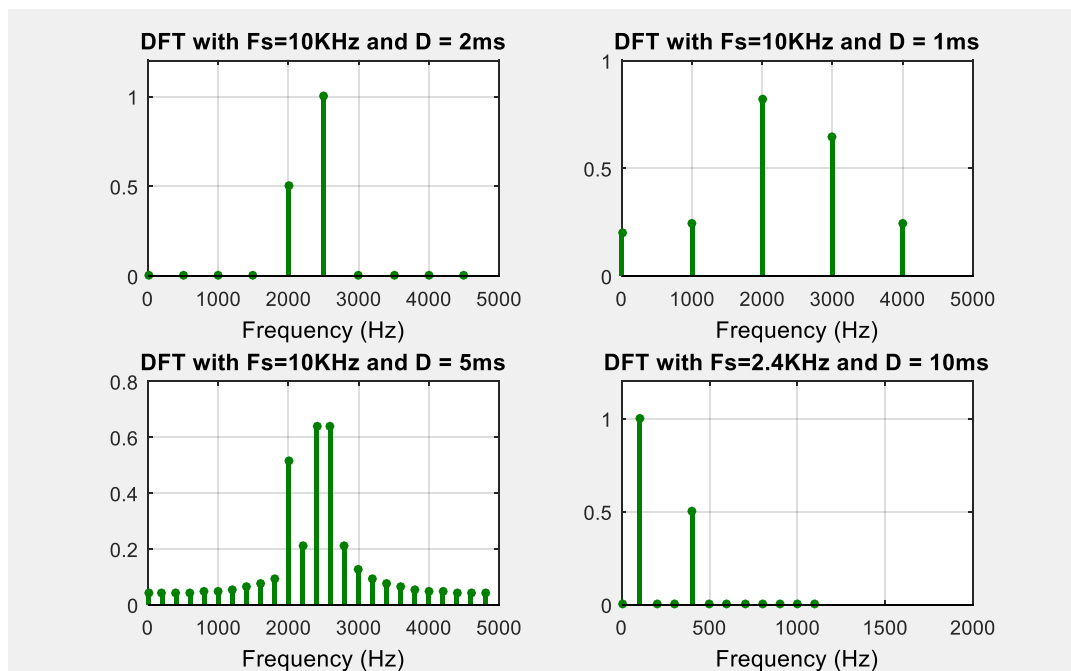


Figure 7.7: Discrete Fourier Transforms for Challenge Question 7.2

7.2 WINDOWING THE DATA TO ENHANCE SPECTRAL ANALYSIS

The effect of leakage was explored in several examples in the previous section. If a sinusoid is sampled for exactly an integer multiple of periods, the DFT will match the spectrum of the sinusoid perfectly yielding frequency components at the frequency of the sinusoid. If the sinusoid is not sampled over an integer number of periods, then the DFT produces frequency components that spread or leak to adjacent frequency bins. The same effect occurs for non-periodic signals. A common method for reducing the effect of leakage is to window the incoming data and calculate the DFT based on the windowed data samples.

The concept of using windowing to design FIR filters was covered in Chapter 5. The same windows introduced in Chapter 5 can be used on the blocks of data collected for spectral analysis. The following example illustrates how windowing the data can reduce the effect of leakage.

Example 7.5: Windowing the Data to Reduce Leakage

Consider an 8 kHz sine wave sampled at 48 kHz. Assume 128 samples of the signal are acquired. Plot the DFT of the input data with no windowing, plot the DFT of the data using a Hamming window, and compare the results.

Solution

The two plots are displayed in Figure 7.8. Since the signal is not being sampled over an integer multiple of periods, leakage will occur. The spectral resolution in this example is $F_s/N = 48000/128 = 375$ Hz. The closest frequency bins to the signal frequency of 8 kHz are 7.875 kHz and 8.25 kHz. The top plot in Figure 7.8 shows the DFT with no windowing. The frequency bins with the highest amplitude are the two bins closest to 8 kHz; however, there is significant spreading or leakage into several other frequency bins. The bottom plot in Figure 7.8 shows the DFT of the windowed data using a Hamming window. Although leakage still occurs, it is confined mainly to the four frequency bins closed to the signal frequency of 8 kHz. The MATLAB code used for this example is listed in Figure 7.9.

There are many different types of windows that can be used for spectral analysis. MATLAB provides sixteen different options using the **window** command. As mentioned in the section on FIR filter design, each window has its own somewhat unique characteristics and will therefore affect the spectral analysis differently from other window functions. None of the windows will eliminate the effect of leakage – the leakage is simply re-distributed among the frequency bins through windowing. The Hamming window used in Example 7.5, widens the main peak while suppressing or attenuating the sidelobe frequencies. This same effect was observed in Chapter 5 for FIR filter design.

In this section, windowing is applied in the time domain by simply multiplying the input data samples by the chosen window function. It is possible (and sometimes more computationally efficient) to compute the DFT for the non-windowed input data samples and then convolve the resulting DFT with the spectrum of the window function. Convolution in the frequency domain is equivalent to windowing in the time domain. This process is discussed in detail in Section 8.3 of Chapter 8.

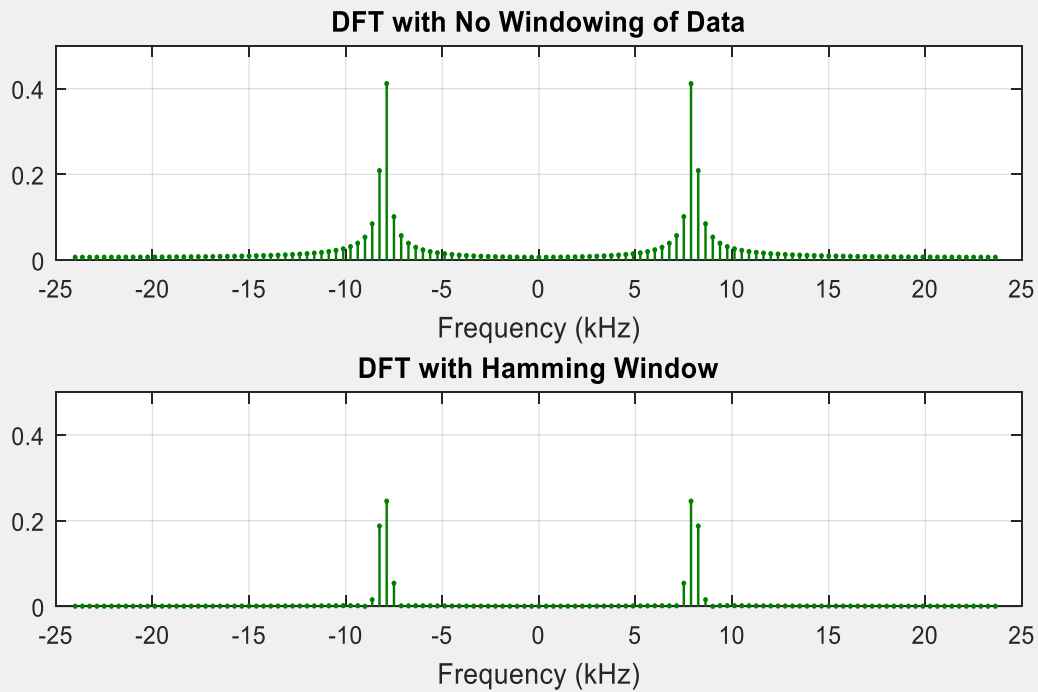


Figure 7.8: DFT with Windowing (Example 7.5)

```

Fs=48000; N=128;
t=0:1/Fs:(N-1)*1/Fs;
x=sin(2*pi*8000*t);
fo=Fs/N;
f=-floor(N/2)*fo:fo:(ceil(N/2)-1)*fo;
% FFT with no windowing of data
Xf=fft(x);Xf=fftshift(Xf);
subplot(2,1,1);stem(f/1000,1/N*abs(Xf));
title('DFT with No Windowing of Data');
xlabel('Frequency (kHz)');grid
% FFT with Hamming Window
x_hamming = x.*window(@hamming,N)';
Xf_hamming = fftshift(fft(x_hamming));
subplot(2,1,2);stem(f/1000,1/N*abs(Xf_hamming));
title('DFT with Hamming Window');
xlabel('Frequency (kHz)');grid

```

Figure 7.9: MATLAB code for Example 7.5

Example 7.6: Windowing the Data to Reduce Leakage

Repeat Example 7.5 for the input signal $x(t) = \sin(2\pi(8000)t) + 0.2\sin(2\pi(9600)t)$.

Solution

The only required change to the MATLAB code for Example 7.5 is to change line 3 in Figure 7.9 to: `x=sin(2*pi*8000*t) + 0.2*sin(2*pi*9600*t);`

The result is shown in Figure 7.10. With no windowing, the 9600 Hz component is not as obvious – it tends to be obscured by the leakage. The Hamming window widens the two main lobes while significantly attenuating the sidelobes which makes the 9600 Hz component much more distinct.

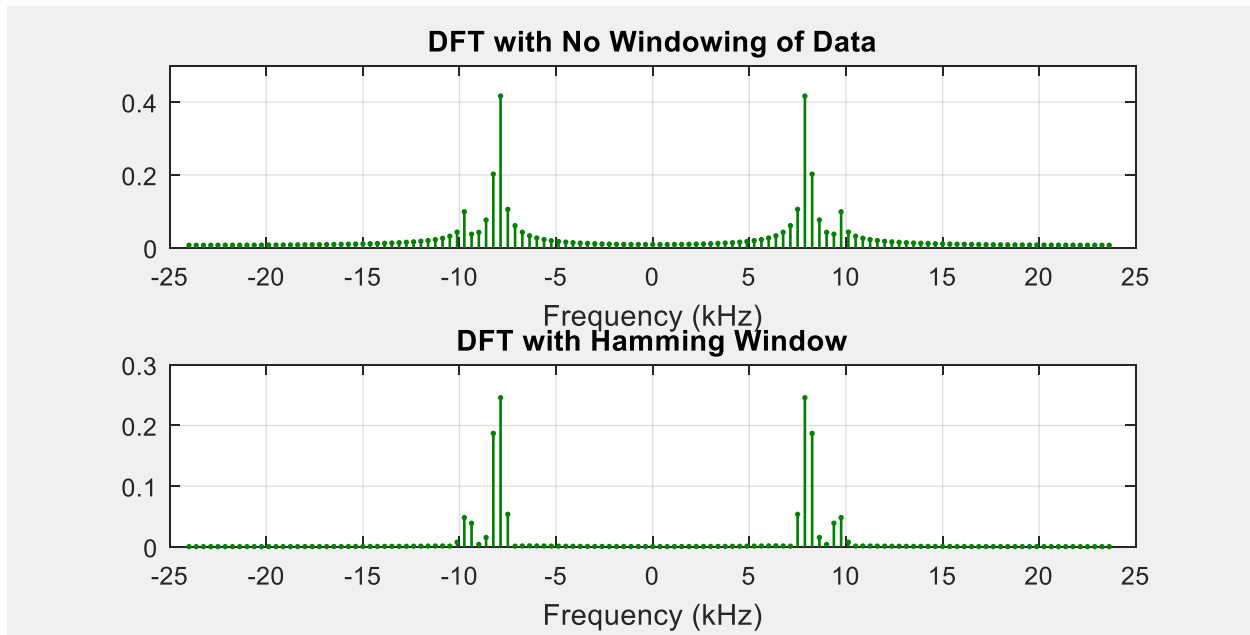


Figure 7.10: DFT with Windowing (Example 7.6)

7.3 FAST FOURIER TRANSFORMS

A fast Fourier transform (FFT) is simply a very efficient algorithm for computing a Discrete Fourier transform (DFT). An FFT takes advantage of the symmetry of the exponential weighting factors in the DFT. The DFT can be re-written as follows:

$$X_{DFT}(nf_o) = \sum_{k=0}^{N-1} x(kT_s) e^{-j2\pi mk/N} = \sum_{k=0}^{N-1} x(kT_s) (W_N)^{nk}$$

$$W_N = e^{-j2\pi/N} = \cos(2\pi/N) - j\sin(2\pi/N) \quad \text{Twiddle Factor} \quad (7.5)$$

Notice that the twiddle factor, W_N , depends only on N which is the size of the FFT or equivalently the number of time samples taken of the signal $x(t)$. W_N has no dependency whatsoever on the time signal $x(t)$ and can therefore be computed ahead of time and stored in memory.

A straight N -point DFT would be calculated by taking N samples of an analog signal, multiplying each sample by the twiddle factor raised to some power, and then summing the results as shown in Equation 7.6.

$$[X_{DFT}(0) \ X_{DFT}(1) \dots X_{DFT}(N-1)] = [x(0) \ x(1) \dots x(N-1)] \begin{bmatrix} (W_N)^0 & (W_N)^0 & (W_N)^0 & \dots & (W_N)^0 \\ (W_N)^0 & (W_N)^1 & (W_N)^2 & \dots & (W_N)^{N-1} \\ (W_N)^0 & (W_N)^2 & (W_N)^4 & \dots & (W_N)^{2(N-1)} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ (W_N)^0 & (W_N)^{N-1} & (W_N)^{2(N-1)} & \dots & (W_N)^{(N-1)(N-1)} \end{bmatrix} \quad (7.6)$$

Notice that the entire weighting matrix consists of the twiddle factor raised to various powers and this matrix could be pre-computed and stored in memory since there is no dependency on the signal, $x(t)$. So, computing the DFT using Equation 7.6 requires taking N samples of the signal $x(t)$ and then performing N^2 complex multiplications. An FFT algorithm reduces the required number of multiplications by taking advantage of the symmetry properties of the twiddle factor when raised to various powers.

Symmetry Properties of the Twiddle Factor

The twiddle factor is a complex number with a magnitude of one that depends only on N , the size of the FFT. Table 7.4 lists values of the twiddle factor for several values of N .

Table 7.4: Twiddle Factors for various N values

SIZE OF FFT, N	TWIDDLE FACTOR: $W_N = e^{-j2\pi/N}$
2	$-1 = 1 \angle -180^\circ$
4	$-j = 1 \angle -90^\circ$
8	$0.707 - j 0.707 = 1 \angle -45^\circ$
256	$0.999698818 - j 0.024541228 = 1 \angle -1.40625^\circ$

Graphically, the twiddle factor can be determined by drawing a circle of radius one, then starting at 0° , rotating clockwise by $360^\circ/N$ to arrive at the twiddle factor.

The key symmetry properties of the twiddle factor are:

1. Periodic in N: $(W_N)^{n+N} = (W_N)^n$
2. 180° Symmetry: $(W_N)^{n+N/2} = -(W_N)^n$
3. Even Power Reduction: $(W_N)^2 = (W_{N/2})$

The first property is important because it means an N-point DFT has only N exponential weighting factors; that is, a 4-point DFT has only four possible weighting factors, an 8-point DFT only has eight possible weighting factors, and so on.

The second property shows that half of the weighting factors are simply the negative of the other half and therefore could be discarded (not stored in memory).

The third property allows a large N-point DFT to be broken down into several simple 2-point DFTs – the heart of an FFT algorithm. To illustrate the first two properties, consider a 4-point DFT which would be computed as follows:

$$\begin{aligned}
 [X_{DFT}(0) \quad X_{DFT}(1) \quad X_{DFT}(2) \quad X_{DFT}(3)] = \\
 [x(0) \quad x(1) \quad x(2) \quad x(3)] & \begin{bmatrix} (-j)^0 & (-j)^0 & (-j)^0 & (-j)^0 \\ (-j)^0 & (-j)^1 & (-j)^2 & (-j)^3 \\ (-j)^0 & (-j)^2 & (-j)^4 & (-j)^6 \\ (-j)^0 & (-j)^3 & (-j)^6 & (-j)^9 \end{bmatrix} \\
 [x(0) \quad x(1) \quad x(2) \quad x(3)] & \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{bmatrix}
 \end{aligned} \tag{7.7}$$

Notice that even though the twiddle factor is raised to many different powers in the weighting matrix, the resulting matrix only has 4 unique values (i.e., the size of the DFT) due to the periodicity in N property. These four values are 1, j, -1, and -j. Notice also that half of the values are simply the negative of the other half due to the 180° symmetry property. The two diagrams in Figure 7.11 graphically illustrate the exponential weighting factors for a 4-point DFT and an 8-point DFT.

As shown in Figure 7.11, the exponential weighting factors for an N-point FFT are simply N complex numbers evenly distributed around the unit circle and spaced by the twiddle factor.

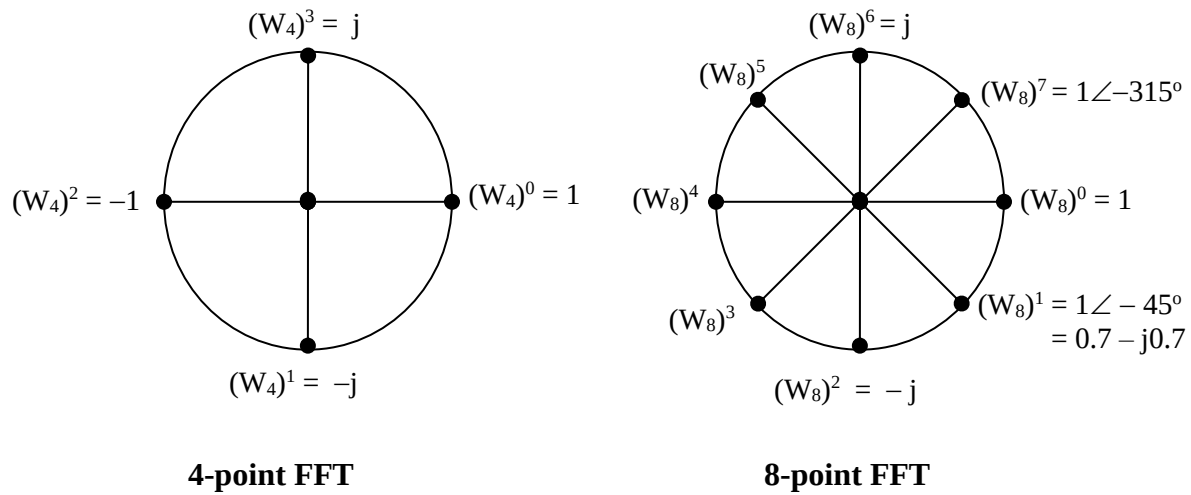


Figure 7.11: Exponential Weighting Factors for 4-point and 8-point DFTs

Radix-2 Decimation in Time FFT

There are several different types of FFT algorithms for efficient computation of a DFT. This text will only cover the Radix-2 decimation in time (DIT) algorithm. The bibliography lists several sources which cover other types of FFT algorithms. The Radix-2 DIT algorithm assumes that N , the number of time samples or equivalently the number of frequency bins, is a power of 2. The simplest DFT to compute is a 2-point DFT:

$$\begin{aligned} X_{DFT}(0) &= x(0) + x(1) \\ X_{DFT}(1) &= x(0) - x(1) \end{aligned} \quad (7.8)$$

A radix-2 FFT algorithm simply breaks a DFT down into several 2-point DFTs by exploiting the symmetry properties of the twiddle factor, particularly the even-power reduction property. First, break the N -point DFT into two $N/2$ -point DFTs by separating the even time samples ($x(0)$, $x(2)$, ...) from the odd time samples ($x(1)$, $x(3)$, ...) as follows:

$$X_{DFT}(nf_o) = \sum_{k=0}^{N-1} x(k)(W_N)^{nk} = \sum_{k=0}^{N/2-1} x(2k)(W_N)^{(2k)n} + \sum_{k=0}^{N/2-1} x(2k+1)(W_N)^{(2k+1)n} \quad (7.9)$$

Applying the even power reduction property of the twiddle factor, $(W_N)^2 = (W_{N/2})$, to equation 7.9 yields:

$$X_{DFT}(nf_o) = \sum_{k=0}^{N/2-1} x(2k)(W_{N/2})^{kn} + (W_N)^n \cdot \sum_{k=0}^{N/2-1} x(2k+1)(W_N)^{kn} \quad n = 0, 1, \dots, N-1 \quad (7.10)$$

The N-point DFT has now been broken down into two N/2-point DFTs. In other words,

1. Use the even samples $[x(0), x(2), \dots, x(N-2)]$ to compute an N/2-point DFT $\rightarrow X^{\text{even}}(nf_o)$.
2. Use the odd samples $[x(1), x(3), \dots, x(N-1)]$ to compute an N/2-point DFT $\rightarrow X^{\text{odd}}(nf_o)$.
3. Then combine the two N/2-point DFTs to create the N-point DFT as follows:

$$X_{\text{DFT}}(nf_o) = X^{\text{even}}(nf_o) + (W_N)^n \cdot X^{\text{odd}}(nf_o) \quad n = 0, 1, \dots, \frac{N}{2} - 1$$

$$X_{\text{DFT}}((n + N/2)f_o) = X^{\text{even}}(nf_o) - (W_N)^n \cdot X^{\text{odd}}(nf_o) \quad n = 0, 1, \dots, \frac{N}{2} - 1$$

(7.11)

This same procedure could be applied to break each of the N/2-point DFTs in half which would yield four N/4-point DFTs. Assuming N is a power of two, continuing this procedure would eventually result in a set of 2-point DFTs.

Figure 7.12 illustrates the basic principles of a Radix-2 Decimation in Time FFT for N = 16. The Radix-2 decimation in time FFT algorithm would simply begin at the bottom of the tree in Figure 7.12 and compute eight 2-point DFTs. These 2-point DFTs would be combined to create four 4-point DFTs using appropriate exponential weighting factors. The 4-point DFTs would then be combined to create two 8-point DFTs again using appropriate exponential weighting factors. Finally, the 8-point DFTs are combined to create the 16-point DFT.

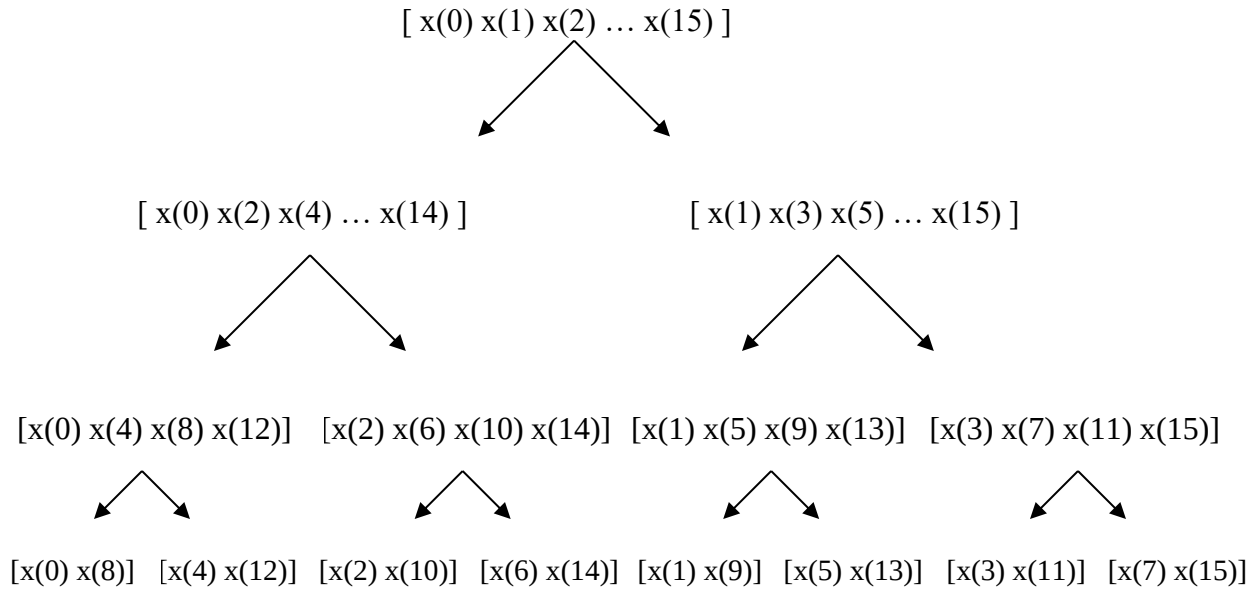


Figure 7.12: Breaking Up Time Samples for FFT

The process of re-combining the smaller DFTs to create the larger DFT is often illustrated using a butterfly diagram. Figure 7.13 illustrates the basic butterfly notation and associated computations.

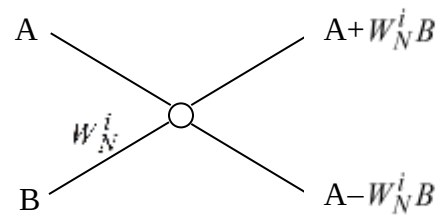


Figure 7.13: Butterfly Notation

A butterfly diagram for an 8-point DFT is shown in Figure 7.14. Starting on the left-hand side of Figure 7.14, time samples are paired together and four 2-point DFTs are computed. In the middle section of the figure, the four 2-point DFTs are combined with appropriate weighting factors to compute two 4-point DFTs. Finally, in the last section of the figure, the 4-point DFTs are combined with appropriate weighting factors to compute a single 8-point DFT.

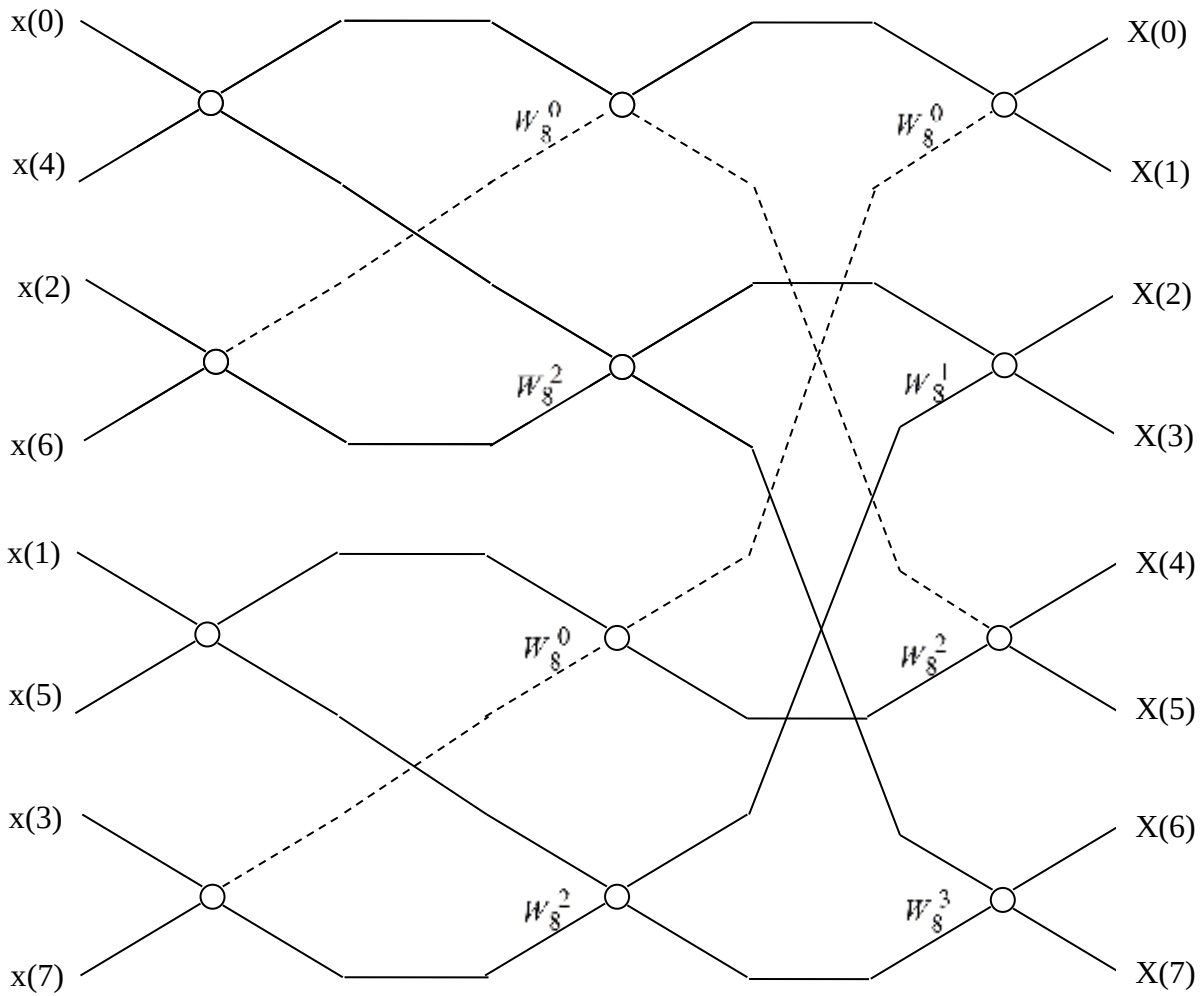
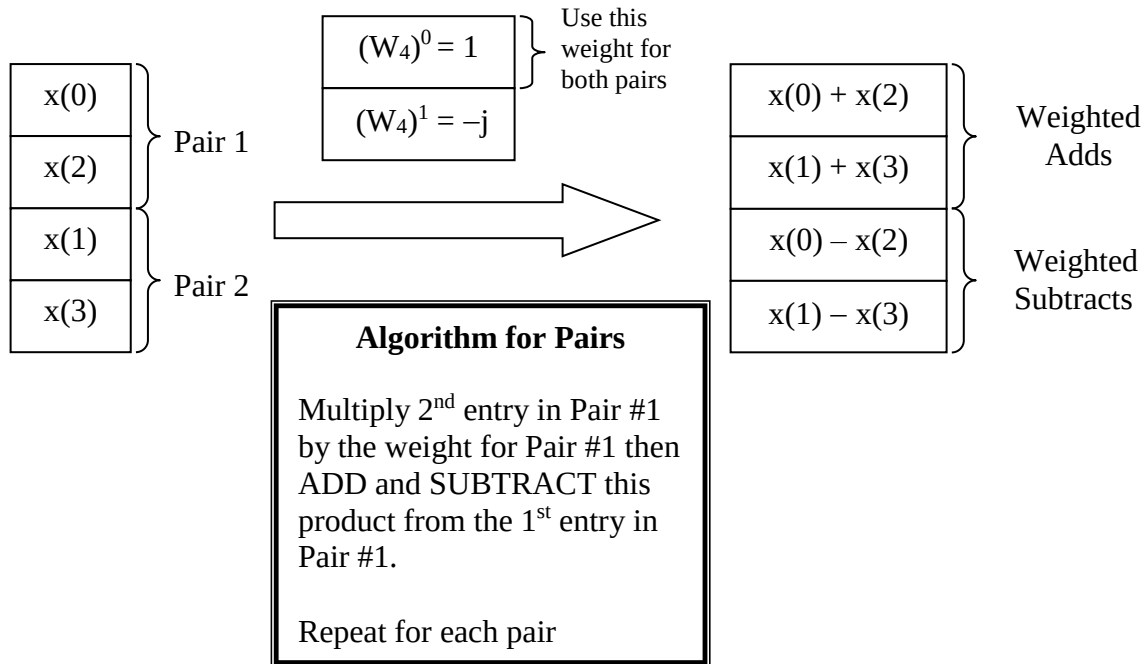


Figure 7.14: Butterfly Diagram for 8-point DFT

Figures 7.15 and 7.16 present an alternative view of a 4-point and 8-point FFT using flow graphs instead of the butterfly diagram.

Flow Graphs for a 4-point FFT

Level 1: 2-pt FFTs



Level 2: 4-pt FFT

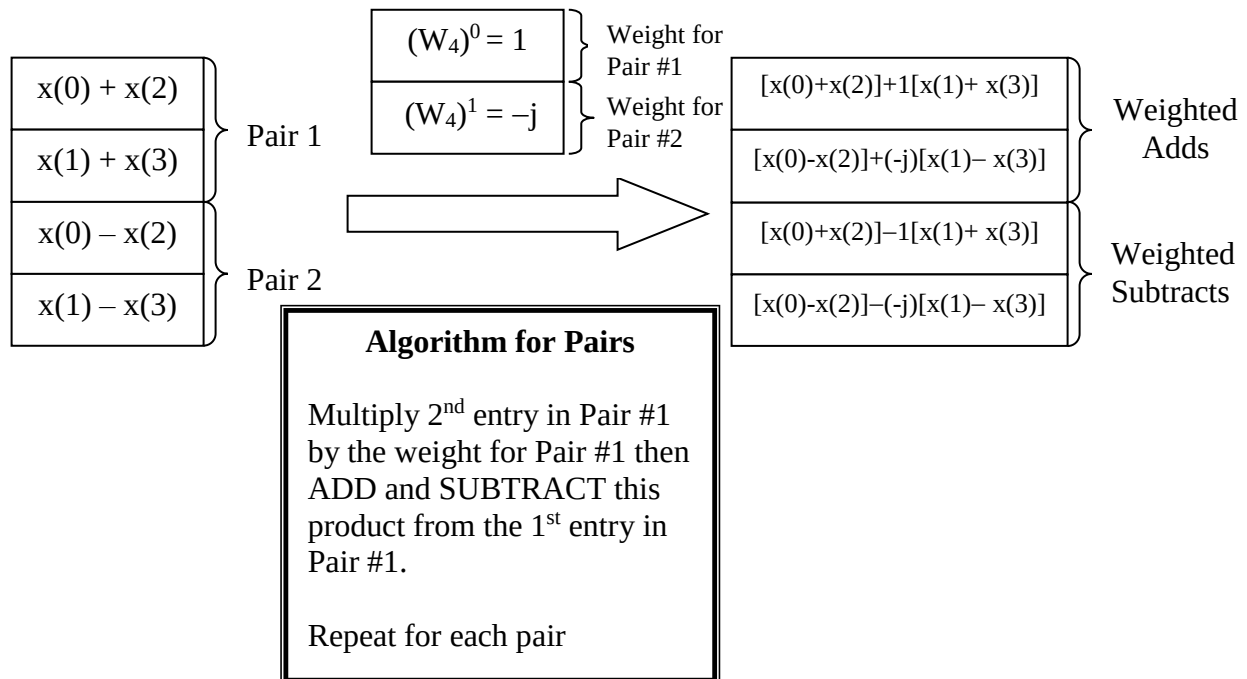
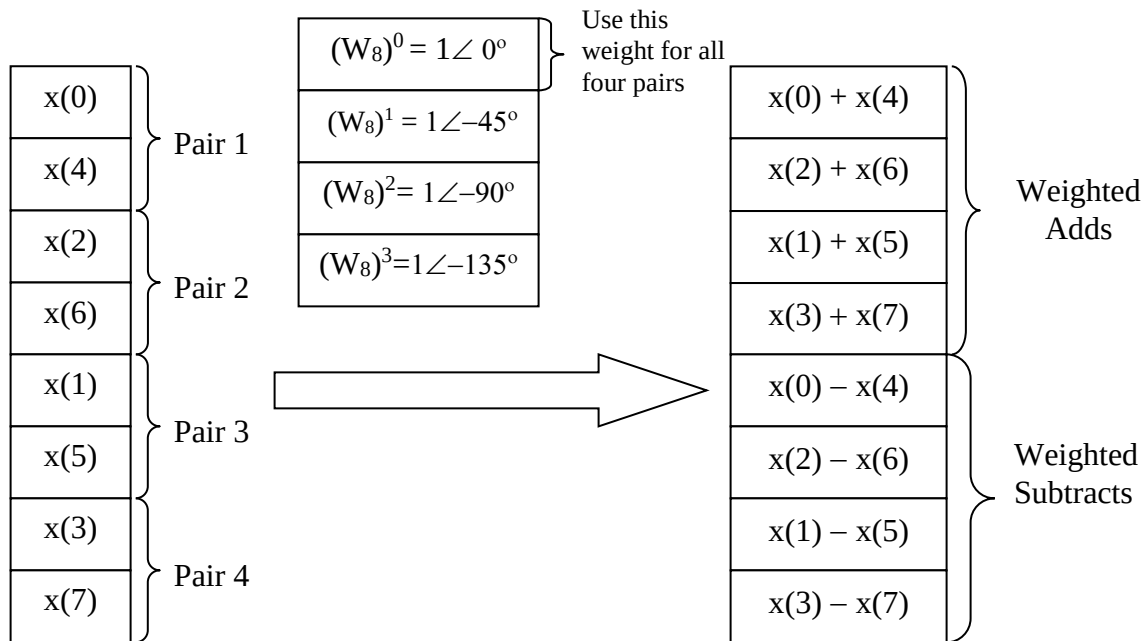


Figure 7.15: Flow Graphs for a 4-point FFT

Flow Graphs for 8-point FFT (Level 1)

Level 1: Four 2-pt FFTs



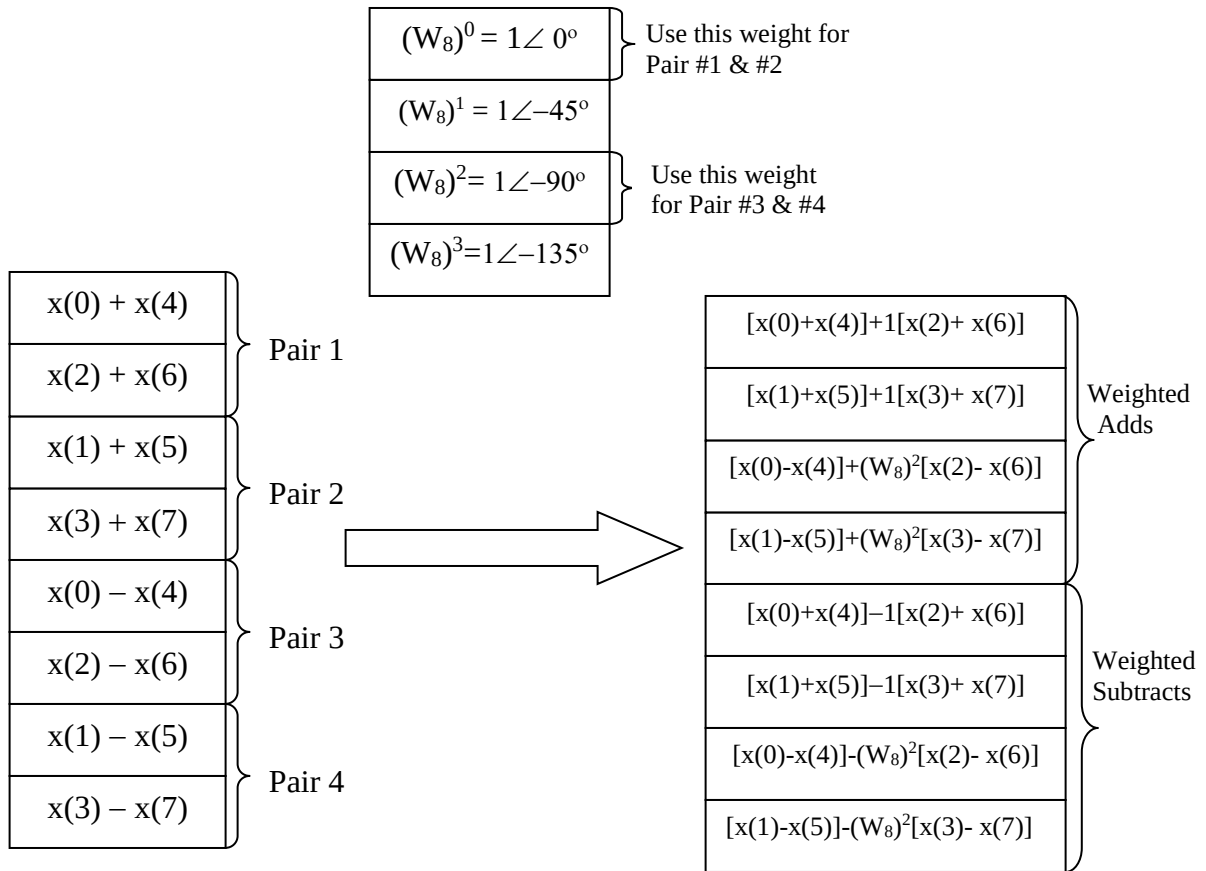
Algorithm for Pairs

Multiply 2nd entry in Pair #1 by the weight for Pair #1 then ADD and SUBTRACT this product from the 1st entry in Pair #1.

Repeat for each pair

Flow Graphs for 8-point FFT (Level 2)

Level 2: Two 4-pt FFTs



Algorithm for Pairs

Multiply 2nd entry in Pair #1 by the weight for Pair #1 then ADD and SUBTRACT this product from the 1st entry in Pair #1.

Repeat for each pair

Flow Graphs for 8-point FFT (Level 3)

Level 3: One 8-pt FFTs

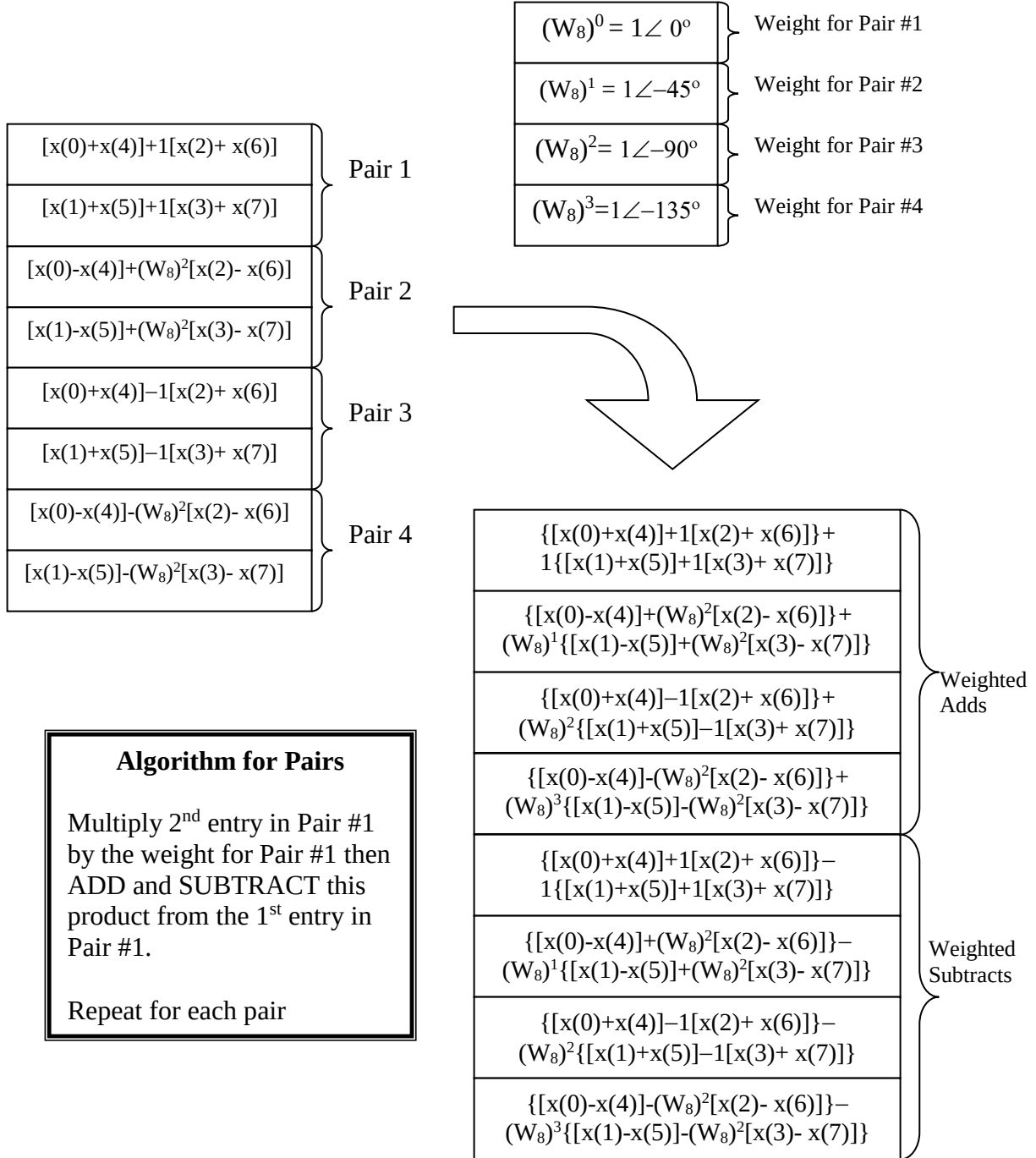


Figure 7.16: Flow Graphs for 8-point FFT

Computational Savings for Radix-2 Decimation in Time FFT

Now that the FFT algorithm has been presented, the obvious question should be why use this complicated algorithm – wouldn't it be easier just to compute the DFT by simply multiplying a vector of time samples of $x(t)$ by the exponential weighting matrix as shown in Equation 7.6? The answer lies in the computational complexity of each method. As the size, N , of the DFT increases, the FFT will require significantly fewer multiplies than a straight DFT computation. The N -point DFT requires N^2 complex multiplies: a $1 \times N$ vector of time samples multiplied by an $N \times N$ weighting matrix. The N -point Radix-2 Decimation in Time FFT requires $N/2 \cdot \log_2(N)$ complex multiplies. The number of complex multiplies for each algorithm are shown in Table 7.5 for several choices of N .

Table 7.5: Computational Complexity of FFT vs. DFT

SIZE, N	N-POINT FFT	N-POINT DFT
	NUMBER OF MULTIPLIES $= \frac{N}{2} \log_2 N$	NUMBER OF MULTIPLIES $= N^2$
32	80	1024
64	192	4096
128	448	16384
256	1024	65536
1024	5120	1048576

The computational savings of an FFT definitely outweigh the added complexity required to program the algorithm. Several applications for fast Fourier transforms including efficient implementation of high order digital filters, fast convolution and correlation, and radar signal processing will be covered in later chapters of this text.

Challenge Question 7.3

Why does a Radix-2 Decimation in Time FFT require $(N/2) \cdot \log_2(N)$ complex multiplies?
Hint: look at the flow graphs in Figures 7.15 and 7.16 and consider the number of multiplies per level as well as the number of levels.

7.4 POWER SPECTRAL DENSITY

The power spectral density of a signal shows how the signal power is distributed over frequency. It can be used to analyze the frequency content of signals. The power spectral density can be estimated from the DFT of the signal as follows:

$$S_{yy}(nf_o) = \frac{1}{N} [Y_{DFT}(nf_o) \cdot Y_{DFT}^*(nf_o)] \quad (7.12)$$

The estimate is only good if the time signal $y(t)$ is sampled fast enough and the resolution, f_0 , is sufficient. The DFT is typically calculated using an FFT algorithm. Power spectral analysis is used in many applications including analysis of cardiovascular signals such as systolic blood pressure and instantaneous heart-rate. Example 7.7 illustrates how the frequency content of a noisy signal can be determined by using an FFT to estimate the power spectral density of the signal.

Example 7.7: Power Spectral Density of a Noisy Signal

Consider the signal $y(t) = 2\sin(2\pi(100)t) + \sin(2\pi(140)t)$. Suppose this signal has been corrupted by random noise. Figure 7.17 shows a plot of the original signal and as well as the noisy signal.

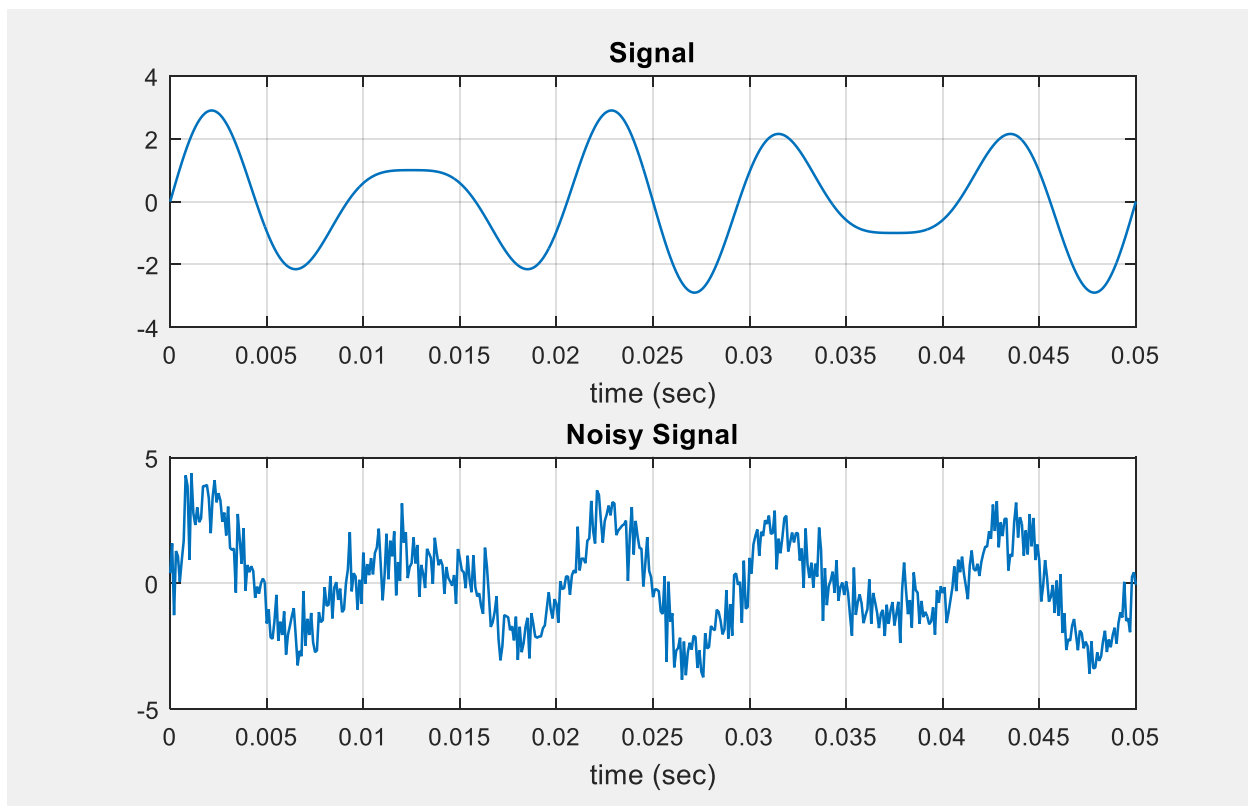


Figure 7.17: Signal and Noisy Signal for Example 7.7

MATLAB code for Figure 7.17:

```
t=0:0.0001:0.05;
y=2*sin(2*pi*100*t)+sin(2*pi*140*t);
ynoisyy = y + 0.75*randn(1,length(t));
subplot(2,1,1); plot(t,y);
title('Signal'); xlabel('time (sec)'); grid;
subplot(2,1,2); plot(t,ynoisyy);
title('Noisy Signal'); xlabel('time (sec)'); grid;
```

The power spectral density is estimated from the discrete Fourier transform of the noisy signal using a variety of sampling frequencies and time durations. The results are shown in Figure 7.18. In the first plot, the sampling frequency and time duration are sufficiently high so the expected frequency components at 100 Hz and 140 Hz appear in the power spectral density. The center plot shows frequency components at 100 Hz and 120 Hz. This is due to not sampling sufficiently fast to capture the 140 Hz component. A sampling frequency of 260 Hz causes the 140 Hz component to alias down to 120 Hz. The last plot illustrates the effect of poor resolution and leakage. With a time duration of only 0.025 sec, the frequency resolution is 40 Hz which means the available frequencies are 0, 40 Hz, 80 Hz, 120 Hz, ... The 100 Hz component spreads mainly to the 80 Hz bin while the 140 Hz component spreads mainly to the 160 Hz bin. Sampling sufficiently fast and over a long enough duration of time are key to getting a good estimate of the signal spectrum.

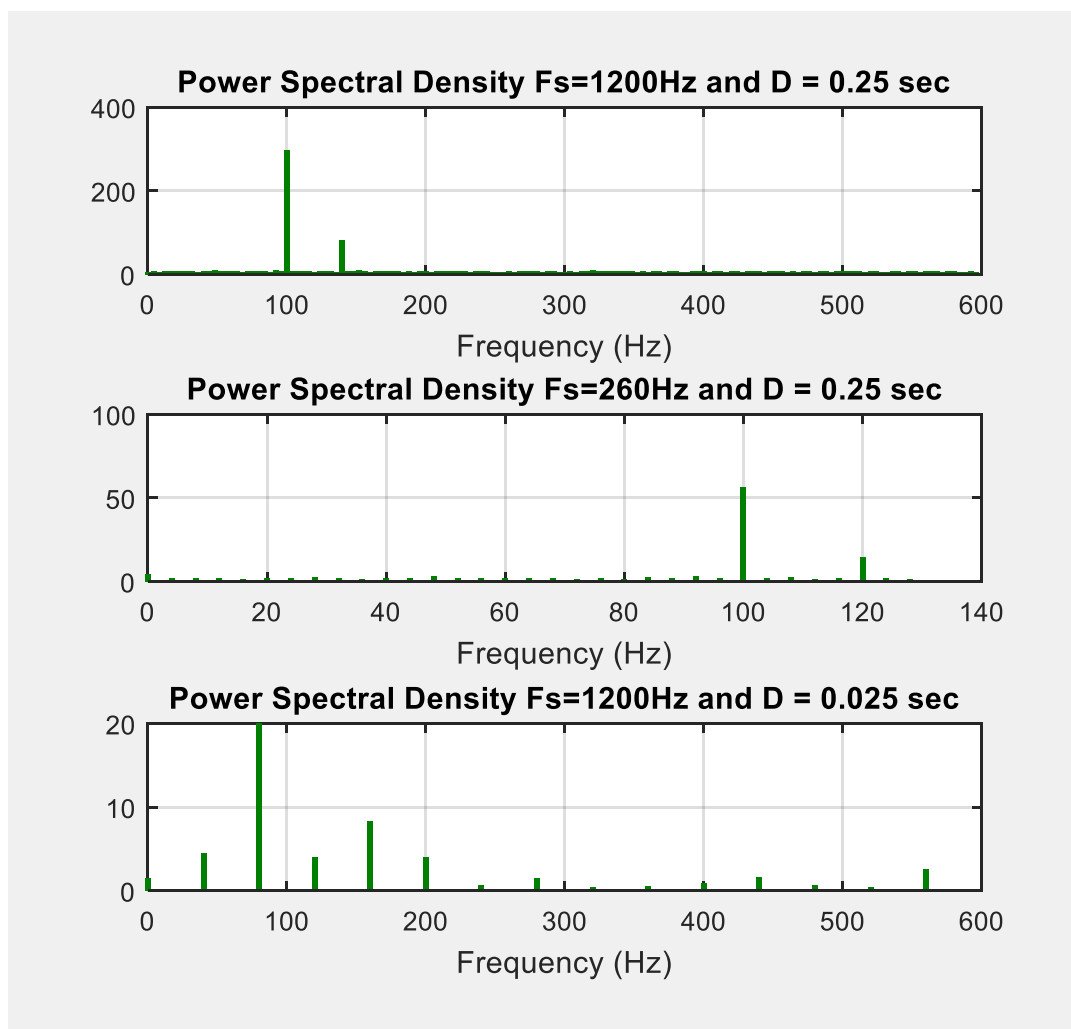


Figure 7.18: Power Spectral Density Estimates

MATLAB code for Figure 7.18:

```
Fs = 1200; D = 0.25; N = Fs*D; fo=1/D;
t = 0:1/Fs:0.25;
ynoisyy = 2*sin(2*pi*100*t)+sin(2*pi*140*t)+
0.75*randn(1,length(t));
Ydft=fft(ynoisyy,N);
Syy = 1/N*Ydft.*conj(Ydft);
f=0:fo:fo*ceil(N/2-1); % Plot only positive frequencies
subplot(3,1,1);stem(f,abs(Syy(1:length(f))));
title('Power Spectral Density Fs=1200Hz and D = 0.25 sec');
xlabel('Frequency (Hz)'); grid

Fs = 260; D = 0.25; N = Fs*D; fo=1/D;
t = 0:1/Fs:0.25;
ynoisyy = 2*sin(2*pi*100*t)+sin(2*pi*140*t)+
0.75*randn(1,length(t));
Ydft=fft(ynoisyy,N);
Syy = 1/N*Ydft.*conj(Ydft);
f=0:fo:fo*ceil(N/2-1);
subplot(3,1,2);stem(f,abs(Syy(1:length(f))));
title('Power Spectral Density Fs=260Hz and D = 0.25 sec')
xlabel('Frequency (Hz)'); grid

Fs = 1200; D = 0.025; N = Fs*D; fo=1/D;
t = 0:1/Fs:0.025;
ynoisyy = 2*sin(2*pi*100*t)+sin(2*pi*140*t)+
0.75*randn(1,length(t));
Ydft=fft(ynoisyy,N);
Syy = 1/N*Ydft.*conj(Ydft);
f=0:fo:fo*ceil(N/2-1);
subplot(3,1,3);stem(f,abs(Syy(1:length(f))));
title('Power Spectral Density Fs=1200Hz and D = 0.025 sec')
xlabel('Frequency (Hz)'); grid
```

7.5 CONVOLUTION AND CORRELATION USING FAST FOURIER TRANSFORMS

Convolution

Convolution of two discrete time signals was introduced in Chapter 4. In Chapter 5 it was shown that the output signal of an FIR filter is simply the convolution of the impulse response (filter coefficients) with the input signal. If the digital filter is high in order, it is more efficient to compute the filter output indirectly using FFTs rather than time domain processing. Fourier Transforms, Laplace Transforms and Z-Transforms, all satisfy the convolution property:

$$x(t) * h(t) \leftrightarrow X(f)H(f) \quad \text{Fourier Transform Convolution Property}$$

$$x(t) * h(t) \leftrightarrow X(s)H(s) \quad \text{Laplace Transform Convolution Property}$$

$$x(k) * h(k) \leftrightarrow X(z)H(z) \quad \text{Z-Transform Convolution Property}$$

The Discrete Fourier Transform **does not** satisfy the standard convolution property as illustrated in Example 7.8.

Example 7.8: Discrete Fourier Transform Doesn't Satisfy Convolution Property

Consider two sequences $x(k) = \{1 \ 2 \ -1 \ 5\}$ and $h(k) = \{1 \ 3 \ 4 \ 0\}$.

- (a) Compute the discrete convolution: $y(k) = x(k) * h(k)$
- (b) Compute the inverse DFT of $[X_{\text{DFT}}(nf_0) \cdot H_{\text{DFT}}(nf_0)]$.

Solution

```
(a)      x=[1  2  -1  5]; h = [1  3  4  0];
          y=conv(x,h)
```

```
y =      1      5      9     10     11     20      0
```

```
(b)      N=4; X=fft(x,N); H=fft(h,N);
          ifft(X.*H)
```

```
ans =      12      25      9     10
```

The two answers do not match!

Although the Discrete Fourier Transform does not satisfy the standard convolution property, it does satisfy a periodic convolution property:

$$x_p(k) \circ h_p(k) \leftrightarrow X_{\text{DFT}}(nf_0)H_{\text{DFT}}(nf_0) \quad (7.13)$$

The sequences $x_p(k)$ and $h_p(k)$ are periodic sequences with period N . How is a periodic convolution performed? First a straight convolution is computed for a single, common length, N , period of each sequence then the result is wrapped around to form a sequence of length N . The process is illustrated in Example 7.9.

Example 7.9: Computing Periodic Convolution using DFT

Consider the two periodic sequences $x_p(k) = \{1 \ 2 \ -1 \ 5 \ 1 \ 2 \ -1 \ 5 \dots\}$ and $h_p(k) = \{1 \ 3 \ 4 \ 0 \ 1 \ 3 \ 4 \ 0 \dots\}$. Compute the periodic convolution, $x_p(k) \circ h_p(k)$. Compare the result to that in Example 7.8b.

Solution

Take a single (common length) period of each signal. Both signals have period 4 so the common period would be four. Compute a straight convolution.

$$x = [1 \quad 2 \quad -1 \quad 5]; \quad h = [1 \quad 3 \quad 4 \quad 0]; \quad y = \text{conv}(x, h)$$

$$y = \quad 1 \quad 5 \quad 9 \quad 10 \quad 11 \quad 20 \quad 0$$

Now wrap the result around to form a sequence of length $N=4$ as follows:

$$y = \begin{array}{cccc} 1 & 5 & 9 & 10 \\ 11 & 20 & 0 & \\ \hline 12 & 25 & 9 & 10 \end{array}$$

$$\text{So, } y_p(k) = x_p(k) \circ h_p(k) = \{12 \ 25 \ 9 \ 10 \ 12 \ 25 \ 9 \ 10 \ \dots\}$$

This is the same answer derived in Example 7.8b by computing the inverse DFT of $X_{\text{DFT}}(nf_0) \cdot H_{\text{DFT}}(nf_0)$.

The periodic convolution property does not appear to be particularly helpful for computing a straight convolution $x(k)*h(k)$ since the input signal and the filter impulse response are not periodic. However, modifying the two sequences by padding them with zeros will force the periodic convolution to match the straight convolution.

Procedure for Computing Convolution, $y(k) = x(k)*h(k)$, using DFTs:

1. Let $N = \text{length}(x) + \text{length}(h) - 1$. Pad each of the original sequences, $x(k)$ and $h(k)$ with zeros until they are both length N .
2. Compute the FFT of each padded sequence: $X_{\text{FFT}}(nf_0)$ and $H_{\text{FFT}}(nf_0)$.
3. Compute the inverse FFT of the product of $X_{\text{FFT}}(nf_0)$ and $H_{\text{FFT}}(nf_0)$. This will be equivalent to $y(k) = x(k)*h(k)$.

Example 7.10: Computing Convolution using DFTs by First Zero Padding Sequences

Consider two sequences $x(k) = \{1 \ 2 \ -1 \ 5\}$ and $h(k) = \{1 \ 3 \ 4\}$. Compute $y(k) = x(k)*h(k)$ using FFTs then compare to the answer in Example 7.8a.

Solution

Let $N = \text{length}(x) + \text{length}(h) - 1 = 4 + 3 - 1 = 6$. Pad each sequence with zeros to make it length $N = 6$. Then compute the FFT of each padded sequence and the inverse FFT of the product as follows:

$$x_{\text{pad}} = [1 \ 2 \ -1 \ 5 \ 0 \ 0]; \quad h_{\text{pad}} = [1 \ 3 \ 4 \ 0 \ 0 \ 0];$$

$$N=6; \quad X = \text{fft}(x, N); \quad H = \text{fft}(h, N); \quad y = \text{ifft}(X.*H)$$

$$y = \quad 1 \quad 5 \quad 9 \quad 10 \quad 11 \quad 20$$

This answer matches than in Example 7.8a and is indeed the straight convolution $x(k)*h(k)$. So why does this work? Padding the sequences makes the periodic convolution of the padded sequences match the straight convolution of the original sequences. Convolver the two padded sequences gives:

```
x_pad=[1  2  -1  5  0  0 ]; h_pad = [1 3 4 0 0 0 0];
y = conv(x_pad,h_pad)
```

```
y=  1  5  9 10 11 20  0  0  0  0  0  0
```

Wrapping the result around to give a sequence of length $N = 6$ yields the desired result:

```
y=  1  5  9 10 11 20
    0  0  0  0  0  0
    1  5  9 10 11 20
```

Notice how the extra zeros extend the wraparound length so all the required data values are properly preserved when wraparound occurs.

Why compute a convolution indirectly using FFTs rather than in the time domain? If the size of the sequences is 128 or higher, computing the convolution indirectly using FFTs requires fewer multiplications than processing in the time domain does. Table 7.6 provides a comparison of the two in terms of required real multiplications.

Table 7.6: Computational Complexity of FFTs for Computing Convolution

SIZE, N	CONVOLUTION USING FFTS NUMBER OF MULTIPLIES $= 8N + 12N \cdot \log_2 N$	CONVOLUTION IN THE TIME DOMAIN NUMBER OF MULTIPLIES $= N^2$
8	448	64
32	2560	1024
128	13,312	16,384
1024	143,360	1,048,576

The FFT requires $8N + 12N \cdot \log_2(N)$ real multiplications. This is not too difficult to derive. If sequence x and h are each of length N , then they will be padded with zeros to length $2N - 1$ which is pretty close to $2N$. As shown in Challenge Question 7.3, a $2N$ -point FFT would require $N \cdot \log_2(N)$ complex multiplications which translates to $4N \cdot \log_2(N)$ real multiplications. Two $2N$ -point FFTs are required (one for padded x and one for padded h), so $8N \cdot \log_2(N)$ real multiplications are required to compute the FFTs of x and h . To multiply the two FFTs requires $2N$ complex multiplications or $8N$ real multiplications. An inverse FFT has the same computational complexity as an FFT so computing the $2N$ -point inverse FFT requires $4N \cdot \log_2(N)$ real multiplications. The total then becomes $8N \cdot \log_2(N) + 8N + 4N \cdot \log_2(N)$ which gives the required result.

Correlation

Correlation is a measure of how well two signals match one another. The equation for computing the cross-correlation for two discrete signals as introduced in Chapter 4 is:

$$R_{xy}(n) = \sum_{k=0}^{N-1} x(k)y(k-n) = \sum_{k=0}^{N-1} x(k+n)y(k) \quad n = -M+1 \dots N-1$$
$$\text{Normalized } R_{xy}(n) = \frac{\sum_{k=0}^{N-1} x(k)y(k-n)}{\sqrt{R_{xx}(0)R_{yy}(0)}} \quad n = -M+1 \dots N-1 \quad (7.14)$$

M = length of signal y(k)

N = length of signal x(k)

The equation for cross-correlation is very similar to the equation for convolution except unlike convolution, the correlation calculation does not require one of the two signals to be reversed (flipped) in time. Like convolution, computing correlations involving long input sequences is more efficient in the frequency domain. The procedure is very similar and the number of required multiplies is exactly the same as listed in Table 7.6 for convolution.

Procedure for Computing Cross-Correlation using FFTs:

1. Let $N = \text{length}(x) + \text{length}(y) - 1$. Pad each of the original sequences, $x(k)$ and $y(k)$ with zeros until they are both length N .
2. Compute the FFT of each padded sequence: $X_{\text{FFT}}(nf_0)$ and $Y_{\text{FFT}}(nf_0)$.
3. Compute the shifted inverse FFT of the product of $X_{\text{FFT}}(nf_0)$ and $\text{conj}(Y_{\text{FFT}}(nf_0))$. This will be equivalent to $R_{xy}(n)$.
4. Divide by the normalization factor $\sqrt{R_{xx}(0)R_{yy}(0)}$ if desired.

Example 7.11: Computing Correlation of Two Sequences using DFTs and Zero Padding

Using FFTs, compute the cross-correlation of the two discrete signals, $x(k)$ and $y(k)$, shown below. Check the result using the MATLAB function ***xcorr***.

$$x(k) = [1 \quad 3 \quad 2 \quad -1 \quad 5]$$

$$y(k) = [2 \quad 5.7 \quad 3.9 \quad -2 \quad 9.8]$$

Solution

```
x=[1 3 2 -1 5]; y = [2 5.7 3.9 -2 9.8];  
x_pad = [x zeros(1,4)]; y_pad = [y zeros(1,4)];  
X = fft(x_pad); Y = fft(y_pad);  
fftshift(iffft(X.*conj(Y)))
```

```
ans =    9.8000    27.4000    17.5000    3.6000    77.9000
    3.5000    17.8000    26.5000    10.0000
```

Checking the result using the MATLAB function ***xcorr*** gives:

```
xcorr(x,y)

ans =    9.8000    27.4000    17.5000    3.6000    77.9000
    3.5000    17.8000    26.5000    10.0000
```

The result can be normalized using MATLAB as follows:

```
fftshift(ifft(X.*conj(Y)))/sqrt((x*x')*(y*y'))

ans =    0.1258    0.3517    0.2246    0.0462    0.9999
    0.0449    0.2285    0.3401    0.1284
```

This shows a very strong linear correlation between x and y when the signals are lined up in time (the center entry in the correlation vector). Not surprising since an inspection of x and y reveals that y is just about double x .

7.6 MATCHED FILTERS

Matched filters are specialized filters designed to pick up a particular signal in the presence of noise. The filter is designed to maximize the signal-to-noise ratio (SNR) for the signal of interest. Matched filters have wide application in image processing, radar processing, and digital communications. Given an input signal, $x(k)$, the coefficients (or impulse response) for an FIR matched filter are:

$$h(k) = \text{ifft}(\text{conj}(\text{fft}(x(k)))) \quad (7.15)$$

The output of the matched filter would be:

$$x(k) * h(k) \leftrightarrow X(nf_o)H(nf_o) = X(nf_o) \cdot \text{conj}(X(nf_o)) \quad (7.16)$$

Recall from the previous section the following correlation property:

$$R_{xx}(n) \leftrightarrow \text{ifft}[X(nf_o) \cdot \text{conj}(X(nf_o))] \quad (7.17)$$

It then follows that the output of the matched filter is the auto-correlation of the desired signal $x(k)$ with itself. Of course, in practice, the desired signal will likely be corrupted by noise. The matched filter essentially performs a correlation calculation to find or detect the desired signal in the presence of noise. This is illustrated in Example 7.12.

Example 7.12: Matched Filter for Linear Chirp Signal

In this example a linear chirp signal will be used to represent the signal to be detected by the filter. A linear chirp signal is simply a signal with a frequency that increases linearly over some specified time interval. The chirp signal for this example is plotted in Figure 7.19a. The frequency varies from 0 Hz to 100 Hz over a time interval of 0.5 sec. The MATLAB commands for this signal are shown Figure 7.20 (lines 2 and 3). As shown in line 5 of Figure 7.20, the matched filter coefficients are computed from the spectrum of the chirp signal using equation 7.15 which is simply $h(k) = \text{ifft}(\text{conj}(\text{fft}(x(k))))$.

The input chirp signal is applied to the matched filter. The resulting filter output is shown in Figure 7.19a. Notice the distinctive spike that occurs in the output signal at $t = 0.5$ sec indicating that the chirp signal has been successfully detected. At $t=0.5$ sec (the exact duration of the chirp signal), the input chirp signal lines up exactly with the matched filter coefficients so the filter output is simply the correlation of the chirp with itself (autocorrelation). Since the chirp signal is pure with no noise corruption, the spike is very distinctive.

The input signal is then corrupted with a fair amount random noise (with mean zero) using the MATLAB function **randn**. The noisy input and the resulting matched filter output are shown in Figure 7.19b. Although the original chirp signal is not at all apparent in the input signal, the filter is still able to detect the presence of the chirp within the noisy input. A distinctive spike is again produced at $t = 0.5$ sec or the duration of the chirp in the noisy input signal. Figure 7.19c illustrates that if the chirp signal is buried in enough noise, the matched filter will not be able to detect the chirp.

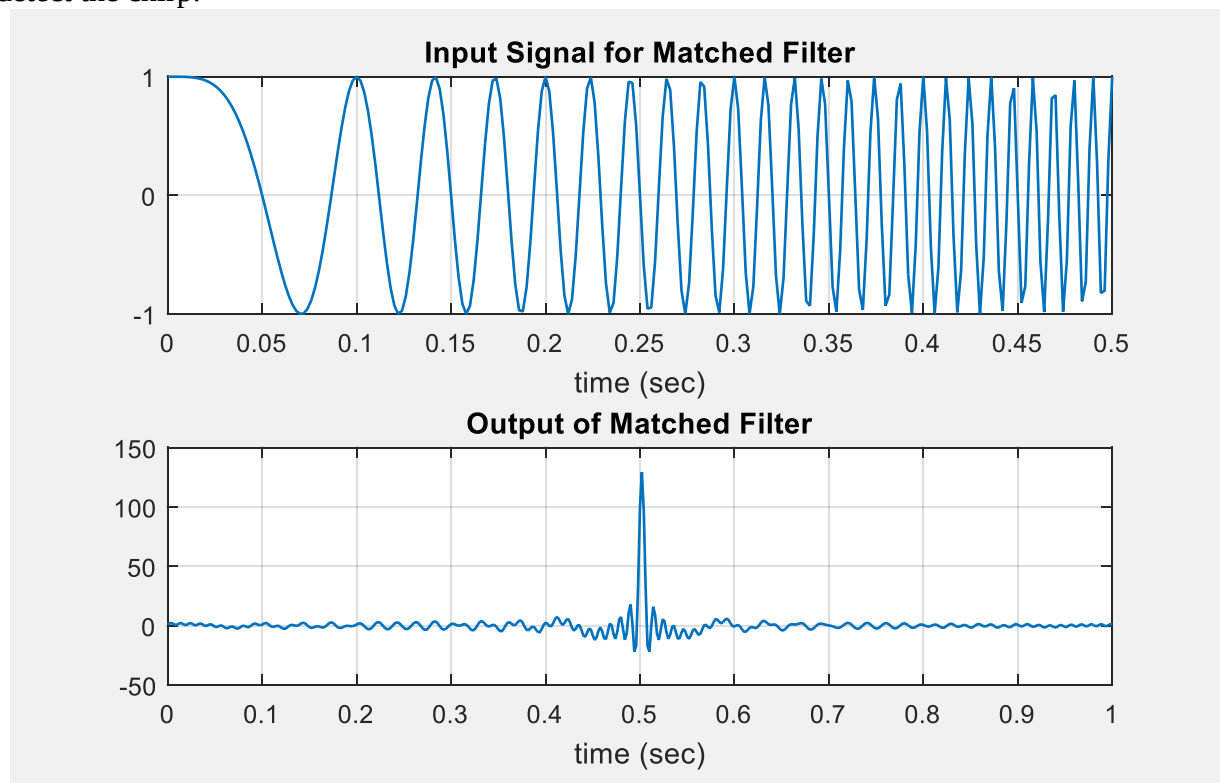


Fig 7.19a: Chirp Input and Matched Filter Output (no noise)

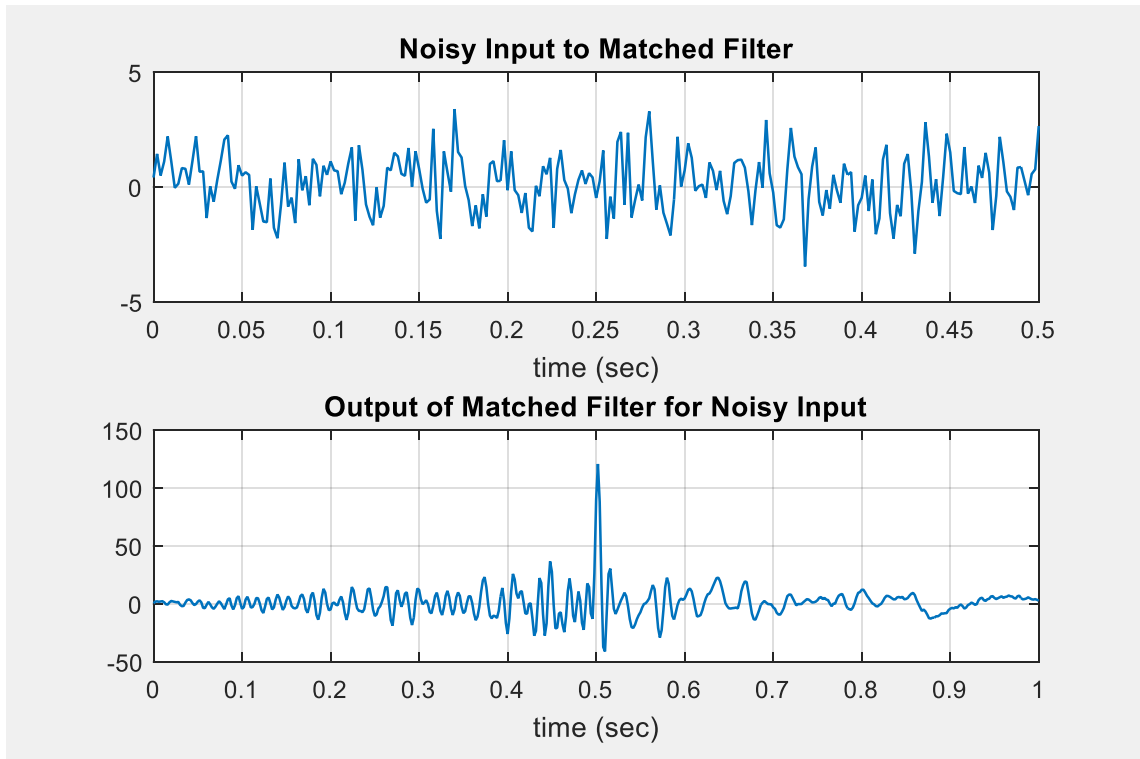


Fig 7.19b: Chirp Input and Matched Filter Output (moderate noise)

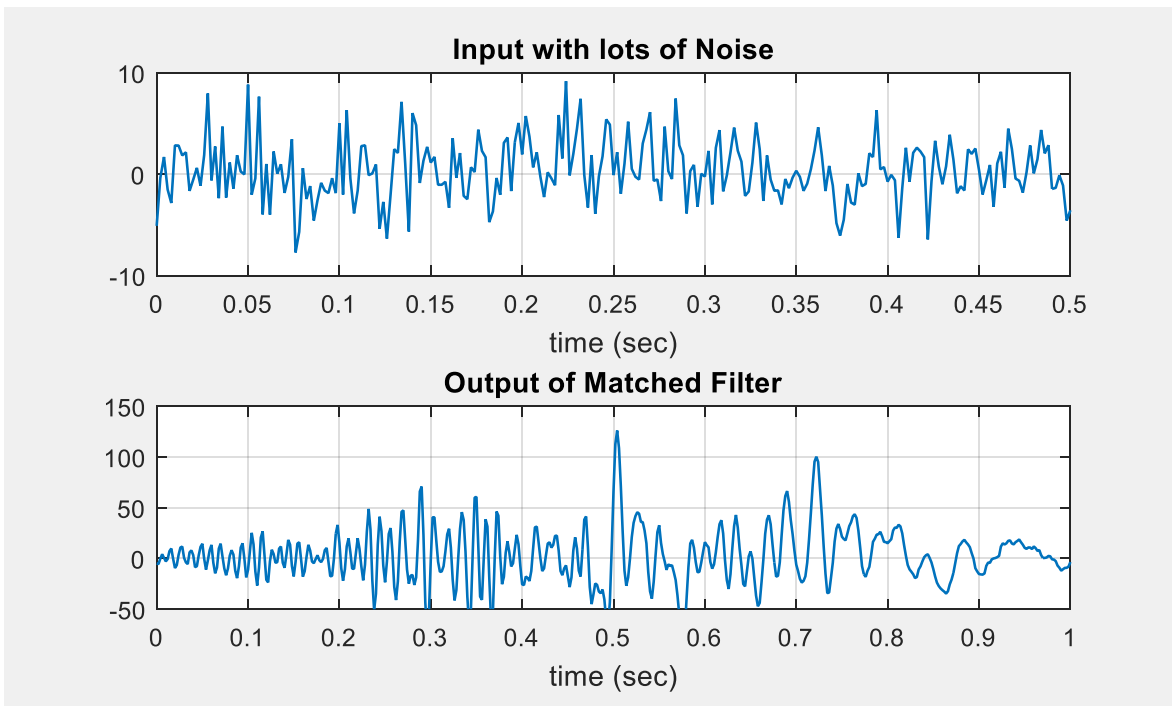


Fig 7.19c: Chirp Input and Matched Filter Output (excessive noise)

Figure 7.19: Matched Filter (Example 7.12)

MATLAB Code for Example 7.12

```
% Create chirp signal input
t = 0:0.002:0.5;
input=chirp(t,0,0.5,100);
% Calculate the Matched Filter Coefficients
h = ifft(conj(fft(input)));
output = conv(h,input);
time = 0:0.002:0.002*(length(output)-1);
subplot(2,1,1); plot(t,input);
title('Input Signal for Matched Filter');
xlabel('time (sec)'); grid
subplot(2,1,2); plot(time,output);
title('Output of Matched Filter');
xlabel('time (sec)'); grid
% Now Create a noisy input
noisy_input = input + randn(size(input));
new_output = conv(h,noisy_input);
figure
subplot(2,1,1);plot(t,noisy_input);
title('Noisy Input to Matched Filter');
xlabel('time (sec)'); grid
subplot(2,1,2);plot(time,new_output);
title('Output of Matched Filter for Noisy Input');
xlabel('time (sec)'); grid
% Now Create an input with lots of noise
noisy_input = input + 3*randn(size(input));
new_output = conv(h,noisy_input);
figure
subplot(2,1,1);plot(t,noisy_input);
title('Input with lots of Noise');
xlabel('time (sec)'); grid
subplot(2,1,2);plot(time,new_output);
title('Output of Matched Filter');
xlabel('time (sec)'); grid
```

Figure 7.20: MATLAB code for Example 7.12

The chirp signal discussed in Example 7.12 is very useful in radar processing for detecting the presence of and distance to a target. Matched filters are an important part of the detection scheme. Radar processing is discussed in considerably more detail in Chapter 10.

7.7 ALGORITHMS FOR FOCUSED SPECTRAL ANALYSIS

Discussion so far in this chapter has assumed that a complete spectral analysis of a signal is required; in other words, all frequency components of the signal must be identified. In some cases, it may only be necessary to focus in on a particular band of frequencies or even a discrete set of frequencies for a signal. For example, RF communications, vibration analysis, and radar signal processing focus in on specific bands of frequencies. Demodulation of FSK (Frequency Shift Keying) signals used for transmitting binary data or detection of DTMF (Dual Tone Multi-Frequency) telephone signals only requires spectral information at a specific set of discrete frequencies. For cases such as these, a Fast Fourier Transform provides more information than necessary and becomes computationally inefficient. A Zoom FFT is an efficient algorithm for focusing in on a particular band of frequencies. The Goertzel algorithm is useful for capturing spectral information for a discrete set of frequencies. Each of these algorithms will be explored in this section.

Zoom FFT

A Zoom FFT is an efficient algorithm for focusing in on a particular band of frequencies rather than estimating the entire spectrum of a signal. Figure 7.21 illustrates the processing associated with a Zoom FFT. The sampled signal is first multiplied by an exponential sequence that will down-shift the frequency band of interest to baseband. The low-pass filter passes the band of interest and eliminates the unwanted portion of the spectrum. The signal is then decimated prior to the FFT calculation.

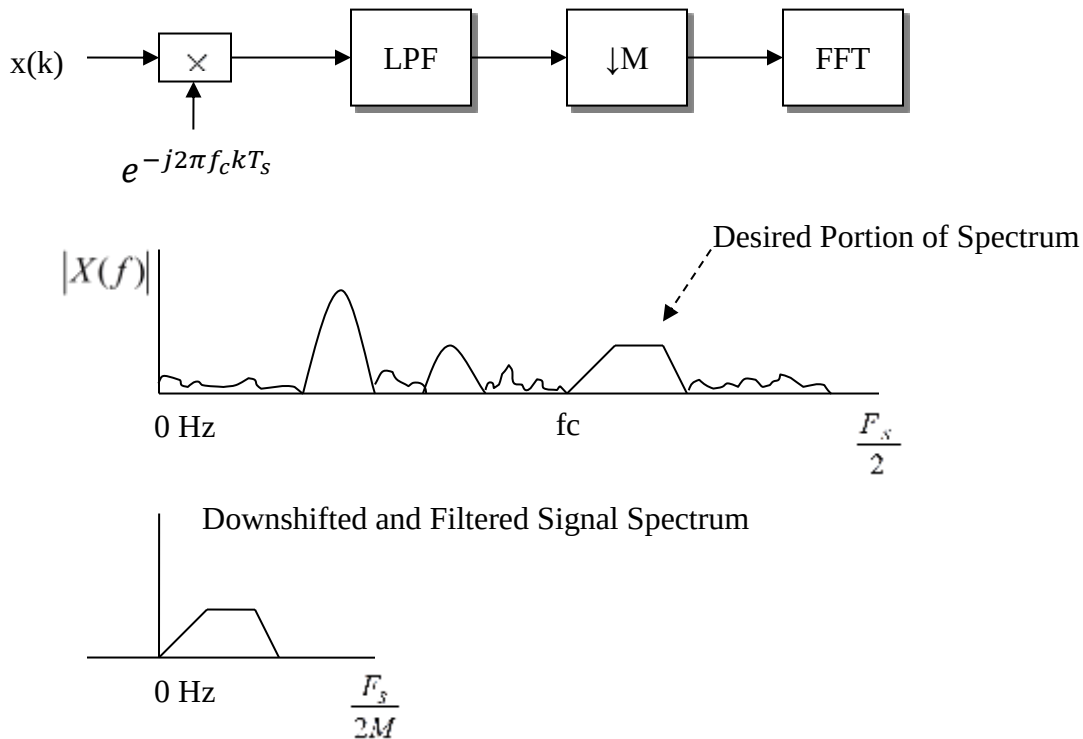


Figure 7.21: Zoom FFT

Example 7.13: Zoom FFT

Suppose the input signal is a wav file which is originally sampled at 44.1 kHz.

- (a) Calculate the spectral resolution for a 1024-pt FFT based on the original signal.
- (b) Assume the frequency band of interest for the wav file ranges from 2 kHz to 4 kHz. What exponential sequence is required to down-shift this portion of the spectrum? What should the decimation factor for this signal be? Why? Calculate the spectral resolution for a 1024-pt FFT based on the focused signal.
- (c) Using the Filter Design and Analysis tool, design an appropriate low-pass filter for the down-shifted input signal prior to decimation.
- (d) Assume the input signal is $\sin(2\pi(2500)t) + \sin(2\pi(2600)t) + \sin(2\pi(3000)t)$ sampled at 44.1 kHz. Compute a 1024-pt FFT on this signal and plot the magnitude of the FFT. Does the FFT adequately reflect the spectrum of the input signal? Why?
- (e) Downshift the input signal by multiplying it by the exponential sequence determined in part b, filter the signal using the filter designed in part c, and then decimate the signal. Compute a 1024-pt FFT on the “zoomed in” signal.

Solution

- (a) The spectral resolution for the original signal would be $f_o = \frac{F_s}{N} = \frac{44100}{1024} = 43.066 \text{ Hz}$.

(b) The desired spectrum ranges from 2 kHz to 4 kHz so the exponential sequence should be chosen as $e^{-j2\pi(2000)kT_s}$. The bandwidth of the filtered signal is 2 kHz so the minimum sampling rate (Nyquist) would be 4 kHz. Decimating by a factor of 10 reduces the sampling rate to 4.41 kHz. The spectral resolution for the filtered and decimated signal would be

$$f_o = \frac{F_s}{N} = \frac{4410}{1024} = 4.3066 \text{ Hz}.$$

- (c) The frequency band of interest in the original signal is from 2 to 4 kHz. After the signal is downshifted, this frequency band will be from 0 to 2 kHz. Fdatool is utilized to design an equiripple low-pass filter with a sampling frequency of 44.1 kHz, a passband of 2 kHz, passband ripple of 0.1 dB, stopband frequency of 2.2 kHz, and stopband attenuation of 20 dB. The numerator coefficients for the filter are the exported and saved in a .mat file.
- (d) and (e) The plots of the original FFT and the zoom FFT are shown in Figure 7.22. The MATLAB code is shown in Figure 7.23. In the plot of the FFT of the original signal, it is not possible to distinguish the three frequencies (2500, 2600, and 3000 Hz). However, the zoom FFT shows three distinct frequencies. Since the signal has been downshifted by 2000 Hz, the original frequencies now appear at 500, 600, and 1000 Hz. The resolution has been improved by a factor of 10. We have “zoomed in” on the spectrum of the signal without increasing the size of the FFT.

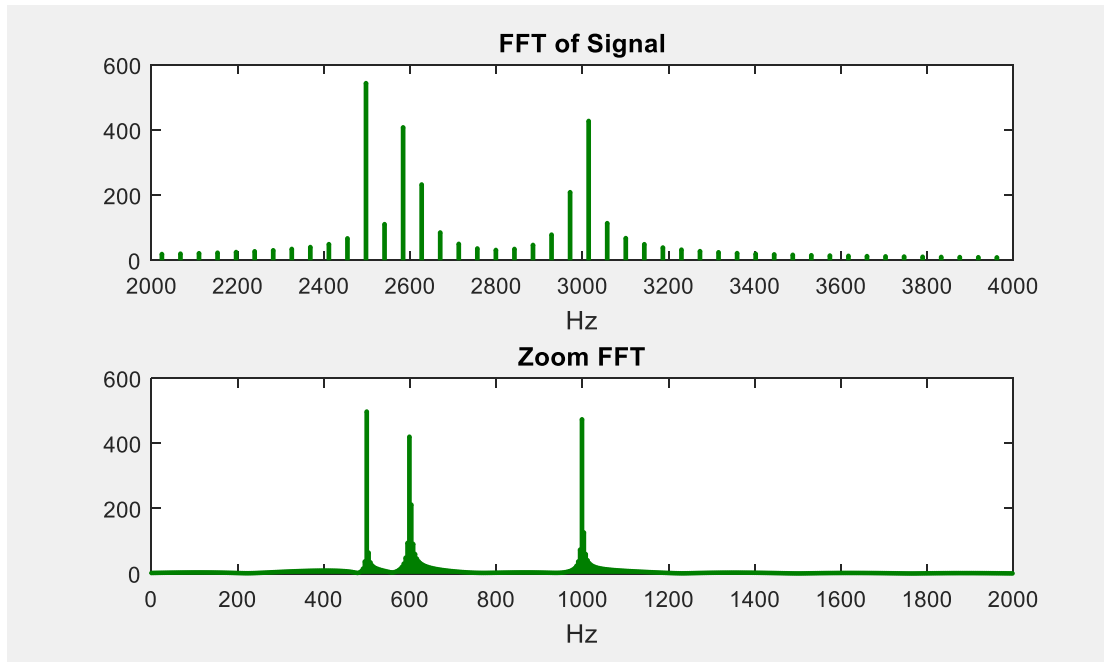


Figure 7.22: Zoom FFT

```
t = 0:1/Fs:(N-1)*1/Fs;
x = sin(2*pi*2500*t) + sin(2*pi*2600*t) + sin(2*pi*3000*t);
X = fftshift(fft(x));
f = -floor(N/2)*fo:fo:(ceil(N/2)-1)*fo;
subplot(2,1,1); stem(f,abs(X));
title('FFT of Signal'); xlabel('Hz'); xlim([2000 4000])
% Downshift by 2000 Hz
% Since decimating by 10, need longer sample of x (10 times as long)
DownShift = 2000;
DecimateFactor = 10;
t = 0:1/Fs:(DecimateFactor*N-1)*1/Fs;
x = sin(2*pi*2500*t) + sin(2*pi*2600*t) + sin(2*pi*3000*t);
x_down = x .* exp(-2*pi*li*DownShift*t);
% fdatool: Design FIR equiripple with Specs: pass: 0-2K, stop 2.2K,
% Fs = 44100, pass ripple 0.1 dB, stop att. 20 dB
% Export numerator as Num and save in EX7_13.mat file
load EX7_13; % loads in numerator coefficients saved from fdatool
Den = zeros(size(Num)); Den(1)=1;
x_filter = filter(Num, Den, x_down);
x_dec = downsample(x_filter,DecimateFactor);
X_zoom = fftshift(fft(x_dec));
fzoom = fo/DecimateFactor;
f = -floor(N/2)*fzoom:fzoom:(ceil(N/2)-1)*fzoom;
subplot(2,1,2); stem(f,abs(X_zoom));
title('Zoom FFT'); xlim([0, 2000]); xlabel('Hz')
```

Figure 7.23: MATLAB Code for Example 7.13

Goertzel Algorithm

The Goertzel Algorithm is a computationally efficient method for computing the spectrum of a signal at only a few select discrete frequencies. The specific application that will be examined in this section is detection of a DTMF (Dual-tone multi-frequency) telephone signal created using the push button keypad on a phone. Figure 7.24 shows the frequencies associated with each button on a standard telephone keypad.

	1209 Hz	1336 Hz	1477 Hz
697 Hz	1	2	3
770 Hz	4	5	6
852 Hz	7	8	9
941 Hz	*	0	#

Figure 7.24: DTMF Frequencies

If the button labeled with a 1 is depressed, a pulse with two frequencies, 697 Hz and 1209 Hz, is transmitted. Therefore, detecting which buttons are depressed becomes a matter of detecting the frequencies present in each pulse. Of course to complicate matters, this pulse could occur in the presence of noise or voice signals.

Computing an FFT of this signal would produce much more spectral information than is actually required and would therefore be computationally inefficient. To see this, assume the pulse will be sampled for a duration of 25.625 ms at a rate of 8000 samples/sec (standard F_s for telephone signals). This would result in 205 time samples so the FFT would produce spectral information at 205 frequencies spaced apart by $f_0 = 1/25.625 \text{ ms} = 39.024 \text{ Hz}$. For this application, only the seven frequencies associated with the buttons are of interest so the FFT provides unnecessary spectral information.

Recall that the DFT of a signal, $x(t)$, is given by

$$X_{\text{DFT}}(nf_0) = \sum_{k=0}^{N-1} x(kT_s) e^{-j\frac{2\pi nk}{N}} \quad n = 0, 1 \dots N-1$$

T_s = sampling interval in time domain

$$f_0 = \frac{F_s}{N} = \text{sampling interval (resolution) in frequency domain}$$

(7.18)

Rather than computing the summation for all n from 0 to $N-1$, the summation could be computed for only select values of n . Assume the pulse is sampled for a duration of 25.625 ms (205 samples at $F_s = 8000$ Hz). This will result in a spectral resolution, f_o of approximately 39.024 Hz. The DFT need only be calculated for the values of n shown in Table 7.7. The discrete frequency bins are not identical to the actual tone frequencies but are very close.

Table 7.7: Discrete Frequencies for Tone Detection

TONE FREQUENCY (HZ)	$n = \text{round}\left[\frac{\text{Tone Frequency}}{f_o}\right]$	DISCRETE FREQUENCY (HZ) $nf_o = n \frac{F_s}{N}$
697	18	702.44
770	20	780.49
852	22	858.54
941	24	936.59
1209	31	1209.76
1336	34	1326.83
1477	38	1482.93

One way to detect which buttons are depressed would be to compute the DFT summation for each of the seven values for n listed in Table 7.7. The DFT for each tone would require the equivalent of $2N$ real multiplications (N complex weightings times N real sample values). A much more efficient method for computing a limited number of DFTs is the Goertzel Algorithm.

For any given n , the DFT at the frequency nf_o can be computed using the Goertzel Algorithm:

$$s(k) = x(k) + 2 \cos\left(\frac{2\pi n}{N}\right) s(k-1) - s(k-2) \quad k = 0, 1 \dots N-1$$

$$s(-1) = s(-2) = 0$$

$$Y(nf_o) = s(N-1) - e^{-j\frac{2\pi n}{N}} s(N-2)$$

$$X(nf_o) = \frac{Y(nf_o)}{e^{j2\pi n(N-1)/N}}$$

(7.19)

In words, iterate through the first difference equation N times. Plug the results of the last two iterations into the equation for $Y(nf_o)$ and then use this result in some threshold decision logic to decide whether or not a tone has been detected. It is not even necessary to calculate $X(nf_o)$ since $X(nf_o)$ and $Y(nf_o)$ have the same magnitude and only differ in phase.

Example 7.14: Tone Detection using Goertzel Algorithm

Create a 50 ms pulse with the frequencies corresponding to button 8 on the telephone keypad using a sampling frequency of $F_s = 8000$ Hz. Use the Goertzel algorithm to take 205 samples of the pulse signal and determine the frequency response at each of the seven discrete frequencies listed in Table 7.7. Plot the results and compare with a standard FFT.

Solution

An m-file was created for this example and the code is shown below. Results are graphed in Figure 7.26. The graphs in Figure 7.26 show definite peaks at the discrete frequency bins of 858 and 1327 Hz which correspond to the tone frequencies of 852 and 1336 Hz for the button 8 on the keypad. Notice that the Goertzel Algorithm matches the FFT exactly at the seven frequencies of interest. It is certainly more efficient since the spectrum is only computed at the seven frequencies of interest; whereas, the FFT computes the spectrum at 205 frequencies.

```
Fs=8000; N = 205; fo=Fs/N;
% Create a Pulse for Button 8
t=0:1/Fs:0.050;
x=sin(2*pi*852*t)+sin(2*pi*1336*t);
% Define the bin numbers for the discrete frequencies
% to be calculated
n=[18 20 22 24 31 34 38];
% Run the Goertzel Algorithm for each discrete frequency
for m=1:7,
    num=1; den=[1 -2*cos(2*pi*n(m)/N) 1];
    s=filter(num,den,x(1:205));
    Y(m)=s(N)-exp(-1i*2*pi*n(m)/N)*s(N-1);
end
% Plot the results
freqs = n*fo;
subplot(3,1,1);stem(freqs,abs(Y));
title('Results of Goertzel Algorithm');
xlabel('Frequency (Hz)');
% For Comparison Purposes, look at FFT
X=fftshift(fft(x,205));
f = -floor(N/2)*fo:fo:(ceil(N/2)-1)*fo;
subplot(3,1,2); stem(f(103:205),abs(X(103:205)));
title('Results of FFT'); xlabel('Frequency (Hz)');
subplot(3,1,3); stem(f(121:141),abs(X(121:141)));
title('Results of FFT - Zoomed In'); xlabel('Frequency (Hz)');
```

Figure 7.25: MATLAB Code for Example 7.14

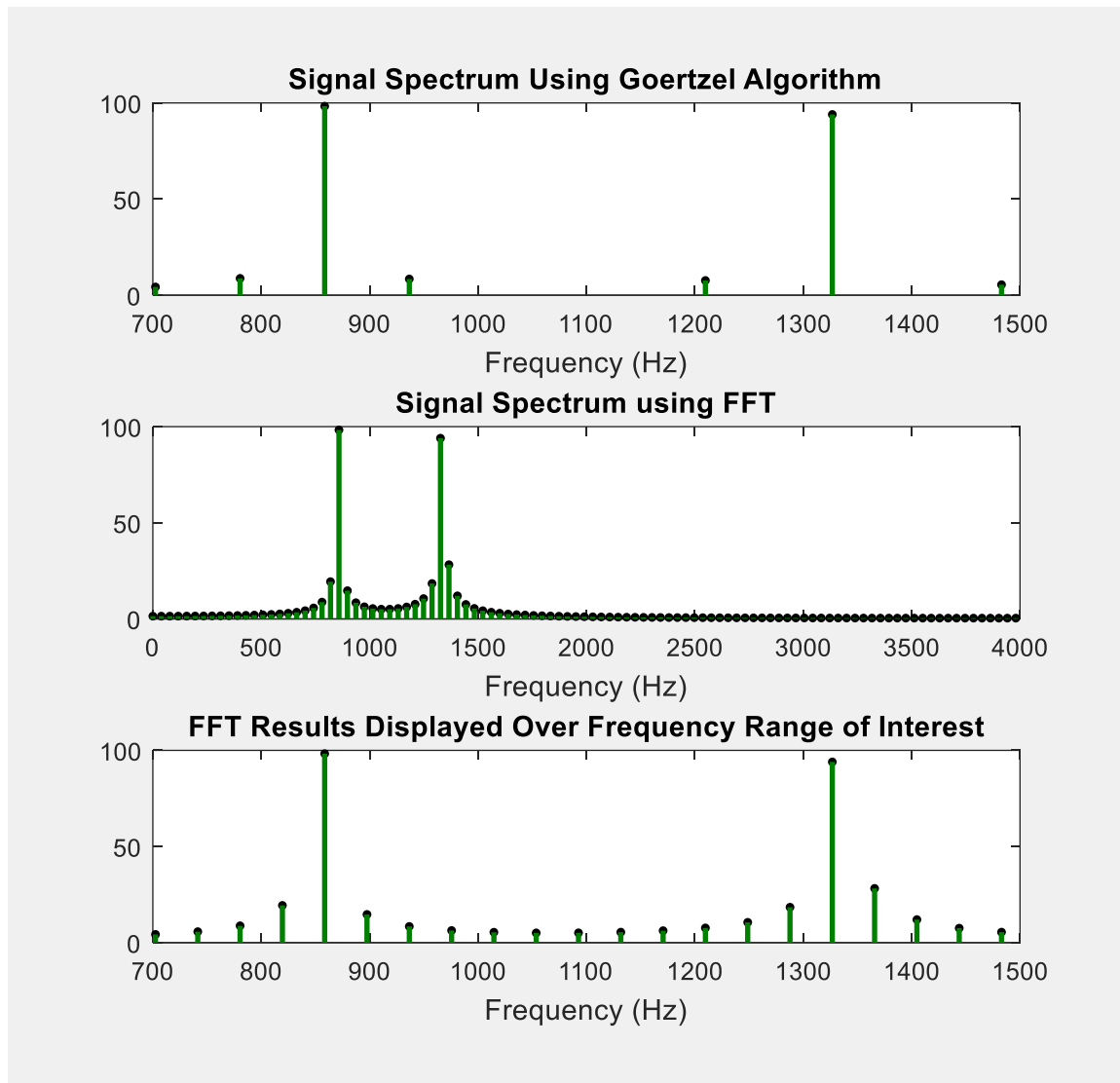


Figure 7.26: Goertzel Algorithm for DTMF Detection

In Example 7.14, the Goertzel Algorithm accurately detected the frequencies associated with the pulse for the push button 8. There are some practical considerations that need to be taken into account:

- Timing is a critical issue. In the previous example, 205 samples of the pulse signal were processed through the algorithm. In reality, data is arriving in real-time and the actual start and end time of pulses is not known. Figure 7.27 shows the results of the Goertzel Algorithm if only 4 ms of the pulse is captured with the remainder of the data over the 25.625 ms time duration being no signal. Detection clearly breaks down here.
- The pulses will be corrupted by noise and voice signals. Also drop outs may occur within the pulse.

- The incoming data must be processed in real-time by capturing sections or windows of data. The Goertzel Algorithm lends itself well to real-time processing since it simply involves iterating through a 2nd order difference equation. Some type of decision making logic based on thresholds must be added to the algorithm to account for noise and to determine whether or not enough pulse data has been captured in a particular window to make a valid decision.

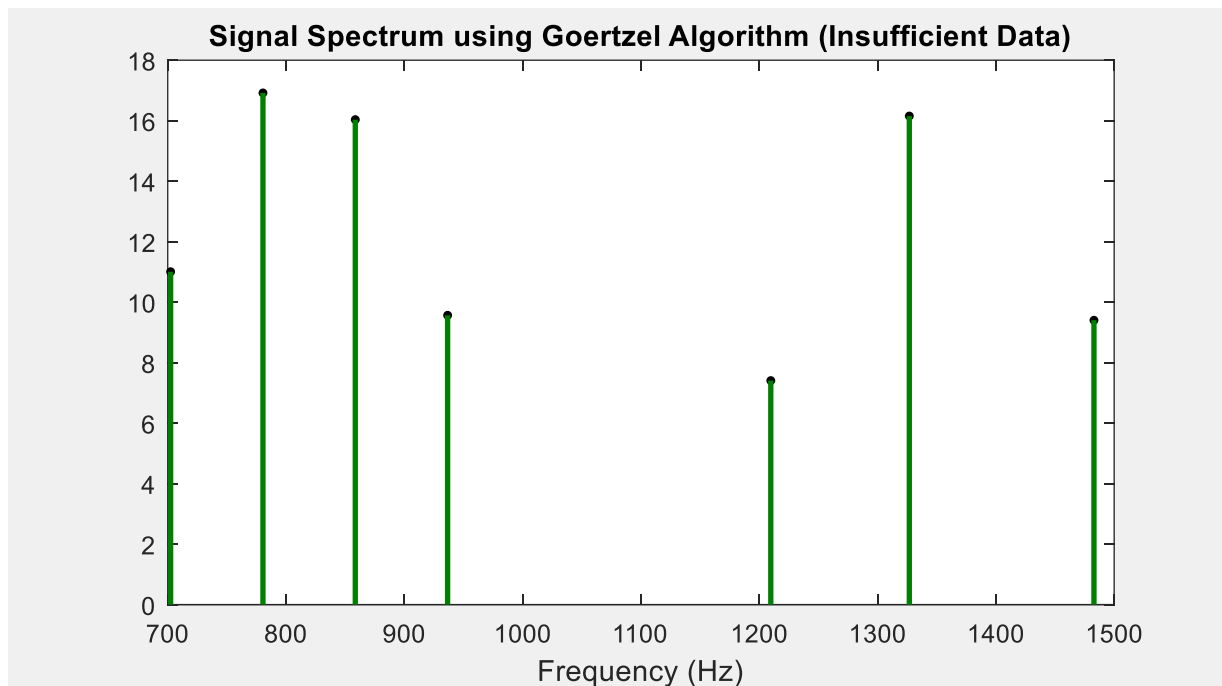


Figure 7.27: Goertzel Algorithm with Insufficient Pulse Data

Challenge Question 7.4

Calculate the number of real multiplications required for the Goertzel algorithm in Example 7.14 and compare it to the number of real multiplications required for the FFT computation.

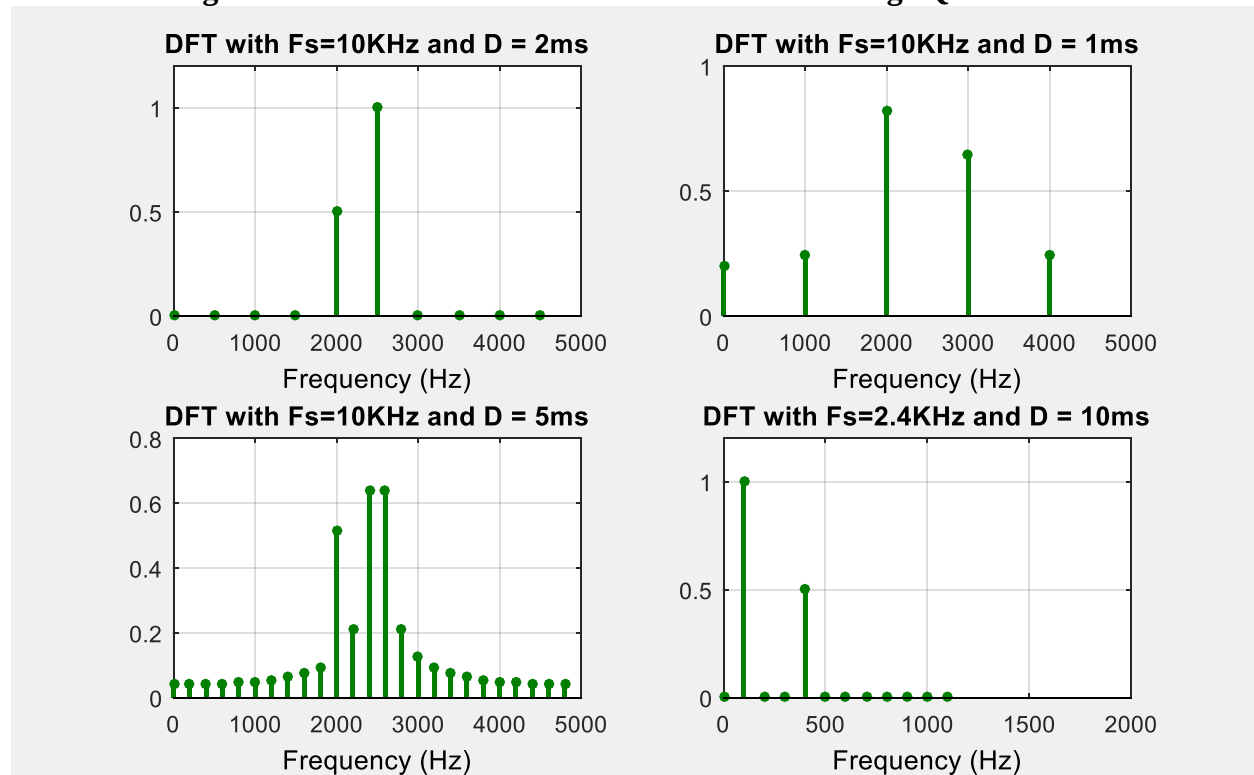
Answers to Chapter 7 Challenge Questions

Question 7.1 The third DFT in Figure 7.2 shows two frequency components at ± 1500 Hz. Explain this. *Hint: consider the sampling frequency chosen for this case.*

The 2.5 kHz sine wave is being sampled at 4 kHz. This is a problem because according to the sampling theorem, a 2.5 kHz sine wave should be sampled above 5 kHz. With a sampling rate of 4 kHz, that 2.5 kHz sine wave will alias as a 1.5 kHz sine wave which is exactly what the DFT shows.

Question 7.2 Figure 7.7 plots the DFT of the signal $x(t) = \sin(2\pi(2000)t) + 2\sin(2\pi(2500)t)$ for several different sampling frequencies and time durations. Which of the DFTs in the plot provide accurate estimates of the spectrum of $x(t)$? For the DFTs which do not accurately reflect the signal spectrum, explain the problem.

Figure 7.7 Discrete Fourier Transforms for Challenge Question 7.2



The spectrum of the signal $x(t)$ should consist of a 2000 Hz and a 2500 Hz component. The plot at the top left of Figure 7.7 is therefore accurate. The signal was sampled sufficiently fast and the time duration, D , was an integer multiple of the period for both sinusoids providing an adequate frequency resolution of $f_0 = 500$ Hz.

The top right plot does not provide an accurate estimate of the signal spectrum due to the selected time duration D . Choosing $D = 1\text{ms}$ results in a frequency resolution of 1000 Hz. So,

the available frequency bins are 0, 1000 Hz, 2000 Hz, 3000 Hz, and 4000 Hz. The 2500 Hz component splits among the adjacent bins – mainly 2000 and 3000 Hz.

The bottom left plot also shows the effect of leakage; that is, the time duration, D , is not an integer multiple of the period of the 2500 Hz signal. Choosing $D = 5$ ms results in a frequency resolution of 200 Hz and the available frequency bins are 0, 200 Hz, 400 Hz, ... Again, the 2500 Hz signal is forced to split among adjacent bins – mainly 2400 and 2600 Hz. The spectrum in this case is certainly better than the spectrum shown in the top right plot due to the improved resolution.

The bottom right plot shows the effect of insufficient sampling rate. The sampling frequency is only 2400 Hz which causes the 2000 Hz signal to alias as a 400 Hz signal and the 2500 Hz signal to alias as a 100 Hz signal.

Question 7.3 Why does a Radix-2 Decimation in Time FFT require $\frac{N}{2} \log_2 N$ complex

multiplies? *Hint: look at the flow graphs in Figures 7.15 and 7.16 and consider the number of multiplies per level as well as the number of levels.*

Start with the 4-point FFT in Figure 7.15. At each level, there is one multiply per pair. There are two pairs ($N/2$) at each level so there are ($N/2$) multiplies per level. How many levels are there? There are two levels since $N = 4 = 2^2$.

Now look at the 8-point FFT in Figure 7.16. Again, at each level, there is one multiply per pair. There are four pairs ($N/2$) at each level so there are ($N/2$) multiplies per level. There are three levels since $N = 8 = 2^3$.

So, the number of multiplies = $[(N/2) \text{ multiplies per level}] \cdot [\text{Number of levels}] = \frac{N}{2} \log_2 N$

Question 7.4 Calculate the number of real multiplications required for the Goertzel algorithm in Example 7.14 and compare it to the number of real multiplications required for the FFT computation.

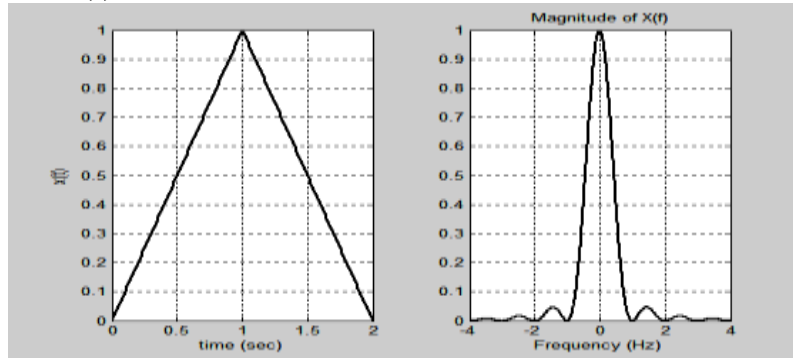
Examine the Goertzel algorithm shown in equation 7.19. The difference equation requires only one real multiplication (middle term) for each iteration of the difference equation. Since there are 205 time samples, the difference equation requires 205 iterations or 205 real multiplications for each tone frequency. Computing $Y(nf_0)$ requires multiplication of a real number times a complex number or equivalently two real multiplications for each tone frequency. Therefore, the Goertzel algorithm for this example requires 207 real multiplications for each of the seven tone frequencies which results in $7 \cdot 207 = 1449$ real multiplications.

A N -point FFT requires $N/2 \cdot \log_2(N)$ complex multiplications or $2N \cdot \log_2(N)$ real multiplications. The 205 time samples would require a 256-point FFT (must be a power of 2). The number of required real multiplications would be $2 \cdot 256 \cdot \log_2(256) = 4096$ real multiplications. This is nearly triple the number of multiplications required for the Goertzel algorithm.

Chapter 7 Problems

Problem 7.1: The spectrum of a signal with a bandwidth of 8 kHz is to be estimated using a DFT. The duration of time, D , over which the signal will be sampled is 0.01 seconds. Determine the frequency resolution, f_0 , for the DFT and the minimum size, N , required for the DFT. Using the minimum size, N , for the DFT, determine the discrete frequencies for the DFT.

Problem 7.2: Consider the time signal $x(t)$ sketched below. The Fourier transform of this triangular pulse is $\text{sinc}^2(f)$.



- (a) Use a sampling frequency of $F_s = 32$ Hz and a duration $D = 6$ sec. Compute the DFT of $x(t)$ and compare it to the actual signal spectrum, $X(f)$. Comment on whether the DFT appears to be a good match to the actual spectrum. If the match is not good, explain why.
- (b) Repeat (a) for $F_s = 8$ Hz and $D = 6$ sec.
- (c) Repeat (a) for $F_s = 32$ Hz and $D = 2$ sec.

Problem 7.3: For each of the signals given below, create the signal in MATLAB, then compute and plot the magnitude of the DFT. Does the DFT accurately reflect the true signal spectrum? If not, why not.

- (a) A 20 Hz sinusoid sampled for three periods at 200 Hz.
- (b) A 20 Hz sinusoid sampled for five periods at 30 Hz.
- (c) A 20 Hz sinusoid sampled for 2.5 periods at 200 Hz.
- (d) A 20 Hz sinusoid sampled for 10.5 periods at 200 Hz.

Problem 7.4:

- (a) What does the spectrum of a 10 Hz square wave look like? (Refer back to Section 2.5 of Chapter 2 if necessary).
- (b) For each of the signals given below, create the signal in MATLAB, then compute and plot the magnitude of the DFT. Does the DFT accurately reflect the true signal spectrum? If not, why not. (Hint: to create the square wave in MATLAB, use $t=0:1/F_s:D-1/F_s$; $y=\text{square}(2\pi*10*t)$)
 - A 10 Hz square wave sampled at 200 Hz for 0.5 seconds.
 - A 10 Hz square wave sampled at 100 Hz for 0.5 seconds.
 - A 10 Hz square wave sampled at 90 Hz for 0.5 seconds.
 - A 10 Hz square wave sampled at 200 Hz for 0.45 seconds.
 - A 10 Hz square wave sampled at 200 Hz for 1.45 seconds.

Problem 7.5: Consider the signal $f(t) = 5\sin(2\pi(261)t) + \sin(2\pi(277)t)$.

- What does the spectrum of $f(t)$ look like?
- Compute and plot the magnitude of the DFT for $f(t)$ assuming a sampling rate of 8000 Hz and a duration of 0.5 seconds. Comment on the results.
- Window the samples of $f(t)$ using a *hann* window then compute and plot the magnitude of the DFT for the windowed data. Comment on the results.
- Repeat part (c) using a *blackman* window.

Problem 7.6: Consider the two sequences given below:

$$x(k) = \{15 \quad 3 \quad 12 \quad 0 \quad 8 \quad -14 \quad 15 \quad -18\}$$

$$h(k) = \{2 \quad -3 \quad -10 \quad 3 \quad 3 \quad 13\}$$

- Convolve the two sequences either by hand or using the MATLAB command **conv**.
- Now convolve the two sequences using FFTs following the procedure outlined in Section 7.5. Verify that the result is indeed identical to the answer in part (a).
- What happens if you skip the zero-padding step when using FFTs to compute the convolution?

Problem 7.7: Create the signals x and y using the MATLAB commands below:

```
t=0:1/20:5; x = exp(-t); y = [zeros(1,10) x];
```

- Plot the signals x and y in MATLAB.
- Calculate the cross-correlation (normalized) for the two sequences using FFTs following the procedure outlined in Section 7.5. Don't print out the long cross-correlation vector. Instead, do a stem plot of the cross-correlation vector. In order to center the correlation results, plot the correlation vector vs. k as shown below (assuming the correlation results are stored under a vector called R):

```
k = -floor(length(R)/2):1:length(R)-floor(length(R)/2)-1;
stem(k,R)
```

By doing this, $R_{xy}(0)$ is located at $k = 0$ and this is the correlation value where x and y are lined up (not shifted) in time. $R_{xy}(-3)$ is located at $k = -3$ and this is the correlation value when y is shifted to the left by 3 sampling instants. Does the correlation vector show 100% correlation anywhere? Why?
- Compute the normalized cross-correlation between x and y using the MATLAB command **xcorr**. Plot the resulting vector as a stem plot following the same procedure as in part (b). Does the cross-correlation vector match the results of the FFT computation in part (b)?

Problem 7.8: Refer to Example 7.12.

- Using the MATLAB command **chirp**, create a linear FM chirp signal with a frequency that varies from 0 to 1 MHz over a pulse duration of 10 ms using a sampling frequency of 10 MHz. Calculate the matched filter coefficients for this signal then plot both the chirp signal and the matched filter coefficients. Compare the plots.

- (b) Compute and plot the filter output when the chirp signal created in part (a) is applied to the matched filter designed in part (a). Where does the peak occur in the output signal? Why?
- (c) Using the MATLAB command **chirp**, create a linear FM chirp signal with a frequency that varies from 0 to 250 kHz over a pulse duration of 10 ms using a sampling frequency of 10 MHz. Calculate the matched filter coefficients for this signal then compute and plot the filter output signal. How does this filter output compare to the output in part b?
- (d) Using the MATLAB command **chirp**, create a linear FM chirp signal with a frequency that varies from 0 to 250 kHz over a pulse duration of 40 ms using a sampling frequency of 10 MHz. Calculate the matched filter coefficients for this signal then compute and plot the filter output signal. How does this filter output compare to the outputs in part b and c?
- (e) For linear FM chirp signals, how does pulse duration affect the matched filter response? How does the frequency sweep (range of frequency variation) affect the matched filter response?

Problem 7.9: Zoom FFT

Refer to Figure 7.21 and Example 7.13. Assume the input signal is sampled at 48 kHz.

- (a) Calculate the spectral resolution for a 256-pt FFT based on the original signal.
- (b) Assume the frequency band of interest for the input signal ranges from 8 kHz to 11.5 kHz. What exponential sequence is required to down-shift this portion of the spectrum? What should the decimation factor for this signal be? Why? Calculate the spectral resolution for a 256-pt FFT based on the focused signal.
- (c) Using the Filter Design and Analysis tool, design an appropriate low-pass filter for the down-shifted input signal prior to decimation. It may be helpful to refer to the subsection on decimation in Chapter 5 (Figure 5.31). Export the filter coefficients to the MATLAB workspace.
- (d) Assume the input signal is $\sin(2\pi(9000)t) + \sin(2\pi(10100)t) + \sin(2\pi(10200)t)$ sampled at 48 kHz. Compute a 256-pt FFT on this signal and plot the magnitude of the FFT. Does the FFT adequately reflect the spectrum of the input signal? Why?
- (e) Now take the input signal in part d, downshift the signal by multiplying it by the exponential sequence determined in part b, filter the signal using the filter designed in part c, and then decimate the signal using **downsample**. Compute a 256-pt FFT on the “zoomed in” signal. Does the FFT adequately reflect the spectrum of the input signal? Why? *Note: if the signal has been down-shifted by 8 kHz then the frequencies in the “zoomed in” signal would be 8 kHz below the original signal frequencies.*

Problem 7.10: Refer to Example 7.14.

Create a 40 ms pulse with the frequencies corresponding to button 6 on the telephone keypad using a sampling frequency of $F_s = 8000$ Hz. Use the Goertzel algorithm to take 256 samples of the pulse signal and determine the frequency response at each of the seven discrete frequencies listed in Table 7.7. Plot the results and compare with a standard FFT.

CHAPTER 7 LAB EXERCISE

Fast Fourier Transforms (FFTs)

Objectives

1. To understand Discrete Fourier Transforms (DFTs) of signals: why the DFT is useful and how the DFT relates to the frequency spectrum of the signal.
2. To understand that a Fast Fourier Transform (FFT) is simply a highly efficient method for computing DFTs.
3. Learn to use MATLAB to compute DFTs of signals: experiment with sampling frequency and time duration in order to understand their effect on accuracy and resolution of the DFT.
4. Use the FFT block in Simulink® in order to find the spectrum of several different types of signals including signals buried in noise.
5. Explore the effects of windowing on the FFT.

Procedure

A. Using MATLAB to compute DFTs of Signals

Decaying exponential $x(t) = e^{-2t} \leftrightarrow X(f) = 1/(2 + j2\pi f)$

1. Plot the time signal $x(t)$ from 0 to 5 sec.
2. Plot the *magnitude* spectrum $X(f)$ from -5 Hz to +5 Hz using the following MATLAB commands:

```
f = -5:0.01:5;
mag_xf = abs(1./(2+2*pi*j*f));
plot(f,mag_xf)
```

3. Compute and plot the fft of signal $x(t)$ using a sampling frequency $F_s = 20$ Hz and a time duration $D = 4$ sec. Use the following MATLAB code to accomplish this:

```
Fs = 20; D = 4; N = Fs*D; % N=Number of time samples
t = 0: 1/Fs: (N-1)/Fs;    % Sampling times for x(t)
x = exp(-2*t);            % Sample values of x(t)
y1 = fft(x,N);            % Compute N-point FFT
y1 = fftshift(y1);        % Shift fft to center it at 0 Hz
fo = 1/D;                 % Spectral Resolution of FFT
% Set up frequency scale for FFT
f1 = -floor(N/2)*fo: fo: ceil(N/2-1)*fo;
stem(f1,1/Fs*abs(y1));    % Plot magnitude of DFT/Fs
title(' ... '); xlabel('Freq (Hz)'); grid
hold on; plot(f,mag_xf);  % Plot true spectrum
hold off;
```

Comment on the match between the DFT and the true frequency spectrum:

4. Repeat the previous step using a sampling frequency $F_s = 20$ Hz and a time duration, $D = 1$ sec. Comment on the match between the DFT and the true frequency spectrum:

5. Repeat the step 3 using a sampling frequency of 4 Hz and a time duration $D = 8$ seconds. Comment on the match between the DFT and the true frequency spectrum.

Another Decaying Exponential: $x(t) = te^{-3t} \leftrightarrow X(f) = 1/(3 + j2\pi f)^2$

1. Plot the time signal over some suitable time range.
2. Plot the spectrum over some suitable frequency range.
3. Vary duration, D , and sampling rate, F_s , and compute the DFT. Complete the table below to document your work. Also include plots.

Sampling Frequency, f_s	<i>Duration, D</i>	Number of Samples, N	<i>Resolution, f_o</i>	Frequency Range

What value of sampling frequency seemed to be sufficient for obtaining a DFT that accurately reflects the true signal spectrum? Why?

What duration, D , seemed sufficient for obtaining a DFT that accurately reflects the true signal spectrum? Why?

Obviously, the DFT becomes more accurate as Sampling Frequency and Duration are increased. What are the disadvantages of increasing sampling frequency and duration?

Sinusoidal Signal $x(t) = 10\cos(2\pi(2000)t)$

1. Sketch the spectrum of the analog signal in the space below.
2. Sample $x(t)$ for a duration of 5msec using a sampling rate of 10 kHz. Plot the magnitude of the FFT and comment on the match to the true spectrum of $x(t)$.
3. Repeat Step 2 using a sampling rate of 3 kHz for a duration of 5 msec. Does the FFT accurately estimate the spectrum? Why or why not?

4. Repeat Step 2 using a sampling rate of 10 kHz for a duration of 5.2 msec. Does the FFT accurately estimate the signal spectrum? Why or why not?

5. Repeat Step 2 using a sampling rate of 10 kHz for a duration of 20.2 msec. How does this spectrum compare with the previous case?

B. Effects of Windowing on FFT

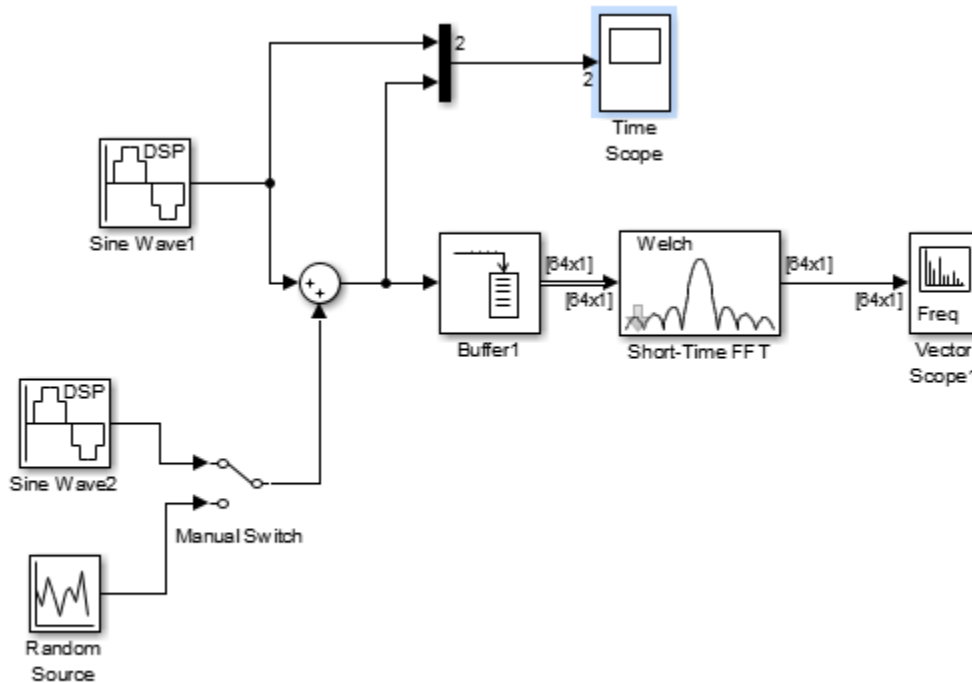


Figure 1: FFTs using SIMULINK

1. Build the system shown in Figure 1 in Simulink. Most of the blocks are in the DSP Blockset.
2. Set up the first sine wave with an amplitude of 1 and a frequency of 100 Hz with a sample time of 0.001 (i.e., sampling frequency is 1000 Hz).
3. Set up the second sine wave with an amplitude of 0, a frequency of 125 Hz, and a sample time of 0.001 (i.e., sampling frequency is 1000 Hz).

4. Set the Buffer Size to 64 and treat Mx1 and unoriented sample-based signals as One Channel.
5. Set up the ST-FFT with a BoxCar window and check inherit FFT length from input block.
6. Set the Vector Scope up for Frequency (under scope properties). Set the minimum y-limit to -40dB and the maximum y-limit to 20dB (under axis properties). These can be re-scaled if necessary or when the scope plot is open simply choose Axes – Autoscale.
7. In the Random Source Block, set the source to Gaussian and the sample time to 0.001.
8. Run the simulation. Since the input is a 100 Hz sine wave, the spectrum should show a resonant peak at 100 Hz.
9. Increase the buffer size (size of FFT). What is the effect on the spectrum?
10. Set the buffer size to 128. Try 2 or 3 other types of windows. Describe in detail the effect on the spectrum for each of the windows you chose.
11. Change the amplitude of the 2nd sine wave to 1. Leave the frequency at 125 Hz and the sampling frequency at 1000 Hz. Ideally, the spectrum should show distinct spikes at 100 Hz and 125 Hz. Vary the buffer size (FFT size) and the type of window. Explain in detail the effects on the spectrum.
12. Switch out the 2nd sine wave and switch in the Random Noise Source. Again, vary both the buffer size and the type of window to explore the effects on the spectrum. Be sure to look at the time scope plot. Is it possible to pull the original sine wave out of the noisy signal?

In many applications including digital communications, spectral analysis, audio processing, and radar processing, data is received and must be processed in real-time. In some applications, the incoming data sequence is very large and simply exceeds the memory capacity of the processor. In other applications, the time delay associated with waiting for all the data to arrive before processing is simply unacceptable. Processing data in real-time involves breaking the input data sequence into blocks of data and processing each block of data separately. The chapter begins with the implementation of difference equations in the time domain and the use of circular addressing. Next, Overlap-Add and Overlap-Save, two algorithms for real-time convolution or correlation computations using FFTs, are presented. Block processing of input data for real-time spectral analysis using FFTs is covered in Section 8.3. The Sliding Discrete Fourier Transform, which is a real-time implementation of a Discrete Fourier Transform, is explored in the fourth section. In the final section, adaptive filters for noise and echo cancellation and system identification are introduced.

8.1 DIFFERENCE EQUATIONS AND CIRCULAR ADDRESSING

Difference equations used for FIR filters, IIR filters, and the Goertzel algorithm naturally lend themselves to real-time processing. Consider the difference equation for an FIR filter introduced in Chapter 5:

$$y(k) = \sum_{i=0}^N b_i x(k-i) \quad (8.1)$$

The output at any time kT_s is computed by simply multiplying the current input and the N most recent past input values by the filter coefficients and summing up the results. At any sampling time instant, the only data values that need to be stored in memory are the current input and N past inputs. In the next sampling instant, the oldest data value will be discarded and replaced with the newest input data value. Real-time implementation of the filter becomes a matter of taking in a new input value at each sampling instant, discarding the oldest value, then performing the $N + 1$ multiplications and N additions to calculate the next output. Of course, the processor must be fast enough to perform the calculations within one sampling instant and the input signal itself must be sampled fast enough to maintain fidelity with the original analog signal. IIR filters require storage of a finite number of past outputs in addition to current and past inputs but the basic principle is the same. The Goertzel algorithm, discussed in Chapter 7 for determining the spectrum at a finite number of select frequencies, is simply a 2nd order difference equation requiring two past outputs and the current input.

The input data values and past output values (if needed), can be stored and accessed using circular addressing. The required data is stored in a buffer of fixed length as shown in Figure 8.1. When the new data value comes in, instead of shifting all the data in memory which would use up valuable processing time, the newest value simply replaces the oldest value with a pointer showing where the newest value is located. As multiplications are performed, the pointer moves through the buffer and “circles” or wraps around to the top when the bottom of the buffer is reached.

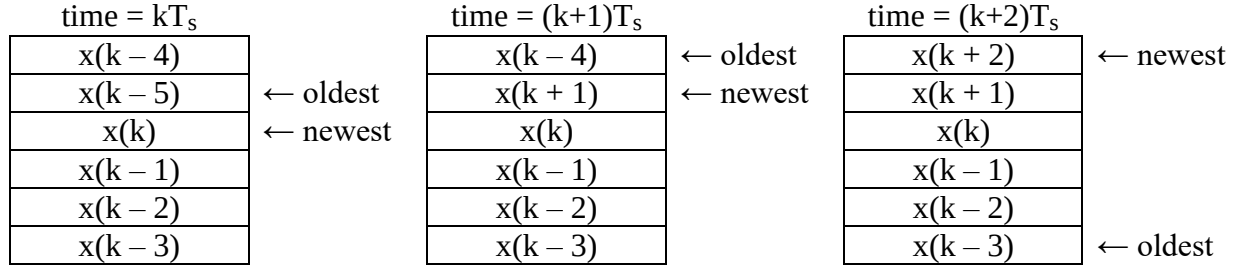


Figure 8.1: Circular Addressing

If the filter order is very large, processing in the time domain is not as efficient as frequency domain processing. As discussed in Chapter 7, FFTs can be used to compute the filter output efficiently for high order filters and long sequences of input data. The next section presents the overlap add and overlap save algorithms for real-time convolution or correlation using blocks of input data and FFTs.

8.2 OVERLAP-ADD AND OVERLAP-SAVE ALGORITHMS FOR CONVOLUTION OR CORRELATION

Overlap-add and overlap-save are useful algorithms for real-time computation of the convolution or correlation of two sequences using FFTs. One practical example of this would be using a digital FIR filter to filter a long sequence of input data. In this case, the filter output is the convolution of the input data and the filter coefficients. Rather than waiting for all the input data to arrive, the convolution can be processed in blocks. Recall from Chapter 7 that if the filter has an order $N \geq 128$, it is computationally efficient to use FFTs to compute the convolution (fast convolution) instead of working in the time domain. Another practical application would be to correlate an incoming stream of data with a template of reference signals for voice identification, finger print identification, radar target identification, or other purposes. The correlation can be computed in real-time using FFTs as blocks of input data become available.

Overlap-Add Algorithm

Assume the data input sequence, $x(k)$ has length L , the filter impulse response, $h(k)$, has length N , and $L \gg N$. The procedure to compute $y(k) = x(k) * h(k)$ is as follows:

1. Partition $x(k)$ into blocks of length N by simply using data buffers of length N . Call these partitioned sequences $x_0, x_1, x_2, \dots$
2. Convolve each block of input data, as it becomes available, with $h(k)$ using fast convolution (FFTs):

$$y_i = \text{ifft} [\text{fft}(x_i^{\text{padded}}) \cdot \text{fft}(h_i^{\text{padded}})]$$

Note that the block of input data and the filter coefficients must be padded with zeros so each have length $2N - 1$. The result of each block convolution will be an output sequence also of length $2N - 1$. Call these blocks of results $y_0, y_1, y_2, \dots$

3. The output sequence is then constructed from the individual convolution results by

$$y(k) = y_0(k) + y_1(k - N) + y_2(k - 2N) + \dots$$

The over-lap add algorithm is illustrated in Example 8.1 using very simple and short sequences.

Example 8.1: Computing Convolution using Overlap Add

Suppose $x(k) = [-1 \ 4 \ 2 \ 5 \ 1 \ 3 \ 7]$ and $h(k) = [1 \ 3]$. Compute $y(k) = x(k) * h(k)$ using the overlap add algorithm and check the results using the MATLAB[®] function, **conv**.

Solution

1. First partition $x(k)$ into sequences of length $N = 2$:

$$x_0 = [-1 \ 4] \quad x_1 = [2 \ 5] \quad x_2 = [1 \ 3] \quad x_4 = [7 \ 0]$$

2. Now compute each individual block convolution (since this example is being done by hand with very short sequences, the convolution will be computed in the time domain rather than using FFTs):

$$\begin{aligned} y_0 &= x_0 * h(k) = [-1 \ 1 \ 12] \\ y_1 &= x_1 * h(k) = [2 \ 11 \ 15] \\ y_2 &= x_2 * h(k) = [1 \ 6 \ 9] \\ y_3 &= x_3 * h(k) = [7 \ 21 \ 0] \end{aligned}$$

3. Now construct $y(k)$ from the individual block convolutions:

$$\begin{array}{cccccccc} & & [-1 & 1 & 12] & & & \\ & & & [2 & 11 & 15] & & \\ & & & & [1 & 6 & 9] & \\ & & & & & [7 & 21 & 0] \\ \hline y(k) = & [-1 & 1 & 14 & 11 & 16 & 6 & 16 & 21 & 0] \end{array}$$

Checking the result with MATLAB yields the same result:

$$\begin{aligned} x &= [-1 \ 4 \ 2 \ 5 \ 1 \ 3 \ 7]; \quad h = [1 \ 3]; \quad y = \text{conv}(x, h) \\ y &= \quad -1 \quad \quad 1 \quad \quad 14 \quad \quad 11 \quad \quad 16 \quad \quad 6 \quad \quad 16 \quad \quad 21 \end{aligned}$$

Overlap-Save Algorithm

Assume the data input sequence, $x(k)$ has length L , the filter impulse response, $h(k)$, has length N , and $L \gg N$. The procedure to compute $y(k) = x(k) * h(k)$ is as follows:

1. Add $N - 1$ leading zeros to the input data sequence, $x(k)$.
2. Partition the input data sequence (with the added zeros) into segments that overlap by $N - 1$. Choose a segment length $M \approx 2N$ which is preferably a power of 2. Call these partitioned sequences $x_0, x_1, x_2, \dots$
3. Zero pad $h(k)$ using trailing zeros so that it is length M .
4. Perform *periodic* convolutions on the blocks of data (as they become available in real time) and the padded $h(k)$ using FFTs. This will yield sequences $y_0, y_1, y_2, \dots$ of length M :

$$y_i = \text{ifft} [\text{fft}(x_i) \cdot \text{fft}(h_i^{\text{padded}})]$$

5. Discard the first $N - 1$ samples from each of the individual convolutions, then paste the resulting segments together to form the output, $y(k)$.

Example 8.2: Computing Convolution using Overlap Save

Suppose $x(k) = [-1 \ 4 \ 2 \ 5 \ 1 \ 3 \ 7]$ and $h(k) = [1 \ 3]$. Compute $y(k) = x(k) * h(k)$ using the overlap save algorithm and compare the results to Example 8.1.

Solution

In this example, $L = 7$ and $N = 2$.

1. First, add $N - 1 = 1$ leading zero to the input sequence:

$$x = [0 \ -1 \ 4 \ 2 \ 5 \ 1 \ 3 \ 7]$$

2. Next, partition x into segments of length $M = 4$ ($\approx 2N$ and a power of 2) that overlap by $N - 1 = 1$:

$$x_0 = [0 \ -1 \ 4 \ 2] \quad x_1 = [2 \ 5 \ 1 \ 3] \quad x_2 = [3 \ 7 \ 0 \ 0]$$

3. Pad $h(k)$ with trailing zeros to make it length $M = 4$:

$$h = [1 \ 3 \ 0 \ 0]$$

4. Convolve the sections using periodic convolution (regular convolution with wraparound to make length = 4). Once again, since the sequences are so short for this example, computations will be done in the time domain instead of using FFTs:

$$x_0 * h = [0 \ -1 \ 1 \ 14] \rightarrow y_0 = x_0 \bullet h = [6 \ -1 \ 1 \ 14]$$

$$x_1 * h = [2 \ 11 \ 16 \ 6] \rightarrow y_1 = x_1 \bullet h = [11 \ 11 \ 16 \ 6]$$

$$x_2 * h = [3 \ 16 \ 21 \ 0 \ \underline{0 \ 0 \ 0}] \rightarrow y_2 = x_2 \bullet h = [3 \ 16 \ 21 \ 0]$$

5. Discard the first $N - 1 = 1$ samples from each section then past directly together to get the output:

$$y(k) = [-1 \ 1 \ 14 \ 11 \ 16 \ 6 \ 16 \ 21 \ 0]$$

The results of this example are identical to the results in example 8.1.

Overlap add and overlap save are both efficient algorithms for computing a convolution or correlation in real-time. How do the two algorithms compare?

- The overlap save algorithm does not require that any of the previous convolution results be saved since the output is determined simply by pasting individual sections together. However, it does require that some of the input data from the previous partition be saved and used in the next data segment and convolution calculation.
- The overlap add algorithm requires that one previous convolution result be saved and combined with the current convolution result. However, once a block of input data is used for convolution, it can be discarded.
- The convolution step in both overlap add and overlap save uses FFTs (fast convolution). Overlap save is set up beautifully for fast convolution since the segment convolutions are periodic convolutions. Overlap add requires zero padding of the input segments and filter coefficients following the procedure outlined in Chapter 7 on fast convolution.
- For both methods, the FFT of the padded filter coefficients, $h(k)$, can be computed ahead of time and stored in memory.
- Both methods have about the same computational complexity – the number of real multiplications is roughly

$$\frac{L}{M} \cdot M^2 \quad \text{using time domain convolution}$$

$$L/M [12\bar{N}\log_2(2\bar{N}) + 8\bar{N}] \quad \text{using FFTs for the convolution} \quad (8.2)$$

where L is the length of the input data sequence, $x(k)$, N is the length of $h(k)$, M is the length of the partitions of $x(k)$, and $\bar{N}$ is the smallest power of 2 that is greater than or equal to M . Compare this equation to the number of multiplies for fast convolution summarized in Section 7.5. The only difference is the pre-multiplier of L/M which is simply the number of partitions of the input sequence, $x(k)$, or equivalently the number of individual convolutions that need to be performed.

8.3 REAL-TIME SPECTRAL ANALYSIS USING FAST FOURIER TRANSFORMS

Analyzing the spectrum of a signal in real-time using Fast Fourier Transforms is accomplished by collecting blocks of input data samples and performing an FFT on each block of data. The process is illustrated in Figure 8.2.

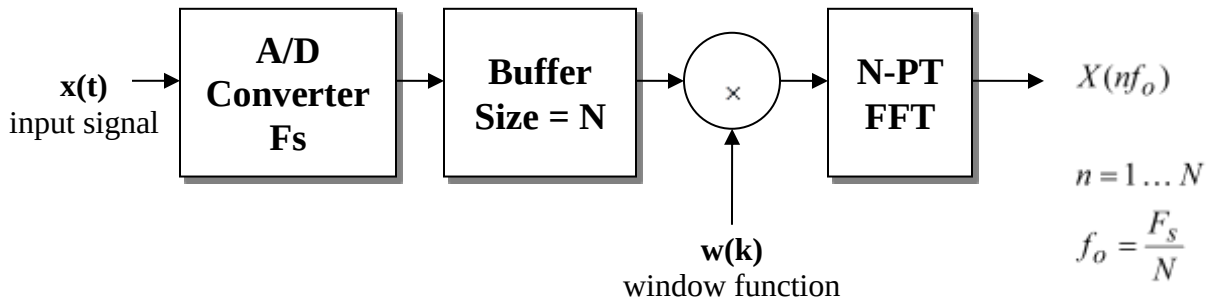


Figure 8.2: Real-Time FFT Processing

As indicated in Figure 8.2, the input signal is sampled and these samples are stored in a buffer of size N . Once the buffer is full, an FFT is computed for the block of N data samples. As discussed in Section 7.2, the input data samples are windowed prior to computing the N -PT FFT to enhance the spectral analysis.

There are several important considerations in performing real-time FFT processing:

- The sampling rate, F_s , must of course be chosen to exceed twice the bandwidth of the input signal to avoid aliasing.
- The duration of time, D , over which the input signal is sampled for each FFT computation must be chosen carefully. Increasing D will improve the resolution of the FFT since $f_o = 1/D$. However, if the time duration is chosen too long, then changes in the spectrum of input signal could be missed. The time duration, D , directly affects the size of the FFT since $N = D \cdot F_s$ which means a longer time duration results in a larger FFT.
- The choice of a window function is application dependent. As discussed in Section 7.2, windowing can be used to reduce the effects of leakage. Each window function has its own somewhat unique characteristics which affect how the leakage is re-distributed.
- Overlapping the blocks of input data can be very beneficial to the spectral analysis, particularly when windowing is included. In other words, if the block size is 1024, the last 256 input samples of a block could become the first 256 samples of the next block.

As mentioned above, the choice of time duration for each block of input data has a big effect on the resulting FFT. If the time duration is too long, then short burst changes in the input signal could be missed (poor time resolution). Decreasing the time duration, D , will certainly improve the time resolution but will definitely degrade the frequency resolution. Example 8.3 illustrates this trade-off between time resolution and frequency resolution.

Example 8.3: Trade-off Between Time Resolution and Frequency Resolution

(a) Suppose the input signal is $x_1(t) = \begin{cases} \sin(2\pi(100)t) & 0s \leq t < 0.75s \\ \sin(2\pi(300)t) & 0.75s \leq t < 1.25s \\ \sin(2\pi(100)t) & 1.25s \leq t \leq 2.0s \end{cases}$

The input signal is sampled at $F_s = 1000$ Hz (plenty fast) and the buffer length (FFT length) is chosen to be $N = 256$. The time duration, D , would therefore be 0.256 sec and the frequency resolution would be $1000/256 \approx 4$ Hz. Using Simulink® and MATLAB, plot the FFT of $x_1(t)$ using block processing.

(b) Now change the input signal to $x_2(t) = \begin{cases} \sin(2\pi(100)t) & 0s \leq t < 0.95s \\ \sin(2\pi(300)t) & 0.95s \leq t < 1.05s \\ \sin(2\pi(100)t) & 1.05s \leq t \leq 2.0s \end{cases}$

Use the same sampling frequency and FFT length from part (a), plot the FFT of $x_2(t)$ using block processing, and comment on the results.

(c) Repeat part (b) with the FFT length reduced to 128. Reducing the FFT length from 256 to 128 will cut the time duration in half ($D = 0.128$ sec) and double the frequency resolution ($f_0 = 1000/128 = 7.8125$ Hz).

(d) Repeat part (b) with the FFT length reduced to 64. In this case, the time duration is further reduced to 0.064 sec while the frequency resolution increases to 15.625 Hz.

Solution

A Simulink model (Figure 8.3) is built for the block processing of the input signal and FFT computation.

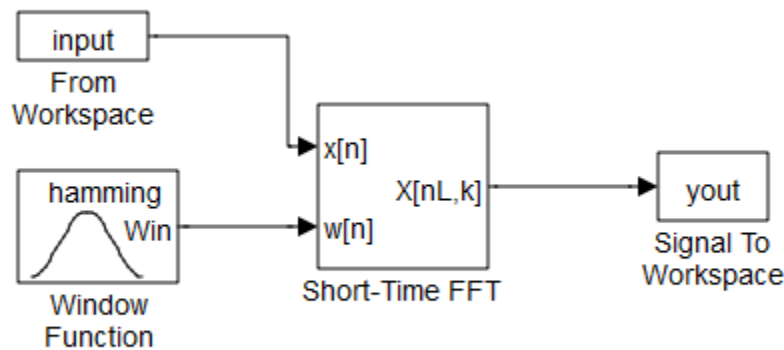


Figure 8.3: Simulink Model for Block Processing of FFT

In the Window Function block, the window length is set equal to the FFT length, sample mode is Discrete, and sample time is 1/Fs. In the Short-Time FFT block, the analysis window length is set equal to the FFT length and the overlap is set to 0. The sampling frequency is specified in the From Workspace block.

- (a) The following commands in MATLAB define the input signal then plot the results of running the Simulink model

```
Fs=1000; N = 256;
t=0:1/Fs:2;          % Sampling times
f=-N/2*Fs/N:Fs/N:(N-1)/2*Fs/N; % Frequency bins for FFT
% Define input signal, x, to be a 300 Hz sine wave from 0.75 to
% 1.25 sec and 100 Hz at all other times
x=sin(2*pi*100*t);
x(750:1250)=sin(2*pi*300*t(750:1250));
input=[t' x'];
[K,F]=meshgrid(t,f); % Set up mesh grid for 3-D plot
```

Now run FFT Block Simulation (stop time should be set to 2 sec) then execute the following commands in the MATLAB workspace

```
yfft(:, :)=yout(:, 1, :);
mesh(F,K,abs(fftshift(yfft,1)))
```

Comments on Part (a):

The resulting FFT spectrum is shown in Figure 8.4 (a). It is clear from the plot that the input initially has a frequency component close to 100 Hz, the spectrum then switches to approximately 300 Hz, and then returns to 100 Hz. Note that there is a time delay of approximately 0.25 sec in the spectrum. This is the length of time required to fill the buffer with 256 input values before the processing can begin.

Challenge Question 8.1

Close examination of the plot in Figure 8.4(a) reveals a very small peak at 300 Hz prior to the much larger peak and a very small peak at 100 Hz in between the two much larger peaks. Why does this occur?

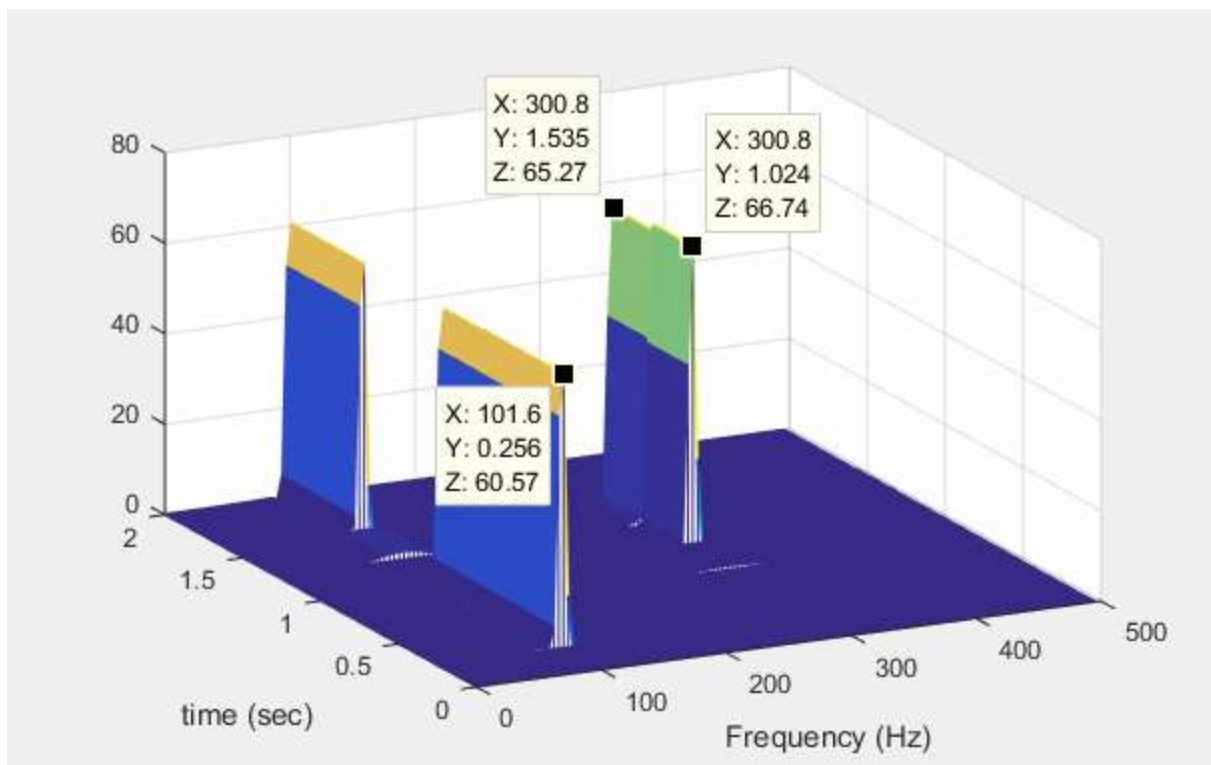


Figure 8.4a: Spectrum for Example 8.3(a)

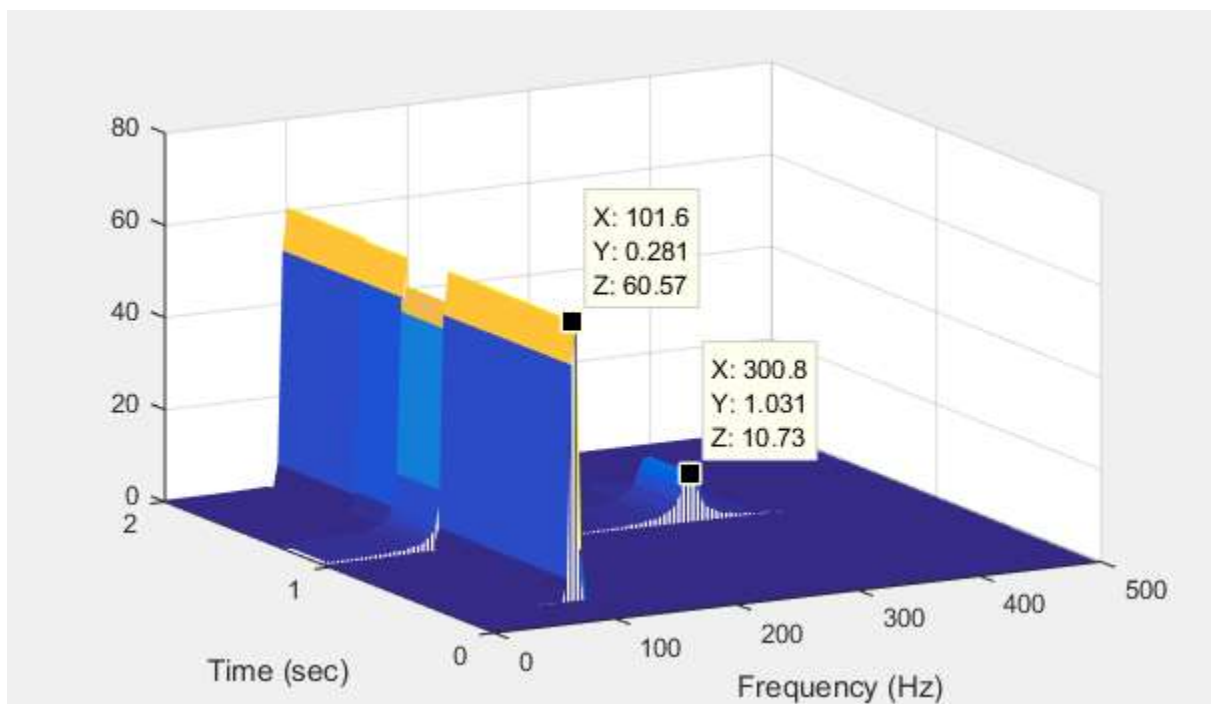


Figure 8.4b: Spectrum for Example 8.3(b)

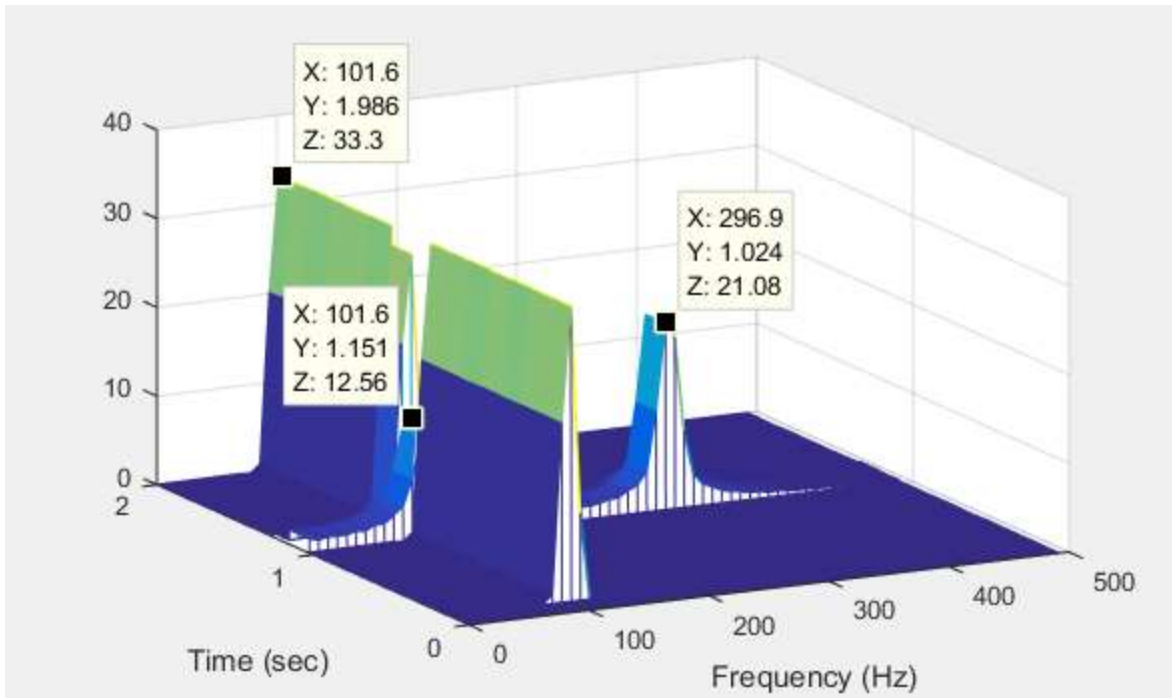


Figure 8.4c: Spectrum for Example 8.3(c)

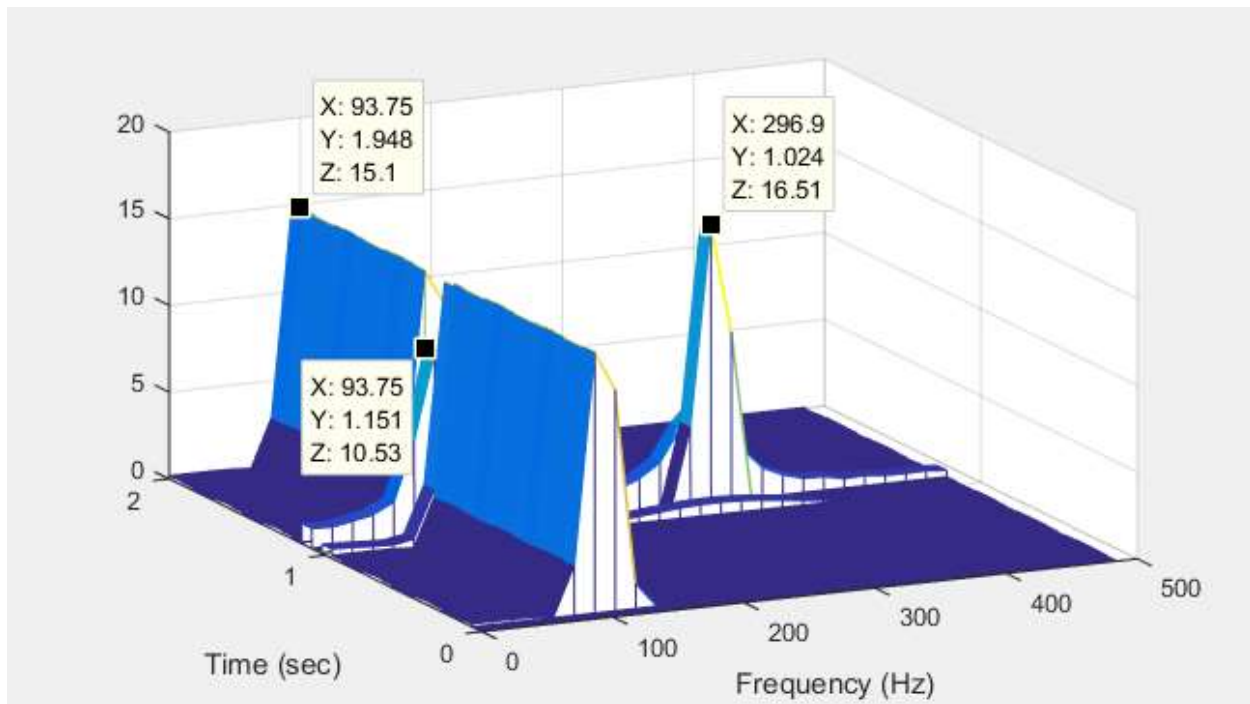


Figure 8.4d: Spectrum for Example 8.3(d)

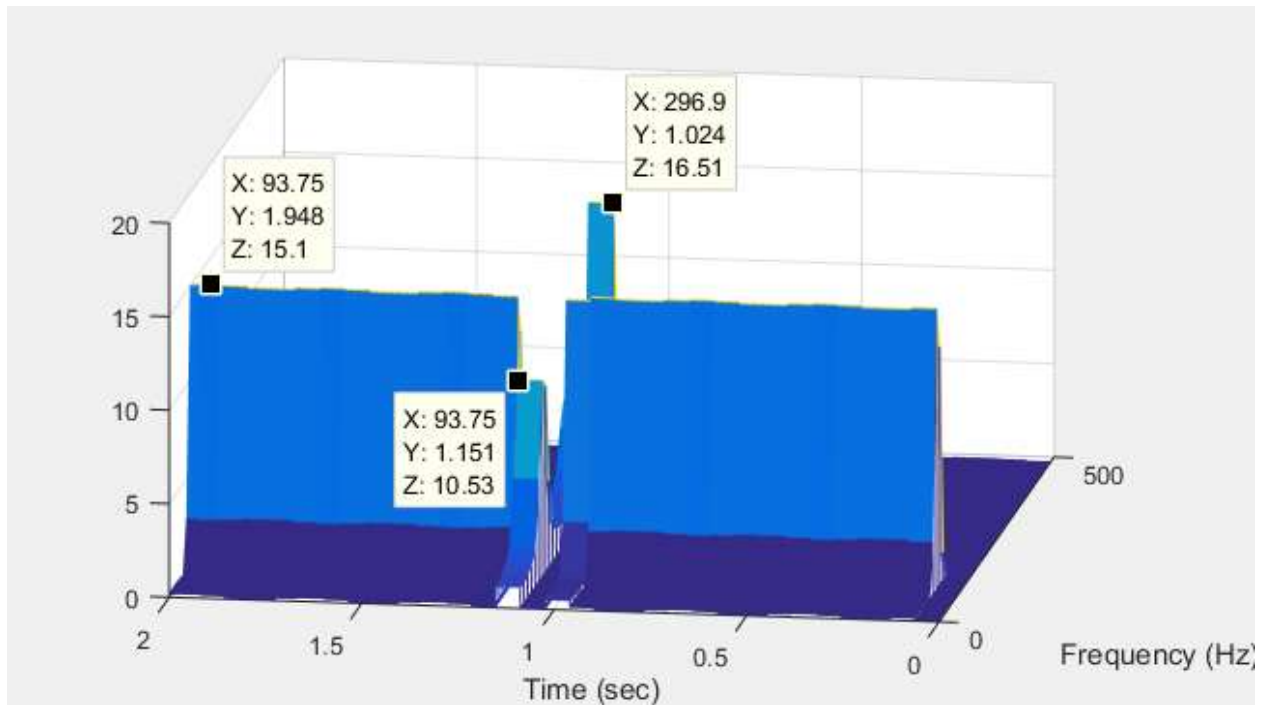


Figure 8.4e: Spectrum for Part (d) – Rotated for a Different Perspective

Figure 8.4: Spectral Plots for Example 8.3

- (b) The following commands in MATLAB define the input signal then plot the results of running the Simulink model

```
x=sin(2*pi*100*t);
x(950:1050)=sin(2*pi*300*t(950:1050));
input=[t' x'];
```

Re- run FFT Block Simulation then run the following commands in MATLAB

```
Clear yfft; yfft(:, :)=yout(:, 1, :);
mesh(F,K,abs(fftshift(yfft,1)))
```

Comments on Part (b):

The resulting FFT spectrum is shown in Figure 8.4(b). Notice that a small peak at 300 Hz does appear in the spectrum; however, during the exact same period of time there is a much larger peak occurring at approximately 100 Hz. The fact that the input signal is a pure 300 Hz wave between 0.95 sec and 1.05 sec is completely lost because the duration of time, D , over which blocks of input data are being analyzed is much longer than the period of time during which the input signal hits 300 Hz. In other words, the buffer of input signal samples does include samples when the input is a 300 Hz tone, but it also includes a lot more samples of the 100 Hz tone. Parts (c) and (d) of this example are an attempt to fix this problem by reducing the length of time over which the input signal is sampled.

(c) The following commands in MATLAB adjust the buffer length to 128

```
% Change N to 128 and adjust frequency bins and meshgrid:
Fs=1000; N = 128;
f=-N/2*Fs/N:Fs/N:(N-1)/2*Fs/N; % Frequency bins for FFT
[K,F]=meshgrid(t,f); % Set up mesh grid for 3-D plot
```

Change the window lengths and FFT length in FFT Block Simulation to 128. Re-run FFT Block Simulation then run the following commands in MATLAB

```
Clear yfft; yfft(:, :)=yout(:, 1, :);
mesh(F,K,abs(fftshift(yfft,1)))
```

Comments on Part (c):

The resulting FFT spectrum is shown in Figure 8.4(c). The 300 Hz signal is much more distinct in this spectrum; however, the spectrum still shows a 100 Hz signal (albeit much smaller) occurring in the same time period. Also notice the widening of the peaks due to poorer frequency resolution.

(d) The following commands in MATLAB adjust the buffer length to 64

```
%Change N to 64 and adjust frequency bins and meshgrid:
Fs=1000; N = 64;
f=-N/2*Fs/N:Fs/N:(N-1)/2*Fs/N; % Frequency bins for FFT
[K,F]=meshgrid(t,f); % Set up mesh grid for 3-D plot
```

Change the window lengths and FFT length in FFT Block Simulation to 64. Re-run FFT Block Simulation then run the following commands in MATLAB

```
Clear yfft; yfft(:, :)=yout(:, 1, :);
mesh(F,K,abs(fftshift(yfft,1)))
```

Comments on Part (d):

The resulting FFT spectrum is shown in Figure 8.4(d) and (e). The time resolution has been improved enough to show the 300 Hz component occurring alone for a small period of time. Notice the degradation in the frequency resolution. There is now much more uncertainty in the actual frequency of the two tones.

Example 8.3 illustrates the effect of time duration on a real-time FFT. The next example shows that overlapping the blocks of input data can be beneficial for real-time spectral analysis.

Example 8.4: Overlapping Blocks of Data for FFT

(a) Suppose the input signal is $x_1(t) = \begin{cases} \sin(2\pi(100)t) & 0s \leq t < 0.5s \\ \sin(2\pi(300)t) & 0.5s \leq t < 0.75s \\ \sin(2\pi(100)t) & 0.75s \leq t \leq 2.0s \end{cases}$

Using Simulink and MATLAB, plot the FFT of $x_1(t)$ using block processing. Assume a sampling frequency of 1000 Hz and an FFT length of 256.

(b) Now change the input signal to $x_2(t) = \begin{cases} \sin(2\pi(100)t) & 0s \leq t < 0.64s \\ \sin(2\pi(300)t) & 0.64s \leq t < 0.89s \\ \sin(2\pi(100)t) & 0.89s \leq t \leq 2.0s \end{cases}$

Use the same sampling frequency and FFT length from part (a), plot the FFT of $x_2(t)$ using block processing, and comment on the results.

(c) Repeat part (b) but overlap the input data by 128 so that every block of 256 data points consists of 128 data points from the previous FFT calculation and 128 new data points.

(d) Repeat part (c) using an overlap of 192 so that every block of 256 data points consists of 192 data points from the previous FFT calculation and 64 new data points.

Solution

The Simulink model used for Example 8.3 (see Figure 8.3) is also used for this example.

(a) The MATLAB commands for part (a) are identical to those of Example 8.3(a) with the following exception:

```
x=sin(2*pi*100*t);  
x(500:750)=sin(2*pi*300*t(500:750));
```

(b) Change the input signal to $x_2(t)$ using the following MATLAB commands, re-run the simulation, then plot the results:

```
x=sin(2*pi*100*t);  
x(640:890)=sin(2*pi*300*t(640:890));  
input=[t' x'];
```

Re- run FFT Block Simulation then run the following commands in MATLAB

```
Clear yfft; yfft(:, :)=yout(:, 1, :);  
mesh(F,K,abs(fftshift(yfft,1)))
```

(c) In the Simulink model, open up the Short-Time FFT block and change the overlap to 128. Re-run the simulation model and plot the results.

- (d) In the Simulink model, open up the Short-Time FFT block and change the overlap to 192. Re-run the simulation model and plot the results.

The spectral plots for this example are shown in Figure 8.5 (a)-(d).

Comments on results of Example 8.4:

- (a) As shown in Figure 8.5(a), the spectral plot for part (a) is a very nice representation of the spectrum of the signal $x_1(t)$. It clearly shows that the signal is initially a 100 Hz tone, switches to a 300 Hz tone, and then reverts back to a 100 Hz tone.
- (b) The only difference between the signals $x_1(t)$ and $x_2(t)$ is the interval of time during which these signals become 300 Hz sine waves. The signal $x_1(t)$ is a 300 Hz sine wave from 0.5 – 0.75 sec while the signal $x_2(t)$ is a 300 Hz sine wave from 0.64 – 0.89 sec. Even though both signals have a 300 Hz component for exactly the same length of time, both signals were sampled at exactly the same rate, and the same length FFT was used in both cases, the spectrum for $x_2(t)$ shown in Figure 8.5(b) is quite different from the spectrum of $x_1(t)$. The spectrum in Figure 8.5(b) would indicate that $x_2(t)$ begins as a 100 Hz tone, then becomes a two tone signal (100 Hz and 300 Hz), and finally reverts back to a 100 Hz tone. What causes the big difference in results between parts (a) and (b)? To answer this question, consider the windows of time over which the FFT is being calculated and what the input signal samples look like within these windows of time. This information is shown in Table 8.1. In the case of $x_1(t)$, most of the samples of the 300 Hz section of the signal fall into the same buffer and just about fill it. In the case of $x_2(t)$, samples of the 300 Hz section get split between two adjacent buffers and fill only about half the buffer. The other half of the buffer is filled with samples of the 100 Hz section. Thus, the FFT calculation for these buffers indicates a two tone signal rather than isolating the 300 Hz tone.
- (c) In part (c), an overlap of 128 is introduced for the FFT processing. Each sampling window consists of 128 sample values from the previous window and 128 new sample values. This creates overlapping windows of time, but does not reduce the size of the FFT so the frequency resolution remains the same. As shown in Table 8.2 and also reflected in the spectral plot in Figure 8.5(c), this overlap allows the 300 Hz tone to be effectively isolated during one sampling window. The spectrum still indicates a two tone signal in the adjacent sampling windows.
- (d) In part (d), the overlap is increased to 192 which means each input buffer consists of 192 past sample values and 64 new sample values. The resulting spectrum plotted in Figure 8.5(d) shows better time resolution as the signal transitions from 100 Hz to 300 Hz then back to 100 Hz.

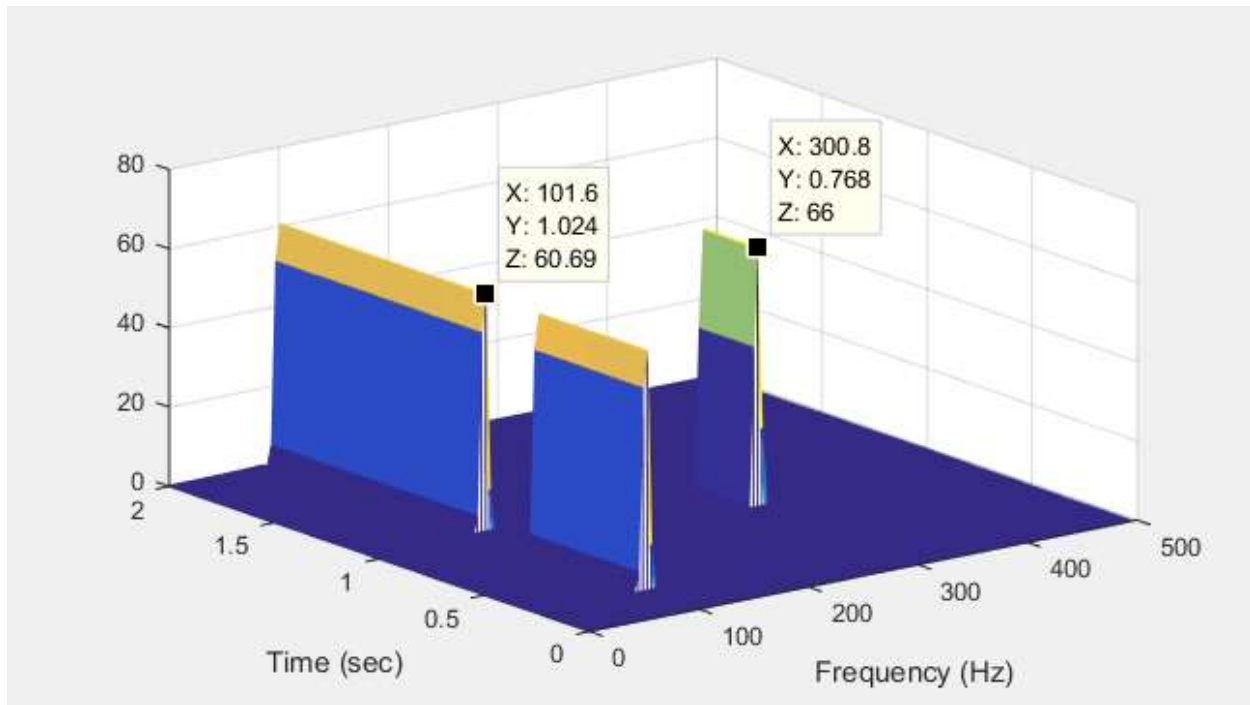


Figure 8.5a

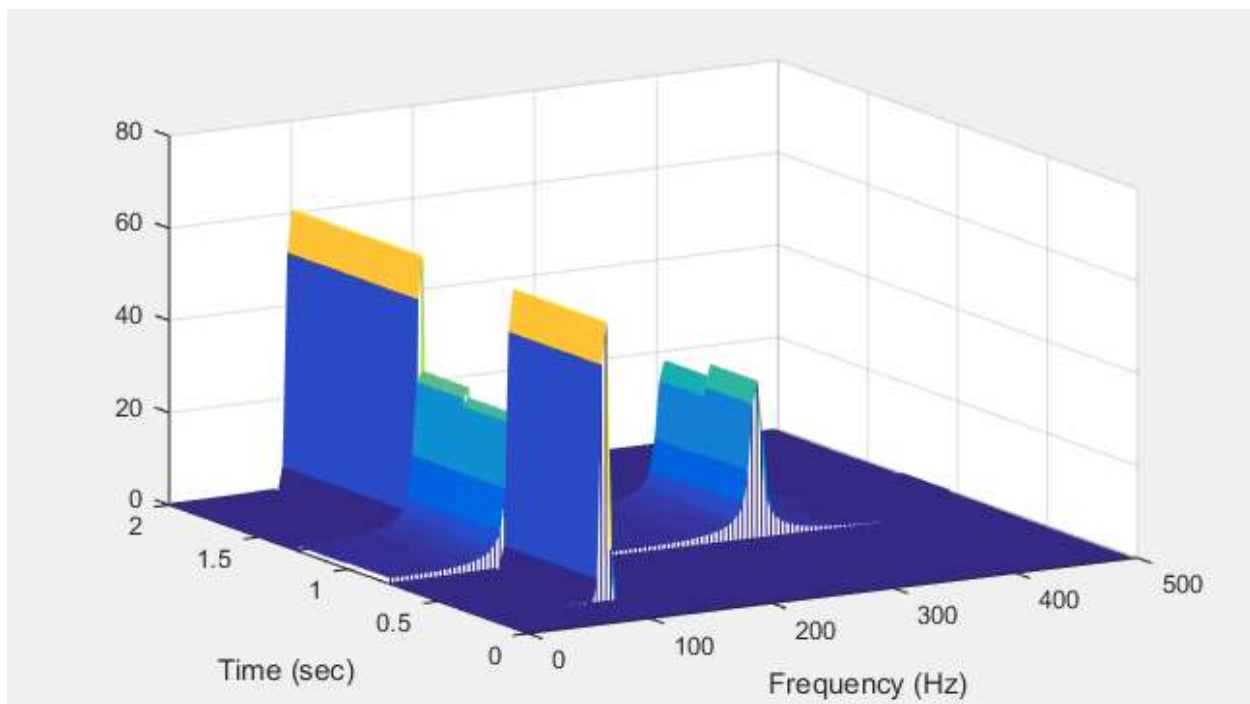


Figure 8.5b

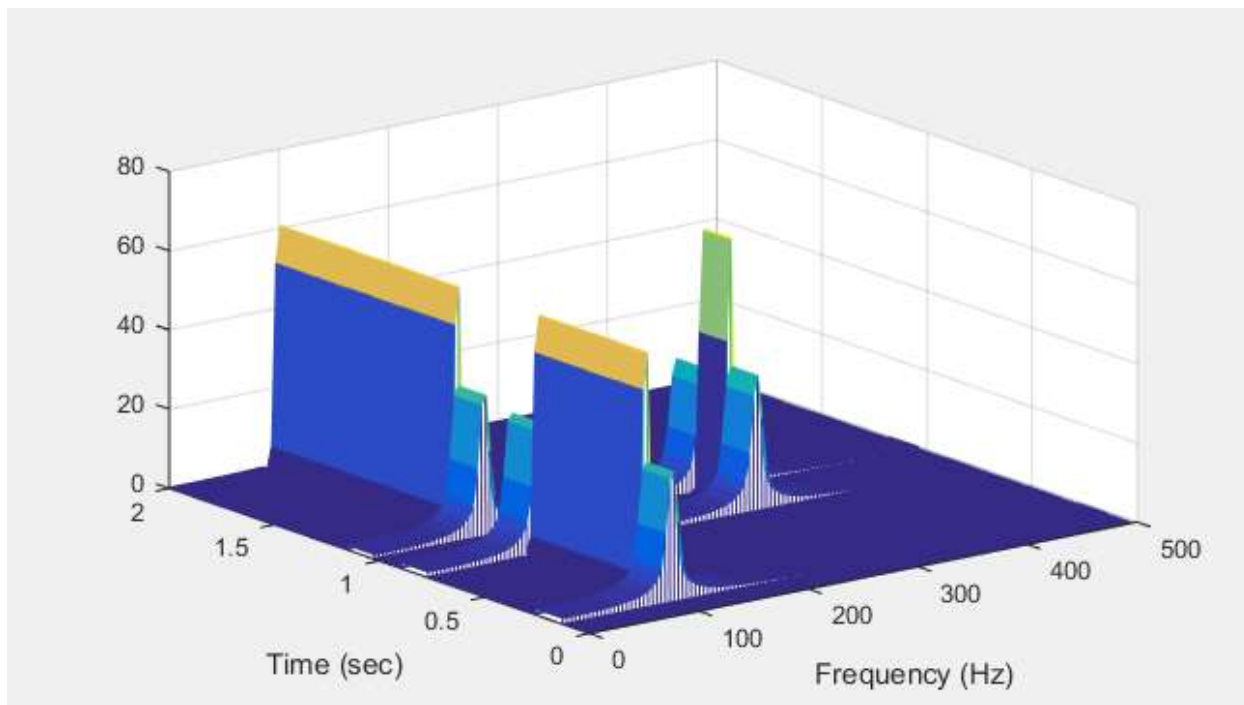


Figure 8.5c

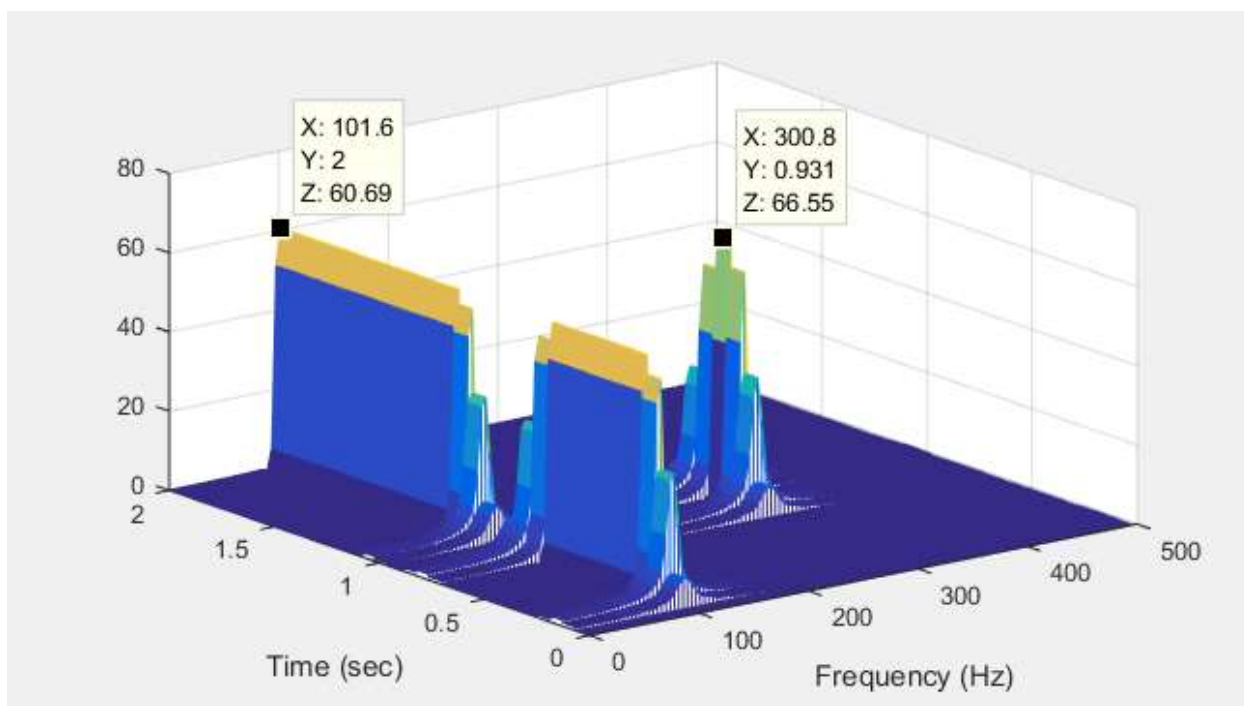


Figure 8.5d

Figure 8.5: Spectral Plots for Example 8.4

Table 8.1: Sample Intervals and Input Signal Content for Example 8.4(a)-(b)

Time Interval for FFT	$x_1(t)$		$x_2(t)$	
	Number of 100 Hz Samples	Number of 300 Hz Samples	Number of 100 Hz Samples	Number of 300 Hz Samples
0 – 0.256 s	256	0	256	0
0.256 – 0.512 s	244	12	256	0
0.512 – 0.768 s	18	238	128	128
0.768 – 1.024 s	256	0	134	122
1.024 – 1.28 s	256	0	256	0
1.28 – 1.536 s	256	0	256	0
1.536 – 1.8 s	256	0	256	0
1.8 – 2.046 s	256	0	256	0

Table 8.2: Sample Intervals and Input Signal Content for Example 8.4 (c)

Time Interval for FFT	$x_2(t)$	
	Number of 100 Hz Samples	Number of 300 Hz Samples
0 – 0.256 s	256	0
0.128 – 0.384 s	256	0
0.256 – 0.512 s	256	0
0.384 – 0.64 s	256	0
0.512 – 0.768 s	128	128
0.640 – 0.896 s	6	250
0.768 – 1.024 s	134	122
0.896 – 1.152 s	256	0
1.024 – 1.28 s	256	0
Remaining Intervals	256	0

Challenge Question 8.2

Example 8.4 shows that increasing the overlap of input sample values for spectral analysis results in a better reflection of how the signal spectrum varies over time. What is the disadvantage of increasing the overlap?

8.4 SLIDING DISCRETE FOURIER TRANSFORM

The sliding DFT is a real-time implementation of a DFT. The sliding DFT computes an N-point spectrum at each sampling instant; that is, every time a new data sample is taken. This would be equivalent to the FFT block processing in the previous section using an overlap of $N - 1$ for an N-point spectrum. The advantages of overlap were illustrated in Example 8.4; however, computing an N-point FFT every single sampling instant is much too complex. The sliding DFT is a practical algorithm for computing the spectrum at each sampling instant. The DFT was introduced in Chapter 7 and is computed as follows:

N-PT DFT

At $t = LT_s$ (i.e., end of a sampling interval), let $x(k)$ be the N input samples taken over the last sampling interval; that is, $(L-1)T_s < t \leq LT_s$.

$$x(k) = [x(L - (N - 1)T_s) \dots x((L - 1)T_s) \ x(LT_s)]$$

$$X_{\text{DFT}}(nf_0) = \sum_{k=0}^{N-1} x(k)e^{-j2\pi nk/N} \quad n = 0, 1, \dots, N-1$$

$$\text{Bin Frequencies: } 0, f_0, 2f_0, \dots, (N-1)f_0 \quad \text{where } f_0 = F_s/N$$

(8.3)

Computing a DFT for each frequency bin simply involves multiplying N consecutive input data values by the exponential weighting functions for that bin then adding the results. It is really easy to understand how the sliding DFT algorithm was derived using the graphical illustration of an 8-pt DFT shown in Figure 8.6. The discrete frequencies (or bin frequencies) for an 8-pt DFT are $0, f_0, 2f_0, 3f_0, 4f_0, 5f_0, 6f_0, 7f_0$. The exponential weights for Bin 0 ($f = 0$ Hz) are all one so that case is not drawn. The exponential weights for Bin 1 ($f = f_0$ Hz) are the eight weights equally distributed clockwise about the unit circle as shown in Figure 8.6. The exponential weights for Bin 2 ($f = 2f_0$ Hz) are the eight weights obtained by taking every other weight in Bin 1 starting with one, moving clockwise, and wrapping around the circle as many times as necessary. The exponential weights for Bin 3 ($f = 3f_0$ Hz) are the eight weights obtained by taking every 3rd weight in Bin 1 starting with one, moving clockwise, and wrapping around the circle as many times as necessary. This same pattern would continue to find the weights for Bins 4-7.

Given eight consecutive data samples, the DFT for each bin frequency is computed by multiplying the eight data values by the weights for that bin and adding the results. In Figure 8.6, it is assumed that the eight consecutive data samples are numbered 0, 1, 2, ..., 7. These sample numbers are shown in the figure next to their corresponding weights.

Challenge Question 8.3

For a 16-pt DFT, assume the sixteen consecutive data samples are numbered 0, 1, 2, ..., 15. Sketch a diagram similar to the one in Figure 8.6 that shows how the data samples are distributed among the sixteen weights for Bin 5.

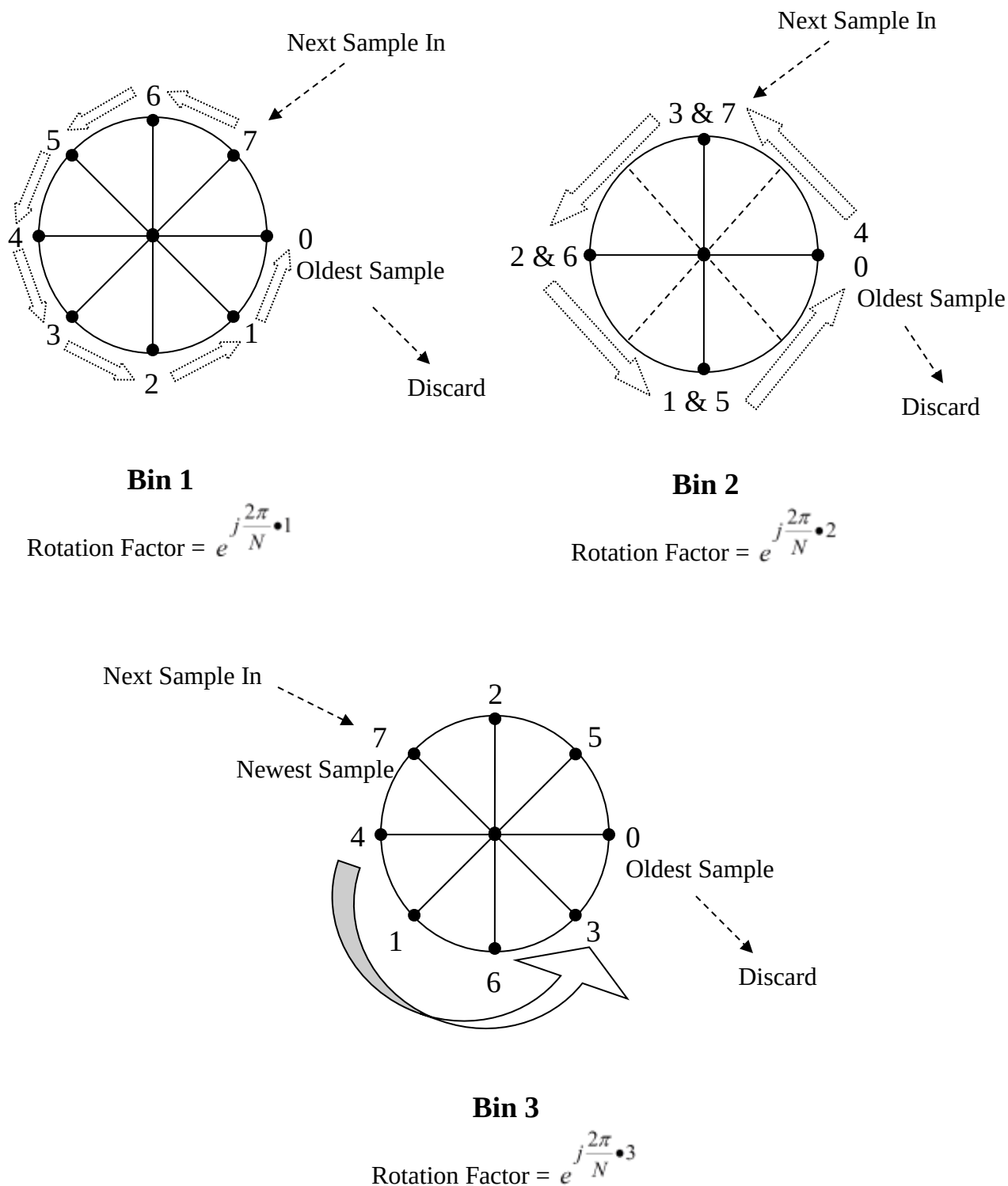


Figure 8.6: Graphical View of Sliding DFT

Assume the DFT has been calculated for all eight bin frequencies using the data samples 0, 1, 2, ..., 7. Now a new data sample value is taken and we wish to compute a new DFT using the eight most recent sample values; that is, 1, 2, 3, ..., 8. While it would certainly be possible to start all over again and compute a new DFT, this would not be smart. Looking at the diagrams in Figure 8.6, three things must happen to find the new DFT:

1. Discard the oldest data sample
2. Rotate the remaining samples to their new corresponding weights
3. Account for the newest data sample

Since the weight on the oldest sample is always one, the oldest sample can be discarded by simply subtracting it directly from the previous DFT calculation. The new weights for the remaining samples are derived by simply shifting each data sample counterclockwise to the next weight in the bin which would be a $\frac{2\pi}{N}n$ radian shift (n is the bin number). Since all the data samples for a specified bin frequency are shifted by the same amount, the product of each sample times its new weight equals the old product times the rotation factor, $e^{j\frac{2\pi}{N}n}$, for Bin n .

So, at this point, we have

$$\text{Bin } n: \quad \text{NEXT DFT} = e^{j\frac{2\pi}{N}n} [\text{PREVIOUS DFT} - \text{oldest data sample}] + ?$$

The only thing not accounted for at this point is inserting the newest data sample multiplied by its corresponding weight. Looking at Figure 8.6, the weight corresponding to the newest data sample just happens to be equivalent to the rotation factor. So, we now have the following:

$$\text{NEXT DFT} = e^{j\frac{2\pi}{N}n} [\text{PREVIOUS DFT} - \text{oldest data sample}] + e^{j\frac{2\pi}{N}n} [\text{newest data sample}]$$

which simplifies to

$$\text{NEXT DFT} = e^{j\frac{2\pi}{N}n} [\text{PREVIOUS DFT} - \text{oldest data sample} + \text{newest data sample}]$$

In more conventional math notation, the algorithm for a sliding DFT is

SLIDING DFT

$$X_{nf_0}(L) = e^{j2\pi n/N} [X_{nf_0}(L-1) - x(L-N) + x(L)] \quad n = 0, 1, \dots, N-1 \quad (8.4)$$

The algorithm must be initialized in order to implement the sliding DFT. There are two options for initializing the algorithm. One option is to collect the first N input data values then calculate an N -PT DFT that would be used as the initializing DFT in the difference equation above. Another option is to simply start at $t = 0$ by loading the input data buffer with all zeros (except the current input $x(0)$) and initializing $X_{nfo}(L-1)$ to zero. In this case, it is important to recognize that the difference equation would not produce valid DFT information until the N^{th} iteration through the difference equation. Both options require the exact same time delay of NT_s before the first valid DFT can be computed. The second option is a lot easier to program because it simply requires implementation of the difference equation and does not include the additional computational burden of the one-time N -PT DFT calculation.

A block diagram realization of the sliding DFT is shown in Figure 8.7.

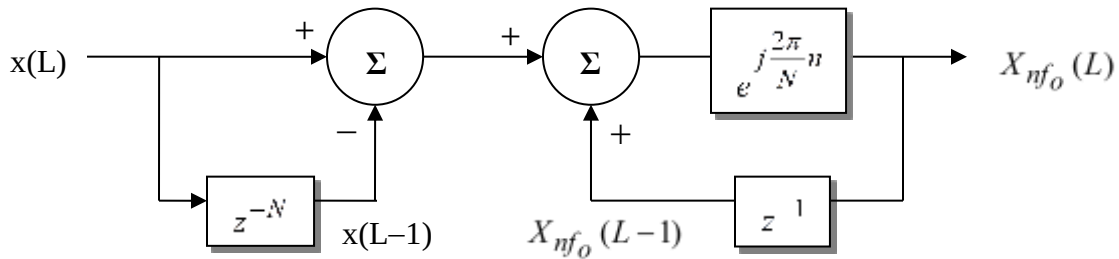


Figure 8.7: Block Diagram Realization of Sliding DFT

The first part of the block diagram in Figure 8.7 is simply a comb filter. The second part of the block diagram involves feedback of the previous output. The transfer function for the Sliding DFT is given by:

$$H(z) = \frac{[1 - z^{-N}] \times e^{j\frac{2\pi}{N}n}}{[1 - e^{j\frac{2\pi}{N}n} z^{-1}]} = \frac{[z^N - 1] \times e^{j\frac{2\pi}{N}n}}{z^{N-1} [z - e^{j\frac{2\pi}{N}n}]} \quad (8.5)$$

Challenge Question 8.4

What are the poles associated with the Sliding DFT algorithm? What does this indicate about the stability of the algorithm?

Modified Sliding DFT

As indicated in the challenge question, the Sliding DFT algorithm is marginally stable and may become unstable due to rounding of the coefficient, $\exp(j2\pi n/N)$. If stability is a problem, the sliding DFT algorithm can be modified to guarantee stability. The modified algorithm for a sliding DFT is given by:

Modified Sliding DFT (to guarantee stability)

$$X_{nf_o}(L) = r \cdot e^{j\frac{2\pi}{N}n} [X_{nf_o}(L-1) - r^N x(L-N) + x(L)] \quad n = 0, 1, \dots, N-1$$

$$0 < r < 1$$

(8.6)

Notice that in the modified algorithm, the numerical coefficient which posed a potential stability problem has been scaled by a real number $r < 1$. This will guarantee stability of the algorithm as

long as the magnitude of $re^{j\frac{2\pi}{N}n} < 1$. How does the scale factor, r , affect the DFT calculation? The modified sliding DFT algorithm will actually calculate the following:

$$X_{DFT}(nf_o) = \sum_{k=0}^{N-1} x(k)r^{N-k}e^{-j2\pi nk/N} \quad n = 0 \dots N-1 \quad (8.7)$$

Notice that the only difference between this modified DFT and the original DFT is the powers of r that have been inserted into the summation. These powers of r could be interpreted either as scaling factors that pull the bin weights inside the unit circle or as a non-symmetric unusual type of window on the input data samples as shown in Figure 8.8 for an 8-pt DFT. Notice that choosing r very close to one minimizes the scaling effect on the input data samples; that is, $r^{N-k} \approx 1$ so long as r is very close to 1.

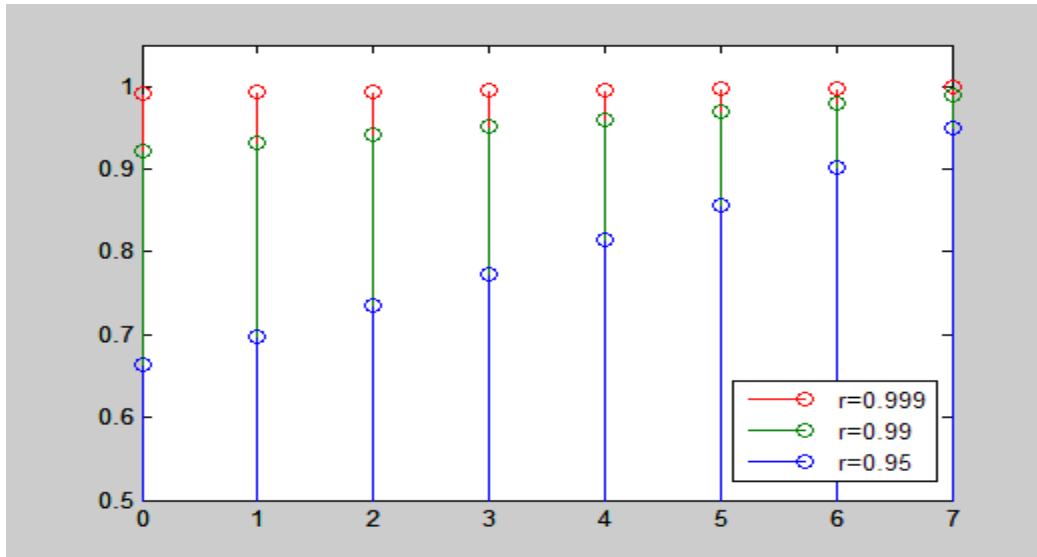


Figure 8.8: Effect of Scale Factor, r for 8-pt DFT

In summary, the scale factor r should be chosen as close to 1 as possible while still ensuring that the magnitude of $re^{j(2\pi/N)n}$ is less than 1 when implemented on the DSP processor. This will guarantee stability of the algorithm while minimizing the effect of scaling on the accuracy of the DFT results. A block diagram for the modified sliding DFT is shown in Figure 8.9.

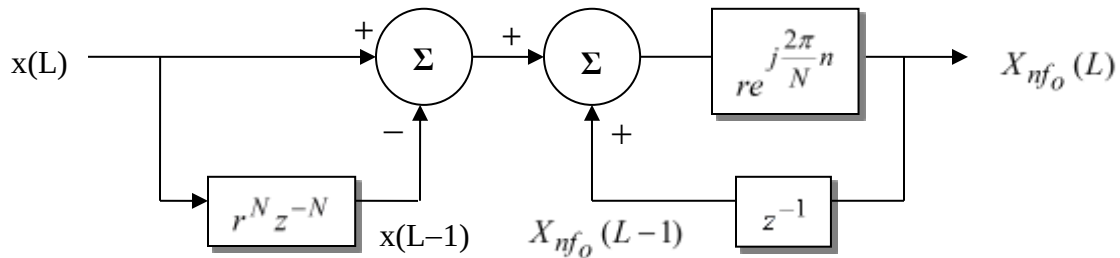


Figure 8.9: Block Diagram Realization of Modified Sliding DFT

Example 8.5: Tone Detection using Sliding DFT

The input signal for this sliding DFT example was generated by playing and recording a series of notes on a digital piano (see Figure 8.10(a)). The Midi recording was then converted to a wav file which was imported into MATLAB using the **audioread** function. A wav file uses a sampling rate of 44,100 Hz which is higher than necessary for this example so the data was decimated by a factor of 10 to reduce the sampling rate to 4410 Hz. A plot of the decimated wav file is shown in Figure 8.10(b). An FFT of the decimated wav file was computed using MATLAB and is plotted in Figure 8.10(c). The frequencies for the notes played are: Middle C = 261.63 Hz, E = 329.63 Hz, G = 392 Hz, and C one octave above Middle C = 523.25 Hz. Notice that the FFT in Figure 8.10(c) shows resonant peaks at the discrete frequency bins closest to these note frequencies.



Figure 8.10a: Series of Notes Played (Middle C, E, G, C, G, E, Middle C)

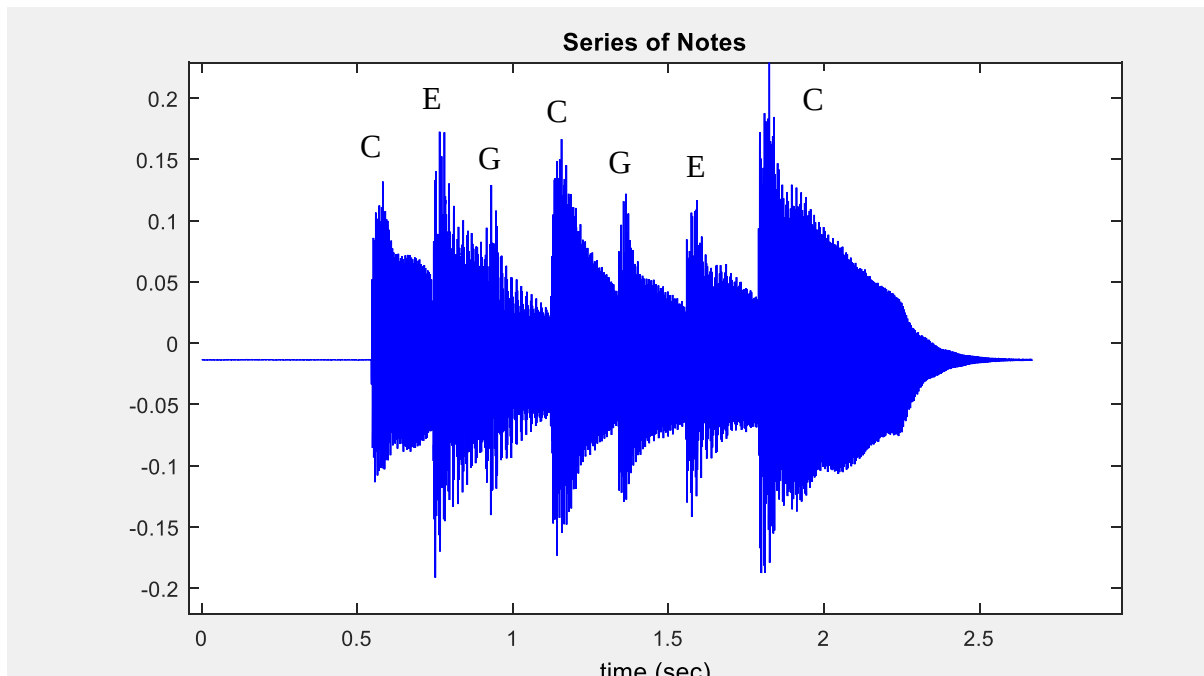


Figure 8.10b: Plot of wav file Data

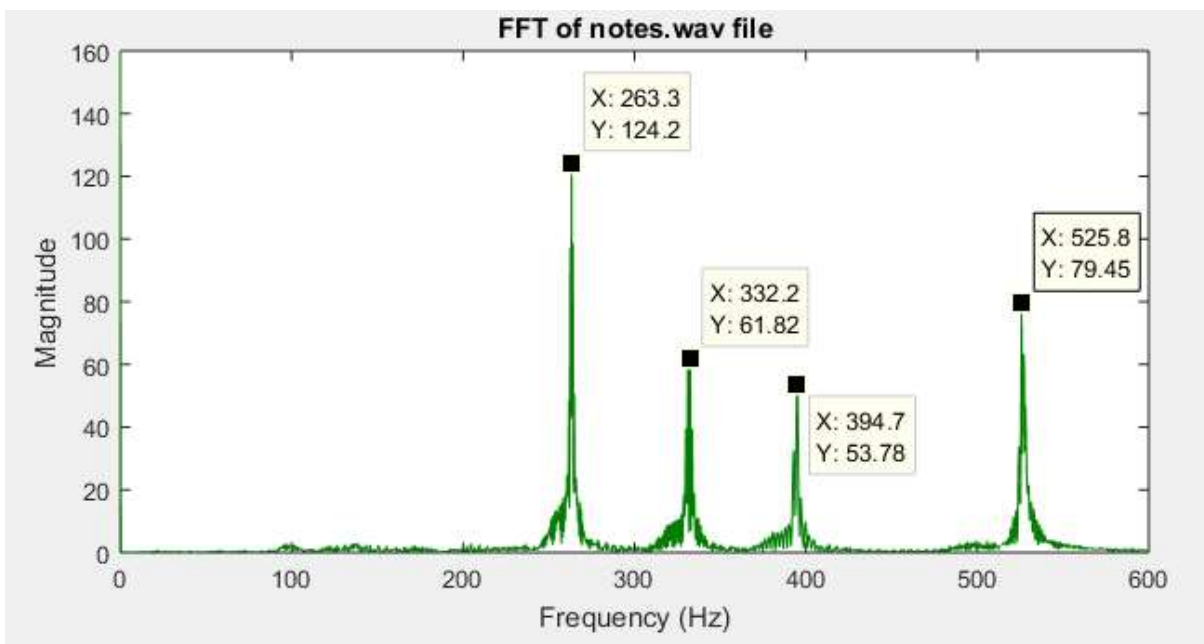


Figure 8.10c: FFT of the Decimated wav file

Figure 8.10: Example 8.5

The MATLAB commands used to generate the plots in Figure 8.10b and 8.10c are listed in Figure 8.11.

```
% Example 8.5
[y,Fs] = audioread('notes.wav');
% Decimate the signal by 10 giving a new sampling rate of 4410
% Just include first channel of stereo signal
ydec=decimate(y(:,1),10);
tdec = 0:1/4410:(size(ydec)-1)*1/4410;
plot(tdec,ydec);

fs=Fs/10;
yf=fftshift(fft(ydec));
N=length(yf); % N= # of time samples = # of freq. comp in fft
fo=fs/N;      % Calculate resolution of fft in Hz
f=-floor(N/2)*fo:fo:(ceil(N/2)-1)*fo; % Set up DFT Frequencies

%Plot the magnitude response of fft
figure;
plot(f,abs(yf));
title('FFT of notes.wav file')
xlabel('Frequency (Hz)');xlim([0 600])
```

Figure 8.11: MATLAB Code for Figure 8.10 (b) and (c)

Although the spectral plot in Figure 8.10c does indeed accurately reflect the notes that were played, it gives no indication of when each note was played or the duration of each note. For the next part of this example, a 256 pt-sliding DFT will be implemented to show how the signal spectrum varies over time. The resolution for the 256-pt DFT is

$$f_o = \frac{F_s}{N} = \frac{4410}{256} = 17.2265625 \text{ Hz}$$

The sliding DFT could easily be calculated for all 256 bin frequencies but in this example it will only be calculated for the bin frequencies closest to the notes played. The bin numbers for each note are shown in Table 8.3.

Table 8.3: Bin Numbers for Notes Played

NOTE	FREQUENCY	FREQUENCY/ f_o	BIN #
Middle C	261.63	15.2	15
E	329.63	19.1	19
G	392	22.76	23
C octave above Middle C	523.25	30.4	30

The results are plotted in Figure 8.12. Each of the sliding DFTs correlates very well with the original wav file. It is possible to examine each of the sliding DFTs in Figure 8.12 and determine when each of the individual notes was played. The bottom graph for the C one octave above Middle C does require comment. It appears that this note was played in conjunction with Middle C at $t = 0.6$ sec and $t = 1.8$ sec. This is actually due to how the digital piano works. The digital piano provides several options for special effects; one of which is that when a note is played, harmonics are also produced. The C above Middle C is the 2nd harmonic of Middle C and will be generated every time Middle C is played (with smaller amplitude).

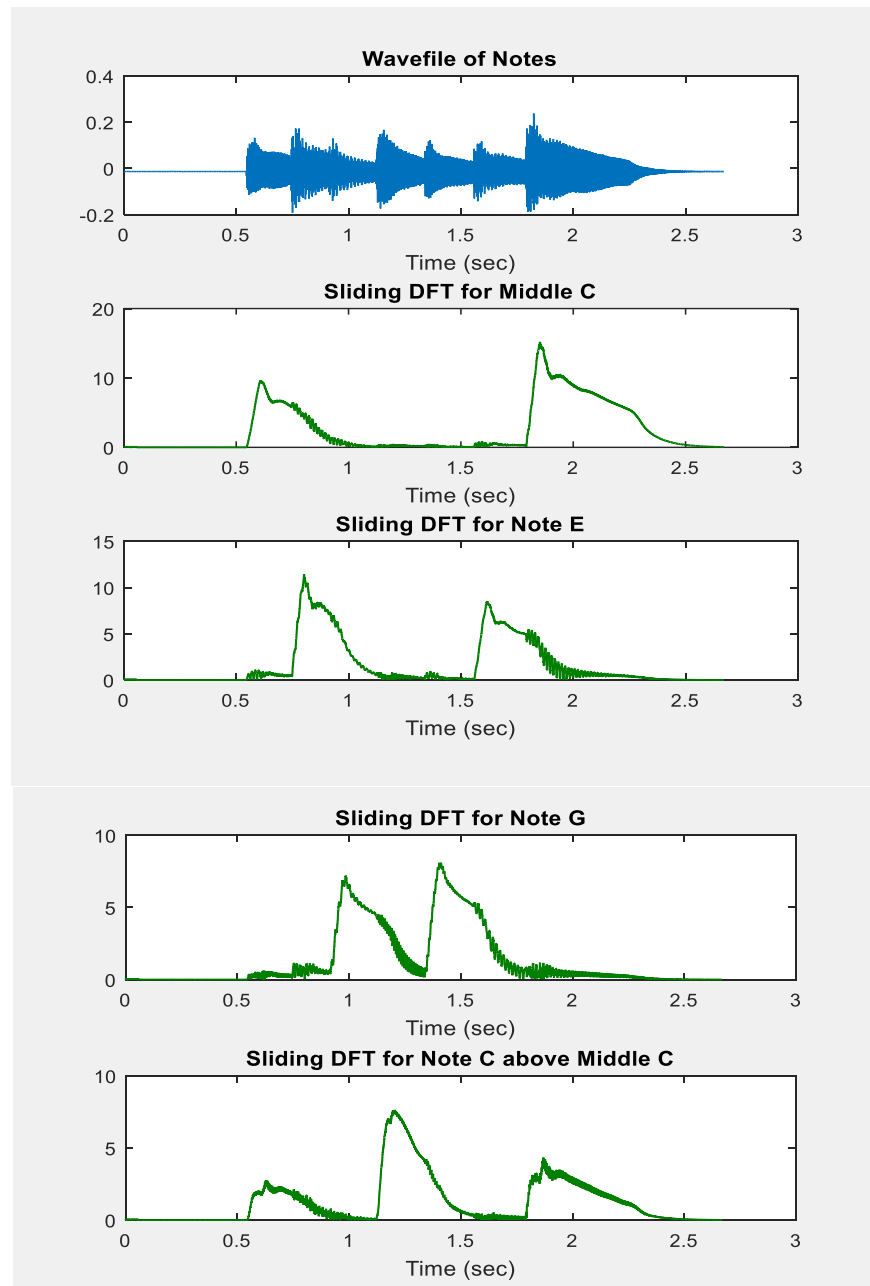


Figure 8.12: Sliding DFT Results (Example 8.5)

```

[y,Fs] = audioread('notes.wav');
ydec=decimate(y(:,1),10); % Decimate Channel 1 of input wav file by 10
tdec = 0:1/4410:(size(ydec)-1)*1/4410;
subplot(3,1,1);plot(tdec,ydec);
title('Wavefile of Notes');xlabel('Time (sec)')

% Sliding DFT
N = 256; % Size of DFT
x = [zeros(1,N) ydec']; % Initialize input data vector with N zeros

% Middle C = Bin 15
n = 15; X_15(1)=0; % Set Bin # and Initialize first DFT to zero
for m = 1:length(x)-N-1,
    X_15(m+1)=exp((1i*2*pi/N)*n)*[X_15(m)-x(m)+x(m+256)];
end
subplot(3,1,2); plot(tdec,abs(X_15));
title('Sliding DFT for Middle C');xlabel('Time (sec)')

% Note E above Middle C = Bin 19
n = 19; X_19(1)=0;
for m = 1:length(x)-N-1,
    X_19(m+1)=exp((1i*2*pi/N)*n)*[X_19(m)-x(m)+x(m+256)];
end
subplot(3,1,3); plot(tdec,abs(X_19));
title('Sliding DFT for Note E');xlabel('Time (sec)')

% Note G above Middle C = Bin 23
n = 23; X_23(1)=0;
for m = 1:length(x)-N-1,
    X_23(m+1)=exp((1i*2*pi/N)*n)*[X_23(m)-x(m)+x(m+256)];
end
figure; subplot(2,1,1); plot(tdec,abs(X_23));
title('Sliding DFT for Note G');xlabel('Time (sec)')

% C above Middle C = Bin 30
n=30; X_30(1)=0;
for m = 1:length(x)-N-1,
    X_30(m+1)=exp((1i*2*pi/N)*n)*[X_30(m)-x(m)+x(m+256)];
end
subplot(2,1,2); plot(tdec,abs(X_30));
title('Sliding DFT for Note C above Middle C');xlabel('Time (sec)')

% Show all 4 sliding DFTs in a single plot
figure; subplot(2,1,1); plot(tdec,ydec);
title('Input Data: Wavefile of Notes');xlabel('Time (sec)')
subplot(2,1,2);
plot(tdec,abs(X_15),tdec,abs(X_19),tdec,abs(X_23),tdec,abs(X_30));
title('Sliding DFTs for all Four Bins');xlabel('Time (sec)')
legend('Middle C','E','G','C octave above Middle C')

```

Figure 8.13: MATLAB Code for Sliding DFT (Example 8.5)

Figure 8.14 shows all four sliding DFTs in a single plot.

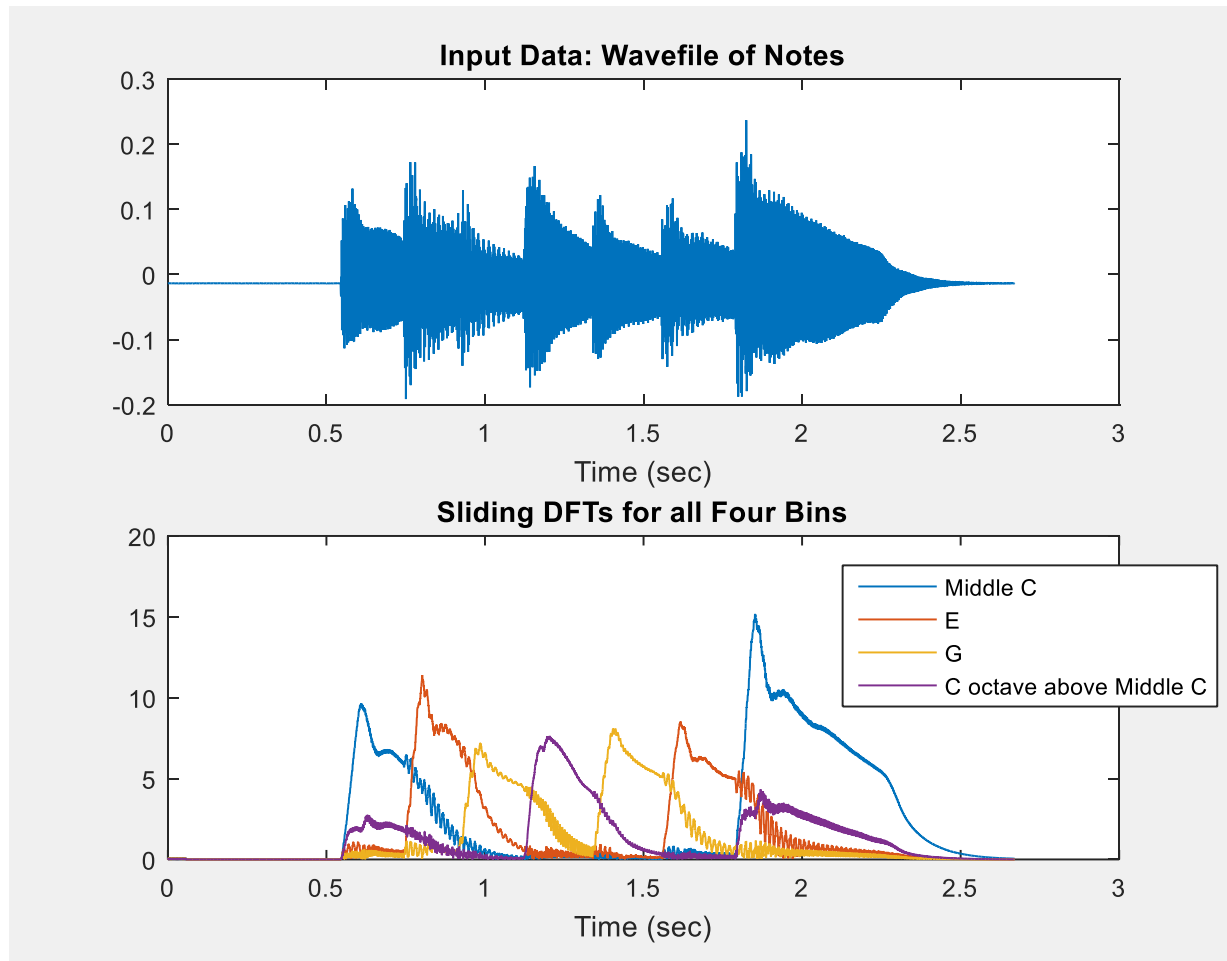


Figure 8.14: Combined Sliding DFT Results (Example 8.5)

Effect of Time Duration, D

In the previous section on real-time spectral analysis using block processing and FFTs, it was mentioned that the choice of time duration, D , was critical. Choosing a time duration too long could result in missing short burst changes in the input signal (poor time resolution), but decreasing the time duration reduces the frequency resolution. This issue was illustrated in Example 8.3. The same issue occurs with a sliding DFT. Even though the spectrum is updated at every sampling instant, the spectrum is still based on N samples of the input signal spanning a time duration of $D = N \cdot F_s$.

Wavelets are an alternative to FFTs or sliding DFTs for real-time spectral analysis. Wavelets attempt to bridge the tradeoff between poor time resolution with large D and poor frequency resolution with small D by offering good time resolution at higher frequencies and good frequency resolution at lower frequencies. The underlying assumption on the input signal is that the lower frequencies persist for longer intervals of time (so D can be longer); whereas, the higher frequencies tend to occur in short bursts (so D must be shorter).

Wavelet processing essentially breaks the input signal into distinct frequency bands through a series of digital filters and provides good frequency resolution in the lower bands and good time resolution in the higher bands of the signal. A detailed analysis of wavelet processing is beyond the scope of this text, but it is a fascinating subject and there are many good texts and websites available to the interested reader.

Sliding DFT with Windowing

In section 7.2, the importance of windowing blocks of data to enhance spectral analysis by reducing the effect of leakage was discussed. With the Sliding DFT, windowing the incoming data samples directly (in the time domain) is not an option.

Challenge Question 8.5

Why is it not possible to window the incoming data samples directly for a sliding DFT?

Although it is not possible to window the incoming input data samples in the time domain, it is possible to produce the same effect by performing a convolution in the frequency domain. Recall the following useful property first introduced in Chapter 4:

$$x(t)w(t) \leftrightarrow X(f) * W(f) \quad (8.8)$$

Multiplication of two signals in the time domain is equivalent to convolving the spectrums of the two signals. One of the common window functions introduced in Chapters 5 and 7 is the Hamming window described by

$$w(kT_s) = 0.54 - 0.46 \cos\left(\frac{2\pi kT_s}{N-1}\right) \quad k = 0 \dots N-1 \quad (8.9)$$

This window function has a very simple Fourier Transform given by:

$$W(f) = 0.54\delta(f) - 0.23[\delta(f + f_o) + \delta(f - f_o)] \quad \text{where} \quad f_o = \frac{F_s}{N} \quad (8.10)$$

Assuming that the sliding DFT provides an accurate sampled approximation to the spectrum of the input data signal $X(f)$ at the bin frequencies nf_o , and applying the convolution property using $W(f)$ for the Hamming window yields

$$X^W(nf_o) = 0.54X(nf_o) - 0.23[X((n+1)f_o) + X((n-1)f_o)] \quad (8.11)$$

In other words,

$$\text{Bin } n \text{ Windowed} = 0.54 [\text{Bin } n] - 0.23 [\text{Bin } (n-1) + \text{Bin } (n+1)] \quad (8.12)$$

Windowing is therefore accomplished by using the sliding DFT algorithm to compute the spectrum at the various bin frequencies, then combining adjacent bins to produce a spectrum for windowed data samples.

Several other window functions were explored in Chapters 5 and 7. Any window function that can be represented as a combination of cosine or sine functions could be used in place of the Hamming window. Table 8.4 provides some common window functions and the corresponding “windowed” DFT.

Table 8.4: DFT using Common Window Functions

WINDOW	WINDOW FUNCTION	WINDOWED DFT
Hamming	$w(k) = 0.54 - 0.46\cos\left(\frac{2\pi k}{N-1}\right)$	$X_n^w = 0.54X_n - 0.23[X_{(n+1)} + X_{(n-1)}]$
Hanning	$w(k) = 0.5 - 0.5\cos\left(\frac{2\pi k}{N-1}\right)$	$X_n^w = 0.5X_n - 0.25[X_{(n+1)} + X_{(n-1)}]$
Blackman	$w(k) = 0.42 - 0.5\cos\left(\frac{2\pi k}{N-1}\right) + 0.08\cos\left(\frac{4\pi k}{N-1}\right)$	$X_n^w = 0.42X_n - 0.25[X_{(n+1)} + X_{(n-1)}] + 0.04[X_{(n+2)} + X_{(n-2)}]$

Notice that the coefficients for the Hanning window can be used to advantage when implementing this window. Multiplication by 0.5 can be accomplished by a single bit shift to the right and multiplication by 0.25 can be implemented by a 2 bit shift to the right.

8.5 ADAPTIVE FILTERING

Adaptive filtering is the process of changing the weights of a filter in real-time based on some type of input/output signal information. Echo cancellation, room acoustic identification, and noise cancellation are examples of adaptive filtering applications.

Echo Cancellation

Echo cancellation is extremely useful in telecommunications. The two types of echo are hybrid echo and acoustic echo. Hybrid echo occurs in a wire-line telephone network switch where the two-wire analog loop connection to the house is switched to a four-wire digital line for transmission. The echo is simply a result of an impedance mismatch in the two-wire to four-wire conversion circuitry. Hybrid echo occurs when one or more speakers are utilizing a land line connection. Acoustic echo results from unwanted feedback between a speaker and a microphone or from reflection of audio signals from surfaces back to the microphone. Acoustic echo occurs in very small phones where the earpiece and microphone are not sufficiently isolated or in “hands-free” conversations. Digital signal processors programmed as adaptive filters are used to cancel unwanted echo.

A block diagram for Hybrid echo cancellation is shown in Figure 8.15. When Speaker #1 talks, the impedance mismatch at the hybrid causes an echo of his or her speech. In the absence of echo cancellation, this echo would be combined with a voice signal (if present) from Speaker #2 and would be transmitted back to Speaker #1. The adaptive FIR filter in the hybrid produces an estimate of the echo and subtracts the estimate from the return signal effectively cancelling the echo. The weights of the adaptive FIR filter are adjusted in real-time based on the incoming signal from Speaker #1 and the residual error for the echo estimate. The filter weights are only adjusted when Speaker #1 is speaking and Speaker #2 is not in which case a residual error of zero would indicate a perfect estimate of the hybrid echo signal. Typically a least mean squares (LMS) algorithm (or version of such) is used to adjust the filter weights to minimize the residual error. The LMS algorithm is covered later in this section.

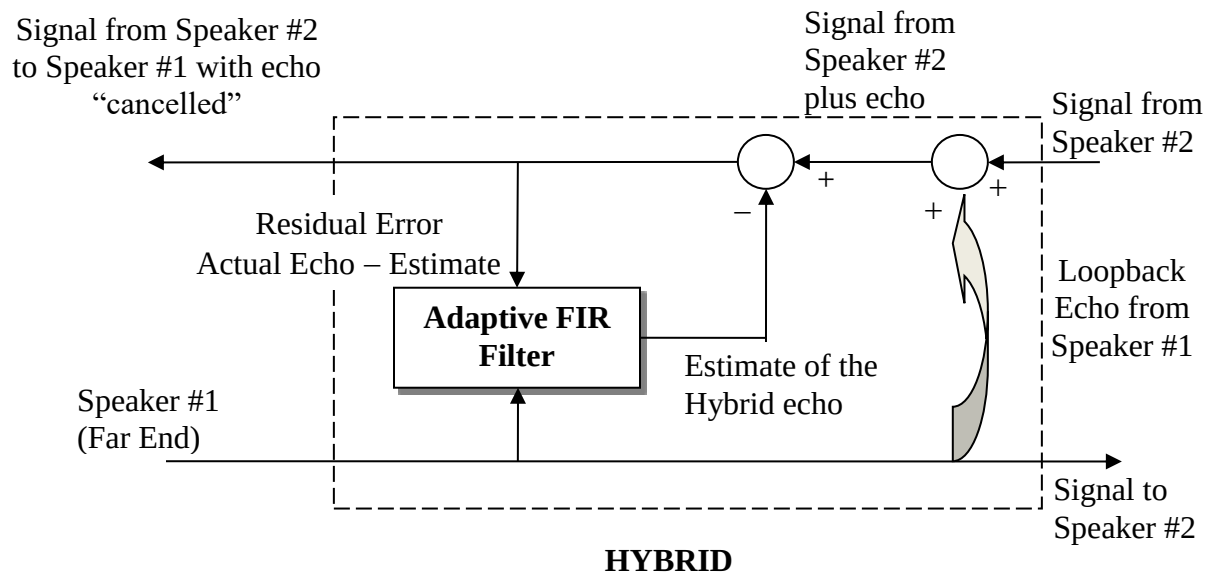


Figure 8.15: Echo Cancellation

The hybrid echo signal is relatively easy to cancel because it is linear and stationary; that is, the echo signal is simply the original speech signal attenuated and time-delayed. Acoustic echo is much more difficult to cancel because it is non-linear and non-stationary. Methods for cancelling acoustic echo vary from vendor to vendor with varying degrees of success (or failure).

Room Acoustic Identification

Adaptive filters can also be used to identify the acoustics of an enclosed space such as a room, an auditorium, a concert hall or even a stairwell. Acoustic identification is useful for new construction, for modifying existing spaces, and for reverb filter design. A basic block diagram

for room acoustic identification is shown in Figure 8.16. An input signal, typically white or pink noise, is applied to an adaptive filter and to a loudspeaker in the room. The signal, modified by the acoustical characteristics of the room, is picked up by the microphone. This signal is compared to the output of the adaptive filter producing a difference or error signal. The filter weights are adjusted based on the applied input signal and on the error signal in order to minimize the error signal. If the error signal converges to zero, then the adaptive filter weights “match” the impulse of the room.

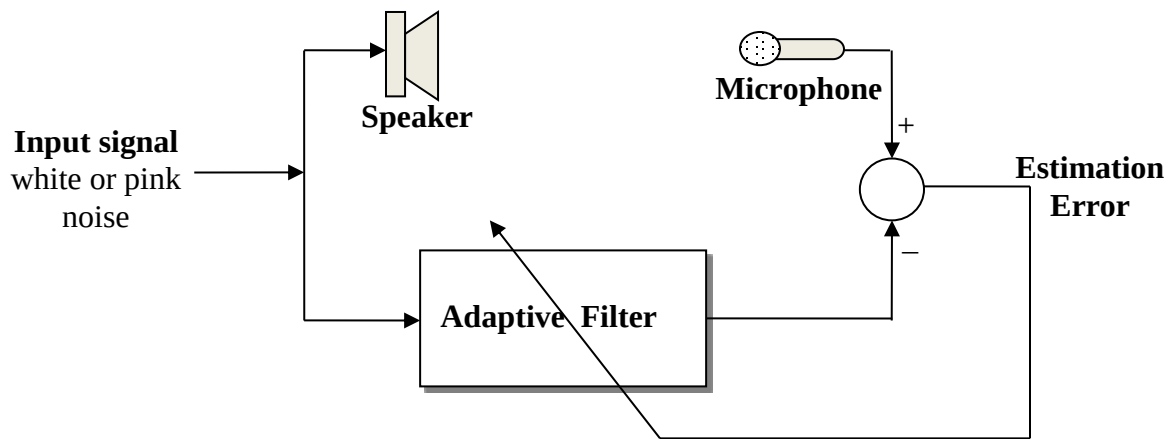


Figure 8.16: System Identification

Noise Cancellation

Adaptive filters can be used to suppress or cancel noise in a signal or in the surrounding environment. Noise-cancelling headphones reduce the level of unwanted ambient noise from fans or engines or other background noise sources. There are numerous biomedical applications for adaptive filtering. One of the earliest biomedical applications for noise cancellation was by Widrow et. al to eliminate the 50 Hz “hum” from an ECG (electrocardiograph) heart signal. Widrow also utilized adaptive filtering to eliminate the mother’s heartbeat signal from a fetal monitor signal in order to produce the heartbeat of the fetus only.

A block diagram for adaptive noise cancellation is shown in Figure 8.17. The reference noise signal is applied to an adaptive filter to produce an “anti-noise” signal that is then subtracted from the noisy signal to reduce the noise level in the signal. In the case of noise-cancelling headphones, the reference noise signal is the background noise picked up by a microphone. For the fetal heartbeat application, the reference noise signal is the mother’s heartbeat signal picked up by a second monitor.

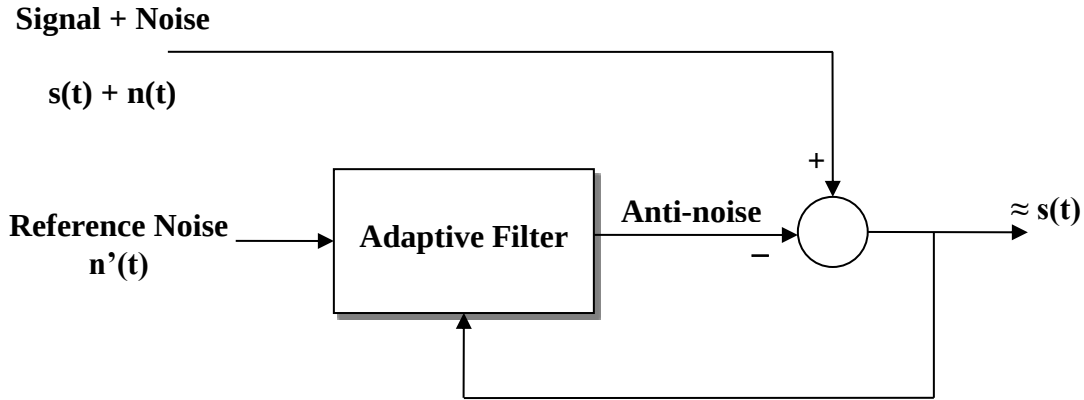


Figure 8.17: Noise Cancellation

Least Mean Squares Algorithm

One algorithm for adjusting the weights of an adaptive filter is the least mean squares (LMS) algorithm. The block diagram in Figure 8.18 is used to define the symbols used in the algorithm for the various signals.

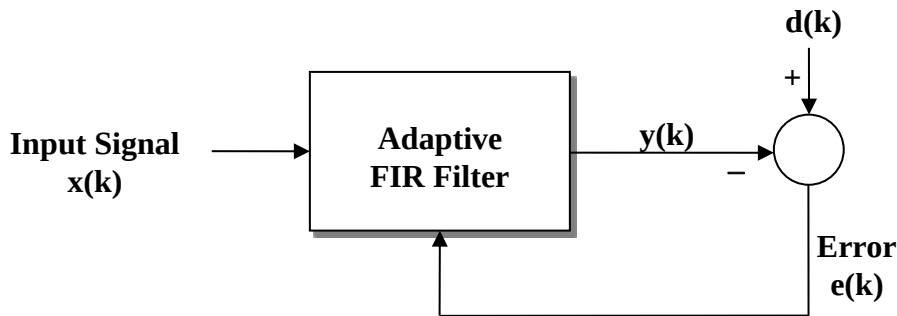


Figure 8.18: Adaptive FIR Filter

The weights (or filter coefficients) of the adaptive FIR filter are adjusted to minimize the error, $e(k)$, according to some constraint. The error signal is simply the difference between the output of the adaptive filter and the desired output. Using $w_0, w_1, \dots, w_{N-1}$ to designate the filter weights, the output of the adaptive filter is

$$y(k) = w_0(k)x(k) + w_1(k)x(k-1) + \dots + w_{N-1}(k)x(k-(N-1)).$$

The LMS algorithm minimizes the expected value of the square of the error signal; that is the power of the error signal, and results in the following update for the filter coefficients:

LMS Algorithm

$$w_i(k + 1) = w_i(k) + \mu e(k) \cdot x(k - i) \quad \text{for } i = 0 \dots N - 1$$

$e(k) = d(k) - y(k)$ = estimation error

μ : adaptation step size

N : order of FIR filter

$w_i(k)$: i^{th} filter coefficient at time kT_s (8.13)

The adaptation step size, μ , affects how much the filter weights will be changed at each iteration of the algorithm and therefore affects the rate of convergence of the algorithm. Choosing a larger value for μ could result in faster convergence (adaptivity) but could also cause less accurate filter weights, larger estimation error, and affect the stability of the algorithm. The adaptation step size is sensitive to the characteristics of the input signal which means there is no “one size fits all” value for μ . The normalized LMS algorithm reduces input signal sensitivity by normalizing the weight adjustment term by the input signal power as shown in Equation 8.14:

Normalized LMS Algorithm

$$w_i(k + 1) = w_i(k) + \mu e(k) \cdot \frac{x(k - i)}{\sum_{i=0}^{N-1} x^2(k - i)} \quad \text{for } i = 0 \dots N - 1 \quad (8.14)$$

Recall in the previous section, it was noted that the Sliding DFT algorithm was marginally stable (poles on unit circle) and a scale factor was introduced to stabilize the algorithm. The LMS algorithms are also marginally stable with a discrete pole at 1. A leakage factor can be added to stabilize the algorithm:

Normalized LMS Algorithm with Leakage Factor

$$w_i(k + 1) = (1 - \mu\alpha)w_i(k) + \mu e(k) \cdot \frac{x(k - i)}{\sum_{i=0}^{N-1} x^2(k - i)} \quad \text{for } i = 0 \dots N - 1$$
$$0 < (1 - \mu\alpha) < 1 \quad (8.15)$$

There are several other algorithms available for adjusting the weights of adaptive filters including recursive least squares (RLS) and Kalman filters. Detailed discussion and derivation of adaptive filters is beyond the scope of this text. Many of these algorithms are available in MATLAB. The lab exercise at the end of this chapter introduces the adaptive filter blocks in Simulink.

Answers to Chapter 8 Challenge Questions

Question 8.1 Close examination of the plot in Figure 8.4(a) reveals a very small peak at 300 Hz prior to the much larger peak and a very small peak at 100 Hz in between the two much larger peaks. Why does this occur?

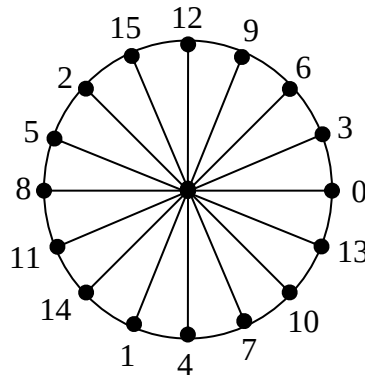
The input samples are fed into a buffer of length 256 and an FFT is computed. The buffer is then filled with 256 new input samples and a new FFT is computed. This process continues over the entire duration of the input signal. The signal in this example begins as a 100 Hz tone then switches to a 300 Hz tone. There are certain windows of time where the buffer of data contains some samples of the 100 Hz tone and some samples of the 300 Hz tone. The small peak at 300 Hz prior to the larger peak indicates that the buffer was filled mostly with samples of the 100 Hz tone but had a few samples of the 300 Hz tone. The small peak at 100 Hz between the two larger peaks at 100 Hz is indicative of a sampling time window that produced a buffer mostly filled with samples at 300 Hz with a few samples at 100 Hz. This concept is explored in more detail in Example 8.4.

Question 8.2 Example 8.4 shows that increasing the overlap of input sample values for spectral analysis results in a better reflection of how the signal spectrum varies over time. What is the disadvantage of increasing the overlap?

Increasing the overlap results in an increase in computational complexity by creating more sampling intervals and shortening the amount of time to compute each FFT. In Example 8.4, an overlap of 128 (for a 256-PT FFT) doubled the number of sampling windows and cut the time in half for computing each new FFT. Instead of waiting for 256 new input samples before computing the next FFT, the FFT is computed based on 128 new input samples and 128 past sample values. Increasing the overlap to 192 in Example 8.4 resulted in four times as many FFTs computed in $\frac{1}{4}$ the amount of time compared to no overlap.

Question 8.3 For a 16-pt DFT, assume the sixteen consecutive data samples are numbered 0, 1, 2, ... 15. Sketch a diagram similar to the one in Figure 8.6 that shows how the data samples are distributed among the sixteen weights for Bin 5.

First distribute 16 weights evenly spaced around a unit circle. The oldest sample, 0, has a weight of one. For the remaining samples, start at one and move clockwise to every 5th sample since we are working with Bin 5. Wrapping around the circle as necessary to use all 16 data samples results in the diagram on the following page.



Question 8.4 What are the poles associated with the Sliding DFT algorithm? What does this indicate about the stability of the algorithm?

As discussed in Chapter 4, the poles are the roots of the denominator of the transfer function.

There are $N-1$ poles at the origin, but there is one pole at $e^{j\frac{2\pi}{N}n}$. This pole lies on the unit circle which means the algorithm is only marginally stable. The system has N zeros evenly distributed about the unit circle, and one of these theoretically cancels the marginally stable pole. However,

coefficient rounding of the term $e^{j\frac{2\pi}{N}n}$ could cause the pole to migrate outside the unit circle which would cause instability of the algorithm.

Question 8.5 Why is it not possible to window the incoming data samples directly for a sliding DFT?

In a sliding DFT, we lose access to the individual data samples. The sliding DFT depends on the previous DFT calculation, the newest input value, and the oldest output value. In order to use windowing, the window values for each of the samples would have to be re-adjusted at each new time interval. This adjustment at each time interval was possible for the DFT weights because every data sample experienced the exact same change in weight (a simple rotation around the unit circle). Unfortunately, the data samples don't experience any type of uniform or consistent change when shifting through a window function.

Chapter 8 Problems

Problem 8.1 For an 8-pt DFT, draw a diagram showing all the weights for Bin 6 and the corresponding data samples assuming the data samples are numbered 0, 1, ... 7 with 0 being the oldest sample.

Problem 8.2: Compute the convolution $y = \text{conv}(x,h)$ using the overlap add method. Do the all the calculations by hand. Check your answer using MATLAB.

$$\begin{aligned}x &= [1 \ -1 \ 3 \ -2 \ 4 \ 2 \ 1] \\h &= [1 \ 2 \ 3]\end{aligned}$$

Problem 8.3: Compute the convolution $y = \text{conv}(x,h)$ using the overlap save method. Do the all the calculations by hand. Check your answer using MATLAB.

$$\begin{aligned}x &= [1 \ -1 \ 3 \ -2 \ 4 \ 2 \ 1] \\h &= [1 \ 2 \ 3]\end{aligned}$$

Problem 8.4:

- (a) Write an m-file to implement the overlap add algorithm. Inputs to the function should be the filter coefficients, h , and the input x . Output of the function should be the filter output, y .
- (b) Verify that the m-file works by doing the following:
 - Design an FIR filter using the Filter Design and Analysis Tool
 - Define some sinusoidal input in the passband of your filter.
 - Pass the input and filter coefficients to your overlap add m-file and plot the results.
 - Compute $y = \text{conv}(x,h)$ and plot the results. The two plots should match.

Problem 8.5:

- (c) Write an m-file to implement the overlap save algorithm. Inputs to the function should be the filter coefficients, h , and the input x . Output of the function should be the filter output, y .
- (d) Verify that the m-file works by doing the following:
 - Design an FIR filter using the Filter Design and Analysis Tool
 - Define some sinusoidal input in the passband of your filter.
 - Pass the input and filter coefficients to your overlap add m-file and plot the results.
 - Compute $y = \text{conv}(x,h)$ and plot the results. The two plots should match.

Problem 8.6: Refer to Example 8.3 Suppose the input signal is

$$x(t) = \begin{cases} \sin(2\pi(1250)t) & 0 \text{ ms} \leq t < 24 \text{ ms} \\ \sin(2\pi(725)t) & 24 \text{ ms} \leq t < 42 \text{ ms} \\ \sin(2\pi(400)t) + \sin(2\pi(550)t) & 42 \text{ ms} \leq t \leq 60 \text{ ms} \end{cases}$$

and the input signal is sampled at $F_s = 8000$ Hz.

- (a) Create the input signal in the MATLAB.
- (b) Build the Simulink model shown in Figure 8.3 for block processing an FFT. Set the FFT length to 256 with no overlap and set the window length to 256. Run the model then plot the 3-D FFT similar to Figure 8.5. Does the FFT reflect the true spectrum of the input signal? Explain.
- (c) Repeat part b for an FFT and window length of 128. How does the decrease in buffer size affect the spectrum?
- (d) Repeat part c for an FFT and window length of 64.

Problem 8.7: Refer to Example 8.4. Suppose the input signal is

$$x(t) = \begin{cases} \sin(2\pi(1250)t) & 0 \text{ ms} \leq t < 24 \text{ ms} \\ \sin(2\pi(725)t) & 24 \text{ ms} \leq t < 42 \text{ ms} \\ \sin(2\pi(400)t) + \sin(2\pi(550)t) & 42 \text{ ms} \leq t \leq 60 \text{ ms} \end{cases}$$

and the input signal is sampled at $F_s = 8000$ Hz.

- (a) Create the input signal in the MATLAB.
- (b) Make a table similar to Table 8.1 and 8.2 that show the time intervals for a 128-pt FFT with no overlap, a 128-pt FFT with an overlap of 32, and a 128-pt FFT with an overlap of 64.
- (c) Build the Simulink model shown in Figure 8.3 for block processing an FFT. Set the FFT length to 128 with no overlap and set the window length to 128. Run the model then plot the 3-D FFT similar to Figure 8.5. Does the FFT reflect the true spectrum of the input signal? Explain. (*Note: this is part c of Problem 8.6*)
- (d) Repeat part c using an overlap of 32 for the FFT. How does the overlap affect the spectrum? Why?
- (e) Repeat part c using an overlap of 64 for the FFT. How does the overlap affect the spectrum? Why?

Problem 8.8: Refer to Example 8.5. Suppose the input signal is

$$x(t) = \begin{cases} \sin(2\pi(349)t) & 0 \text{ s} \leq t < 0.25 \\ \sin(2\pi(440)t) & 0.25 \text{ s} \leq t < 0.5 \text{ s} \\ \sin(2\pi(349)t) + \sin(2\pi(523)t) & 0.5 \text{ s} \leq t \leq 1.0 \text{ s} \end{cases}$$

and the input signal is sampled at $F_s = 8000$ Hz.

- (a) Assuming a 256-pt DFT, calculate the discrete bin frequencies (and bin numbers) closest to the frequencies in the input signal.
- (b) Perform a sliding DFT on the input signal but only calculate the DFT at the bin frequencies determined in part a. Plot the results for each of the bin frequencies.
- (c) Repeat part b with a windowed sliding DFT using one of the windows from Table 8.4. Compare the results with part b.

CHAPTER 8 LAB EXERCISE

Adaptive FIR Filters

Objectives

1. To experiment with adaptive FIR filters for system identification using Simulink.
2. To explore the effectiveness of adaptive FIR filters for noise cancellation.

Procedure

A. System Identification using a Normalized LMS Adaptive Filter

1. Build the Simulink system shown in Figure 1.

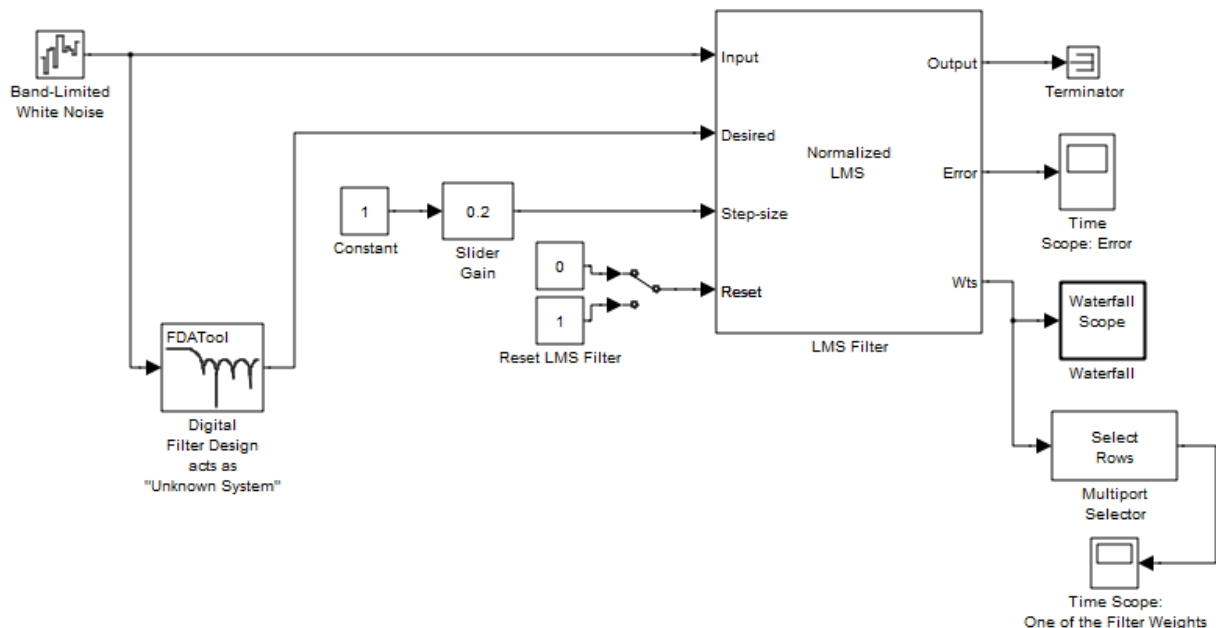


Figure 1: Simulink Model for System Identification

2. Set up the block parameters for the LMS filter block and the Digital Filter Design as shown in Figure 2.
3. Set the sample time for the band-limited white noise equal to 1/8000.
4. Set up the parameters for the Waterfall Scope as shown in Figure 3.
5. For the Multiport Selector, leave the selection as Rows and set the index to {1}. This will cause row 1 of the weights vector to be displayed on the time scope.
6. Set the stop time for the simulation to 0.1 seconds.

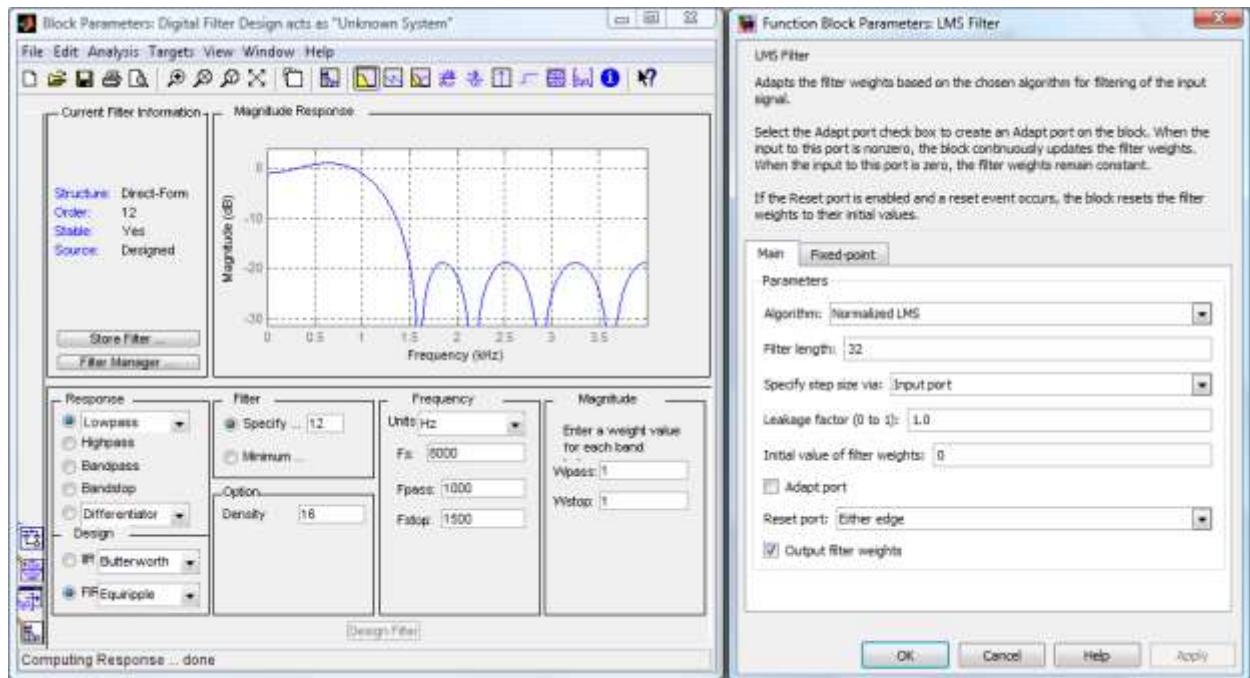


Figure 2: Block Parameters for Digital Filter and LMS Filter

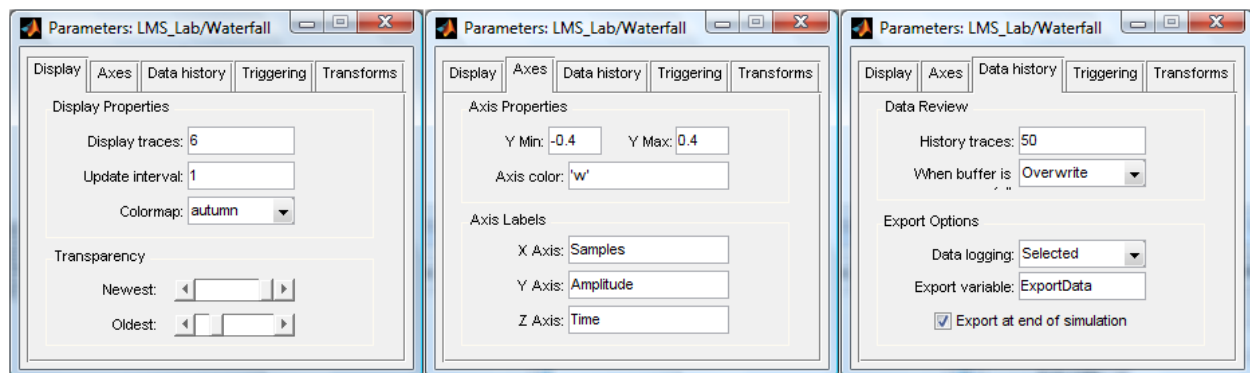


Figure 3: Block Parameters for Waterfall Scope

- Run the simulation with the Waterfall Scope and each of the time scopes open.

Comments:

The error time scope should display an error that starts out relatively high but is very close to zero by the end of the simulation (0.1 seconds). The time scope for the weights simply shows how the first weight changes over time (all weights begin at zero initially). You could view any of the other weights by simply clicking on the Multiport Selector and choosing a different row. The Waterfall scope shows how all 32 weights (samples) change over time. Only the last six values (six traces) for each weight is shown. The final values for all 32 weights are available in the MATLAB workspace in a vector called *ExportData*.

8. Now experiment with the slider gain to change the step size of the adaptation for the filter weights. Comment on the effect of the slider gain. What happens when the adaptation step goes up to 2?

9. Set the slider gain to 0.5. Experiment with the filter length in the LMS filter block. The “Unknown System” is 12th order which means it has 13 numerator coefficients. Answer the following questions:

What happens if the filter length in the LMS filter block is shorter than the “unknown” system filter length?

What happens if the filter length in the LMS filter block is exactly equal to the number of “unknown” filter coefficients? To really answer this question, you will need to look at the variable `ExportData` in the MATLAB workspace after running the simulation. Compare the LMS filter weights to the filter coefficients in the FDAT block.

What happens if the filter length in the LMS filter block exceeds the “unknown” system filter length?

Since the system in practice is unknown, the order of the system will presumably also be unknown. What are the advantages and disadvantages of simply setting the LMS adaptive filter length really high?

10. Design a different type of “unknown” system. Choose a low-pass IIR Butterworth filter (minimum order) with a sampling frequency of 8000 Hz, pass-band 1000 Hz, stop band 2000 Hz with 40 dB of attenuation in the stop band. Can you get the LMS FIR filter to properly model the IIR “unknown” filter? How many LMS filter coefficients were needed?

11. Try a different filter type (FIR band-pass or notch) for the “unknown” system. Experiment with the adaptation step size and the LMS filter length. Comment on observations.

B. Noise Cancellation Using Adaptive Filtering

1. Build the system in Simulink shown in Figure 4.
2. Set the stop time in the model block to 90 (seconds).
3. In the **Random Source** block, set the Source Type to Uniform, the Maximum to 0.25, and the sample time to 1/44100.
4. In the **From Multimedia File** block, set the file name to songmono.wav, click the loop box, and set the number of times to repeat to 1.
5. Locate the wav file (songmono.wav) on the text website and add it to your current directory. Or, create your own songmono.wav file by saving one of your own wav files to the current directory then converting it to mono as follows:

```
[y,Fs] = audioread('nameofyourfile.ext');  
ymono = y(:,1); % read the first channel of the stereo file  
audiowrite('songmono.wav',ymono,Fs);
```

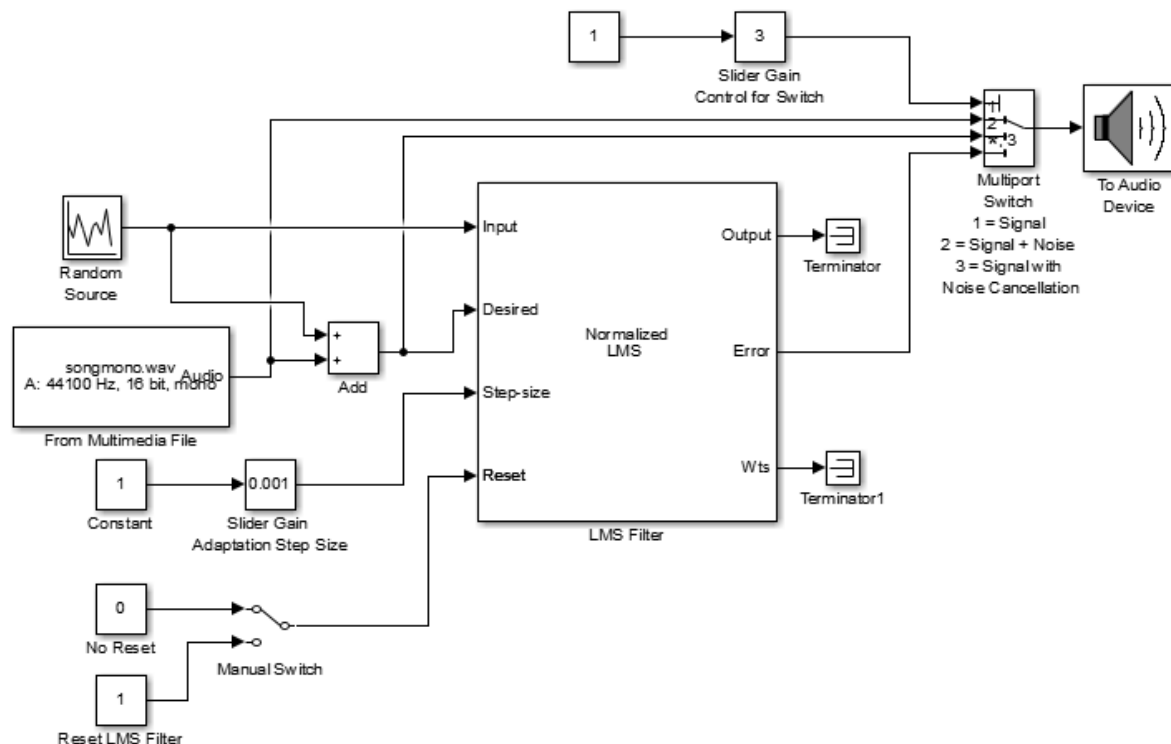


Figure 4: Noise Cancellation Model

6. On the Slider Gain Control for the switch, set the minimum to 1 and the maximum to 3. Setting this gain to 1 will cause the wav file to play. Setting the gain anywhere between 2 and 3 will cause the wav file with the noise added in to play. A gain of 3 will play the output of the LMS filter; that is the “noise reduced” wav file. Start with a slider gain of 3 (output of LMS filter).
7. The Slider Gain for the Adaptation Step Size should be set to a minimum of 0 and a maximum of 2. This slider will allow the user to adjust the adaptation step size while the simulation is running.
8. The manual switch allows the user to reset the LMS filter while the simulation is running.
9. For the LMS filter, set the algorithm to normalized LMS and the filter length to 32.
10. Run the simulation, listening to “noise reduced” or filtered signal (slider gain control switch on 3, the noisy wav file (slider gain control switch on 2 - the noise will really dominate here), and the original non-noisy signal (slider gain control switch on 1).

Note: when you first start the simulation, the noise-reduced signal sounds noisy because the LMS filter is still modeling the noise but you should notice that noise dies off within 5 or 6 seconds .

11. Experiment with the adaptation step size.

Does an increase in adaptation step size help or hurt in the noise reduction?

What happens if the adaptation step size is made very small?

How do these results compare to the effect of adaptation step size for system identification as explored in Part A?

12. Experiment with the size of the Normalized LMS filter and comment on the results.

CHAPTER 9 INTRODUCTION TO IMAGE PROCESSING

This chapter explores image processing and some of the many practical applications associated with image processing. The chapter begins with basic image terminology and descriptions of some of the common formats used for image files. Two dimensional linear filters for blurring, sharpening, or detecting edges in images are described in the second section. The third section explores filters for removing noise from images. The final section provides a detailed description of the JPEG format used for image compression and introduces the Discrete Cosine Transform (DCT) and Huffman codes.

9.1 TERMINOLOGY AND COMMON IMAGE FORMATS

Terminology

Pixel

Pixel is an abbreviation for Picture Element and represents the smallest component, a single rectangular cell, within a digital image. A 1024×768 digital image is simply a rectangular grid with 1024 columns and 768 rows with a total of 786,432 pixels. Larger images require more pixels so that the eye cannot distinguish individual pixels in the image. Individual pixels can be viewed by zooming in on a digital image as shown in Figure 9.1.

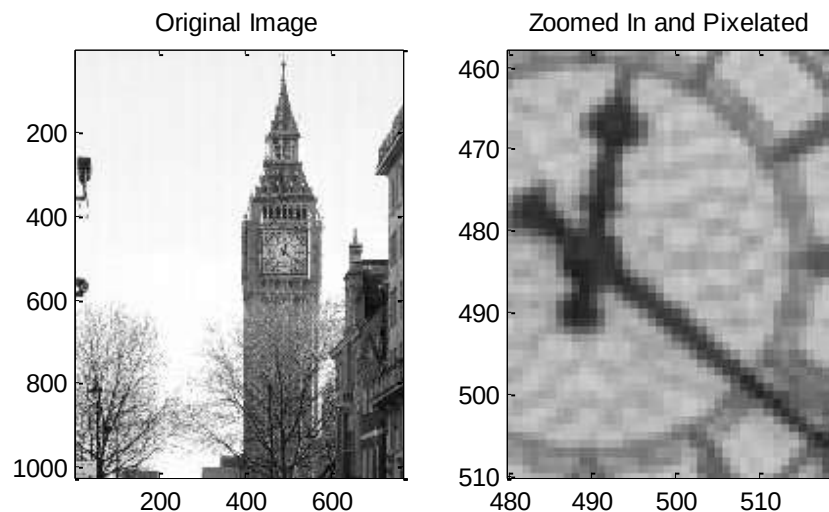


Figure 9.1: Individual Pixels in an Image

Color Depth

Color depth is the number of bits used to represent the color or shading for each pixel. Black and white images require only 1 bit per pixel (0 = black and 1 = white). Grayscale images use 8 bits per pixel resulting in 256 (2^8) different shades of grey ranging from black to white. For color images, more bits are needed to allow for a rich variety of colors. “True color” images use 24 bits per pixel with 8 bits for red, 8 bits for green, and 8 bits for blue which results in approximately 16.8 million color combinations ($256 \times 256 \times 256$). Higher end graphics systems may use 32 bits or 48 bits per pixel resulting in many more color combinations.

Example 9.1: Size of Raw Image File

How large is a 2736 x 3648 image file that uses 24 bits per pixel (8-red, 8-green, and 8-blue)? How many of these images could be stored in raw bit format on a 2 GB SD card?

Solution

A 2736 x 3648 image file has 9,980,928 pixels. At 24 bits per pixel, this image would require $239.542272 \cdot 10^6$ bits or 29.942784 MB (8 bits/byte) of storage space. Only about 67 of these digital images could be stored in raw bit format on a 2 GB SD card which is one good reason why digital cameras use special file formats to compress the size of each image.

Lossless and Lossy Compression

As shown in Example 9.1, images can become quite large as the number of pixels and the number of bits per pixel increase. There are many different formats for digital images that attempt to reduce or compress the size of the image. Any compression technique which allows the original image to be completely recovered for viewing or analysis is referred to as lossless compression. Compression techniques which cause some loss of information in the original image are referred to as lossy compression.

Frequency Content (Spectrum) of Images

The concept of the frequency of a sinusoidal signal is very easily understood; it is simply the number of cycles per second. The spectrum of a time signal reflects all the various frequencies contained within the time signal at appropriate magnitudes. How can an image have frequency content? Well, think of frequency as a measure of rate of change. The fine detail in an image (edges, texture, rapid color variations) is the high frequency content of an image. The uniform sections of an image are the low frequency content of an image.

Some Common Image Formats

BMP (Windows bitmap)

The .bmp format was designed for graphics files used within Microsoft Windows applications. The color depth ranges from 1 bit for black and white images to 24 bits for true color images. Files in this format are normally uncompressed and tend to be very large.

GIF (Graphics Interchange Format)

The .gif format uses a proprietary LZW (Lempel Zev Welch) lossless compression algorithm, and has a color depth of 8 bits per pixel which limits the number of colors to 256. This format works well for diagrams, cartoons, and other images that do not have a large variety of color or textures. The format also supports animation.

PNG (Portable Network Graphics)

The .png format was developed for Web pages as an open source or nonproprietary alternative to the .gif format. It utilizes a lossless compression technique and offers a much wider range of color than GIF with up to 48 bits per pixel. This format does not directly support animation. The PNG format works well for editing photographs but is not the best choice for final distribution or storage of photographs since a PNG file for a detailed photographic image with lots of texture and color is unacceptably large.

JPEG (Joint Photographic Expert Group)

The JPEG format uses a lossy compression technique that will be discussed in some detail in a later section of this chapter. It is the format of choice for storing or distributing photographic images. The JPEG format supports “true color” images (24 bits per pixel) and JPEG files are smaller than PNG files. Because the compression is lossy, repeatedly opening, editing, and re-saving JPEG files can result in poor image quality. A better alternative is to edit the photographic image in a lossless format such as the PNG format then save the final image as a JPEG file.

TIFF (Tagged Image File Format)

The TIFF format is flexible in that it allows for a choice of compression techniques including JPEG, LZW, and many others as well as a choice of color depth. While this flexibility results in a huge variety of TIFF file types, the TIFF format is not well supported by Web browsers and readers may read certain types of TIFF files and not others. Some digital cameras store images in the TIFF format.

9.2 IMAGE EFFECTS

Most of the effects discussed in this section will involve two dimensional linear filtering. Pixels in the new image are formed by combining the original pixel with a set of surrounding pixels in some linear manner. This concept was first introduced in Section 5.6 with moving average filters and Gaussian filters for smoothing or blurring images. To illustrate the concept, consider the following very simple weighting matrix or filter:

$$h = \begin{bmatrix} \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\ \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \end{bmatrix}$$

If an image were filtered with this weighting matrix, pixel (i, j) of the filtered image would be created by centering the filter over pixel (i, j) in the original image, taking the product of the filter coefficients and the image pixels under the filter, and then summing the resulting products. Figure 9.2 shows a 7 x 7 image matrix with the filter centered over pixel (3,4).

80	64	66	64	60	64	67
75	72	68 ^{1/16}	67 ^{1/8}	65 ^{1/16}	50	48
70	75	64 ^{1/8}	63 ^{1/4}	61 ^{1/8}	56	52
70	78	62 ^{1/16}	58 ^{1/8}	67 ^{1/16}	58	56
76	80	60	56	73	62	60
84	88	62	58	75	64	64
88	92	70	64	80	68	60

Figure 9.2: Filtering an Image

Pixel (3,4) of the filtered image would then be computed as follows:

$$\text{Filtered Pixel (3,4)} = \frac{1}{16} \times 68 + \frac{1}{8} \times 67 + \frac{1}{16} \times 65 + \frac{1}{8} \times 64 + \frac{1}{4} \times 63 + \frac{1}{8} \times 61 + \frac{1}{16} \times 62 + \frac{1}{8} \times 58 + \frac{1}{16} \times 67$$

The entire image is filtered by first centering the filter matrix over the top left pixel of the image, computing the new pixel value, then sliding the filter to the right by one pixel, computing a new pixel, and repeating till the end of the row is reached. The filter is then centered over the left-most pixel of the second row and new pixels are computed as the filter slides across the second row. The process is repeated until the last pixel in the image is reached. At the edge of the image, some of the filter coefficients will hang over the border. Non-existent pixels are assumed to be zero which can darken the border of the picture. There are other options for handling the border pixels. One option is to simply start and end the filtering process so that the filter will not hang over the border. In the simple example shown in Figure 9.2, the filtering would start in row 2 and column 2 and end in row 6 and column 6. Pixels in the first and last row or in the first and last column would not be filtered and would remain at the original values. Another option is to again start and end the filtering process where the filter won't hang over the border, then after filtering, eliminate all border pixels that were not filtered. This, of course, results in a slightly smaller image.

Importing, Exporting, and Displaying Images using MATLAB®

Images are imported into MATLAB using the **imread** command. The command **X = imread(filename, format);** will import the file specified by filename and format into an array X in the MATLAB workspace. The image file must be in the current directory. Images are exported from MATLAB using the **imwrite** command. The command **imwrite(X, filename, format);** will export the array, X, as an image file using the specified filename and format. MATLAB supports many types of image formats.

Images can be displayed using the **image** function or the **imshow** function. The command **image(X)** will display the array, X, as an image. If X is 2-d, a colormap should be specified. If X is 3-d, the image is displayed as a truecolor RGB image. X should be of type uint8 or of type double. If the pixels are type double, the values must range from 0 to 1. The function **imshow** is available only through the image processing toolbox. The command **imshow(X)** will display a 2d array as a grayscale image (no colormap required) and a 3-d array as a true color RGB image. **Imshow** preserves the scale of the image and does not add axis labels.

Filtering Images in MATLAB

Filtering images in MATLAB is easily accomplished using **filter2** for two dimensional filtering or **imfilter** for multi-dimensional images. The command **Imf = filter2(F, Im)** produces a 2-d image, Imf, that is exactly the same size as the original 2-d image, Im, but has been filtered by the filter matrix, F. There is an optional 3rd argument that allows some adjustment of the size of the resulting filtered image. The function **imfilter** is only available with the image processing toolbox and extends the filtering capability to multi-dimensional images. The command **Imf = imfilter(Im, F)** produces an image, Imf, that is exactly the same size as the original image, Im, but has been filtered by the filter array, F. **Notice the reversal in the 1st and 2nd input arguments**

between **filter2** and **imfilter**. The examples in this section will make use of **filter2** since this function does not require the image processing toolbox.

Converting Color (RGB) Images to Grayscale

Grayscale images will be used throughout this section, but the concepts easily extend to color images. The m-file, **bw.m**, (code shown in Figure 9.3) can be used to convert color jpeg files to grayscale images. The pixels in the resulting image will be unsigned 8 bit integers. A pixel value of 0 corresponds to the color black and a pixel value of 255 corresponds to the color white. Pixel values in between 0 and 255 represent various shades of grey. The **colormap(gray(256))** command is used to set up the figure window to display the image properly. Anyone with access to the Image Processing Toolbox in MATLAB could use the command **rgb2gray** instead of **bw.m**.

```
function Y = bw(X)
% Converts a jpeg image, X, into black and white image
% First need to read image into MATLAB:
% X = imread('filename','fmt');
% usage >> Y = bw(X);
dimensions = size(X); % Read the size of the image file
Rows = dimensions(1); Cols = dimensions(2);

% Convert Trucolor RGB Image to a Luminance (Black and White) Image
Y(1:Rows,1:Cols)=0.299*X(1:Rows,1:Cols,1)+0.587*X(1:Rows,1:Cols,2)+
0.114*X(1:Rows,1:Cols,3);
Y = uint8(Y);

% Setup the colormap of the figure for a black and white image
colormap(gray(256))
image(Y)
end
```

Figure 9.3: m-file to convert a JPEG Image to a Black and White Image

Adjusting Contrast

As mentioned previously, all images in this section will be grayscale images with pixels represented by 8 bit unsigned integers. The pixel values therefore range from 0 to 255 with a value of 0 representing black and a value of 255 representing white. An image that doesn't effectively span this allowable range may appear rather grey. The image contrast can be stretched or adjusted to make better use of the color range as follows:

$$Im(i,j) = \begin{cases} 0 & 0 \leq Orig(i,j) \leq Min \\ 255 * \left[\frac{Orig(i,j) - Min}{Max - Min} \right] & Min \leq Orig(i,j) \leq Max \\ 255 & Max \leq Orig(i,j) \leq 255 \end{cases} \quad (9.1)$$

This function simply takes all the image pixels in the range [Min, Max] and expands them linearly to occupy the full range [0, 255]. Any pixels below the Min value would be mapped to 0 and any pixels above the Max value would be mapped to 255. A nice choice for Min is a value which just exceeds 1% of the original pixel values while a nice choice for Max is a value that just exceeds 99% of the original pixel values. The image processing toolbox has functions called **imadjust** for adjusting contrast and **imhist** to display a histogram of the pixel values. The histogram is useful for choosing the threshold values, Min and Max.

Brighten or Darken

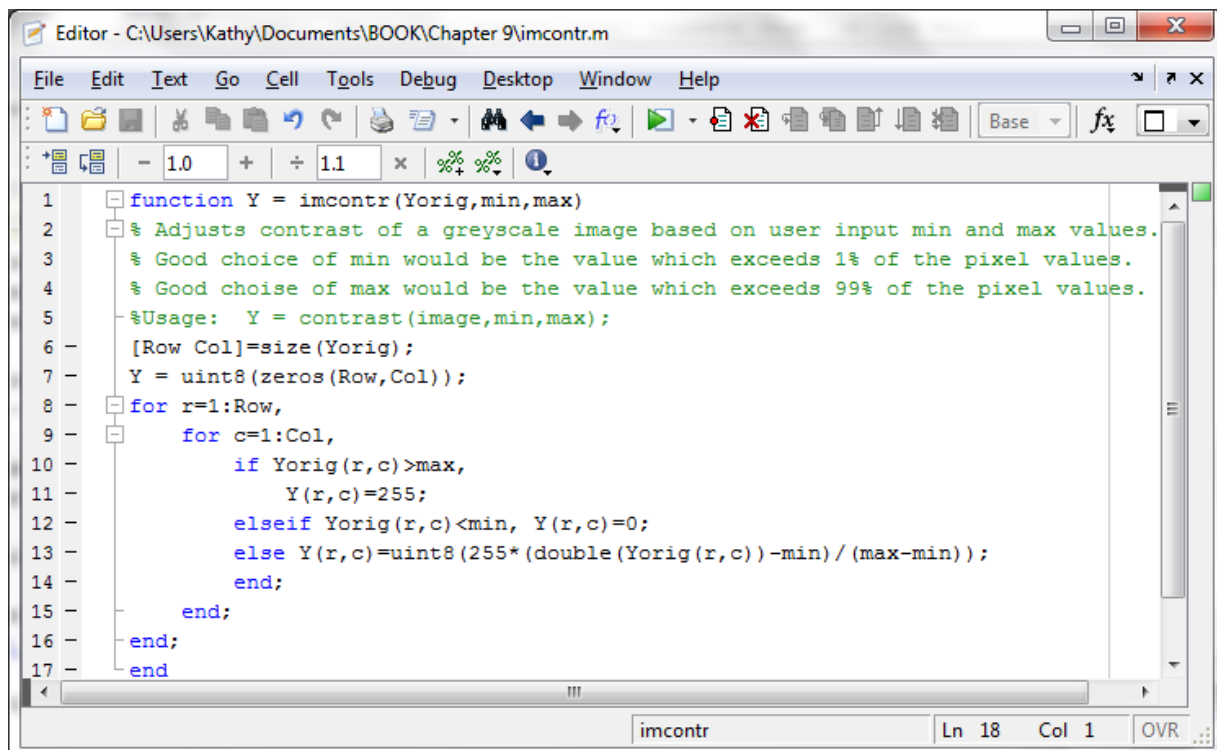
Images can be brightened either by multiplying all pixels in the image by some number larger than one or by adding some constant value to every pixel. Images are darkened by multiplying all pixels in the image by some number between zero and one or subtracting some constant value from each pixel. Any computed value outside of the allowable range of 0 to 255 will saturate.

Example 9.2: Adjusting Image Contrast and Brightness

An image of a butterfly is imported into MATLAB. First the contrast is adjusted then the image is scaled to brighten and darken the image.

Solution

An m-file, **imcontr.m**, to perform the contrast adjustment was created (students with access to the image processing toolbox could simply use **imadjust**):



```
1 function Y = imcontr(Yorig,min,max)
2 % Adjusts contrast of a greyscale image based on user input min and max values.
3 % Good choice of min would be the value which exceeds 1% of the pixel values.
4 % Good choice of max would be the value which exceeds 99% of the pixel values.
5 %Usage: Y = contrast(image,min,max);
6 [Row Col]=size(Yorig);
7 Y = uint8(zeros(Row,Col));
8 for r=1:Row,
9     for c=1:Col,
10         if Yorig(r,c)>max,
11             Y(r,c)=255;
12         elseif Yorig(r,c)<min, Y(r,c)=0;
13         else Y(r,c)=uint8(255*(double(Yorig(r,c))-min)/(max-min));
14         end;
15     end;
16 end;
17 end
```

The following commands import the image into MATLAB, convert the image to grayscale, adjust the contrast, and then lighten and darken the image:

```
X = imread('Butterfly2','jpeg');           % Import a jpeg image
Yorig = bw(X);                             % Convert to Grayscale
imshow(Yorig);
Yc = imcontr(Yorig,40,220);                 % Adjust Contrast
imshow(Yc)

BW = imcontr(Yorig,120,121);                % Purely Black/White Image
imshow(BW);

imshow(Yc+60);                             % Lighten by addition
imshow(Yc*1.5);                             % Lighten by multiplication
imshow(Yc*5);                               % Over-lightened image
(Yc-60);                                    % Darken by subtraction
imshow(Yc*0.75);                             % Darken by multiplication
imshow(Yc*0.5);                             % Too Dark
```

The effects are illustrated in Figure 9.4. Notice that the image with the contrast adjustment makes much better use of the grayscale range.

The purely black and white image was created by choosing two consecutive values for Min and Max (120 and 121) in ***imcontr.m*** so all pixel values at or above 121 became white and all pixel values 120 and below became black. Notice that brightening an image by multiplication has no effect on the color black since black has a value of zero.

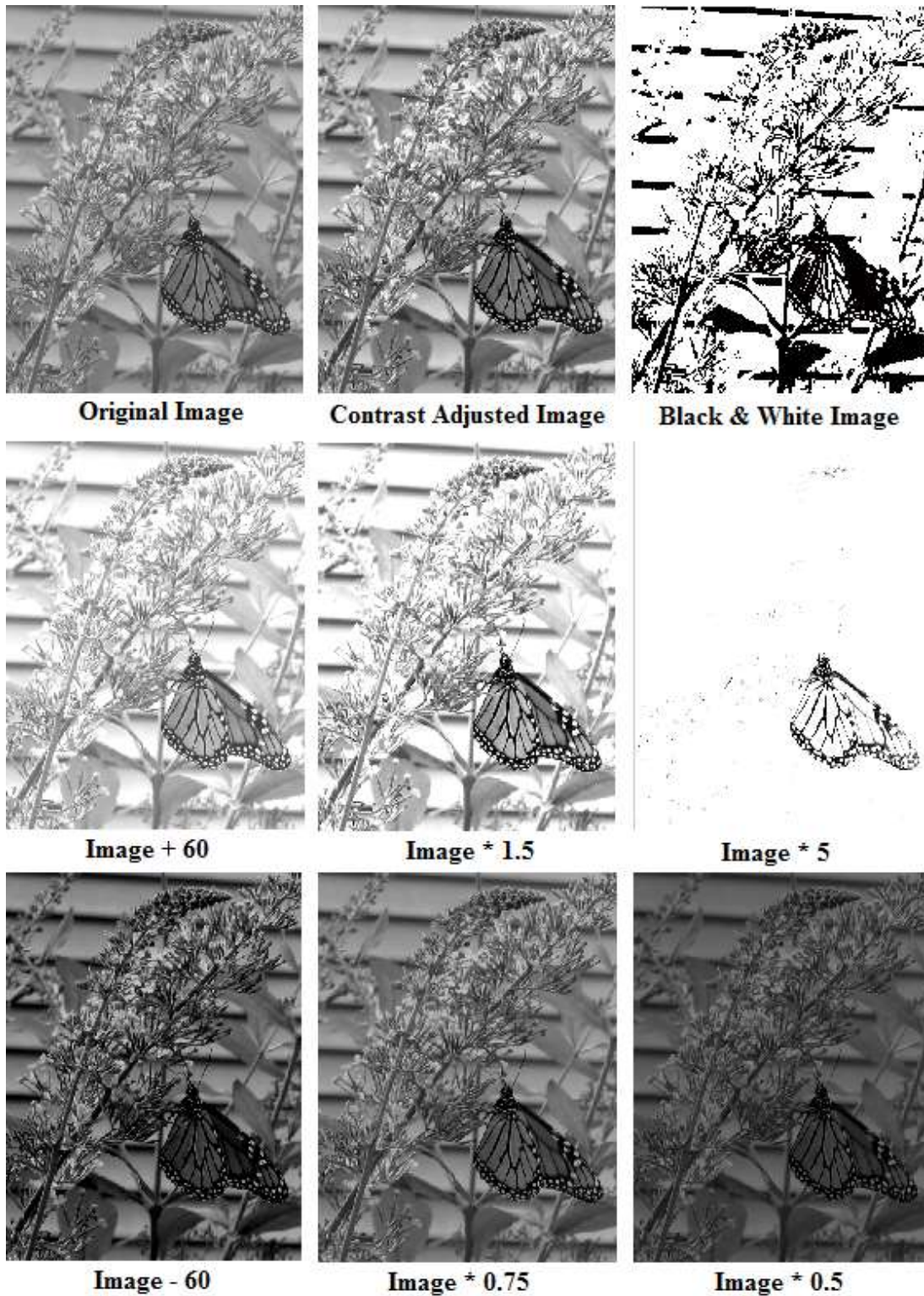


Figure 9.4: Adjusting Image Contrast and Brightness

Blurring

An image is blurred or softened by using a low-pass filter to blend surrounding pixels. The pixels in a section could all simply be averaged using a two dimensional moving average filter; however, a Gaussian filter provides a nice softening effect without sacrificing as much detail in the image. A Gaussian Filter has a bell curve shape with the sharpness of the curve defined by the standard deviation. The bell curve becomes wider as the standard deviation is increased. The $N \times N$ Gaussian filter matrix is defined by the following equation for the $(i, j)^{\text{th}}$ entry of the matrix:

$$h(i,j) = e^{-\frac{1}{2}[(i-\frac{N}{2})^2+(j-\frac{N}{2})^2]/\sigma^2} \quad \text{for } i,j = 0 \dots N$$

σ = standard deviation

(9.2)

The code for a function, `blurfilt`, used to create a Gaussian filter matrix that allows the user to select size, N , and standard deviation, σ , is shown in Figure 9.5.

```
function h = blurfilt(N,Sigma)
% This function creates an NxN Gaussian filter
% with standard deviation, Sigma.
% Usage >> h = blurfilt(N,Sigma);

h = zeros(N,N);
for r=1:N,
    for c=1:N,
        h(r,c)=exp(-0.5*((r-ceil(N/2))^2+(c-ceil(N/2))^2)/(Sigma^2));
    end;
end;

h=h/(sum(sum(h)));
```

Figure 9.5: MATLAB Function for Creating NxN Gaussian Filter

Anyone with access to the image processing toolbox in MATLAB could create the same filter using the command

```
h = fspecial('gaussian',N,Sigma);
```

The function, `blurfilt`, is run for size $N=5$ and various choices for standard deviation.

```
h1=blurfilt(5,0.2)
h1 =
    0.0000    0.0000    0.0000    0.0000    0.0000
    0.0000    0.0000    0.0000    0.0000    0.0000
    0.0000    0.0000    1.0000    0.0000    0.0000
    0.0000    0.0000    0.0000    0.0000    0.0000
    0.0000    0.0000    0.0000    0.0000    0.0000
```

This weighting matrix appears to do no filtering. All pixels surrounding the center pixel are multiplied by zero while the center pixel is multiplied by 1. Actually, if we expanded the precision for displaying h1, we would see a couple of the surrounding pixels do get multiplied by really small values but the effect on an image would not be noticeable.

```
h2=blurfilt(5,0.5)
h2 =
    0.0000    0.0000    0.0002    0.0000    0.0000
    0.0000    0.0113    0.0837    0.0113    0.0000
    0.0002    0.0837    0.6187    0.0837    0.0002
    0.0000    0.0113    0.0837    0.0113    0.0000
    0.0000    0.0000    0.0002    0.0000    0.0000
```

```
h3=blurfilt(5,1)
h3 =
    0.0030    0.0133    0.0219    0.0133    0.0030
    0.0133    0.0596    0.0983    0.0596    0.0133
    0.0219    0.0983    0.1621    0.0983    0.0219
    0.0133    0.0596    0.0983    0.0596    0.0133
    0.0030    0.0133    0.0219    0.0133    0.0030
```

In the case of h2 and h3, the surrounding pixels are combined with the center pixel. Notice that the center pixel is weighted by the highest value while the weights of surrounding pixels drop off.

```
h4=blurfilt(5,6)
h4 =
    0.0378    0.0394    0.0400    0.0394    0.0378
    0.0394    0.0411    0.0417    0.0411    0.0394
    0.0400    0.0417    0.0423    0.0417    0.0400
    0.0394    0.0411    0.0417    0.0411    0.0394
    0.0378    0.0394    0.0400    0.0394    0.0378
```

The standard deviation is set so high for h4 that all pixels are basically weighted by the same value. As standard deviation is increased, a Gaussian filter will widen out and start to look like a moving average filter.

In Figure 9.6, the 5 x 5 Gaussian filter matrices are plotted using the **mesh** command to provide a nice visual illustration:

```
subplot(2,2,1); mesh(h1);
subplot(2,2,2); mesh(h2);
subplot(2,2,3); mesh(h3);
subplot(2,2,4); mesh(h4);
```

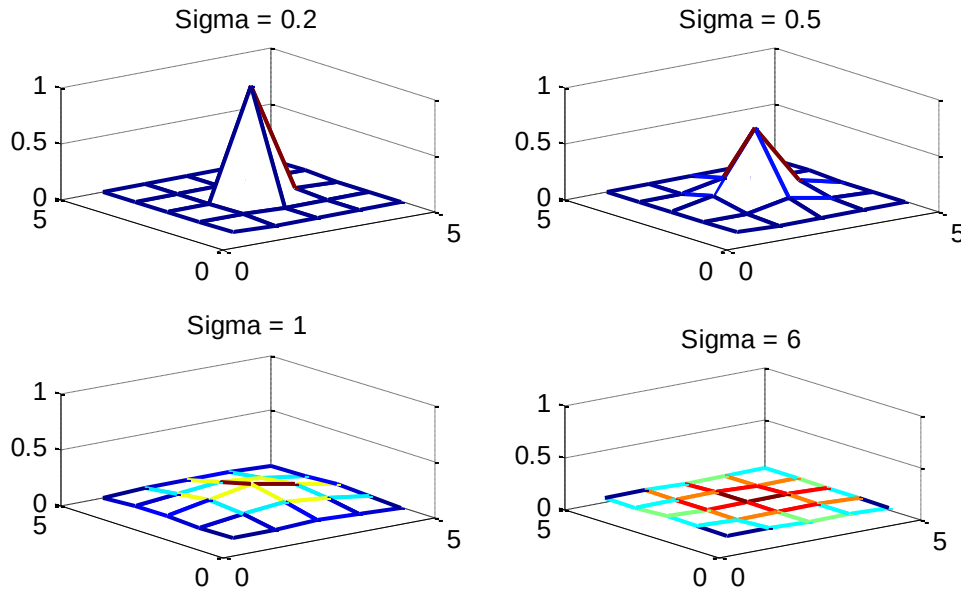


Figure 9.6: Gaussian Filter (5x5) with Varying Standard Deviation

It should also be noted that the sum of all the values in the Gaussian filter matrix is one regardless of the size of the matrix or the standard deviation. This ensures that the new pixels of the filtered image will stay within the range of the image file to avoid saturation.

Example 9.3: Blurring an Image with Gaussian Filters

An image is imported into MATLAB then filtered with several different Gaussian filters to blur the image.

Solution

The following commands import a jpeg file into MATLAB, convert the image to black and white, and then filter the image with three different Gaussian filters.

```
X = imread('Butterfly2','jpeg');
Y = bw(X);
Y = 1.2*Y;      % Lighten Image a bit before Blurring
Y1 = filter2(blurfilt(5,2),Y);
Y2 = filter2(blurfilt(10,2),Y);
Y3 = filter2(blurfilt(10,12),Y);
colormap(gray(256));
subplot(2,2,1); imagesc(Y); subplot(2,2,2); imagesc(Y1);
subplot(2,2,3); imagesc(Y2); subplot(2,2,4); imagesc(Y3)
```


The effects are illustrated in Figure 9.7. The first filter (top right image) has only a slight softening effect on the original image. The second filter (bottom left image) uses the same standard deviation as the first but expands the size of the filter so 10x10 sections of the image are processed rather than 5x5 sections. More softening of the fine details in the original image is now apparent. The last filter (bottom right image) processes 10x10 sections of the image, but the standard deviation of the Gaussian function is increased to 12. As mentioned previously, increasing the standard deviation, σ , widens the Gaussian filter. A standard deviation of twelve will essentially weights all pixels in the 10x10 section equally making this filter a 2-d moving average filter. This results in an image that is considerably more blurred than the image shown in the bottom left of Figure 9.7. A sharper Gaussian filter will soften an image while preserving the detail, while a wider Gaussian filter blurs the details.

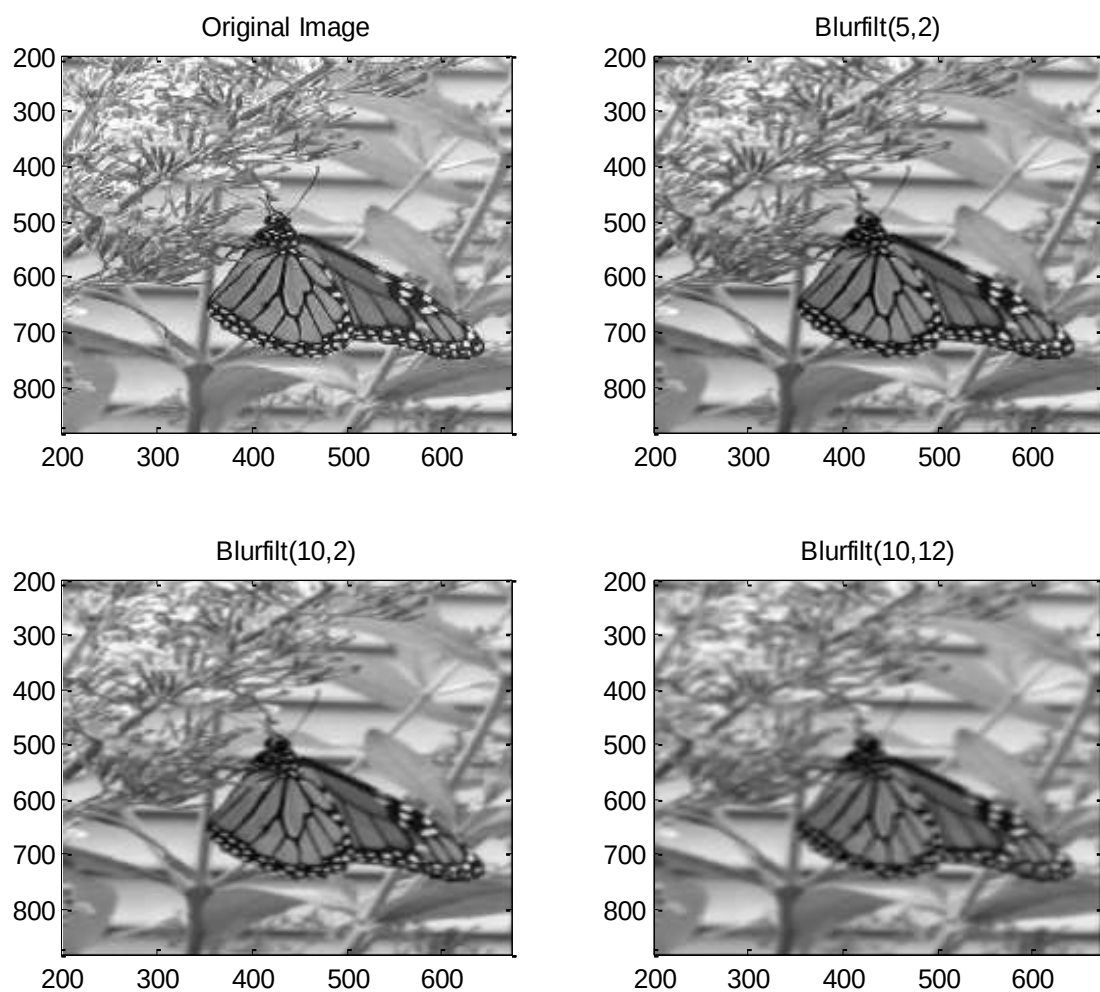


Figure 9.7: Blurring Images

Sharpening

An image is sharpened by using a high-pass filter to accentuate the detail in the image. A very simple filter that sharpens the detail in an image by basically taking a first derivative is described by:

$$h = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad (9.3)$$

This filter processes 3x3 sections of an image. If all the pixels in a 3x3 section are exactly the same (no color variation whatsoever), then the new pixel will remain exactly the same as the original pixel. However, any significant variations in an image section will be accentuated.

To see this, create a very simple two-tone image in MATLAB, filter the image, and then plot the results using the following set of commands:

```
Y = [50*ones(8,4) 150*ones(8,4)]

Y =
    50    50    50    50   150   150   150   150
    50    50    50    50   150   150   150   150
    50    50    50    50   150   150   150   150
    50    50    50    50   150   150   150   150
    50    50    50    50   150   150   150   150
    50    50    50    50   150   150   150   150
    50    50    50    50   150   150   150   150
    50    50    50    50   150   150   150   150

h = [0 -1 0;-1 5 -1; 0 -1 0];

Y_sharp = filter2(h,Y)

Y_sharp =
   150   100   100     0   400   300   300   450
   100    50    50   -50   250   150   150   300
   100    50    50   -50   250   150   150   300
   100    50    50   -50   250   150   150   300
   100    50    50   -50   250   150   150   300
   100    50    50   -50   250   150   150   300
   100    50    50   -50   250   150   150   300
   150   100   100     0   400   300   300   450

colormap(gray(256));
subplot(1,2,1); image(Y);
subplot(1,2,2); image(Y_sharp)
```

The original image and the filtered image are shown in Figure 9.8. Notice that the color contrast in the center of the image is amplified by the sharpening filter. The filtered image matches the original in the areas where the original has no color variation.

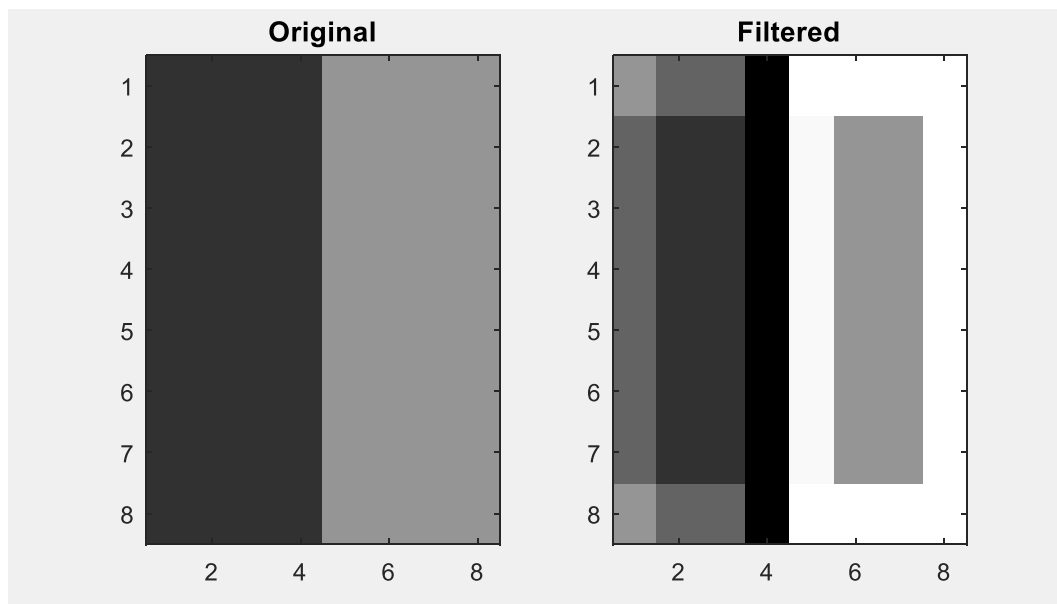


Figure 9.8: Effect of Sharpening Filter on Two-Tone Image

Challenge Question 9.1

Why is there a distinct border around the left and right sides of the filtered image in Figure 9.8?

Example 9.4: Sharpening an Image

An image is imported into MATLAB then filtered to sharpen the image.

Solution

The following commands import a jpeg file into MATLAB, convert the image to black and white, and then filter the image to sharpen it.

```
X=imread('Butterfly2','jpeg');
Y=bw(X);
Y = 1.2*Y; % Lighten Image a bit before Sharpening
h = [0 -1 0; -1 5 -1; 0 -1 0];
colormap(gray(256));
subplot(1,2,1); image(Y);
subplot(1,2,2); image(filter2(h,Y));
```

The original image and the filtered image are shown in Figure 9.9. The filter definitely accentuates the fine details in the image.

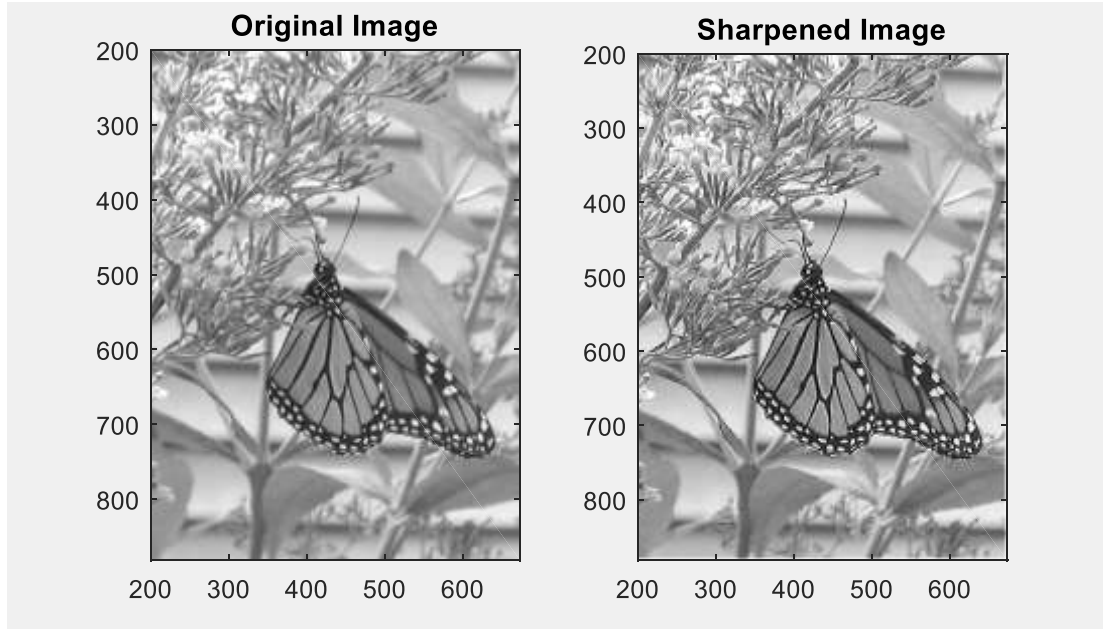


Figure 9.9: Sharpening an Image (Example 9.4)

The image processing toolbox in MATLAB uses the following filter matrix for sharpening an image:

$$h = \frac{1}{\alpha+1} \begin{bmatrix} -\alpha & \alpha-1 & -\alpha \\ \alpha-1 & \alpha+5 & \alpha-1 \\ -\alpha & \alpha-1 & -\alpha \end{bmatrix} \quad 0 \leq \alpha \leq 1 \quad (9.4)$$

If $\alpha = 0$, the matrix is identical to the filter used in Example 9.4. For any value of α , the sum of all entries in the matrix h is one. This ensures that the new pixels of the filtered image will stay within the range of the image file to avoid saturation. The matrix h can be entered directly into MATLAB for any value of α or, if the image processing toolbox is available, created using the command *fspecial('unsharp',alpha)*. Although 'unsharp' is a counterintuitive term to use in this command, it does indeed produce a filter that sharpens the image.

Edge Detection

Sharpening filters accentuate the detail or contrast in an image while leaving uniform areas unchanged. Edge detection filters accentuate the high contrast areas or edges of an image while essentially filtering out the low frequency portions of an image. One type of edge detection filter is a Sobel filter described by:

$$h = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad v = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad (9.5)$$

The matrix h emphasizes the horizontal differences in an image while matrix v emphasizes the vertical edges. An image can be filtered with each matrix separately then the results can be combined in a gradient as follows:

$$Im_f = \sqrt{Im_h^2 + Im_v^2} \quad (9.6)$$

Example 9.5: Edge Detecting using Sobel Matrix

Take the Butterfly image used in Example 9.4 and filter it using the Sobel matrix.

Solution

The following commands import a jpeg file into MATLAB, convert the image to black and white, and then filter the image to accentuate the horizontal and vertical edges.

```
X = imread('Butterfly2','jpeg');
Y = bw(X);
Y = imcontr(Y,40,200); % Adjust the image contrast
h = [1 2 1; 0 0 0; -1 -2 -1]; v = h';
Yh = imfilter(Y,h); Yv = imfilter(Y,v);
Yf = uint8(sqrt(double(Yh).^2 + double(Yv).^2));
colormap(gray(256));
subplot(2,2,1); image(Y); title('Grayscale Image');
subplot(2,2,2); image(Yh); title('Horizontal Filter');
subplot(2,2,3); image(Yv); title('Vertical Filter');
subplot(2,2,4); image(Yf);
title('Horizontal and Vertical Filters');
```

The results are shown in Figure 9.10. Notice that the horizontal lines of the aluminum siding in the background of the original image are detected by the horizontal filter (top right image) but not by the vertical filter (bottom left image). The vertical branch of the butterfly bush is evident in the image filtered by the vertical filter but does not appear in the image filtered by the horizontal filter. The lines on the butterfly wings are on a diagonal and therefore appear in both filtered images. The image on the bottom right formed by combining the horizontally filtered image and the vertically filtered image emphasizes all edges or areas of contrast in the image.

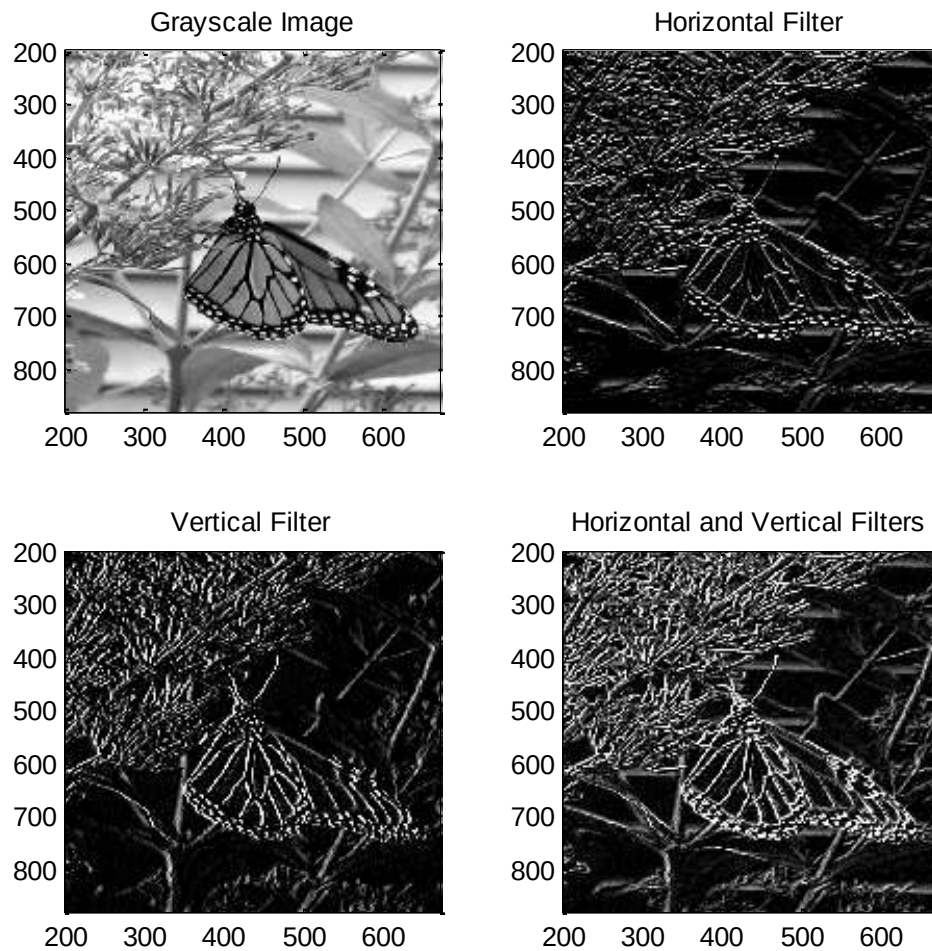


Figure 9.10: Edge Detection Using a Sobel Filter (Example 9.5)

Comments:

- There are many other types of blurring, sharpening, and edge detection filters other than the types covered in this section.
- If the filter matrix, h , can be separated into two vectors, $h = h_{vertical} \times h_{horizontal}$, it is computationally advantageous to perform the 2-D filtering using two one-dimensional convolutions (one along the rows and followed by another along the columns). For example, the Sobel matrix can be split as follows:

$$h = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \times [1 \quad 2 \quad 1]$$

Challenge Question 9.2

Explain the second comment. What is the savings in terms of the number of multiplications if the single 2-D matrix convolution is replaced with two 1-D convolutions? Also, which of the filter matrices discussed in this section can be broken up into vertical and horizontal vectors?

9.3 DE-NOISING IMAGES

There are several possible sources for noisy images: scanning a print or producing digital images from slides or video tape, electronic transmission of a digital image file, or the digital camera itself. Some digital cameras produce excellent images outdoors under good lighting conditions but create somewhat grainy images in settings with poor lighting such as indoors or at night. Image noise can be reduced using digital filters. There are many possible choices of digital filters to reduce noise and the optimal choice will depend on the type of noise in the image. Moving average filters or Gaussian filters will reduce grainy noise in an image, but these filters will also eliminate some of the fine detail in the image which will cause the image to blur. Refer back to Figure 5.27 in Chapter 5. The Gaussian filters and the averaging filter reduced the graininess present in the dog's fur but also blurred the image to some degree depending on the filter size.

Alternatives to averaging or Gaussian filters for noise reduction include median filters, Wiener filters, and many other image de-noising algorithms that are beyond the scope of this text. The median filter simply replaces each original pixel value with the median value of that pixel combined with a set of surrounding pixels. It is very similar to an averaging filter; however, calculating a median instead of an average significantly reduces the effects of outliers. The Wiener filter also provides smoothing but the degree of smoothing is adjusted based on the variance among the neighboring pixels. In the areas of the image with high variance (sharp detail), less smoothing is used; whereas, more smoothing is applied to the areas of the image with small variance.

Figure 9.11 illustrates the effect of various filters on reducing “salt and pepper” noise. Salt and pepper noise means that random pixels throughout the image are set to either white or black. The original image is shown in Figure 9.11(a) and the noisy image shown in Figure 9.11(b). The effect of various filters in reducing the noise is shown in Figure 9.11 (c-f).

The 3x3 median filter effectively eliminates the noise but also causes some reduction in image detail and contrast. Those extra white and black pixels introduced as noise are basically thrown out as outliers in the median calculation. The loss of contrast can easily be adjusted using contrast enhancement as discussed in the previous section. The 3x3 averaging filter does not do an effective job of eliminating the salt and pepper noise since those extra black and white pixels are averaged right in with the surrounding pixels. While increasing the averaging filter order to 8x8 certainly decreases the noise level, it also results in significant blurring of the image. The Wiener filter is the least effective of the three filters investigated for reducing salt & pepper noise. Some of the white and black noisy pixels are still present in the filtered image and the Wiener filter causes more blurring than the averaging filter of comparable size.



(a) Original Image



(b) Image with Salt & Pepper Noise



(c) Noisy Image Filtered by 3x3 Median Filter



(d) Noisy Image Filtered by 3x3 Averaging Filter



(e) Noisy Image Filtered by 8x8 Averaging Filter



(f) Noisy Image Filtered by 8x8 Wiener Filter

Figure 9.11: Digital Filtering to Reduce Salt & Pepper Noise

The MATLAB code for creating the noisy image and the filtered images of Figure 9.11 utilizes several files available within the Image Processing Toolbox:

```
% Read image, convert to Greyscale (bw.m), then display
Y=imread('Cypress2.jpg'); Y=bw(Y); imshow(Y);

% Add Salt & Pepper Noise (Density 1%) to image and display
figure; Ynoise=imnoise(Y,'salt & pepper',0.01);
imshow(Ynoise);

% Filter with a 3x3 Median filter
figure; Ymed=medfilt2(Ynoise,[3 3]); imshow(Ymed);

% Filter with a 3x3 and 8x8 Averaging Filters
figure; Yavg=filter2(fspecial('average',3),Ynoise)/255;
imshow(Yavg);

figure; Yavg=filter2(fspecial('average',8),Ynoise)/255;
imshow(Yavg);

% Filter with an 8x8 Wiener Filter
figure; Yw=wiener2(Ynoise, [8 8]); imshow(Yw);
```

Figure 9.12 illustrates the effect of various filters on reducing Gaussian noise in an image. The original image and corrupted image are shown in Figure 9.12 (a) and 9.12 (b) respectively. The filtered images using a 4x4 median filter and a 4x4 Wiener filter are shown in Figure 9.12 (c-d). In the case of Gaussian noise, the Wiener filter does a much better job than the median filter of reducing the Gaussian noise and preserving the detail in the image. Clearly, the choice of filter type and size for removing image noise will depend on the type of noise and the extent of the noise corruption. The MATLAB code for creating the images in Figure 9.12 is:

```
% Read image, convert to grayscale (bw.m), then display:
Y=imread('Cypress2.jpg'); Y=bw(Y); imshow(Y);

% Add Gaussian noise: mean = 0, variance = 0.005
Ynoise=imnoise(Y,'gaussian',0,0.005);imshow(Ynoise);

% Filter noisy image with 4x4 Median filter
figure; Ymed=medfilt2(Ynoise,[4 4]); imshow(Ymed);

% Filter noisy image with 4x4 Wiener filter
figure; Yw=wiener2(Ynoise, [4 4]); imshow(Yw);
```

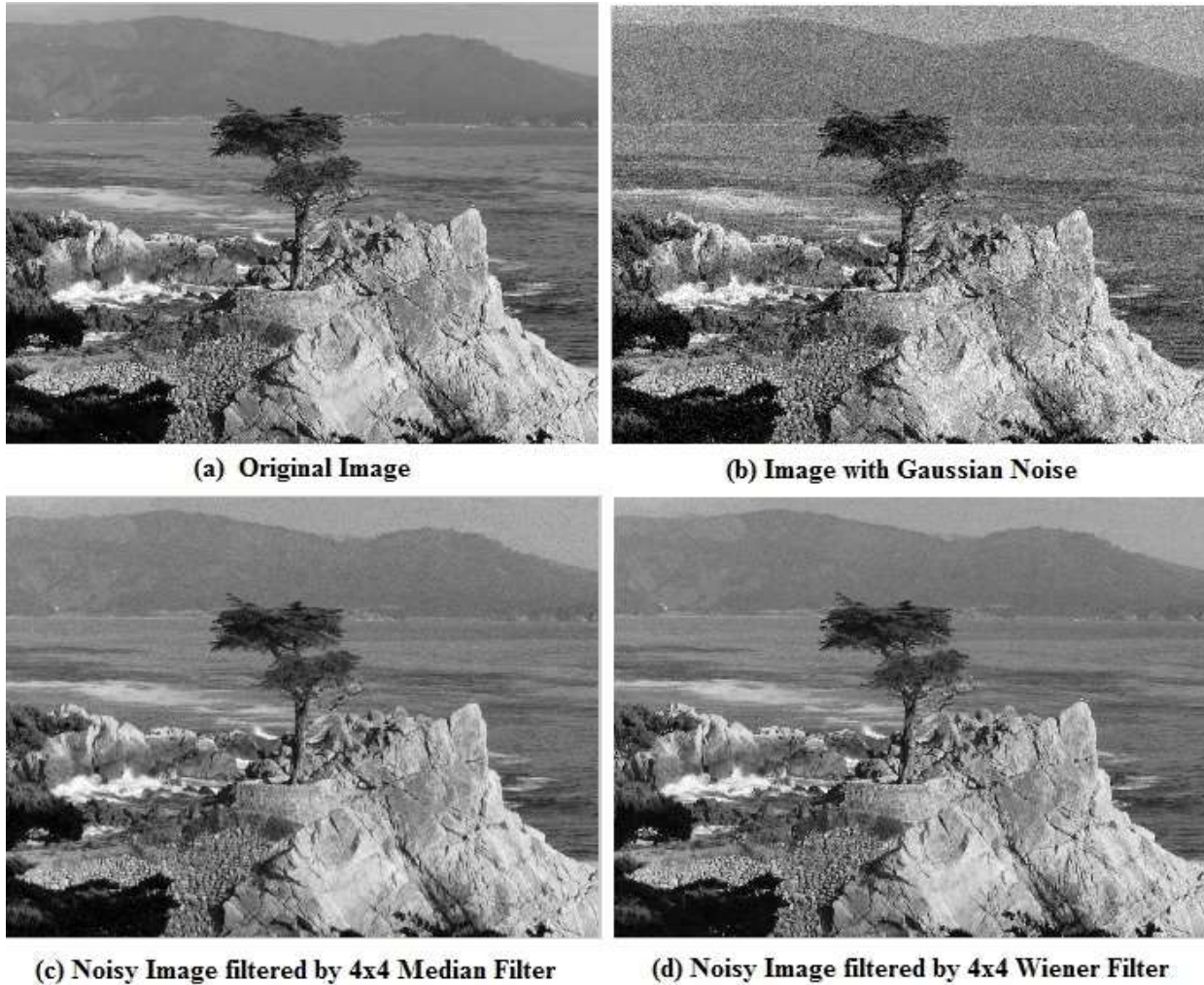



Figure 9.12: Digital Filtering to Reduce Gaussian Noise

9.4 IMAGE COMPRESSION: JPEG FORMAT

As illustrated in Example 9.1, image sizes can be very big resulting in large memory storage requirements and unacceptably long download or transmission times. Image compression algorithms attempt to reduce image sizes while preserving the quality of the image. The JPEG format is commonly used for photographic images and uses a lossy compression algorithm to reduce image size. The JPEG compression algorithm is illustrated in Figure 9.13. As indicated in the figure, the JPEG compression algorithm is pretty sophisticated and involves color space conversion, a Discrete Cosine Transform, and Huffman coding. Displaying an image from a JPEG file requires running the process in reverse. The processing that occurs in each of the blocks of Figure 9.13 will be discussed in detail in this section.

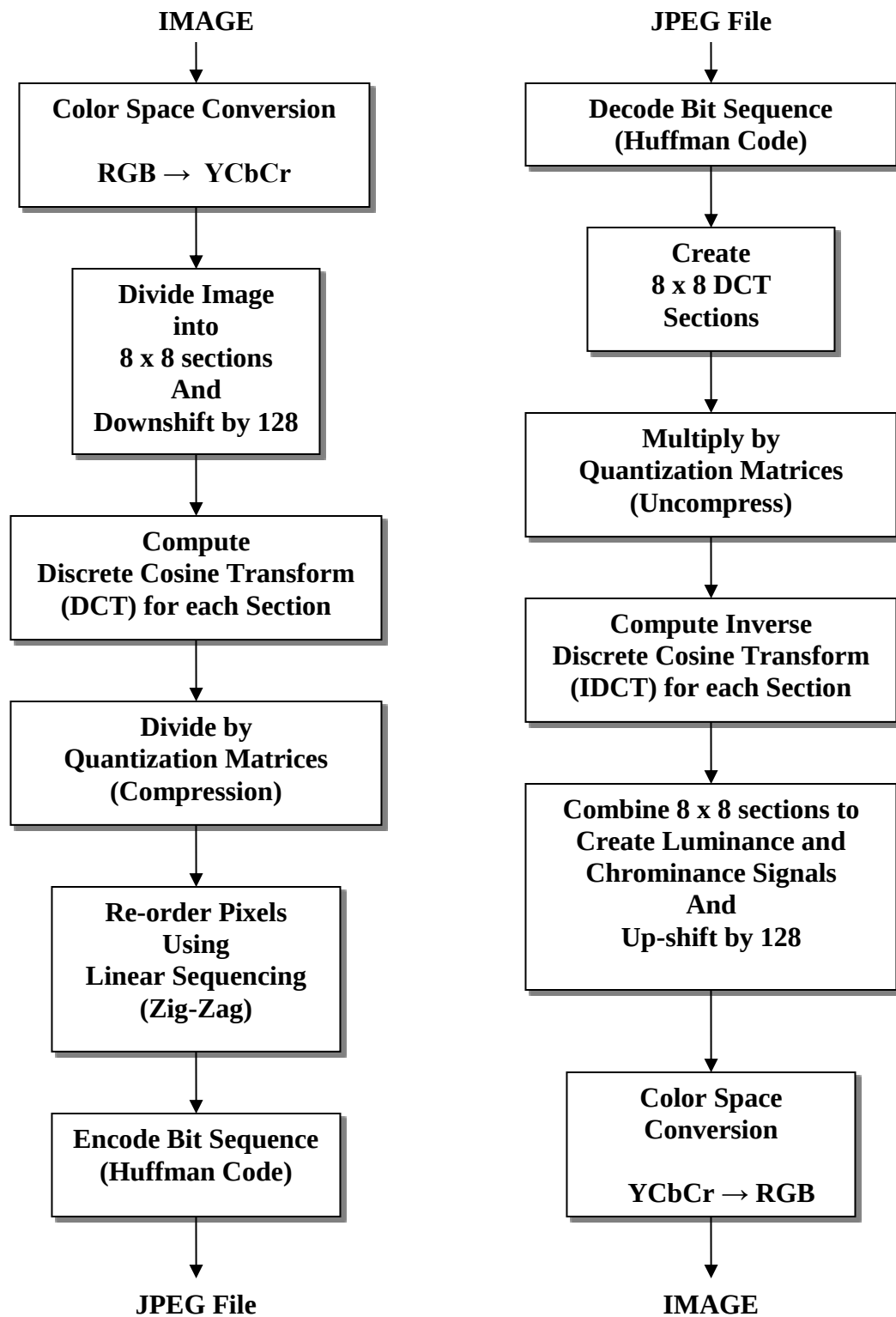


Figure 9.13: JPEG Compression Algorithm

Color Space Conversion

The first step in the JPEG compression is to convert from the RGB (Red – Green – Blue) color space to the YCbCr (Luminance – Chrominance Blue – Chrominance Red) color space. The conversion equations are:

$$\begin{aligned} Y &= 0.299 \times R + 0.587 \times G + 0.114 \times B \\ C_b &= 0.564 \times (B - Y) \\ C_r &= 0.713 \times (R - Y) \end{aligned} \quad (9.7)$$

The Y signal is the luminance (brightness) component. The Cb (chrominance blue) and Cr (chrominance red) signals are the color components. Color space conversion was first developed so that black and white televisions would be able to receive and properly decode the signals transmitted for color television. Black and white television receivers would only pick up the luminance signal and therefore still operate properly.

Color conversion is useful for images because the human eye is very sensitive to high frequency (rapid variation in) brightness or luminance but is not sensitive to high frequency chrominance or color information. This fact allows the two chrominance signals to be compressed or quantized considerably more than the luminance signal.

The equations to recover the RGB signals from the luminance and chrominance signals are:

$$\begin{aligned} R &= Y + 1.4 \times C_r \\ G &= Y - 0.344 \times C_b - 0.714 \times C_r \\ B &= Y + 1.77 \times C_b \end{aligned} \quad (9.8)$$

Discrete Cosine Transform (DCT)

The JPEG compression algorithm next divides the image into non-overlapping 8×8 square sections resulting in 64 pixels per section. A two-dimensional Discrete Cosine Transform (DCT) is computed for each section. An N-point DCT is computed as follows:

$$\begin{aligned} X_{DCT}(n) &= w(n) \sum_{k=1}^N x(k) \cos \frac{\pi(2k-1)(n-1)}{2N} \quad n = 1, \dots, N \\ w(n) &= \begin{cases} \frac{1}{\sqrt{N}} & n = 1 \\ \sqrt{\frac{2}{N}} & 2 \leq n \leq N \end{cases} \end{aligned} \quad (9.9)$$

Notice the similarities between an N-point DCT and an N-point DFT. Both transforms take N points of a signal and produce N “spectral” points. The discrete cosine transform fits the signal to cosine weighting functions rather than the exponential weighting functions associated with a DFT. Hence, the DCT does not require any complex number arithmetic, all values are strictly real. One method for rapidly computing a DCT is to first compute a DFT using a computationally efficient FFT algorithm and then extract the real part to find the DCT. There are also efficient DCT algorithms designed for commonly used sizes such as N=8.

For JPEG, a two-dimensional 8-point DCT is computed for each 8×8 section of the image. In other words, each non-overlapping 8x8 image section is converted into an 8x8 DCT spectrum. One way to accomplish this is to first calculate an 8-point DCT on each column of the original section then compute an 8-point DCT on each row of the “column processed” section. The pixel in the i^{th} row and j^{th} column of the DCT spectrum is:

$$X_{\text{DCT}}(i, j) = W_i \times [8 \times 8 \text{ image section}] \times W_j^T \quad i = 1, \dots, 8 \text{ and } j = 1, \dots, 8 \quad (9.10)$$

The weight vectors are calculated as:

$$W_n = \begin{cases} \frac{1}{\sqrt{N}} \cos\left(\frac{(2k-1)(n-1)}{2 * 8}\right) & n = 1 \text{ and } k = 1, \dots, 8 \\ \sqrt{\frac{2}{N}} \cos\left(\frac{(2k-1)(n-1)}{2 * 8}\right) & n = 2, \dots, N \text{ and } k = 1, \dots, 8 \end{cases} \quad (9.11)$$

For an 8-point DCT, the weight vectors are:

W₁ =	0.35355	0.35355	0.35355	0.35355	0.35355	0.35355	0.35355	0.35355
W₂ =	0.49039	0.41573	0.27778	0.09754	-0.09755	-0.27779	-0.41573	-0.49039
W₃ =	0.46194	0.19134	-0.19134	-0.46194	-0.46194	-0.19134	0.191342	0.46194
W₄ =	0.415735	-0.0975	-0.49039	-0.27779	0.277785	0.490393	0.097545	-0.41573
W₅ =	0.353553	-0.3535	-0.35355	0.353553	0.353553	-0.35355	-0.35355	0.353553
W₆ =	0.277785	-0.4904	0.097545	0.415735	-0.41573	-0.09755	0.490393	-0.27779
W₇ =	0.191342	-0.4619	0.46194	-0.19134	-0.19134	0.46194	-0.46194	0.191342
W₈ =	0.097545	-0.2778	0.415735	-0.49039	0.490393	-0.41573	0.277785	-0.09755

This is pretty difficult to visualize since it is just a bunch of numbers. To help visualize the DCT, start by re-writing Equation 9.10 as follows:

$$X_{\text{DCT}}(i,j) = \sum_{\text{all entries}} [W_i^T \times W_j] .* [8 \times 8 \text{ image section}] \text{ for } i = 1, \dots, 8 \text{ and } j = 1, \dots, 8$$

.* indicates entry by entry multiplication (9.12)

For a DCT spectrum, each non-overlapping 8x8 image section is multiplied (entry by entry) by 64 different 8x8 weighting matrices, and the products are summed to calculate the pixels in the DCT spectrum. So, each of the 64 weighting matrices produces one pixel in the 8x8 DCT spectrum.

For example, the pixel in the top left corner (row 1 and column 1) is computed by first multiplying the image section (entry by entry) by the weighting matrix:

$$W1^T \times W1 = \begin{bmatrix} 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 \\ 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 \\ 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 \\ 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 \\ 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 \\ 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 \\ 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 \\ 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 \\ 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 \end{bmatrix}$$

then summing all the entries in the resulting matrix. In other words, the pixel in the top left corner is computed by weighting every single pixel in the original section by 1/8 then summing the results; essentially an averaging computation which calculates the “DC or 0 Hz” content of the section. Hence, the top left corner pixel of the DCT spectrum is referred to as the DC component.

The pixel in row 1 and column 2 (an AC component) is computed by multiplying the image section (entry by entry) by the matrix:

$$W1^T \times W2 = \begin{bmatrix} 0.1734 & 0.1470 & 0.0982 & 0.0345 & -0.0345 & -0.0982 & -0.1470 & -0.1734 \\ 0.1734 & 0.1470 & 0.0982 & 0.0345 & -0.0345 & -0.0982 & -0.1470 & -0.1734 \\ 0.1734 & 0.1470 & 0.0982 & 0.0345 & -0.0345 & -0.0982 & -0.1470 & -0.1734 \\ 0.1734 & 0.1470 & 0.0982 & 0.0345 & -0.0345 & -0.0982 & -0.1470 & -0.1734 \\ 0.1734 & 0.1470 & 0.0982 & 0.0345 & -0.0345 & -0.0982 & -0.1470 & -0.1734 \\ 0.1734 & 0.1470 & 0.0982 & 0.0345 & -0.0345 & -0.0982 & -0.1470 & -0.1734 \\ 0.1734 & 0.1470 & 0.0982 & 0.0345 & -0.0345 & -0.0982 & -0.1470 & -0.1734 \\ 0.1734 & 0.1470 & 0.0982 & 0.0345 & -0.0345 & -0.0982 & -0.1470 & -0.1734 \end{bmatrix}$$

then summing all the entries in the resulting matrix. Notice that each row of the original image section is weighted by the same set of values, so in this case, the variation across the columns in the image is being measured.

Figure 9.14 is a graphical illustration the 64 weighting matrices used to calculate the DCT. Notice that the weighting matrix in the top left corner of Figure 9.14 is solid gray indicating that all pixels are given equal weighting. The weighting matrix in row 1 and column 2 of Figure 9.14 shows that variations in the weights occurring across the columns. The color white indicates the most positive value in the weighting matrix, the color black indicates the most negative value in the weighting matrix, and shades of gray are values in between. It is important to note that each of these weighting matrices are individually scaled so a white section in one graph or subplot does not correspond to the same positive number (weighting) as a white section in another graph or subplot.

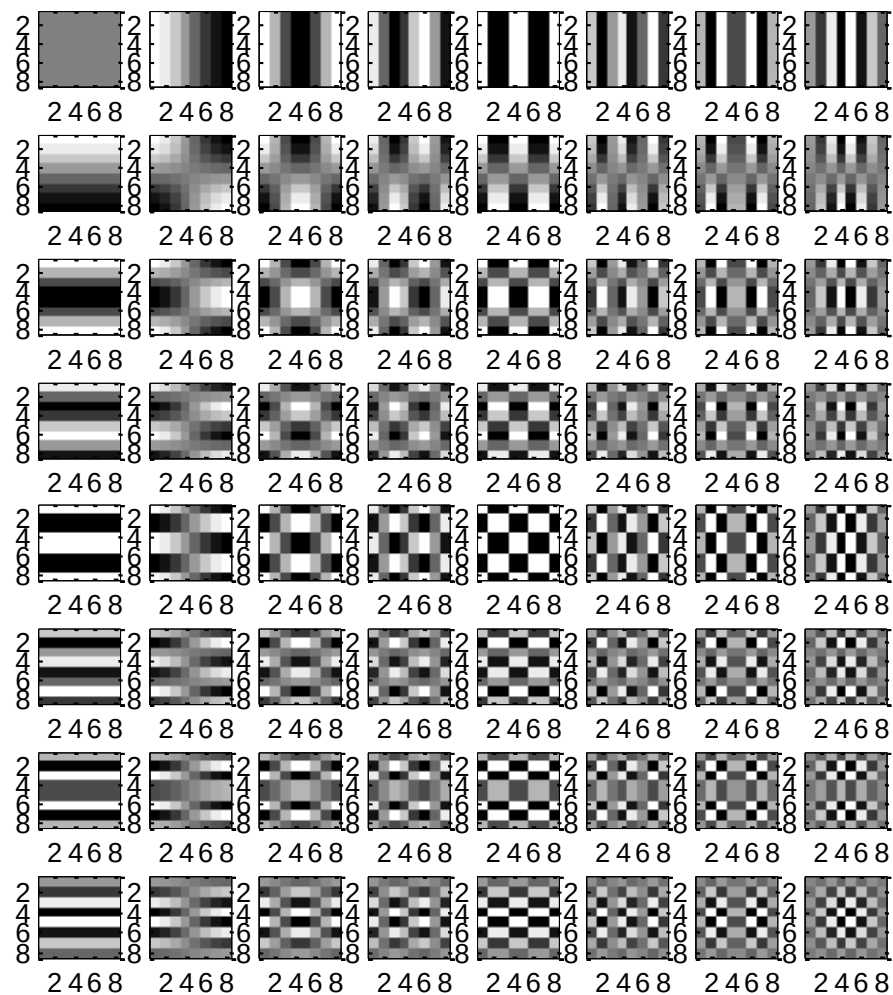


Figure 9.14: Graph of DCT Weighting Matrices

Notice that the weighting matrices in the bottom right section of 9.14 have much more variation than the weighting matrices in the top left section of 9.14. What does this mean? The pixels in the top left corner of the DCT spectrum reflect the low frequency content of the image section while the pixels in the bottom right corner reflect the high frequency content of the image section. An image section with very little detail or variation will have very small values in the bottom right entries of the DCT spectrum. An image section with a lot of variation would have significant values in the bottom right portion of the DCT spectrum. The top left pixel in the DCT spectrum is the DC component and the remaining 63 pixels are the AC components.

The MATLAB code to generate Figure 9.14 is:

```

for k=1:8
    y(1,k)=sqrt(1/8);
end
for n=2:8,
    for k=1:8
        y(n,k)=sqrt(2/8)*cos(pi*(2*k-1)*(n-1)/(2*8));
    end
end
colormap(gray(256));

p=1; % Subplot Number
for row=1:8
    for col=1:8
        w=y(row,:)'*y(col,:);
        subplot(8,8,p);
        imagesc(w); p=p+1;
    end
end

```

Example 9.6 illustrates how the frequency content of an image affects the DCT spectrum.

Example 9.6: Computing Discrete Cosine Transform of Simple Images

Compute the DCT spectrum for the four image sections shown in Figure 9.15. Comment on the results.

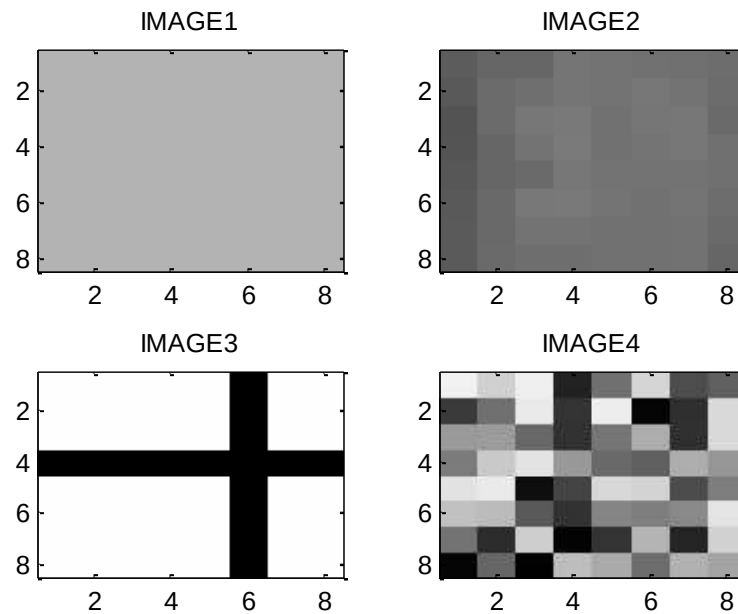


Figure 9.15: Images for Example 9.6

Solution

The image sections graphed in Figure 9.15 are:

IMAGE1 =

180	180	180	180	180	180	180	180
180	180	180	180	180	180	180	180
180	180	180	180	180	180	180	180
180	180	180	180	180	180	180	180
180	180	180	180	180	180	180	180
180	180	180	180	180	180	180	180
180	180	180	180	180	180	180	180
180	180	180	180	180	180	180	180

IMAGE2 =

92	102	102	117	115	113	112	109
89	107	111	117	115	119	115	108
83	107	119	121	113	118	119	106
84	102	115	122	113	116	119	112
87	102	106	119	115	115	115	112
89	105	120	121	116	113	116	109
90	105	115	115	113	113	114	106
90	106	110	110	113	113	114	102

IMAGE3 =

255	255	255	255	255	0	255	255
255	255	255	255	255	0	255	255
255	255	255	255	255	0	255	255
0	0	0	0	0	0	0	0
255	255	255	255	255	0	255	255
255	255	255	255	255	0	255	255
255	255	255	255	255	0	255	255
255	255	255	255	255	0	255	255

IMAGE4 =

242	209	239	35	113	214	78	96
59	113	234	52	238	5	48	219
155	157	105	51	119	174	49	218
124	202	228	154	107	97	174	151
227	235	15	69	216	212	77	127
194	188	90	51	134	128	138	229
116	45	207	4	52	181	38	210
5	103	3	190	171	109	178	164

The image section is first down-shifted by 128 resulting in signed integers rather than unsigned integers. The two-dimensional DCT of each section is then computed using the MATLAB command **dct** first along the columns then along the rows as shown below for IMAGE1.

```
IMAGE1_DCT = (dct(dct(IMAGE1-128)'))';  
round(IMAGE1_DCT)
```

The results are:

IMAGE1_DCT =

416	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

IMAGE1 is an 8×8 section of a single color. Since there is no variation in the image at all, the DCT spectrum shows only a single frequency component (dc) in the top left pixel.

IMAGE2_DCT =

-149	-42	-51	-20	-19	3	-10	-5
2	-5	-1	1	3	0	-1	-6
-7	7	5	9	1	-4	-1	2
0	0	2	1	10	-1	0	0
-7	-5	4	5	7	-1	-3	-2
0	0	-1	-4	0	3	1	1
1	2	-3	-3	-2	2	-1	2
-3	-1	1	2	1	-1	-1	-2

IMAGE2 does not have a lot of detail, just some minor variation in color. The DCT spectrum shows that the frequency components of greatest magnitude are the lower frequencies in the top left corner of the spectrum. The high frequency components in the lower right portion of the spectrum are of much smaller magnitude.

IMAGE3_DCT =

538	175	121	-309	223	62	-292	262
-62	-7	-5	12	-9	-2	11	-10
292	33	23	-58	42	11	-54	49
175	20	14	-35	25	7	-33	29
-223	-25	-17	44	-32	-9	42	-37
-262	-29	-20	52	-37	-10	49	-44
121	14	9	-24	17	5	-23	20
309	35	24	-61	44	12	-58	52

IMAGE3 has distinct black lines crossing a white background. Although the lower frequency elements still show the largest magnitude, the higher frequency components have become much more significant for this image section.

IMAGE4_DCT =

47	0	113	-6	2	-190	87	38
59	143	2	-32	-40	-52	68	86
-68	-48	-61	-53	-11	26	51	39
56	106	89	112	-39	-33	71	19
70	52	-68	89	-48	61	-121	-70
14	54	56	-41	-122	85	151	-20
41	-9	64	92	4	11	-64	-92
-29	20	30	155	6	-64	96	-77

IMAGE4 was generated using the MATLAB function **rand** to create a random image section. The large variations from pixel to pixel are reflected in the DCT spectrum.

Computing a DCT does not result in any loss of information whatsoever from the original image section. The original image can be completely recovered at this point using an inverse DCT. In the previous example, all four images can be completely recovered using a MATLAB command of the form:

```
IMAGE1_RECOVERED = (idct(idct(IMAGE1_DCT)'))'+128;
```

for the non-rounded DCT. The loss associated with JPEG compression occurs mainly in the next step: compression using quantization.

Quantization (Lossy Compression)

An image section is pretty small: only 8 pixels by 8 pixels. For most of the 8x8 sections in a photographic image, the amount of detail or variation should be small, and the significant frequency content of the image will be contained in the top left portion of the DCT spectrum. The quantization step involves dividing the DCT spectrum by a quantization matrix (entry by entry) then rounding the result to the nearest integer.

So what does the quantization matrix look like? There is a huge variety of quantization matrices in use. The quantization matrix varies with the system (digital camera type or photo-editing software program) and the amount of compression desired. Also, the quantization matrix for the luminance signal is different from the quantization matrix for the chrominance signals because the human eye is much more sensitive to the high frequency content of the luminance signal. Therefore the luminance signal is not compressed as much as the chrominance signals. The following example illustrates the compression step.

Example 9.7: Effect of Quantization on Image Quality

Compress the DCT spectrums for IMAGE2 and IMAGE3 from Example 9.6 using the quantization matrices Q1 and Q2 shown in Table 9.1. Recover the images from the quantized spectrum using an inverse DCT. Compare the recovered images to the original images and comment on the effects of the two quantization matrices.

Table 9.1: Quantization Matrices

Q1=	1	1	1	1	1	2	2	3	Q2=	16	11	10	16	24	40	51	61
	1	1	1	1	1	3	3	3		12	12	14	19	26	58	60	55
	1	1	1	1	2	3	3	3		14	13	16	24	40	57	69	56
	1	1	1	1	2	4	4	3		14	17	22	29	51	87	80	62
	1	1	3	4	4	6	6	4		18	22	37	56	68	109	103	77
	1	2	3	3	4	5	6	5		24	35	55	64	81	104	113	92
	2	3	4	4	5	6	6	5		49	64	78	87	103	121	120	101
	5	5	5	5	5	5	5	5		72	92	95	98	112	100	103	99

Solution

Enter IMAGE2, IMAGE3, Q1, and Q2 into MATLAB. The following MATLAB commands down-shift the image data by 128, compute the DCT of each image, and quantize the results.

```

IMAGE2_DCT=(dct(dct(IMAGE2-128))');
IMAGE3_DCT=(dct(dct(IMAGE3-128))');
IMAGE2_Q1 = round(IMAGE2_DCT./Q1);
IMAGE2_Q2 = round(IMAGE2_DCT./Q2);
IMAGE3_Q1 = round(IMAGE3_DCT./Q1);
IMAGE3_Q2 = round(IMAGE3_DCT./Q2);

```

The DCT of each image is displayed along with the quantized DCT of each image in Table 9.2. When IMAGE2 is quantized by Q1, several pixels in the lower right corner, or high frequency section, of the spectrum become zero. When IMAGE2 is quantized by Q2, even more of the pixels become zero; in fact, only seven of the 64 pixels contain non-zero values. These quantized spectrums require much fewer bits to store the information than the original DCT spectrum. Notice that when IMAGE3 is quantized, there are not nearly as many pixels with zeros in the quantized spectrum. This is due to the fact that IMAGE3 had significant high frequency components; whereas, IMAGE2 did not.

Table 9.2: DCT Spectrum and Quantized DCT Spectrums

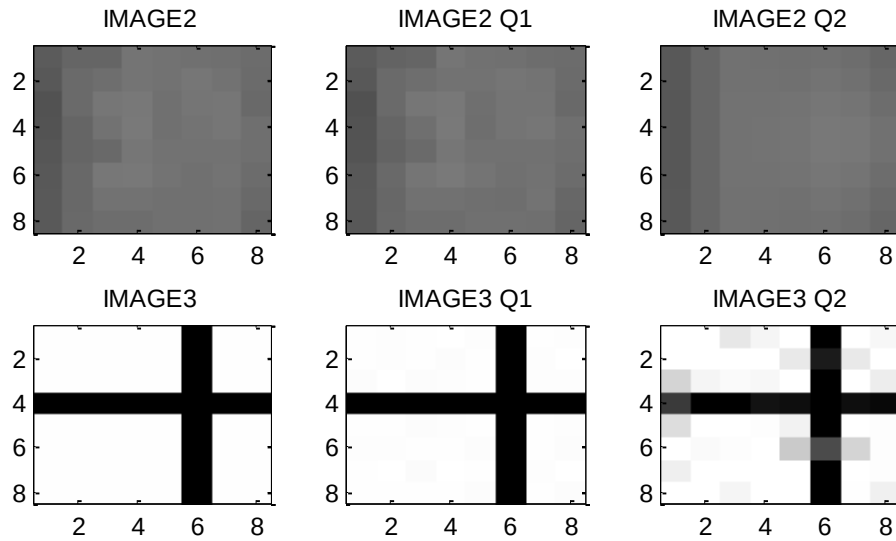
IMAGE2_DCT							
-149	-42	-51	-20	-19	3	-10	-5
2	-5	-1	1	3	0	-1	-6
-7	7	5	9	1	-4	-1	2
0	0	2	1	10	-1	0	0
-7	-5	4	5	7	-1	-3	-2
0	0	-1	-4	0	3	1	1
1	2	-3	-3	-2	2	-1	2
-3	-1	1	2	1	-1	-1	-2
IMAGE2 QUANTIZED BY Q1							
-149	-42	-51	-20	-19	2	-5	-2
2	-5	-1	1	3	0	0	-2
-7	7	5	9	1	-1	0	1
0	0	2	1	5	0	0	0
-7	-5	1	1	2	0	0	-1
0	0	0	-1	0	1	0	0
1	1	-1	-1	0	0	0	0
-1	0	0	0	0	0	0	0
IMAGE2 QUANTIZED BY Q2							
-9	-4	-5	-1	-1	0	0	0
0	0	0	0	0	0	0	0
-1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
IMAGE3_DCT							
538	175	121	-309	223	62	-292	262
-62	-7	-5	12	-9	-2	11	-10
292	33	23	-58	42	11	-54	49
175	20	14	-35	25	7	-33	29
-223	-25	-17	44	-32	-9	42	-37
-262	-29	-20	52	-37	-10	49	-44
121	14	9	-24	17	5	-23	20
309	35	24	-61	44	12	-58	52
IMAGE3 QUANTIZED BY Q1							
538	175	121	-309	223	31	-146	87
-62	-7	-5	12	-9	-1	4	-3
292	33	23	-58	21	4	-18	16
175	20	14	-35	13	2	-8	10
-223	-25	-6	11	-8	-1	7	-9
-262	-15	-7	17	-9	-2	8	-9
60	5	2	-6	3	1	-4	4
62	7	5	-12	9	2	-12	10
IMAGE3 QUANTIZED BY Q2							
34	16	12	-19	9	2	-6	4
-5	-1	0	1	0	0	0	0
21	3	1	-2	1	0	-1	1
13	1	1	-1	0	0	0	0
-12	-1	0	1	0	0	0	0
-11	-1	0	1	0	0	0	0
2	0	0	0	0	0	0	0
4	0	0	-1	0	0	-1	1

The first step in recovering the image from the quantized spectrum is to multiply the quantized DCT by the quantization matrix. It is important to recognize that this will not match the original DCT spectrum because of the rounding that occurred in the quantization process. The majority of the loss in the JPEG compression algorithm occurs when dividing by the quantization matrix then rounding to the nearest integer. The second step in recovering the image is to perform an inverse DCT then up-shift by 128. The MATLAB commands to recover and plot the images are:

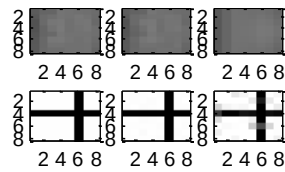
```
IMAGE2_R1 = round((idct(idct(IMAGE2_Q1.*Q1)))'+128);
IMAGE2_R2 = round((idct(idct(IMAGE2_Q2.*Q2)))'+128);
IMAGE3_R1 = round((idct(idct(IMAGE3_Q1.*Q1)))'+128);
IMAGE3_R2 = round((idct(idct(IMAGE3_Q2.*Q2)))'+128);
figure; colormap(gray(256));
subplot(2,3,1);image(IMAGE2);title('IMAGE2');
subplot(2,3,2);image(IMAGE2_R1);title('IMAGE2 Q1');
subplot(2,3,3);image(IMAGE2_R2);title('IMAGE2 Q2');
subplot(2,3,4);image(IMAGE3);title('IMAGE3');
subplot(2,3,5);image(IMAGE3_R1);title('IMAGE3 Q1');
subplot(2,3,6);image(IMAGE3_R2);title('IMAGE3 Q2');
```

The original images and the images recovered from the quantized spectrums are shown in Figure 9.16. The numerical differences between the original image and the recovered image for each pixel are displayed in Table 9.3 for each of the quantization matrices. Both Table 9.3 and Figure 9.16 indicate that quantization by Q1 results in very little difference between the recovered image and the original image. However, quantization by Q2 does cause noticeable differences between the recovered image and the original image. Notice that the pixels with the largest numerical errors in Table 9.3 are also the pixels in Figure 9.16 with the most noticeable differences between the original and the recovered image. A positive value for the difference indicates that the recovered image pixel will be darker than the original pixel, while negative error indicates the recovered image will be lighter than the original.

It is important to put things in perspective here. These 8×8 image sections are part of a much larger image and will be significantly smaller in area than shown in Figure 9.16a. Individual pixels are not supposed to be distinguishable in an image. In Figure 9.16b, the image sections are considerably reduced in area and the differences are much more difficult to perceive.



(a)



(b)

Figure 9.16: Original Image and Recovered Images

Table 9.3: Differences Between Original Image and Recovered Image

ERROR IN RECOVERED IMAGE2 USING Q1								ERROR IN RECOVERED IMAGE3 USING Q1							
0	2	-1	-1	1	0	1	0	0	1	0	-1	1	1	0	0
0	0	0	1	-1	0	-1	0	0	0	0	2	0	-1	1	-1
1	0	0	0	1	1	1	0	1	-3	2	0	0	-1	0	1
-1	0	2	-1	1	-2	-1	0	1	2	-1	-1	0	0	-1	0
1	1	-2	0	1	0	1	0	-1	0	1	1	0	0	0	0
0	-2	1	-1	-1	0	0	0	0	0	-1	0	1	0	0	1
0	1	0	1	1	1	-3	2	0	-1	3	-1	0	0	1	-2
0	1	0	-1	-1	0	1	-1	0	0	0	-1	1	-1	-1	1

ERROR IN RECOVERED IMAGE2 USING Q2								ERROR IN RECOVERED IMAGE3 USING Q2							
2	-2	-12	3	3	-1	1	5	-9	-5	23	9	-17	12	-27	7
0	3	-4	2	1	3	1	1	-10	-12	-13	-24	21	-29	20	-2
-6	3	4	5	-3	-2	1	-5	42	9	4	7	-1	8	-18	16
-4	-1	0	5	-5	-6	-1	-2	-59	10	29	-20	-16	35	-15	-10
-1	-1	-9	2	-3	-7	-5	-2	33	-8	-19	1	12	1	-20	1
0	1	5	5	0	-7	-2	-2	-12	3	0	-11	52	-76	41	1
1	1	0	0	-1	-3	0	-1	14	-12	-5	-3	-9	27	-25	1
0	2	-4	-4	1	-1	3	-2	-5	-5	9	-9	-6	3	0	14

Example 9.7 helps to explain why JPEG is considered a poor format for line drawings, logos, and other graphics with sharp detail. Although IMAGE3 could be recovered accurately when quantized by Q1, the quantized spectrum has non-zero values in almost all the pixels, so there won't be significant savings in size for the image. Quantizing IMAGE3 by Q2 results in more pixels with zeros but it also filters out some significant high frequency components in the image resulting in poor recovery of the original image.

For any image, there is a tradeoff between quantization error and size of the compressed image. A quantization matrix with larger values will result in more zeros in the quantized spectrum which reduces the image file size. However, larger values in the quantization matrix results in larger quantization errors. Digital cameras use quantization matrices similar to Q1 in Example 9.7 for the luminance signal. This results in about a 10:1 compression ratio over storing raw 24-bit RGB data and yields excellent quality images. Many photo-editing software programs allow the user to choose different quality levels when saving an edited image. Higher quality corresponds to less quantization and will result in a larger image file size. Lower quality means larger entries in the quantization matrix which will result in a smaller image file size, but will also reduce the quality of the image. Of course, a reduction in quality only matters if the artifacts or differences are actually noticeable when viewing or analyzing the image. Example 9.8 illustrates the compression process and recovery for a color image.

Example 9.8: Compression for a Color Image

The color image shown in the top left corner of Figure 9.17 is to be imported into MATLAB. Convert the RGB signals to YCbCr signals, break the image signals into 8×8 sections, take a DCT of each section, then quantize the DCT spectrum using the quantization matrices shown below. Recover the image and comment on results.

Q1 (LUMINANCE)							
16	11	17	27	34	39	34	17
11	12	16	26	28	23	12	12
17	16	21	28	23	12	12	12
27	26	28	23	12	12	12	12
34	28	23	12	12	12	12	12
39	23	12	12	12	12	12	12
34	12	12	12	12	12	12	12
17	12	12	12	12	12	12	12

Q1 (CHROMINANCE)							
17	17	22	34	20	20	17	17
17	19	22	14	14	12	12	12
22	22	14	14	12	12	12	12
34	14	14	12	12	12	12	12
20	14	12	12	12	12	12	12
20	12	12	12	12	12	12	12
17	12	12	12	12	12	12	12
17	12	12	12	12	12	12	12

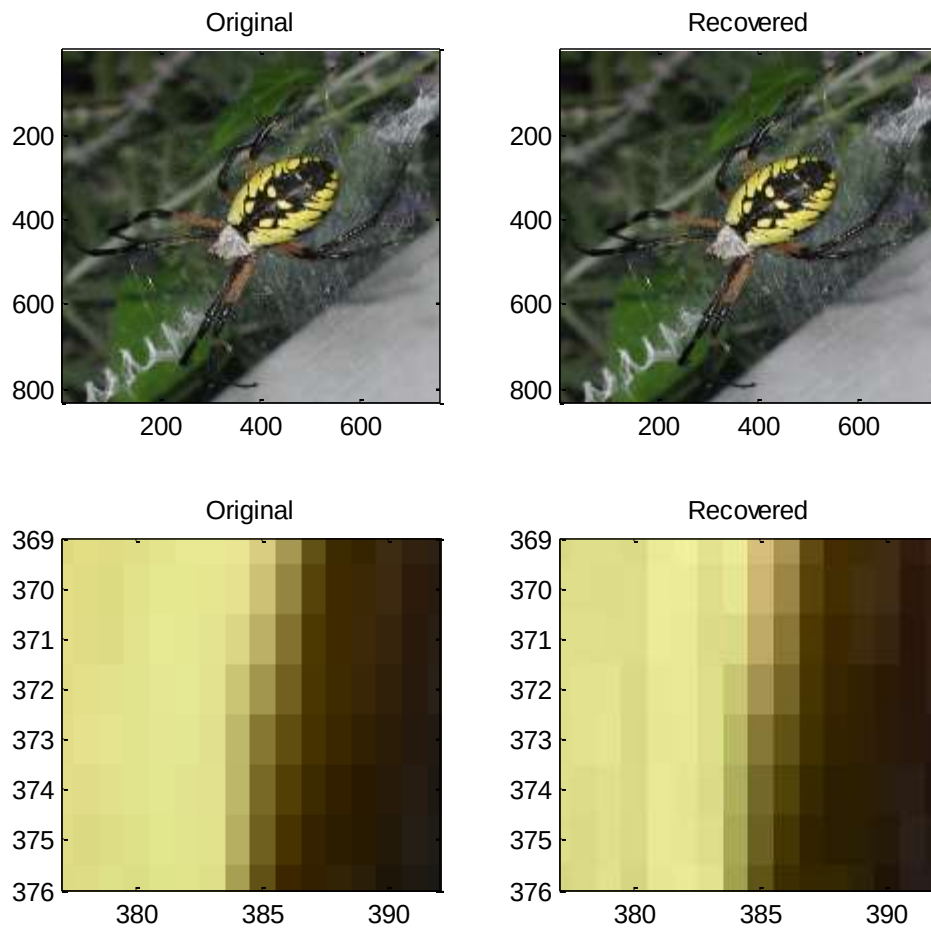


Figure 9.17: Argiope Aurantia (Black and Yellow Garden Spider)

Solution

The photograph of the spider in the top left corner of Figure 9.17 is a 170 kB JPEG image (760 x 832 pixels) taken by the author. The raw 24-bit RGB format would require 1.9 MB. So this image was already compressed by approximately an 11:1 ratio when stored as a jpeg image by the digital camera. It can be imported into MATLAB as follows:

```
X = imread('Spider2', 'jpeg');
```

The MATLAB m-file created for this example is shown in Figure 9.18. The image resulting from quantization using the given set of compression matrices is shown in the top right corner of Figure 9.17. The recovered image looks like the original image. If we zoom in on the images, then differences become apparent. Notice that columns 380 and 383 of the recovered image are noticeably darker than these columns in the original image. The high frequency abrupt transition from light to dark in the original image was filtered out when the DCT spectrum was quantized resulting in a slightly more gradual transition from light to dark in the recovered image.

MATLAB Code for Example 9.8

```
function [X_recv, Y_dct, Cb_dct, Cr_dct] = colorjpeg(X,Q_Lum,Q_Chrom)
% colorjpeg takes an image, X, in RGB format, converts to YCbCr,
% compresses image using quantization matrices: Q_lum and Q_Chrom.
% The output, X_recv is RGB format of compressed signal.
% Usage: >> [X_recv] = colorjpeg(X,Q_Lum,Q_Chrom);

% Image will be cropped if necessary to make the number of rows and
% columns divisible by 8.
X = double(X);
dims = size(X); Row = dims(1)-mod(dims(1),8);
Col = dims(2)-mod(dims(2),8);
R = Row/8; C = Col/8; %Determines number 8x8 sections

% Convert RGB to YCbCr
Y(1:Row,1:Col)=0.299*X(1:Row,1:Col,1)+0.587*X(1:Row,1:Col,2)+
0.114*X(1:Row,1:Col,3);
Cb(1:Row,1:Col) = 0.564*(X(1:Row,1:Col,3)-Y(1:Row,1:Col));
Cr(1:Row,1:Col) = 0.713*(X(1:Row,1:Col,1)-Y(1:Row,1:Col));

% JPEG Compression for Luminance Signal
for r = 1:R,
    for c = 1:C,
        temp = Y((r-1)*8+1:8*r, (c-1)*8+1:8*c); %Break into 8x8 sections
        temps = temp-128; %Shift by 128
        tempc = dct(temps); % take dct of columns of 8x8 section
        tempr = (dct(tempc'))'; % take dct of rows of 8x8 section
        tempq = round(tempr./Q_Lum); %Quantize the matrix and round
        Y_dct((r-1)*8+1:8*r, (c-1)*8+1:8*c)=tempq;
    end
end

% JPEG Compression for Chrominance Blue: Same Steps - Use Q_Chrom
for r = 1:R,
    for c = 1:C,
        temp = Cb((r-1)*8+1:8*r, (c-1)*8+1:8*c);
        temps = temp-128;
        tempc = dct(temps);
        tempr = (dct(tempc'))';
        tempq = round(tempr./Q_Chrom);
        Cb_dct((r-1)*8+1:8*r, (c-1)*8+1:8*c)=tempq;
    end
end

% JPEG Compression for Chrominance Red
for r = 1:R,
    for c = 1:C,
        temp = Cr((r-1)*8+1:8*r, (c-1)*8+1:8*c);
        temps = temp-128;
        tempc = dct(temps);
```

```

        tempr = (dct(tempc'))';
        tempq = round(tempr./Q_Chrom);
        Cr_dct((r-1)*8+1:8*r, (c-1)*8+1:8*c)=tempq;
    end
end

% Recover the image from the quantized jpeg image
for r = 1:R,
    for c = 1:C,
        tempunq = Y_dct((r-1)*8+1:8*r, (c-1)*8+1:8*c).* Q_Lum;
        %Unquantize Matrix
        tempc = idct(tempunq);           %Take inverse dct by columns
        tempr = round((idct(tempc'))'); %Take inverse dct by rows
        temps = tempr + 128;             %Shift by 128
        Y_recv((r-1)*8+1:8*r, (c-1)*8+1:8*c)=temps;
    end
end

for r = 1:R,
    for c = 1:C,
        tempunq = Cb_dct((r-1)*8+1:8*r, (c-1)*8+1:8*c).* Q_Chrom;
        tempc = idct(tempunq);           %Take inverse dct by columns
        tempr = round((idct(tempc'))'); %Take inverse dct by rows
        temps = tempr + 128;             %Shift by 128
        Cb_recv((r-1)*8+1:8*r, (c-1)*8+1:8*c)=temps;
    end
end

for r = 1:R,
    for c = 1:C,
        tempunq = Cr_dct((r-1)*8+1:8*r, (c-1)*8+1:8*c).* Q_Chrom;
        tempc = idct(tempunq);           %Take inverse dct by columns
        tempr = round((idct(tempc'))'); %Take inverse dct by rows
        temps = tempr + 128;             %Shift by 128
        Cr_recv((r-1)*8+1:8*r, (c-1)*8+1:8*c)=temps;
    end
end

% Convert YCbCr back to RGB
X_recv(1:Row,1:Col,1)= Y_recv(1:Row,1:Col)+
1.40*Cr_recv(1:Row,1:Col); % Red Pane

X_recv(1:Row,1:Col,2)= Y_recv(1:Row,1:Col) -
0.344*Cb_recv(1:Row,1:Col) - 0.714*Cr_recv(1:Row,1:Col); % Green Pane

X_recv(1:Row,1:Col,3) = Y_recv(1:Row,1:Col) +
1.77*Cb_recv(1:Row,1:Col); % Blue Pane
X_recv = uint8(X_recv); % Convert X_recv to 8 bit unsigned integer

```

Figure 9.18: MATLAB Code for Example 9.8

Table 9.4 shows the pixel values in the red, green, and blue panes of the original image for the 8 x 16 “zoomed in” section of Figure 9.17. The table also displays the DCT spectrum values for Y, Cb, and Cr. Looking at Table 9.4 should make it very clear why JPEG files are so much smaller than the raw bit files. Each pixel in the red, green, and blue panes would be stored as an 8-bit unsigned integer, and each of the pixels contains information that cannot be discarded. JPEG files store the spectrum information instead. The non-zero pixels in the DCT spectrum for the Y, Cb, and Cr signals must be stored. It is not necessary to waste bits by storing the many zeros in the spectrum. Instead, the number and location of zeros in a string can be specified. Sequencing through the spectrum data and specifying number and location of zeros is detailed in the next two sub-sections.

Table 9.4: Pixel Values for 8 x 16 Section of Image

RED PANE															
222	227	224	224	228	233	233	234	217	166	97	60	54	61	49	45
228	224	220	226	229	229	230	230	205	148	86	62	60	61	44	39
228	224	221	227	231	229	228	219	188	131	76	63	61	54	41	40
225	228	227	225	229	229	227	202	162	114	74	62	57	48	39	41
227	230	230	225	228	227	225	190	134	96	70	58	49	43	37	33
227	225	226	225	227	225	226	189	120	82	60	49	41	39	38	29
225	219	221	222	226	225	226	182	111	71	53	45	40	34	37	31
221	222	222	219	223	225	223	167	97	59	50	50	42	29	30	26
GREEN PANE															
217	222	220	222	228	233	233	229	205	151	81	42	36	44	34	33
222	220	219	226	232	232	230	225	193	133	67	41	40	41	27	27
222	220	219	227	234	232	227	214	176	113	57	42	41	36	25	30
219	224	225	228	231	233	226	197	150	96	54	43	38	32	25	32
221	225	228	228	230	231	224	184	119	78	50	39	32	28	25	26
222	220	224	228	231	227	225	183	105	63	42	31	25	26	28	24
220	217	221	225	231	228	226	175	96	51	34	29	26	25	30	28
219	220	225	222	228	228	223	160	81	38	32	35	30	21	26	25
BLUE PANE															
126	131	131	135	140	147	147	147	129	82	21	0	0	18	15	21
134	131	129	138	143	143	144	143	117	64	9	0	5	16	11	15
136	133	132	141	147	145	144	133	102	47	1	0	8	12	10	20
133	137	140	141	147	148	143	116	78	32	1	0	6	9	12	23
137	141	143	141	146	146	142	106	50	16	0	0	2	7	13	18
138	136	139	141	144	143	142	105	38	5	0	0	0	9	18	18
136	130	133	136	141	141	140	97	31	0	0	0	0	8	20	23
134	133	136	133	138	139	137	82	21	0	0	4	8	8	17	21
Y_DCT SPECTRUM															
42	2	-3	2	-1	0	0	1	-36	21	8	3	1	0	0	0
3	-3	2	-1	1	-1	1	-1	10	8	3	1	0	-1	0	-1
0	-1	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	-1	0	0	0
0	0	0	0	-1	0	0	0	0	0	0	-1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Cb_DCT SPECTRUM															
-80	0	0	0	0	0	0	0	-70	-5	0	0	0	0	0	0
0	0	0	0	0	0	0	0	-1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Cr_DCT SPECTRUM															
-57	0	0	0	0	0	0	0	-55	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Although not used in Example 9.8, it should also be mentioned that often chrominance sub-sampling is used to further reduce the size of the JPEG file. Chrominance sub-sampling involves decimating the chrominance signals prior to breaking the image into 8 x 8 sections and computing the DCT. 2 x 1 chrominance sub-sampling means every other column is eliminated which reduces the number of 8 x 8 sections by a factor of two. 2 x 2 Chrominance sub-sampling means every other column and every other row is eliminated which reduces the number of 8 x 8 sections by a factor of four. When the image is recovered, interpolation is used to fill in the missing pixels.

Challenge Question 9.3

Why break the image up into 8x8 sections rather than running a discrete cosine transform on the entire image?

Linear Sequencing to Re-order Pixels

When the DCT spectrum is quantized, the 8 x 8 sections are populated with a lot of zeros in the bottom right corner of the section. The pixel information is stored and downloaded in the zig-zag pattern illustrated in Figure 9.19 for one of the 8 x 8 sections of the Y_dct from Example 9.8. The pixel sequence for storage/transmission for this section is 42, 2, 3, 0, -3, -3, 2, 2, -1, four zeros, -1, -1, 0, 1, nine zeros, -1, 0, 1, 1, nine zeros, -1, 0, 0, -1, twenty-one zeros. The same zig-zag pattern is repeated for each of the 8x8 sections. 8x8 images sections are scanned left to right across rows then down. The next two sub-sections detail how this bit sequence is coded using Huffman coding.

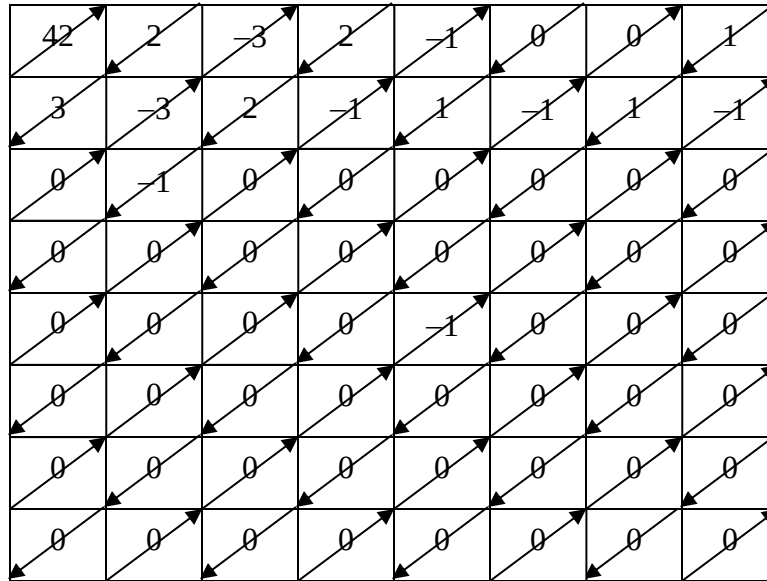


Figure 9.19: Linear Sequencing (Zig-Zag Pattern)

Huffman Code

JPEG files use Huffman coding for the bit sequence in each 8 x 8 section. The actual Huffman code used can vary greatly between different systems so this section will focus on illustrating the basic concepts associated with Huffman coding and how it applies to JPEG. Many, many binary codes use a fixed code length for transmitting, storing, or processing information. For example, the ASCII code uses 7 bits per character. Huffman coding allows the bit length for characters or other values to vary depending on the statistics of how often the characters or values are likely to appear in the message or file. For example, when the Huffman code is applied to text messages, the letters of the alphabet that appear more frequently (like “a” and “e”) would have much shorter bit codes than letters that are not as common (like “z” or “x”). Huffman coding is a form of lossless compression because as long as the code is known, the original file can be completely recovered. Example 9.9 shows how a Huffman code can be derived based on the statistics of the database.

Example 9.9: Huffman Code

Suppose Size is a variable that can range from 0 to 11. Table 9.5 displays the probability distribution for the variable, Size. For example, the probability that Size = 0 is 10% while the probability that Size = 11 is only 0.5%.

Notice that the table is arranged in order of decreasing probability rather than by the numerical values of Size. This is necessary in order to build the binary tree that will determine the Huffman code. The binary tree is formed by grouping together the two values of lowest probability, re-writing the probability table to reflect the combination, and then repeating till all values appear in the tree. Each branch is assigned a 1 or a 0.

Table 9.5: Probability Distribution for the Variable, Size

SIZE	PROBABILITY (%)
4	16
5	15
3	14
2	13
6	12
1	12
0	10
7	4
8	2
9	1
10	0.5
11	0.5

The process is illustrated in Table 9.6 and Figure 9.20. Since 10 and 11 have the lowest probabilities in Table 9.5, these two values are combined as shown in Figure 9.20a. The choice of which value is assigned a 1 and which value is assigned a 0 is completely arbitrary. A new probability distribution table is then constructed reflecting the combination of 10 and 11 as shown in Table 9.6. Now the two lowest probabilities are the 10 & 11 combination and the value 9 which are highlighted in yellow in the first column of Table 9.6. These are combined as shown in Figure 9.20b. This procedure is repeated till everything has been combined together. Notice in Table 9.6 that after 0 and 7-11 are combined, the values 1 and 6 have fall to the bottom of the probability distribution table. As shown in Figure 9.20e, this requires the start of a new branch for the tree. This occurs again with the values 2 and 3 and with the values 4 and 5 as shown in Figure 9.20f.

Table 9.6: Probability Distribution as Values are Combined

SIZE	PROB.		SIZE	PROB.		SIZE	PROB.		SIZE	PROB.		SIZE	PROB.
4	16		4	16		4	16		4	16		0 & 7-11	18
5	15		5	15		5	15		5	15		4	16
3	14		3	14		3	14		3	14		5	15
2	13		2	13		2	13		2	13		3	14
6	12		6	12		6	12		6	12		2	13
1	12		1	12		1	12		1	12		6	12
0	10		0	10		0	10		0	10		1	12
7	4		7	4		7	4		7-11	8			
8	2		8	2		8-11	4						
9	1		9-11	2									
10-11	1												

SIZE	PROB.		SIZE	PROB.		SIZE	PROB.		SIZE	PROB.
1 & 6	24		2 & 3	27		4 & 5	31		0 & 1 & 6-11	42
0 & 7-11	18		1 & 6	24		2 & 3	27		4 & 5	31
4	16		0 & 7-11	18		1 & 6	24		2 & 3	27
5	15		4	16		0 & 7-11	18			
3	14		5	15						
2	13									

Constructing Binary Tree for Huffman Code

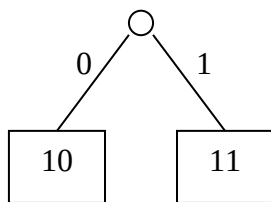


Figure 9.20a

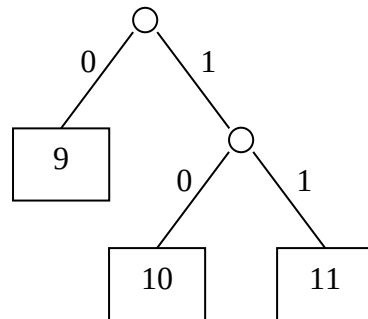


Figure 9.20b

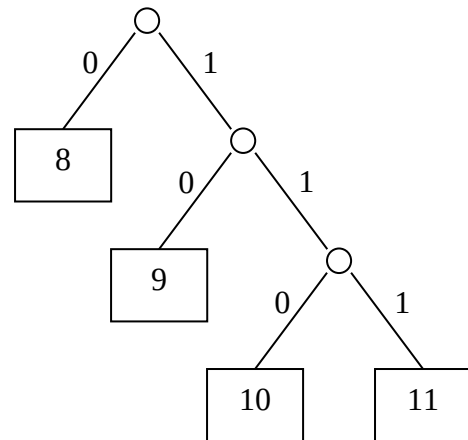


Figure 9.20c

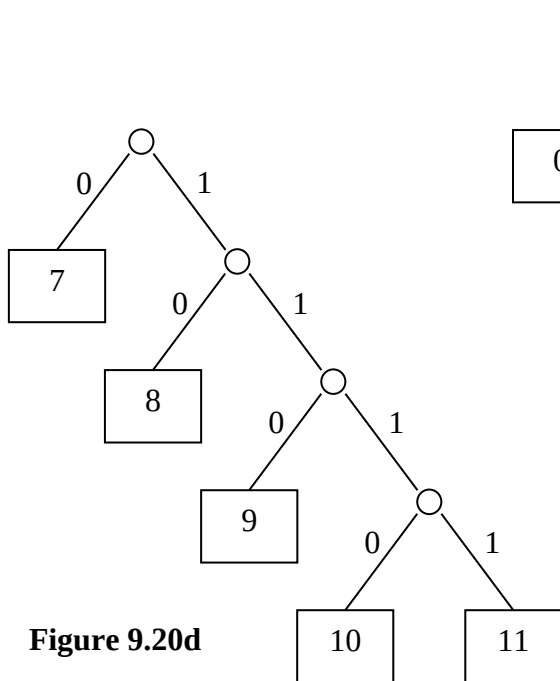


Figure 9.20d

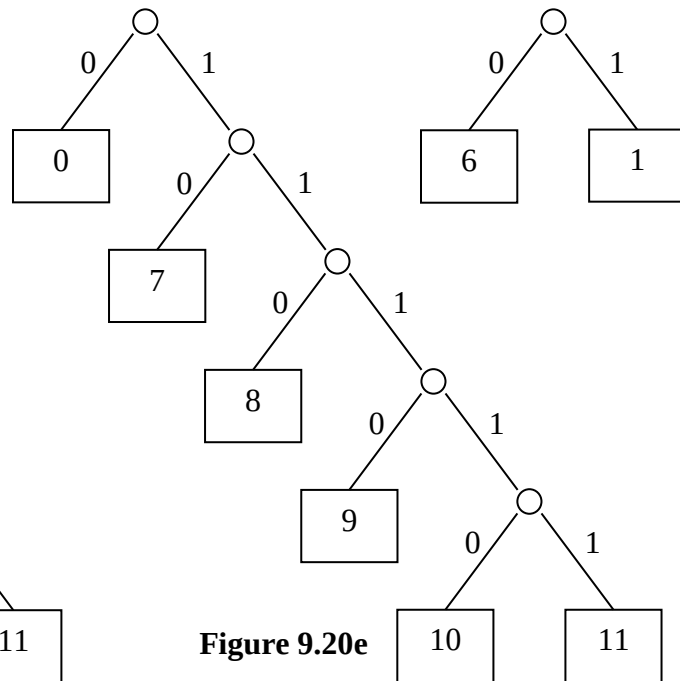


Figure 9.20e

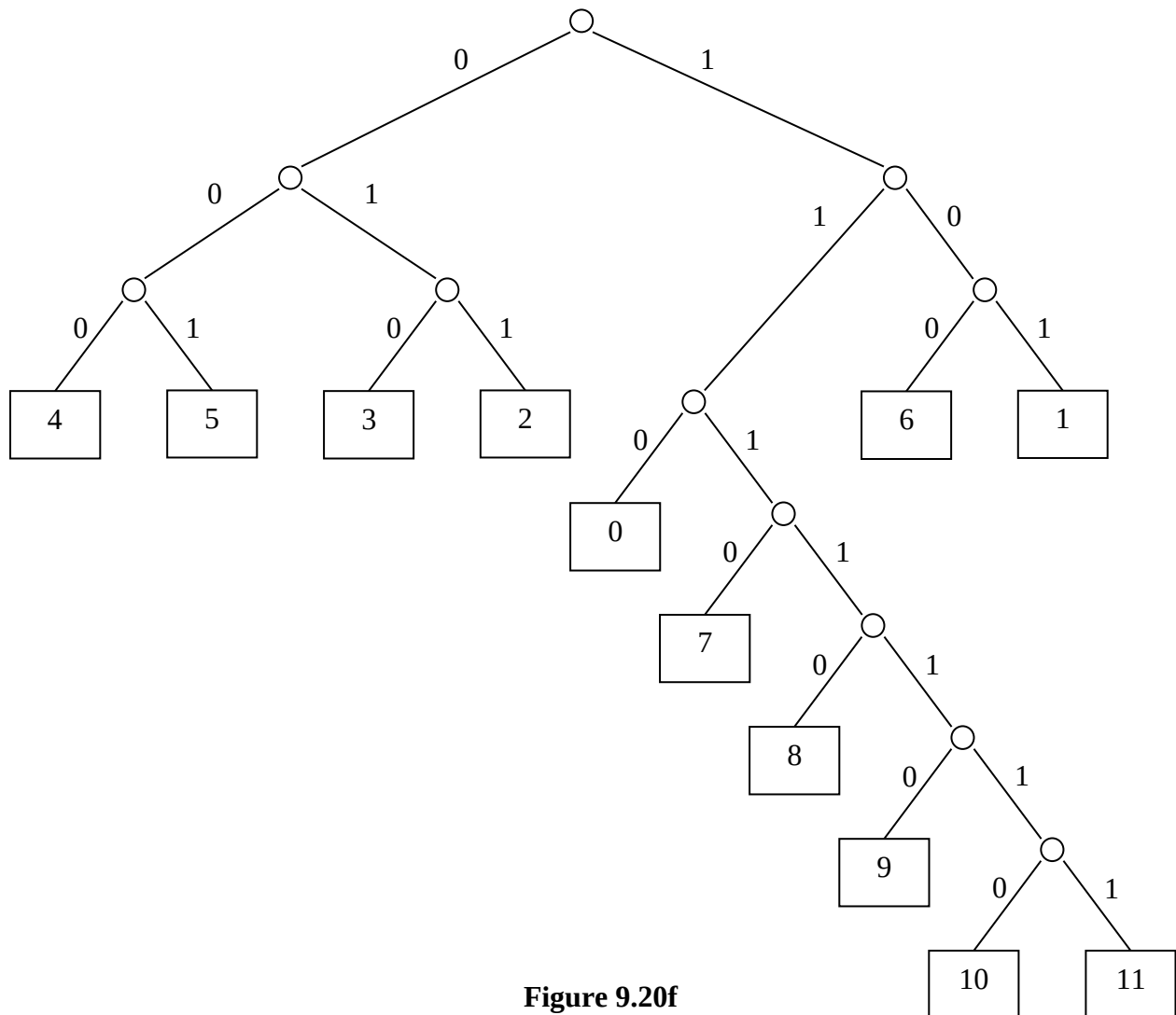


Figure 9.20f

Figure 9.20: Constructing Binary Tree for Huffman Code

The resulting Huffman code is shown in Table 9.7.

Table 9.7: Huffman Code for Example 9.9

SIZE	PROBABILITY (%)	HUFFMAN CODE	CODE LENGTH (BITS)
4	16	000	3
5	15	001	3
3	14	010	3
2	13	011	3
6	12	100	3
1	12	101	3
0	10	110	3
7	4	1110	4
8	2	11110	5
9	1	111110	6
10	0.5	1111110	7
11	0.5	1111111	7

The code for a particular value is determined by starting at the parent (top) node and working down the binary tree to that value recording all the 1s and 0s along the way. Notice that values of higher probability have shorter codes than values with smaller probability. A fixed length code would require 4 bits to represent all possible values for the variable Size (0000 to 1011). The average (expected) length of the Huffman code for the variable Size is:

$$\text{Expected Number of Bits} = 0.92 \times 3\text{bits} + 0.04 \times 4\text{bits} + 0.02 \times 5\text{bits} + 0.01 \times 6\text{bits} + 0.01 \times 7\text{bits}$$

$$\text{Expected Number of Bits} = 3.15\text{ bits}$$

This seemingly small difference pays off for image files!

The choice of which branches are assigned 0 and which branches are assigned 1 is completely arbitrary but will obviously change the resulting code. A Huffman table with the code must be made available in order to decode the message or image. Another key characteristic to recognize about Huffman codes is that the codes each have a unique prefix. Examine the codes in Table 9.7. None of the listed codes is a prefix for any other code. For example, the code for a value of 4 is 000. None of the other codes in the table begin with 000. The unique prefix of each code is in fact what makes it possible to decode a Huffman sequence. A sequence of three zeros is decoded as a 4, it cannot mean anything else.

Huffman Coding Applied to JPEG Images

Huffman codes are used to efficiently store the DCT spectrum information for each 8 x 8 section. Due to differences between the statistics of the luminance signal and the chrominance signals and between the DC and AC components of each, four different Huffman codes are used for JPEG images. There is a Huffman code for the DC component (top left corner pixel) of the luminance signal, a Huffman code for the AC components (remaining 63 pixels) of the luminance signal, a Huffman code for the DC component of the chrominance signals, and finally a Huffman code for the AC components of the chrominance signals.

The DC component is the top left pixel of each 8 x 8 image section. Since the DC component is determined by weighing all 64 pixels by 1/8 then adding the results, the DC component ranges from -1024 to +1016 (8×-128 to 8×127). A DC component of -1024 would indicate that the entire 8x8 section was black while a DC component of 1016 would indicate that the entire 8x8 section was white. In a JPEG file, differential encoding is used for the DC components which means that instead of storing the actual DC component value, the difference between the DC component in the current section and the DC component in the previous section in the row is stored. At the start of each row, it is assumed that the previous DC section value is zero.

Since the DC component can range from -1024 to +1016, the difference in consecutive sections can range from -2040 to +2040. Using a fixed length binary code for this range would require 12 bits. However, the probability statistics for the DC difference between two sections in a row show that most of the time, the differences are reasonably small and very rarely does the DC difference hit the outer range. The only way the difference could hit ± 2040 would be to have a completely white 8x8 section in a row immediately followed by a completely black 8x8 section or vice versa. Huffman coding is used to take advantage of the fact that the DC difference is relatively small most of the time.

Table 9.8 shows one possible Huffman code for the DC component difference for the luminance signal. It is important to note that although Huffman tables were provided in the original JPEG standard, many systems use their own uniquely developed tables. The table provided here is therefore useful only in illustrating how the coding works.

Table 9.8: Example Huffman Code for DC Component Difference of Luminance Signal

SIZE (BITS)	HUFFMAN CODE	DC DIFFERENCE (BITS)	DC DIFFERENCE (VALUE)
4	000	1000 to 1111; 0100 to 0000	+8 to +15; -8 to -15
5	001	10000 to 11111; 01111 to 00000	+16 to +31; -16 to -31
3	010	100 to 111; 011 to 000	+4 to +7; -4 to -7
2	011	10 to 11; 01 to 00	+2 to +3; -2 to -3
6	100	100000 to 111111; 011111 to 000000	+32 to +63; -32 to -63
1	101	1; 0	+1; -1
0	110	0	EOB
7	1110	1000000 to 1111111; 0111111 to 0000000	+64 to +127; -64 to -127
8	11110	10000000 to 11111111; 01111111 to 00000000	+128 to +255; -128 to -255
9	111110	100000000 to 111111111; 011111111 to 000000000	+256 to +511; -256 to -511
10	1111110	1000000000 to 1111111111; 0111111111 to 0000000000	+512 to +1023; -512 to -1023
11	11111110	10000000000 to 11111111111; 01111111111 to 00000000000	+1024 to +2040; -1024 to -2040

The DC component difference is stored as a (Size, Value) with a Huffman code used for Size followed by a bit string of length equal to Size to indicate value. The most significant bit of the value bit string does double duty as both a magnitude bit and a sign bit. A 1 indicates a positive value and a 0 indicates a negative value (opposite of what most signed number systems use). Example 9.10 illustrates how to use the information in Table 9.8 to extract the DC value.

Example 9.10: Calculating DC Component of Image

Assume each of the bit strings of data begin at the DC component of an 8x8 section of the luminance signal. Use Table 9.8 to determine the DC difference value; that is, the difference between the DC component in the current section and the DC component in the previous section.

- (a) 0 1 0 1 1 0 1 1 0 0
- (b) 1 0 0 0 1 1 0 1 1 1 1 0 0
- (c) 1 1 1 1 0 1 0 0 0 0 0 0 1 1 1 0
- (d) 1 1 0 1 1 1 0

Solution

- (a) First extract the Size using the Huffman code in Table 9.8. The first three bits of the given sequence 0 1 0 1 1 0 1 1 0 0 match the code for Size = 3 bits. This means that the DC difference value is encoded in the next 3 bits in the sequence. Therefore, the DC difference value is 1 1 0 or +6. The remaining bits are part of the AC string.
- (b) The first three bits of the sequence 1 0 0 0 1 1 0 1 1 1 1 0 indicate that Size = 6 bits. The DC difference value is therefore determined from 0 1 1 0 1 1. This is a negative value (MSB = 0). Invert all the bits to determine the magnitude which yields 1 0 0 1 0 0 = 36. So, the DC difference value = -36. Remaining bits are part of the AC string.
- (c) The first five bits of the sequence 1 1 1 1 0 1 0 0 0 0 0 0 1 1 1 0 indicate that Size = 8 bits. The DC difference value is therefore determined from 1 0 0 0 0 0 0 1 which is +129.
- (d) The first three bits of the sequence indicate that Size = 0 bits and is also the code for End of Block (EOB).

The DC difference value for the chrominance signal is stored in exactly the same way as a (Size, Value) but the Huffman code for Size is different because the probability distribution for the DC components of the chrominance signal is different from that of the luminance signal.

The 63 AC values for the luminance and chrominance signals are stored in a different format than the DC difference value. As previously noted, a large number of the AC components will be zero and the linear sequencing pattern (see Figure 9.19) leads to long runs of zeros. So, the AC components are stored using a (Run, Size, Value) format. Run indicates how many pixels with zeros occur before the non-zero pixel. Size indicates how many bits are used for the value in the non-zero pixel. Unlike DC where differences are stored, value indicates the actual AC spectrum pixel value. A Huffman code is used for the (Run, Size) pair based on statistical properties of the AC components. Again the code will be different for the luminance signal and for the chrominance signals, and although standard tables are in existence, codes do vary among different systems. Also included in the Huffman code is a code for EOB indicating that all remaining pixels in the section are zero, and a code for ZRL which indicates 15 consecutive zeros. An example of a Huffman code for the AC components of the luminance signal is shown in Table 9.9.

Table 9.9: Example Huffman Code for AC Components of Luminance Signal

(Run,Size)	HUFFMAN CODE	(Run,Size)	HUFFMAN CODE
(0,1)	00	(4,1)	111011
(0,2)	01	(0,6)	1111000
(0,3)	100	(1,3)	1111001
(EOB)	1010	(5,1)	1111010
(0,4)	1011	(6,1)	1111011
(1,1)	1100	(0,7)	11111000
(0,5)	11010	(2,2)	11111001
(1,2)	11011	(7,1)	11111010
(2,1)	11100	(1,4)	111110110
(3,1)	111010	(ZRL)	11111111001

Example 9.11: Huffman Coding of an Image Section

Suppose that the DCT of an 8x8 luminance signal image section has been computed, quantized, and then re-ordered using linear sequencing. The resulting pixel sequence is: 42, 2, -3, 2, 3, 0, 0, 1, 2, -3, 2, -1, 1, -1, 1, six zeros, -1, five zeros, 1, -1, sixteen zeros, 1, seven zeros, 1, ten zeros. Assuming this image section is the first section in a row of the image, code the section using the Huffman codes from Tables 9.8 and 9.9.

Solution

- The DC component is 42. Since the assumption is being made that this is the first section in a row, the previous DC component in the row is taken as zero. The DC difference value to be stored is therefore 42. In binary, $42 = 101010$ which requires a size = 6 bits. According to Table 9.8, the code for a size of 6 bits is 100. So the code for the DC component of this section would be 100, indicating a size of six bits, followed by 101010 indicating a value of 42.
- The first AC value of +2 has no preceding zeros and would require a size of 2 bits. Using Table 9.9, the code for the first AC component would be 01, indicating a (Run,Size) = (0,2), followed by the bits 10 indicating a value of +2.
- Consider the AC component in the string which is a +1 preceded by 16 zeros. Referring to Table 9.9, the code for this component would consist of the ZRL code (11111111001) indicating a run of 15 zeros, followed by 1100, the code for (Run,Size)=(1,1), then followed by a single bit 1 indicating the value of +1.
- The code for the entire pixel sequence would be:

DC Size 6, 42, (0,2), 2, (0,2), -3, (0,2), 2, (0,2), 3, (2,1), 1, (0,2), 2, (0,2), -3,
100 101010 01 10 01 00 01 10 01 11 11100 1 01 10 01 00

(0,2), 2, (0,1), -1, (0,1), 1, (0,1), -1, (0,1), 1, (6,1), -1, (5,1), 1, (0,1), -1
01 10 00 0 00 1 00 0 00 1 1111011 0 1111010 1 00 0

ZRL (1,1) 1 (7,1) 1 EOB
11111111001 1100 1 11111010 1 1010

Answers to Chapter 9 Challenge Questions

Question 9.1

Why is there a distinct border around the left and right sides of the filtered image in Figure 9.8?

The filter is applied to every pixel in the image. When the filter matrix is centered over a pixel that lies right on the border of an image, part of the filter matrix will hang over the image (by one row, or one column, or one row and one column for corner pixels). Unless the user specifies otherwise, zeros are used as the “image” values in this case. In other words, the original image is assumed to be completely surrounded in black. So, the filtered image in Figure 9.8 makes sense. On the right side, the original image is light grey providing a distinct contrast with the assumed surrounding black background. The sharpening filter would make that contrast (edge) even more noticeable by lightening that portion of the image even more. The left side of the original image is much darker and provides much less of a contrast with black so this portion of the image is only slightly lightened by the filter.

Question 9.2

Explain the second comment. What is the savings in terms of the number of multiplications if the single 2-D matrix convolution is replaced with two 1-D convolutions? Also, which of the filter matrices discussed in this section can be broken up into vertical and horizontal vectors?

The computational complexity can be reduced if the filter matrix can be broken down into two one dimensional vectors. The 2-D convolution can then be performed as two 1-D convolutions. What does this mean in terms of multiplications? A 2-D convolution using an $N \times N$ filter matrix on a $W \times L$ image would require $N^2 \cdot W \cdot L$ multiplications. Why? The $N \times N$ matrix must be centered over each pixel in the image, so there are N^2 multiplications for each original image pixel. Two 1-D convolutions on the same $W \times L$ image would only require $2 \cdot N \cdot W \cdot L$ multiplications. Why? One of the convolutions is a row convolution with a vector of length N centered over each image pixel moving across the rows. So, there are N multiplications for each original image pixel. The second convolution is a column convolution with a vector of length N centered over each image pixel moving down the columns. The column convolution also requires N multiplications for each original image pixel. The savings in terms of number of multiplications would increase significantly with the size of the filter matrix. A 3×3 filter matrix that can be broken into vertical and horizontal vectors would yield a 33% reduction in the number of multiplications.

The filter matrices in Section 9.2 that can be broken up into vertical and horizontal vectors are the Gaussian matrix used for blurring, and the Sobel matrix for edge detection. The Sharpening filter cannot be broken up into vertical and horizontal vectors. The Sobel matrix decomposition into vertical and horizontal components was already shown in Section 9.2. The Gaussian matrix can also be easily separated by noting that each term in the matrix can be broken up as follows:

$$h(i, j) = e^{-\frac{1}{2}[(i-\frac{N}{2})^2 + (j-\frac{N}{2})^2]/\sigma^2} = e^{-\frac{1}{2}(i-\frac{N}{2})^2/\sigma^2} \cdot e^{-\frac{1}{2}(j-\frac{N}{2})^2/\sigma^2}$$

So, the Gaussian blurring filter matrix can be rewritten as:

$$\begin{bmatrix} e^{-\frac{1}{2}(1-\frac{N}{2})^2/\sigma^2} \\ e^{-\frac{1}{2}(2-\frac{N}{2})^2/\sigma^2} \\ \vdots \\ e^{-\frac{1}{2}(N-\frac{N}{2})^2/\sigma^2} \end{bmatrix} \cdot \begin{bmatrix} e^{-\frac{1}{2}(1-\frac{N}{2})^2/\sigma^2} & e^{-\frac{1}{2}(2-\frac{N}{2})^2/\sigma^2} & \dots & e^{-\frac{1}{2}(N-\frac{N}{2})^2/\sigma^2} \end{bmatrix}$$

Question 9.3

Why break the image up into 8x8 sections rather than running a discrete cosine transform on the entire image?

The image compression that results from taking a discrete cosine transform relies on the image not varying too much over each 8x8 section. Most 8x8 sections have a DC component and a few AC components reflecting any detail variation in that section. An entire image has a huge amount of variation in most cases. While it is certainly possible to calculate a discrete cosine transform for the entire image, the resulting matrix would not have a lot of zeros or very small values so any compression would not be very efficient.

Chapter 9 Problems

Problem 9.1: A certain color digital image has dimensions of 1944 x 2592 pixels and, in jpeg format, has a size equal to 1.28 MB.

- (a) Calculate the size of the digital image if it were stored in 24 bit true color instead of jpeg format. What is the compression ratio achieved for this image by using jpeg?
- (b) Calculate the size of the digital image if it were converted to black and white using 8-bit grayscale. How does the size of the black and white image compare to the color jpeg image?

Problem 9.2: The matrix below represents a 6 x 6 section of an image file.

$$I = \begin{bmatrix} 108 & 237 & 246 & 173 & 233 & 35 \\ 130 & 186 & 139 & 101 & 203 & 184 \\ 22 & 125 & 133 & 94 & 25 & 27 \\ 67 & 148 & 59 & 252 & 67 & 167 \\ 204 & 61 & 125 & 10 & 86 & 126 \\ 7 & 117 & 159 & 226 & 173 & 199 \end{bmatrix}$$

- (a) Using the Gaussian filter shown below, calculate pixel (5,4) for the filtered image.

$$h = \begin{bmatrix} \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\ \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \end{bmatrix}$$

- (b) Repeat part (a) for pixel (3,6).

Problem 9.3: Use the following MATLAB commands to create an image, Y:

```
Y1 = [200*ones(100,100) 40*ones(100,100) 120*ones(100,100)];
Y2 = [40*ones(100,100) 200*ones(100,100) 120*ones(100,100)];
Y3 = [120*ones(100,100) 40*ones(100,100) 200*ones(100,100)];
Y = [Y1; Y2; Y3];
```

- (a) Plot the image using grayscale, **colormap(gray(256));** for the figure.
- (b) Filter the image with the horizontal and the vertical Sobel filters, combine the two filtered images using the gradient in Equation 9.6, and then plot the resulting image in grayscale using **image**. Discuss the results.
- (c) Plot the resulting image from part (b) using **imagesc** instead of **image**. What is the difference between the images in (b) and (c)? Why?

Problem 9.4: Create the image, I, defined below in MATLAB.

```
I =
    180    180    180    180    180    180    180    180
    180    180    169    112    54    180    180    180
    180    180    158    106    85    180    180    180
    180    180    99     37    41    180    180    180
    180    180    180    180    180    180    180    180
    180    180    180    180    180    10     10    180
    180    180    180    180    180    10     10    180
    180    180    180    180    180    180    180    180
```

- Plot the image using ***colormap(gray(256))***.
- Downshift the image by 128 and then compute the DCT for the image. Comment on the DCT and how it relates to the original image spectrum.
- Quantize the DCT of the image using the quantization matrices, Q1 and Q2, shown in Table 9.1. Round the quantized DCT matrices. Do you expect the quantization by Q1 or Q2 to adversely affect the image quality? Why?
- Compute the inverse DCT for each of the quantized images from part (b), upshift by 128, then round the results. Using ***colormap(gray(256))***, plot the quantized images in MATLAB using and compare them to the original image. Comment on the results.

Problem 9.5:

- Develop a Huffman code for the punctuation symbols shown in the table below.
Note: the probability distribution was made up at random by the author.
- Calculate the expected number of bits per symbol transmission.

Punctuation Symbol	.	,	:	'	;	?	“	!
Probability Distribution (%)	57	28	4.5	4.0	2.0	2.0	1.5	1.0

Problem 9.6: Suppose that the DCT of an 8x8 luminance signal image section has been computed, quantized, and then re-ordered using linear sequencing. The resulting pixel sequence is:

-149 -7 0 5 2 3 9 2 0 0 -1 0 0 -3 0 0 0 0 0 0 1 0 ... 0

Assuming this image section is the first section in a row of the image, code the section using the Huffman codes from Tables 9.8 and 9.9.

CHAPTER 9 LAB EXERCISE

Image Processing

Objectives

1. To become familiar with using the colormap and image functions in MATLAB to create simple images.
2. To explore image effects including blurring, sharpening and edge detection using 2-dimensional convolution.
3. To experiment with de-noising images.

Background

MATLAB has several functions for image processing, some of which will be explored in this lab exercise.

The command

```
colormap(map)
```

opens a figure window and applies a color scheme defined by map to that figure. Map is an Nx3 array where N is the number of colors desired. The first column represents the amount of red, the second column is the amount of green, and the last column is the amount of blue. All entries in map must range from 0 to 1. There are several maps available in MATLAB (for a list, type help colormap), or the user can create his/her own map. A specific color in the map is called by referring to the row number in map.

The command

```
image(X)
```

Creates an image in a figure window. Each entry in X refers to a color (row number) in map. Therefore the **colormap(map)** command should precede this command in order to define colors. A few simple exercises should make the process more clear.

Procedure

A. Creating Colormaps and Simple Images

Exercise 1: Creating and plotting a map of colors

Enter the following command in MATLAB:

```
map = [0 0 0; 1 0 0; 0 1 0; 0 0 1; 1 1 1]
```

The first row [0 0 0] corresponds to the color black, the second row [1 0 0] is red, the third row [0 1 0] is green, the fourth row [0 0 1] is blue, and the last row [1 1 1] is white.

Type the command:

```
colormap(map);
```

This opens a new figure window. If you close the Figure window, you will have to do the **colormap** command again. So, just minimize the Figure window then execute the following commands:

```
y = [1 2 3 4 5];  
image(y)
```

Describe and explain your image

Try the following:

```
y = [1 2 3 4 5; 2 3 4 5 1; 3 4 5 1 2; 4 5 1 2 3;  
5 1 2 3 4];  
image(y)
```

Describe and explain your new image

Try the following:

```
map = [0 0 0; 0.5 0 0; 0 0.5 0; 0 0 0.5; 0.5 0.5  
0.5];  
colormap(map);  
image(y)
```

Describe and explain your new image

Exercise 2: Create a Checker board

Create a checker board (8x8 alternating red and black squares). Record your MATLAB statements below.

Exercise 3: Standard map, gray(256)

There are several maps available in MATLAB. For a complete list and description, type help colormap. Try the following command (leave off the semi-colon).

```
gray(256)
```

Gray is a 256x3 matrix with the same weightings on red, green, and blue. It creates 256 shades of gray from black to white. To see this, do the following

```
colormap(gray(256));  
y = 1:1:256;  
image(y)
```

Describe and explain your image

Exercise 4: Loading an image

MATLAB images are stored in .mat files in Toolbox →MATLAB →Demos. To load the clown image and its corresponding colormap, type

```
load clown
```

This command loads an 81x3 “map” and a 200x320 array, X, with entries which specify a color from map for each pixel. To see the image, type the following

```
colormap(map)  
image(X)
```

Now change the colormap to grayscale using the following command:

```
colormap(gray(256))
```

What happens to the image? Explain why it is so dark.

(Hint: take a look at the entries in X and think about what colors they are calling in gray(256))

Exercise 5: Scaling an image

Try the following commands. For each command, describe what happens to the image and why.

```
image(3*X)
```

```
image(10*X)
```

```
imagesc(X)
```

Exercise 6:

A negative turns black to white and white to black in an image. Create a negative for the clown image. List the commands you used below.

B. Image Effects

Load a jpeg image of your choice into MATLAB using the `imread` command.

Exercise 7: Blurring an Image

Experiment with blurring the image using Gaussian filters of various sizes. The Gaussian filter can either be created using the `blurfilt.m` file listed in Figure 9.5, or for students with access to the Image Processing toolbox, the ***fspecial*** command works well. Filter your image using the command ***filter2***. Save one blurred image for your lab report.

Exercise 8: Sharpening an Image

Experiment with sharpening your image using the following filter with various choices for α .

$$h = \frac{1}{\alpha + 1} \begin{bmatrix} -\alpha & \alpha - 1 & -\alpha \\ \alpha - 1 & \alpha + 5 & \alpha - 1 \\ -\alpha & \alpha - 1 & -\alpha \end{bmatrix} \quad 0 \leq \alpha \leq 1$$

Students with access to the image processing toolbox can create the filter using the command ***fspecial***. Students without access to the image processing toolbox can easily enter in the matrix filter directly. Again, filter your image using the command ***filter2***. Save one sharpened image for your lab report.

Exercise 9: Edge Detection

Filter your image in both the vertical and horizontal direction using the Sobel filter:

$$h = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad v = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

Refer to Example 9.5. Include the resulting filtered image in your lab report.

Exercise 10: De-Noising Images

This exercise requires the image processing toolbox. Use the command ***imnoise*** to add some type of noise to your image (refer to Section 9.3). Experiment with different filter types and sizes to find the filter that is most effective for removing the noise without significant degradation in image quality. Include your noisy image and your de-noised image in your lab report.

CHAPTER 10 INTRODUCTION TO RADAR PROCESSING

This chapter explores some of the ways in which digital filtering and Fast Fourier Transforms are applied to radar signal processing. Radar signal processing is widely used by the military for actively tracking and/or identifying targets (aircraft, ships, missiles), for weapon guidance, for navigation, and for terrain mapping and surveillance. Radar processing is also used by policemen for detecting speeding violations, by pitchers to determine fastball speeds, and by meteorologists for their Doppler weather forecasts. The chapter begins with an introduction to radar processing including basic terminology and performance metrics. Next, a commonly used radar waveform, the LFM chirp signal, and the concept of a digital matched filter designed to pick out or find this chirp signal in the presence of noise, is introduced. Finally, range to target calculations based on signal return time, and target velocity calculations based on Doppler shift are covered.

10.1 INTRODUCTION

Radar (**radio detection and ranging**) has three basic functions: detection, tracking, and imaging. Figure 10.1 shows a simple illustration of radar tracking. The antenna on the aircraft transmits a signal that strikes the target (scatterer) and is then reflected back to the aircraft antenna. Detection involves determining whether or not there is an object located in some range space. The decision is based on processing the return signal in the presence of receiver noise, clutter echoes, and possible jamming, and determining whether or not the processed signal exceeds a pre-determined threshold. Tracking is more sophisticated than detection because in addition to detecting the existence of an object, tracking systems attempt to calculate distance to and/or radial velocity of the target. Imaging radar systems develop image maps of a region, and are used to monitor climate, weather conditions, oil spills, land use, and many other things. Although not of the same quality as optical systems, imaging radar systems have the advantage of being able to “see through” clouds, fog, rain, and darkness.

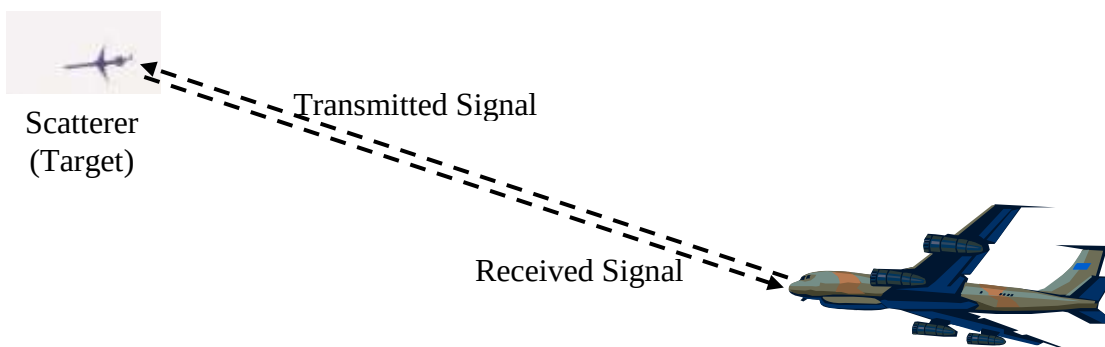


Figure 10.1: Simple Illustration of Radar Tracking

(Airplane sketches from Microsoft Office Clip Art – Used with permission from Microsoft)

A *monostatic* radar means that the receiver and the transmitter are collocated and in fact, often utilize the same antenna. The focus of this chapter will be on *pulsed* radar systems where the transmitted signal consists of a series of pulse bursts rather than a continuous wave (CW) signal. The pulsed signal shown in Figure 10.2 consists of a high frequency sine wave that is turned on for a short period of time, producing pulse bursts at discrete intervals. In Section 10.3, signals that are superior in performance to constant frequency sinusoids will be introduced.

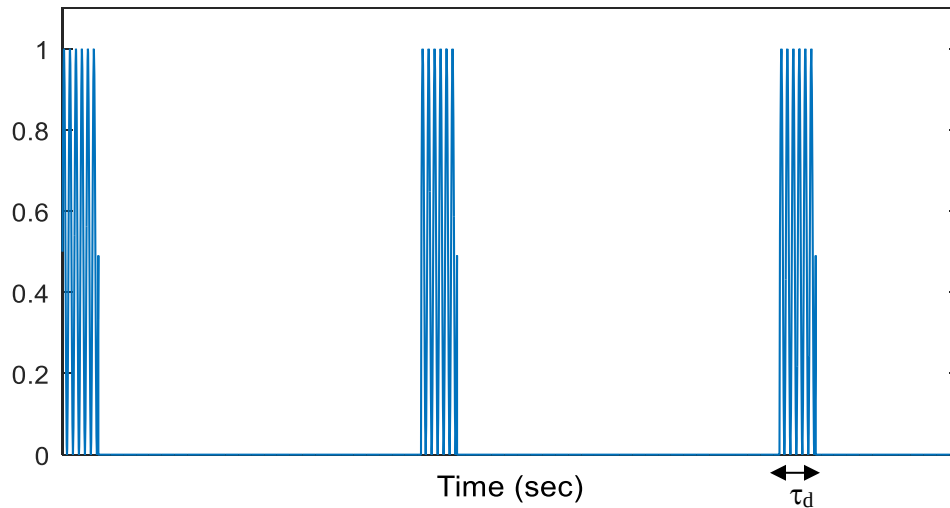


Figure 10.2: Simple Pulsed Radar Signal

The simple pulsed radar signal can be described by the following equation:

$$x(t) = a(t) \sin(2\pi \cdot F_T \cdot t) \quad (10.1)$$

F_T is the transmit frequency. Most radar systems operate in the microwave frequency range from 200 MHz to 95 GHz. The signal $a(t)$ is the on/off amplitude modulation signal, and simply consists of a sequence of discrete pulses of amplitude one that control when each transmit signal burst occurs, and the duration of the burst. The *pulse repetition interval* (PRI) is the time interval from the start of one pulse to the start of the next pulse. The *pulse repetition frequency* (PRF) is the frequency of the pulses and typically ranges from several hundred to several thousand pulses per second (pps). The pulse length or duration of each pulse burst, τ_d , typically ranges from 100ns to 100 μ s. The duty cycle or percent of time the pulse is on during each pulse repetition interval is typically much less than 1%.

The range to a target is determined by measuring the total amount of time it takes the transmitted pulse to reach the target and reflect back to the antenna. Using the speed of light as the velocity of the transmitted waveform, the range is calculated as follows:

$$R = c \cdot T_{\text{echo}}/2 \quad (10.2)$$

R = range (m)
 $c = 3 \cdot 10^8$ (m/s)
 T_{echo} = round trip travel time of pulse (s)

In this calculation, the velocity of the target and movement of the radar platform are ignored. This is often a reasonable assumption since these velocities will be significantly smaller than the velocity of the transmitted pulse.

Doppler processing is used to determine radial velocity of the target (velocity in the direction of the radar). If a target is traveling toward the radar, the frequency of the received signal will exceed the transmit signal frequency. On the other hand, if a target is moving away from the radar, the received signal frequency will be smaller than the transmit signal frequency. It will be shown in section 10.5 that the radial velocity of the target can be determined using the estimated Doppler Shift (difference between return and transmit frequencies) as follows:

$$v = \frac{(c \cdot F_D)}{(2 \cdot F_T)} \quad (10.3)$$

v = target radial velocity (m/s)
 $c = 3 \cdot 10^8$ (m/s)
 F_D = Doppler Shift (Hz)
 F_T = Transmit Frequency (Hz)

Radar processing has several challenges that are somewhat unique to radar systems:

- Radar signal bandwidth is large, typically from a few MHz to several hundred MHz and, in some applications, extending to a few GHz. The large signal bandwidth requires very fast A/D converters, and processors that can manage the speed of the incoming data.
- Radar signals have high dynamic ranges (ratio of strong to weak signals), typically several tens of decibels. Gain control at the receiver and side-lobe control are critical so weak signals are not masked by stronger signals.
- The signal-to-interference ratio (SIR) is relatively low for radar systems; less than 0 dB at the receiver antenna. The low-power return signal must be detected in the presence of receiver noise, echoes from clutter, and possible jamming signals.

10.2 PERFORMANCE METRICS

Radar measures the reflectivity in a three dimensional coordinate system (R, θ, ϕ) where R indicates the range to the object in the line of sight direction, θ is the azimuth angle, and ϕ is the elevation angle. The azimuth and elevation angles determine the position of an object in space relative to the observation point. As indicated in Figure 10.3, the azimuth angle is measured in the clockwise direction from true North along the horizon and ranges from 0 to 360°. The elevation or altitude angle is the angle above the horizon and generally ranges from 0 to 90°. In cases where the observer is above the object, for example in terrain mapping or surveillance from an aircraft, the elevation angle range would be extended downward to -90°. Radar applications

such as tracking or searching for multiple objects in space require changing the direction of the antenna beam in order to effectively map the area surrounding the observer. Electronic beam steering of phased array antennas (composite antennas) allows for an almost instantaneous change in beam direction (microseconds) and significantly increases the multi-mode capability of the radar system. Mechanical beam steering, which requires physical movement of the antenna to change beam direction, is considerably slower than electronic beam steering and suffers from inertial and vibration effects. Antenna theory and electronic beam steering are beyond the scope of this text. The focus in this chapter will be on the transmitted radar signal, and the processing that occurs on the return signal reflected from an object to determine range and radial velocity.

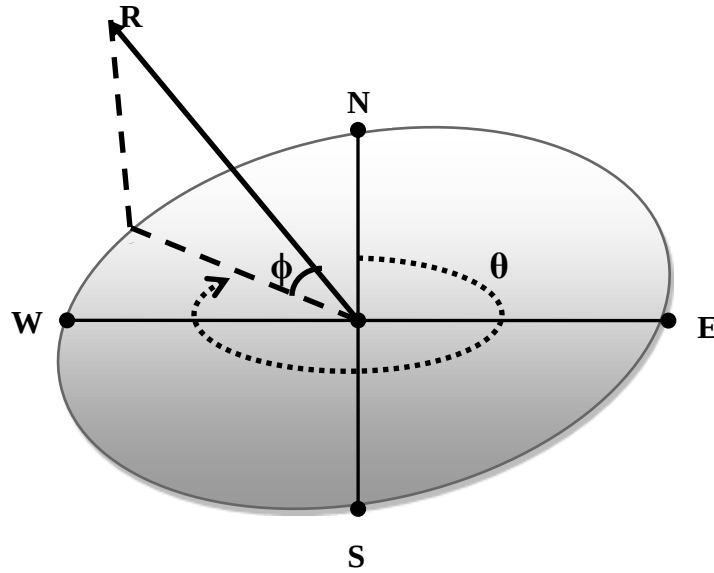


Figure 10.3: Azimuth and Elevation Angles

Some of the common metrics used to measure performance of a radar system are signal-to-noise ratio (SNR), range resolution, Doppler resolution, range ambiguity, and Doppler ambiguity.

Signal to Noise Ratio (SNR)

The SNR is the ratio of signal power to noise power and is calculated at various points in the receiver/processing system beginning at the antenna. Higher SNR means stronger signal power which increases the probability of detecting the desired signal and locating targets and target information. The SNR is directly proportional to the energy in the transmitted signal. Since the amplitude modulation for the pulsed radar signal is on/off, the antenna transmits at full-power every time a pulse is sent. Also, lengthening the duration of a pulse burst, τ_d , will increase SNR since signal energy increases with pulse length.

A simple form of the radar equation for a pulsed radar system, presented here without derivation, is:

$$\text{SNR} = \frac{P_t G^2 \lambda^2 \sigma}{(4\pi)^3 k T_0 B F L R^4} \quad (10.4)$$

SNR: Signal to Noise Ratio at Receiver Output (before signal processing)

P_t : Peak transmitted Power (W)

G : Directional Antenna Gain

λ : Wavelength = c/F_T (m)

σ : RCS (radar cross section) of target (m^2)

τ_d : Transmitted pulse duration (sec)

R : Range to target (m)

k : Boltzmann's constant ($1.38 \cdot 10^{-23}$ J/K)

T_o : Antenna Effective Temperature (290K)

B : Radar Operating Bandwidth = $1/\tau_d$ (Hz)

F : Receiver Noise Figure

L : Total radar losses

Typically, the receiver noise figure, F , and total radar losses, L , are given in units of dB, and must be converted to linear units before being plugged into equation 10.4 using $F = 10^{F(\text{dB})/10}$ and $L = 10^{L(\text{dB})/10}$. Alternatively, equation 10.4 can be converted to dB units using $\text{SNR}_{\text{dB}} = 10 \cdot \log(\text{SNR})$. The RCS (radar cross section) of the target measures how much of the radiated energy that impinges on the target is radiated back in the direction of the antenna. This target specific parameter is proportional to target size, physical shape of the target, target material, and orientation.

The SNR in equation 10.4 is the signal to noise ratio at the receiver output prior to signal processing. Notice that the SNR is inversely proportional to the range to the target raised to the 4th power. The minimum SNR required by the processor to successfully detect a target determines the maximum range capability of the system. Given the minimum required SNR along with the antenna characteristics and transmission parameters, equation 10.4 can be used to determine the maximum detectable target range, R_{max} .

Range Resolution

Range resolution is the minimum distance required between two targets in order to distinguish between them. For the simple pulsed radar signal shown in Figure 10.2, the range resolution is given by:

$$\Delta R = c \cdot \tau_d / 2 \quad (10.5)$$

ΔR = range resolution for simple pulse (m)

c = $3 \cdot 10^8$ (m/s)

τ_d = pulse width (s)

If two targets are closer together than ΔR , their return pulses will overlap at the receiver making it impossible to distinguish between the two for a simple pulse waveform. It should also be clear that a simple pulsed radar signal forces a tradeoff between SNR and range resolution. Resolution is improved by reducing the pulse duration, τ_d , but this will also reduce the energy of the signal which in turn reduces the SNR. In the next section, chirp signals and matched filtering will be introduced which will effectively decouple range resolution from pulse duration.

Doppler Resolution

The Doppler resolution is the smallest detectable Doppler shift and is inversely proportional to the signal observation time. Remember for Discrete Fourier Transforms, the frequency resolution is inversely proportional to the total time over which the signal is sampled. The same

principle applies here. For radar systems, the pulse duration is typically 100ns to 100μs which would result in a Doppler resolution of 10 MHz for a single 100ns pulse, and 10 kHz for a single 100μs pulse. It will be shown in Section 10.5 that the expected Doppler shifts for targets fall considerably below these values, and it is therefore necessary to process several consecutive pulses in order to improve the Doppler resolution and hence the accuracy of the radial velocity estimate.

Range Ambiguity

Range ambiguity refers to a phenomenon where two targets are spaced in such a way that the reflected echo pulses return to the receiver antenna at the same time. Note that this is an entirely different situation from range resolution, where two targets that are spaced closer together than the range resolution, ΔR , cannot be distinguished as two separate targets. Figure 10.4 illustrates the concept of range ambiguity.

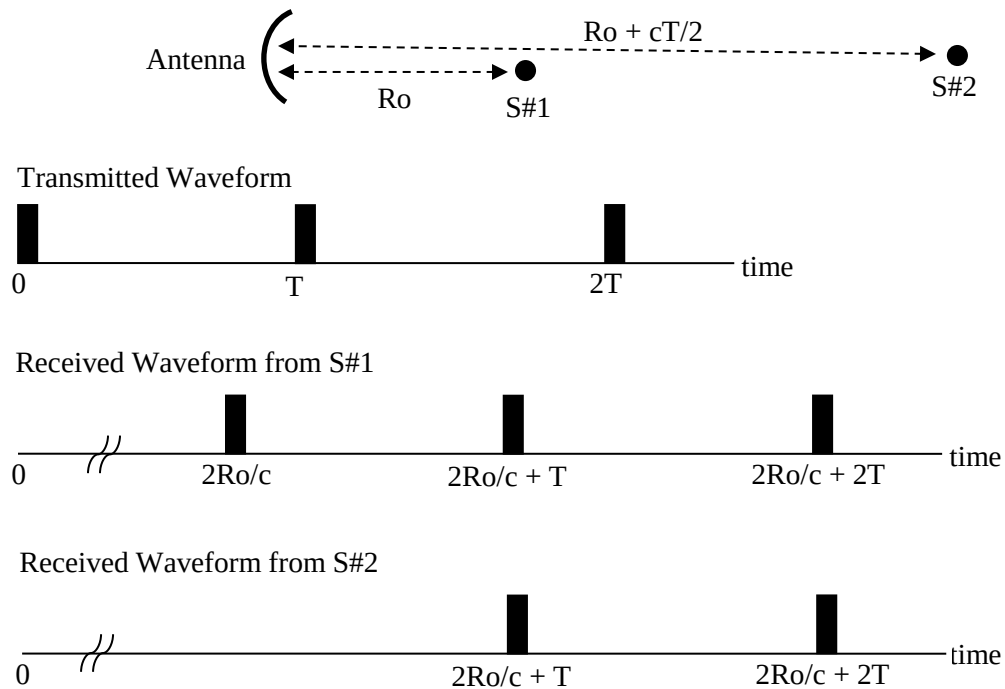


Figure 10.4: Illustration of Range Ambiguity

A pulse burst waveform with a pulse repetition interval (PRI) of T is transmitted by an antenna and is reflected back by two scatterers (S#1 and S#2) that are both within the antenna beamwidth. Scatterer #1 is located at a range of R_0 meters from the antenna and Scatterer #2 is at a range of $R_0 + cT/2$ meters. The total time for a pulse to travel from the antenna to S#1 and back would be $2R_0/c$, while the total time for a pulse to travel from the antenna to S#2 and back would be $2(R_0 + cT/2)/c = 2R_0/c + T$. As shown in Figure 10.4, the first transmitted pulse echoed back from S#2 arrives back at the receiver at exactly the same time as the second transmitted pulse reflected from S#1 hits the receiver. This range ambiguity would occur for all targets spaced apart by any integer multiple of $cT/2$; that is, targets at ranges of R_0 and $R_0 +$

$k \cdot (cT/2)$, where k is any integer, would have return pulses that coincide in time at the receiver. It should be noted that Figure 10.4 does not give any indication of return signal strength. Signal strength drops as travel distance increases, but return signal strength also depends on the radar cross section (RCS) of each scatterer. The unambiguous range, $cT/2$, can be increased by increasing the pulse repetition interval (PRI) or equivalently by decreasing the pulse repetition frequency (PRF). Long range surveillance radars would require relatively low PRFs. The PRF is chosen to make the unambiguous range large enough to meet the radar systems operating specifications.

Doppler Ambiguity

The unambiguous Doppler spectrum width is half of the pulse repetition frequency, $PRF/2$ (Hz). Any Doppler shift outside this bandwidth ($F_D > PRF/2$) will alias to a frequency, $F_{D\text{-alias}}$, within the unambiguous Doppler spectrum ($-PRF/2 \leq F_{D\text{-alias}} \leq PRF/2$). Doppler shift is dependent on several factors including the relative velocity between the radar platform and the target, and the wavelength of the transmitted signal. As long as the range of possible Doppler shifts falls below the unambiguous Doppler spectrum ($PRF/2$), aliasing will not present a problem.

Clearly, the unambiguous Doppler spectrum can be increased by simply increasing the PRF, or equivalently, by decreasing the PRI. However, increasing the PRF will have a detrimental effect on range estimation since it will decrease the unambiguous range. Thus, there is a tradeoff in selecting the pulse repetition frequency, PRF, to allow for both a reasonable unambiguous range and a sufficiently wide Doppler bandwidth. For fast moving targets at long range, it may very well not be possible to use a single PRF for both range and velocity estimation. In this case, some radar systems utilize multiple PRFs to resolve range and Doppler ambiguities.

10.3 MATCHED FILTERS, CHIRP SIGNALS, AND PULSE COMPRESSION

In radar processing, short pulse bursts are transmitted and the return signal (reflected back from objects within the antenna beamwidth) is analyzed to determine range, radial velocity, and possibly target identification. As indicated previously, the signal to interference ratio (SIR) at the receiver antenna is very small. The signal power weakens considerably as the signal travels to an object and is reflected back. Also the signal becomes corrupted with noise from clutter (unwanted echoes from the earth's surface, weather phenomena such as rain or clouds, or man-made clutter such as chaff clouds), possible interference from electronic jamming, and echoes from other nearby objects.

Matched Filters

A matched filter is a filter designed to match one specific input signal, and it is capable of recovering that signal in the presence of quite a lot of noise and interference. The transfer function of a filter matched to a specified input signal, $x(t)$, and the impulse response of the matched filter are shown in equation 10.6.

$$\begin{aligned} \text{Matched Filter Transfer Function: } H(f) &= X^*(f) \\ \text{Matched Filter Impulse Response: } h(t) &= x^*(\tau_d - t) \end{aligned} \quad (10.6)$$

Equation 10.6 indicates that the transfer function of the matched filter is simply the complex conjugate of the Fourier Transform of the input signal $x(t)$. The impulse response of the matched filter is the complex conjugate of the input signal reversed in time, where τ_d is the duration of the input signal. So what would the matched filter output look like if the input signal is indeed the specified signal, $x(t)$? The output of a filter is the convolution of the input signal with the impulse response of that filter, so in the case of the matched filter, the output response would be:

$$\begin{aligned} y(t) &= h(t) * x(t) = \int_{-\infty}^{\infty} h(t - \tau)x(\tau)d\tau \\ &= \int_{-\infty}^{\infty} x^*(\tau - (t - \tau_d))x(\tau)d\tau = R_{xx}(t - \tau_d) \end{aligned} \quad (10.7)$$

In the absence of noise, the output response of a matched filter to the input signal on which the matched filter design was based is simply the correlation of that input signal with itself, in other words, an autocorrelation. At $t = \tau_d$ (the input signal duration), the filter output response would peak at the value $R_{xx}(0)$. When the incoming signal is corrupted with noise, the matched filter runs a correlation with an uncorrupted reference signal attempting to find a distinguishable peak or match. The impulse response for a digital FIR filter matched to a specified input signal, $x(t)$, sampled at a frequency of F_s is given by:

$$h(k) = \text{ifft}\{[\text{fft}(x(k))]^*\} \quad (10.8)$$

In other words, sample the input signal, calculate the FFT of the input samples, take the complex conjugate of the FFT, and then take an inverse FFT to compute the filter coefficients of the digital matched filter for $x(t)$. This process is illustrated in Example 10.1 for a simple sinusoidal pulse signal.

Example 10.1: Calculating Matched Filter Coefficients

Consider the simple sinusoidal pulse signal shown in Figure 10.5. The frequency of the sinusoid is a constant 2 MHz and the pulse duration is 5 μs .

- Assuming a sampling frequency of 20 MHz and a sampling duration of 5 μs , calculate and plot the digital FIR coefficients for the matched filter.
- Plot the output of the matched filter when the input is the sinusoidal pulse signal shown in Figure 10.5.
- Plot the matched filter coefficients and the matched filter output if the pulse duration of the sinusoidal signal is doubled to 10 μs . What changes?
- Using the matched filter designed in part (a) for the 2 MHz sinusoidal pulse of duration 5 μs , plot the filter output for a 2.4 MHz sinusoidal input pulse of duration 1 μs and the filter output for a 1.8 MHz sinusoidal input pulse of duration 5 μs . Comment on the results.

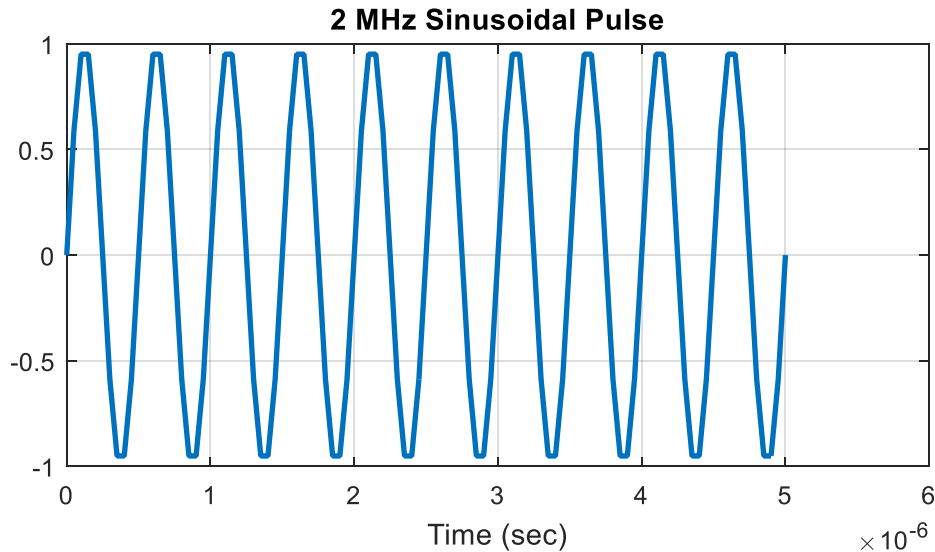


Figure 10.5: Input Sinusoidal Pulse Signal for Matched Filter Design

Solution

The MATLAB[®] Code for (a), (b) is:

```
% 2MHz sin pulse of duration 5 us sampled at 20MHz
Fs = 20e6;
t=0:1/Fs:5e-6;
x = sin(2*pi*2e6*t);

h = ifft(conj(fft(x))); % Matched Digital FIR filter to x(t)
k = 1:length(h);
subplot(2,1,1); stem(k,h);
title('Matched Filter Coefficients for 5us Pulse');

% Output of Matched Filter to input on which design was based
filter_out = conv(h,x);
tout = 0:1/Fs:1/Fs*(length(filter_out)-1);
subplot(2,1,2); plot(tout, filter_out);
title('Output of Matched Filter');
xlabel('Time (sec)'); grid
```

Figure 10.6 shows the 101 matched FIR filter coefficients. Note the similarity with the specified input function in Figure 10.5. As indicated by equation 10.6, the filter coefficients are simply samples of the input signal reversed in time. Figure 10.6 also shows the matched filter output to the input upon which the filter design was based. Note that a peak in the output response occurs at 5 μ s (duration of the input pulse signal), and the width of the output signal is twice the duration of the sinusoidal pulse. The height of the peak depends directly on the energy of the input signal.

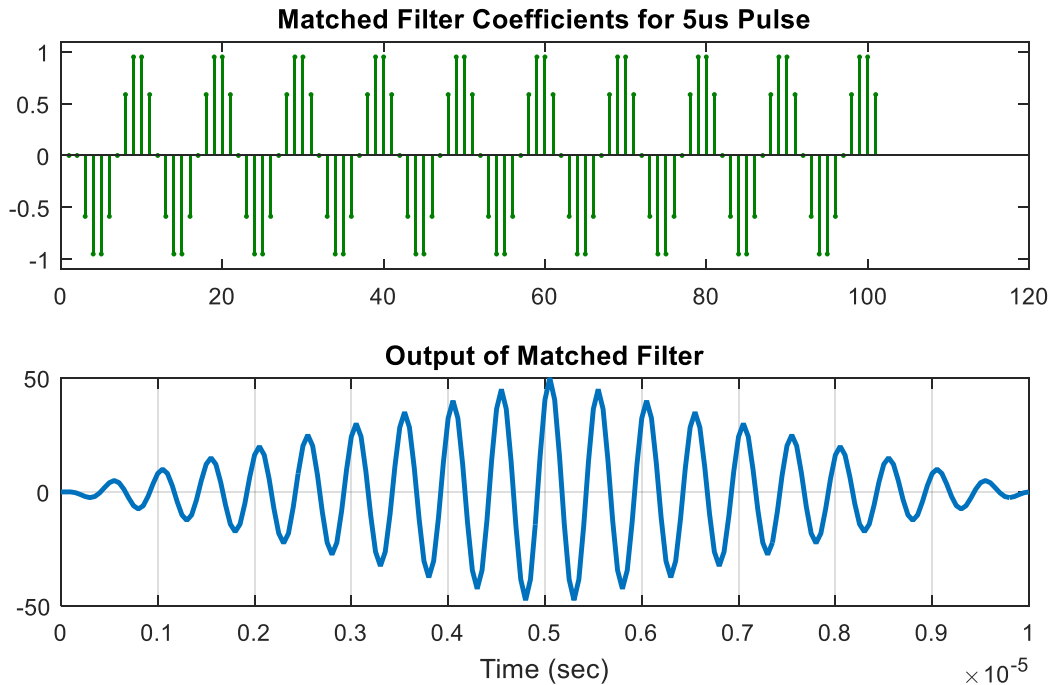


Figure 10.6: Matched Filter Coefficients and Matched Filter Output

The MATLAB code for part (c) is:

```
% Effect of longer pulse duration on Filter Design & Output
tlong = 0:1/Fs:10e-6;
x = sin(2*pi*2e6*tlong);
h = ifft(conj(fft(x))); % Matched Digital FIR filter to x(t)
figure;
k = 1:1:length(h);
subplot(2,1,1); stem(k,h);
title('Matched Filter Coefficients for 10us Pulse');
filter_out = conv(h,x); % Output of Matched Filter
tout = 0:1/Fs:1/Fs*(length(filter_out)-1);
subplot(2,1,2); plot(tout, filter_out);
title('Output of Matched Filter with 10us Pulse Duration');
xlabel('Time (sec)'); grid
```

The results are shown in Figure 10.7. Comparing Figures 10.6 and 10.7, it should be clear that doubling the pulse duration doubles the number of filter coefficients, doubles the width of the filter output, and doubles the peak output (since input signal energy doubles when the input pulse duration is doubled).

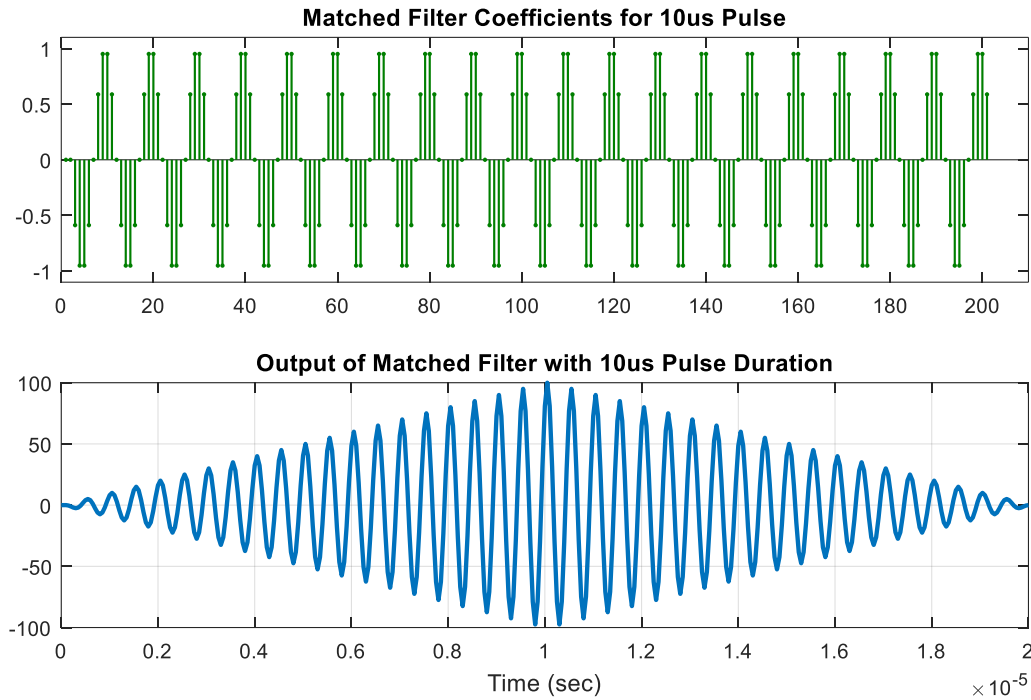


Figure 10.7: Matched Filter Coefficients and Output for a Longer Pulse Duration

The MATLAB code for part (d) is:

```
% Effect of using "unmatched" inputs
x = sin(2*pi*2e6*t);
h = ifft(conj(fft(x))); % Matched Digital FIR filter to x(t)
filter_out = conv(x,h);
x_24 = sin(2*pi*2.4e6*t);
filter_out_24 = conv(h,x_24);
x_18 = sin(2*pi*1.8e6*t);
filter_out_18 = conv(h,x_18);
tout = 0:1/Fs:1/Fs*(length(filter_out)-1);
figure
subplot(3,1,1); plot(tout,filter_out);
title('Matched Filter Output for Specified 2MHz Sinusoidal Input Pulse');
xlabel('Time (sec)'); grid
subplot(3,1,2); plot(tout, filter_out_24);
title('Matched Filter Output for 2.4MHz Sinusoidal Input Pulse');
xlabel('Time (sec)'); grid
subplot(3,1,3); plot(tout, filter_out_18);
title('Matched Filter Output for 1.8 MHz Sinusoidal Input Pulse'); xlabel('time');
xlabel('Time (sec)'); grid
```


The results are shown in Figure 10.8. The first plot shows the filter output response to the 2MHz input signal that the filter was designed to match. As previously noted, this output response produces a peak in the output at 5 μs (the pulse duration) and this peak is directly proportional to the energy in the input signal. The next two plots show the filter response to the “unmatched” inputs; that is, sinusoidal pulses at 2.4 MHz and 1.8 MHz. As expected, the filter output for these “unmatched” input signals is significantly smaller than the output to the 2 MHz signal, particularly at the expected peak time of 5 μs , because the filter was designed to “find” a 2 MHz sinusoidal pulse using correlation.

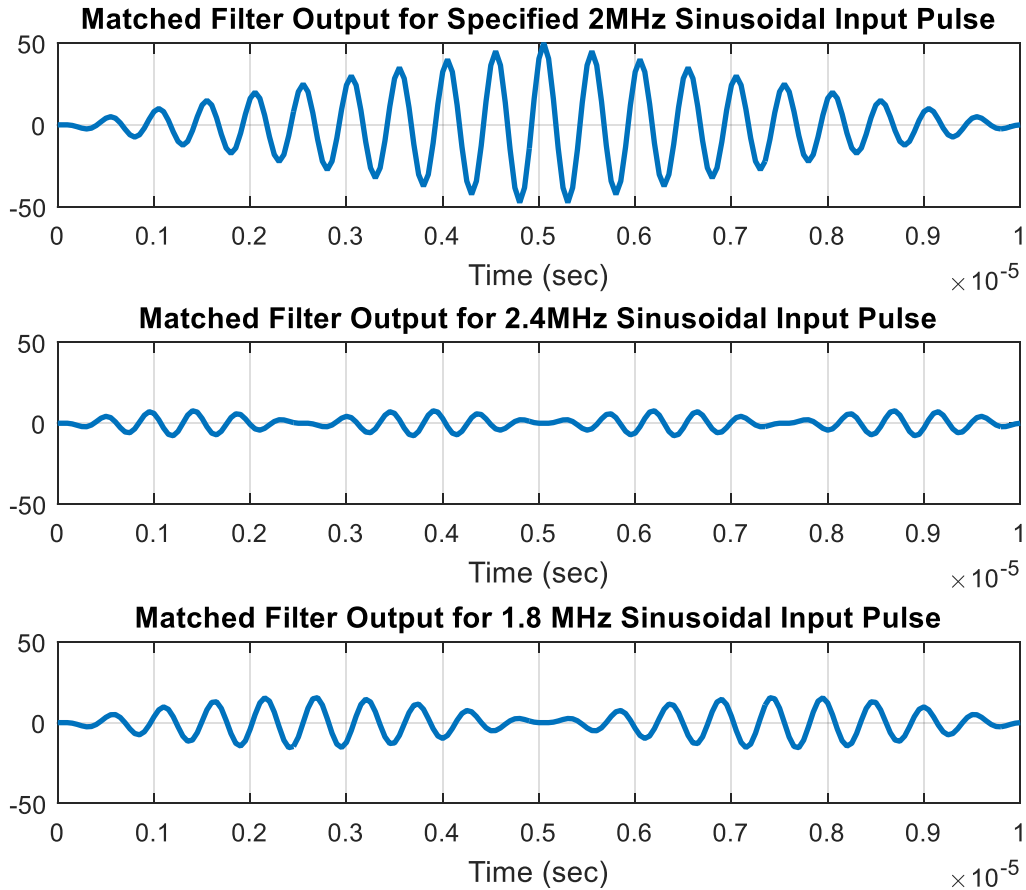


Figure 10.8: Filter Output for “Unmatched” Input Signals

Example 10.1 illustrates a problem that was alluded to in the previous section; that is, the conflict between achieving good signal to noise ratio (SNR) and good range resolution using a simple constant frequency sinusoidal pulse burst signal. The SNR is directly proportional to the signal energy and can be increased by increasing the duration of the sinusoidal pulse. However, increasing the pulse duration has a detrimental effect on range resolution. In order to detect two closely spaced objects as being distinct objects, the return pulses from the targets must produce peaks in the matched filter output signal that are sufficiently separated in time to be recognizable

as two distinct peaks. As illustrated in Example 10.1, as pulse duration increases, the matched filter output response spreads out in time which in turn increases the required distance between peaks, and therefore increases the minimum distance required between objects in order to make them distinguishable as two separate objects. When using a simple sinusoidal pulse of duration, τ_d , the return pulses from two closely spaced objects cannot overlap in time so the objects must be separated in time by τ_d or equivalently in distance by $c \cdot \tau_d / 2$. Processing the return pulses through a matched filter doesn't change the situation. Fortunately, there are alternative types of pulse burst signals that decouple range resolution from pulse duration by using frequency or phase modulation on the transmitted carrier signal. Processing frequency or phase modulated signals through matched filters results in significant pulse compression. A pulse burst signal that uses linear frequency modulation (LFM), aptly named an LFM chirp signal, is explored in detail in the following subsection.

Chirp Signals

A chirp signal is a short burst sinusoidal signal with constant amplitude and a frequency that varies in some manner over the pulse width. A chirp signal where the frequency varies linearly over the pulse time is called an LFM (linear frequency modulated) chirp signal. Although there are many other types of pulse compression waveforms including non-linear FM chirp signals, and bi-phase and poly-phase signals which utilize phase modulation (PM), the remainder of this chapter will focus on LFM chirp signals. An LFM chirp signal can be described by:

$$x(t) = a(t) \sin \left(2\pi F_T t + \pi \frac{\beta}{\tau_d} t^2 \right) \quad (10.9)$$

$a(t)$ is the on/off amplitude modulation
 F_T is the transmit frequency
 τ_d is the pulse burst duration
 β is the frequency sweep range

As mentioned previously, $a(t)$ is one over the duration of the pulse, τ_d , and zero otherwise so that the pulse burst is transmitted at full-power to maximize signal power. The instantaneous frequency of the LFM signal is determined by simply taking the derivative of the sin function argument in equation 10.9 as follows:

$$\frac{d}{dt} \left(2\pi F_T t + \pi \frac{\beta}{\tau_d} t^2 \right) = 2\pi F_T + 2\pi \frac{\beta}{\tau_d} t = 2\pi \left(F_T + \frac{\beta}{\tau_d} t \right) \quad (10.10)$$

Equation 10.10 indicates that the frequency of the LFM chirp signal varies linearly over time from F_T to $(F_T + \beta)$ Hz over the pulse duration time range $t = 0$ to $t = \tau_d$. The signal in equation 10.9 is referred to as an *upchirp* signal since the frequency increases over time from F_T to $F_T + \beta$. Changing the plus sign to a minus sign in equation 10.9 would create a *downchirp* signal where the transmitted frequency decreases linearly from F_T to $F_T - \beta$ over the pulse duration, τ_d .

The MATLAB function **chirp.m** can be used to create chirp signals by specifying start time, start frequency, end time, end frequency, and method of frequency variation (linear, logarithmic, or quadratic). Figure 10.9 shows the plot of an LFM chirp signal and the corresponding FFT. This

particular chirp signal has a pulse width of 25 μs , sweeps linearly from 0 Hz to 2 MHz over the pulse width, and is sampled at 40 MHz (ten times the Nyquist rate). The change in frequency over time is clearly shown in Figure 10.9. Note that the LFM chirp signal bandwidth is equal to the sweep frequency range which is 2 MHz in this case.

Key parameters for the chirp signal are pulse duration, frequency sweep range, sampling frequency, and the *time bandwidth product*. The time bandwidth product is the product of the pulse width or burst time, τ_d , and the pulse bandwidth or equivalently the frequency sweep range, β . The LFM chirp signal shown in Figure 10.9 has a time bandwidth product of 50 ($25\mu\text{s} \cdot 2 \text{ MHz}$). The Nyquist sampling rate for an LFM chirp signal would be $2\cdot\beta$, but in practice, the signal should be significantly oversampled. The effects of pulse duration, frequency sweep and the time bandwidth product on the output of a matched filter tuned to an LFM chirp signal is explored in the next subsection.

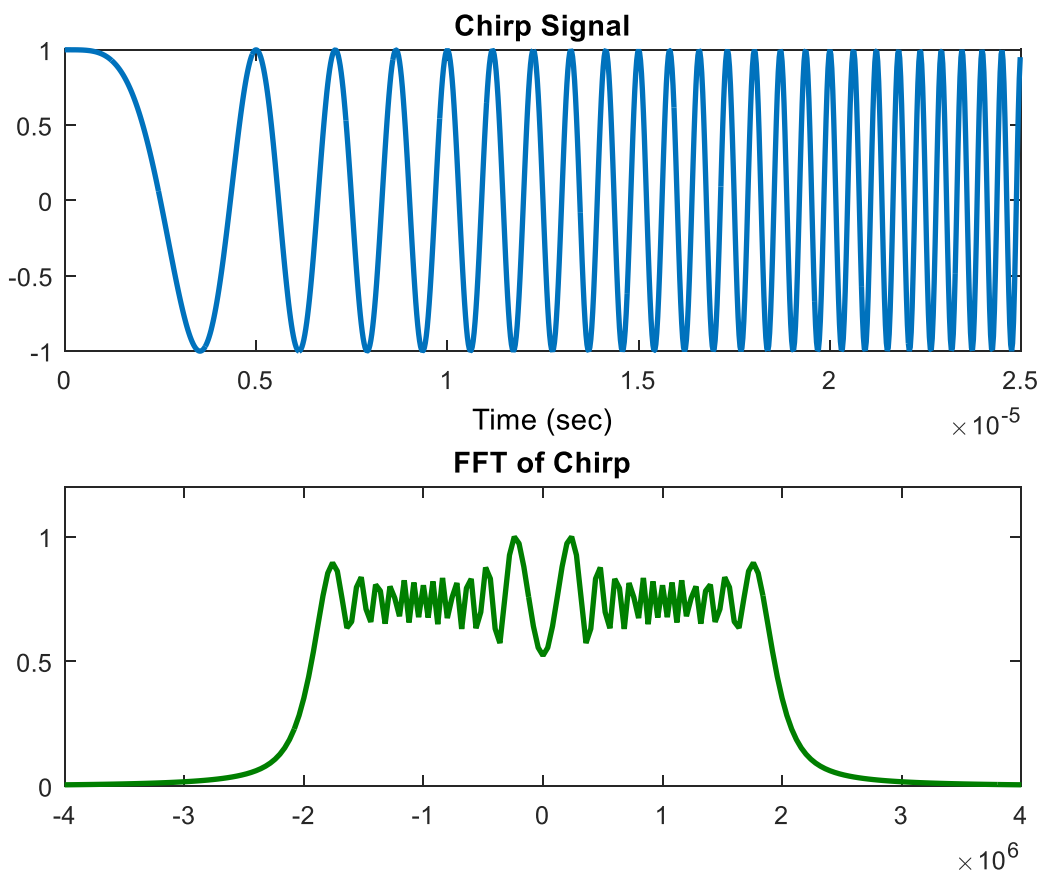


Figure 10.9: LFM Chirp Signal

Pulse Compression of LFM Chirp Signals using Matched Filters

In Example 10.1, a simple sinusoidal pulse with a constant frequency was processed through a matched filter. When the duration of the sinusoidal pulse was doubled, the output response of the matched filter spread out in time (doubled) which had a detrimental effect on range resolution while improving signal to noise ratio. An LFM chirp signal, when processed through a filter matched to the chirp signal, produces a much narrower, better-defined peak in the output response. The pulse compression that occurs when an LFM chirp signal is processed through a matched filter is illustrated in Example 10.2.

Example 10.2: Pulse Compression using Matched Filters

- (a) Consider an LFM chirp signal with a pulse duration of $\tau_d = 5 \mu\text{s}$, a sweep frequency range of $\beta = 10 \text{ MHz}$, and a sampling rate of $F_s = 200 \text{ MHz}$. Plot the chirp signal, find the matched filter coefficients for the chirp signal, and then plot the matched filter output response to the LFM chirp signal.
- (b) Repeat (a) for $\tau_d = 10 \mu\text{s}$, $\beta = 10 \text{ MHz}$, and $F_s = 200 \text{ MHz}$.
- (c) Repeat (a) for $\tau_d = 10 \mu\text{s}$, $\beta = 5 \text{ MHz}$, and $F_s = 200 \text{ MHz}$.
- (d) Repeat (a) for $\tau_d = 5 \mu\text{s}$, $\beta = 20 \text{ MHz}$, and $F_s = 200 \text{ MHz}$.

Solution The MATLAB script function for each part of Example 10.2 is:

```
function Ex10_2(tau,Beta,Fs)
t = 0:1/Fs:tau-1/Fs;
% LFM Chirp signal of duration tau, sweep frequency range Beta
x = chirp(t,0,tau,Beta);
figure
subplot(2,1,1); plot(t*1e6,x);
title(['LFM Chirp Signal (tau = ' num2str(tau*1e6) 'us, Beta = '
num2str(Beta/1e6) 'MHz, Fs = ' num2str(Fs/1e6) 'MHz']]);
xlim([0 10]); xlabel('Time (us)'); grid
% Matched Digital FIR filter to LFM Chirp Signal x(t)
h = ifft(conj(fft(x)));
filter_out = conv(h,x);
tout = 0:1/Fs:1/Fs*(length(filter_out)-1);
subplot(2,1,2); plot(tout*1e6, filter_out);
title('Output of Matched Filter');
xlim([0 20]); xlabel('Time (us)'); grid
```

The results are shown in Figure 10.10. Figure 10.10 (a) shows the $5 \mu\text{s}$ LFM chirp signal and the corresponding output signal from the matched filter. Compare this filter output response to the matched filter output response shown in Figure 10.6 to a $5 \mu\text{s}$ simple sinusoidal input pulse. In both cases, the output response peaks at the pulse duration of $5 \mu\text{s}$ as expected. However, the output response drops off very rapidly for the LFM chirp signal compared to the simple sinusoidal pulse. It will be shown in the next section that this pulse compression effect significantly improves range resolution allowing return pulses from distinct objects to overlap in time and still present as separate peaks once the return signal is processed through the matched filter.

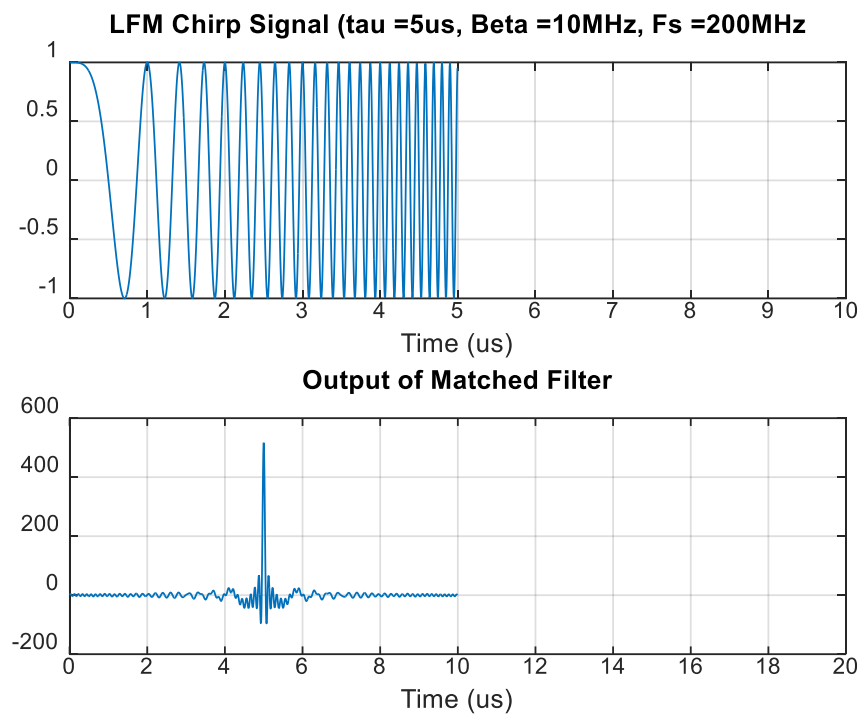


Figure 10.10(a)

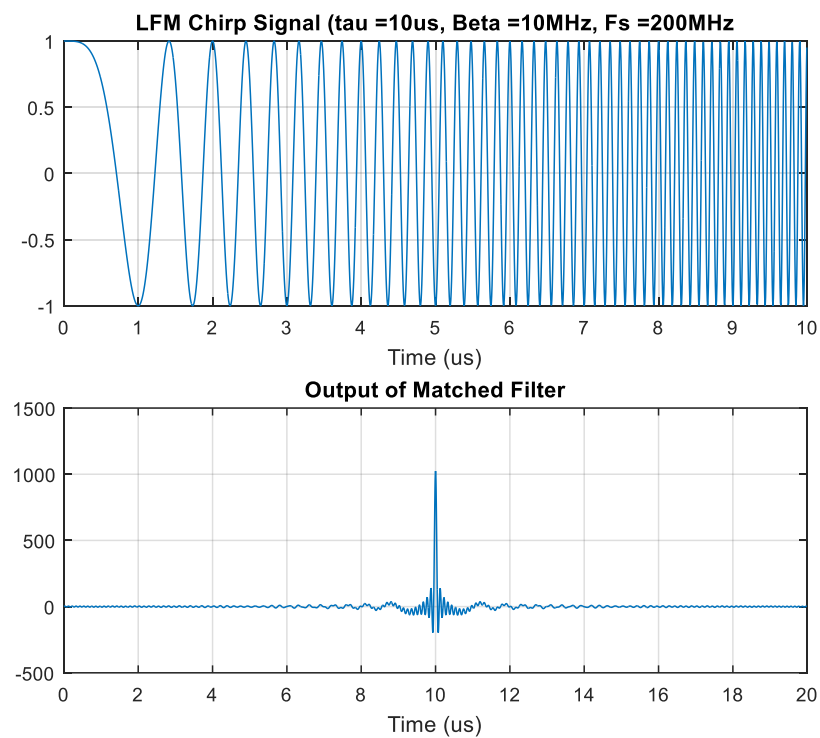


Figure 10.10(b)

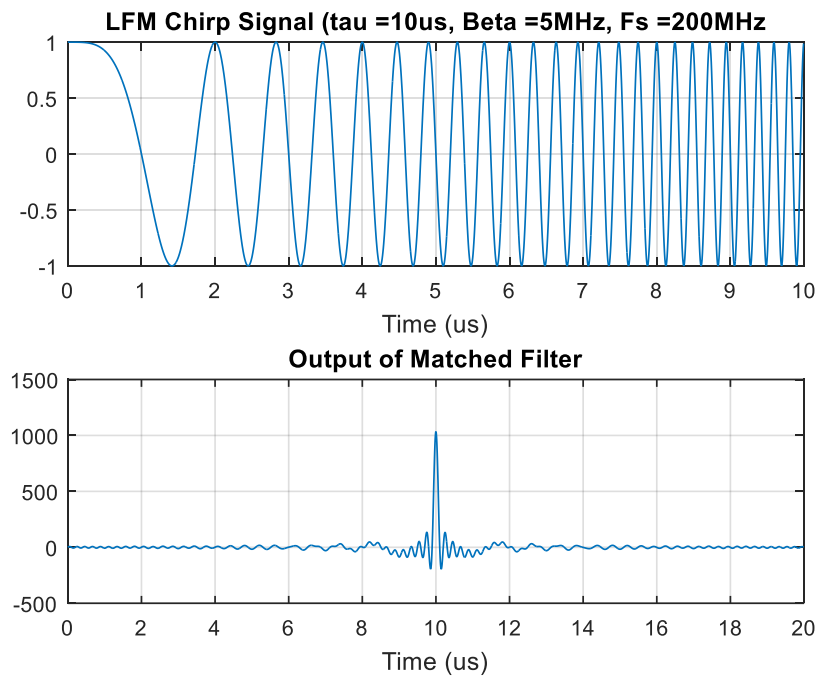


Figure 10.10(c)

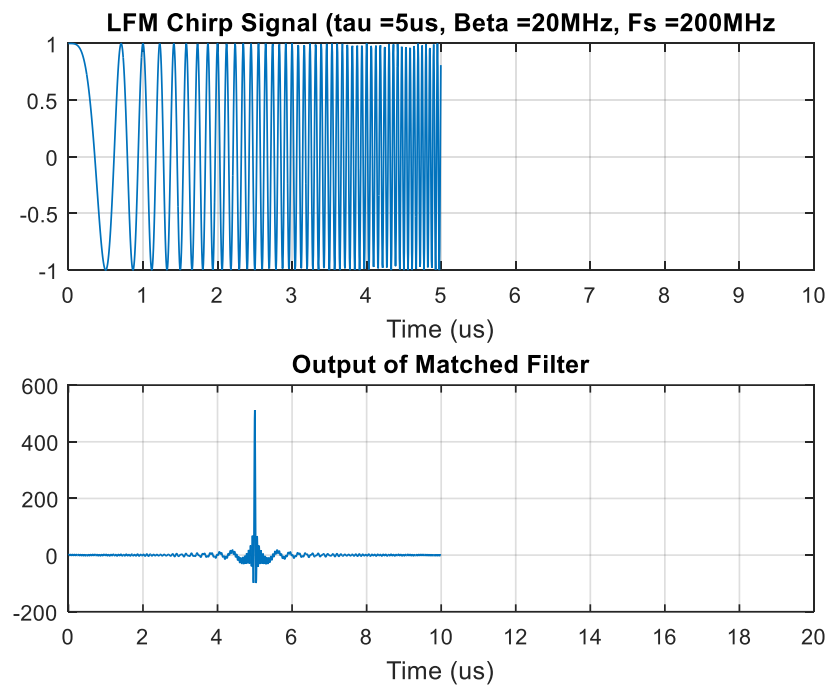


Figure 10.10(d)

Figure 10.10: Pulse Compression - Matched Filter Processing of LFM Signals

Figure 10.10 (b) shows the effect of doubling the pulse duration while leaving sweep rate and sampling rate the same. The output peak shifts to $10\ \mu\text{s}$ (the new pulse duration) and, as expected, the peak doubles because the input signal energy doubles. Notice that the width of the main peak stays the same even though pulse duration is doubled. With the simple sinusoidal pulse of Example 10.1, the duration of the output response doubled when the simple sinusoidal pulse duration was doubled.

Figure 10.10 (c) shows the effect of cutting the frequency sweep rate in half (to 5 MHz) while leaving pulse duration at $10\ \mu\text{s}$ and sampling frequency the same. The filter output response still peaks at $10\ \mu\text{s}$ with the same magnitude as 10.10(b). However, reducing the sweep frequency widened the main lobe of the output response which would reduce the range resolution.

Figure 10.10 (d) shows the effect of increasing the sweep frequency range, β , and reducing the pulse duration to the original $5\ \mu\text{s}$. The increase in sweep frequency range causes the main lobe of the output response to narrow which would improve range resolution. The peak in the output response occurs at the pulse duration of $5\ \mu\text{s}$ and the peak magnitude matches that of part (a).

The peak in the matched filter output response occurs at the pulse duration, τ_d . The width of the main output lobe is determined by the sweep frequency range, β . Larger frequency ranges result in narrower peaks and better range resolution. Although not shown in the example, increasing the sampling frequency will increase the peak filter output. Of course increasing the sampling frequency increases the filter size, which in turn increases the computational complexity and reduces the computational time interval.

The pulse compression that occurs when processing an LFM chirp signal through a matched filter will occur for other types of chirp signals, including non-linear FM and stepped FM, and for bi-phase and poly-phase code signals that utilize phase modulation (PM) rather than frequency modulation. This should be somewhat intuitive based on the fact that processing a signal through its matched filter is equivalent to computing an autocorrelation. Recall from Chapter 4 that an auto-correlation is calculated by taking a copy of a signal, sliding that copy under the original signal, and computing the sum of the products where the signals overlap. The peak occurs when the two signals are in perfect alignment. When the second signal is shifted right or left, the signals become misaligned and the correlation decreases. Consider the simple sinusoidal pulse in Example 10.1 and imagine a second copy of this sinusoidal signal shifting underneath. Even when these signals aren't lined up exactly, they are still very similar. In fact, when shifting by an integer multiple of the period, the samples match perfectly although the non-zero overlap window is smaller. So, the matched filter output for the sinusoidal pulse shows a gradual decrease from the peak output; in fact, an amplitude envelope for the matched filter response looks linear. Now consider the LFM chirp signal of Example 10.2 and imagine a second copy of this chirp signal shifting underneath. Once these two signals are misaligned, they look very different from one another so the correlation function drops very rapidly. Increasing the frequency sweep range, β , causes the chirp signal to vary more rapidly which in turn causes the correlation function to drop off even more rapidly for small misalignment producing a narrower peak in the filter output response.

Effect of Pulse Compression on SNR and Range Resolution

The SNR at the output of the receiver prior to processing is shown in equation 10.4. Notice that the SNR at the receiver output is independent of the type of modulation (frequency or phase or none) used on the transmitted sinusoidal pulse. The ratio of the SNR out of the matched filter to the SNR into the matched filter; that is, the improvement in SNR caused by the matched filter processing, is directly related to the time bandwidth product of the radar pulsed signal:

$$\text{SNR}_{\text{out(peak)}} / \text{SNR}_{\text{in}} = 2 \cdot \beta \cdot \tau_d \quad (10.11)$$

$\text{SNR}_{\text{out(peak)}}:$	Output peak instantaneous SNR
$\text{SNR}_{\text{in}}:$	Signal to Noise ratio into matched filter
$\tau_d:$	Pulse duration (sec)
$\beta:$	Pulse bandwidth (frequency sweep for LFM chirp)

For a simple sinusoidal pulse, the time bandwidth product is equal to one. For LFM chirp signals, the time bandwidth product is greater than one; ranging from as low as 5 to several 1000 for high-end radar systems.

The range resolution, or minimum distance required between two objects to distinguish the objects as separate, is given by:

$$\Delta R = c / (2 \cdot \beta) \quad (10.12)$$

For a simple sinusoidal pulse, the range resolution is $c \cdot \tau_d / 2$ because the signal bandwidth is $1/\tau_d$. As mentioned previously, this direct dependence on pulse duration creates a conflict between SNR and resolution; that is, while decreasing the pulse duration improves radar resolution it also reduces SNR. For pulses that utilize either frequency modulation (FM) or phase modulation (PM), the pulse bandwidth is decoupled from the pulse duration. For example, the LFM chirp signal has a bandwidth equal to the frequency sweep. So, the frequency sweep can be chosen to give an acceptable range resolution and the pulse duration can be chosen to give an acceptable SNR.

10.4 DETERMINING RANGE

Figure 10.11 is a very simple block diagram of a monostatic pulsed radar system. The duplexer is a switch which controls when the radar system is transmitting a pulse and when the system is in receiver mode. The waveform signal generator and transmitter generate the pulsed waveform signal that is then radiated by the antenna. During pulse transmission, the radar system is “blind” in the sense that any pulses that echo off an object and return within the transmission time will not be detected. The duty cycle of the pulsed waveform is typically less than 1% so the radar system spends over 99% of the time in receiver mode searching for objects in the range space.

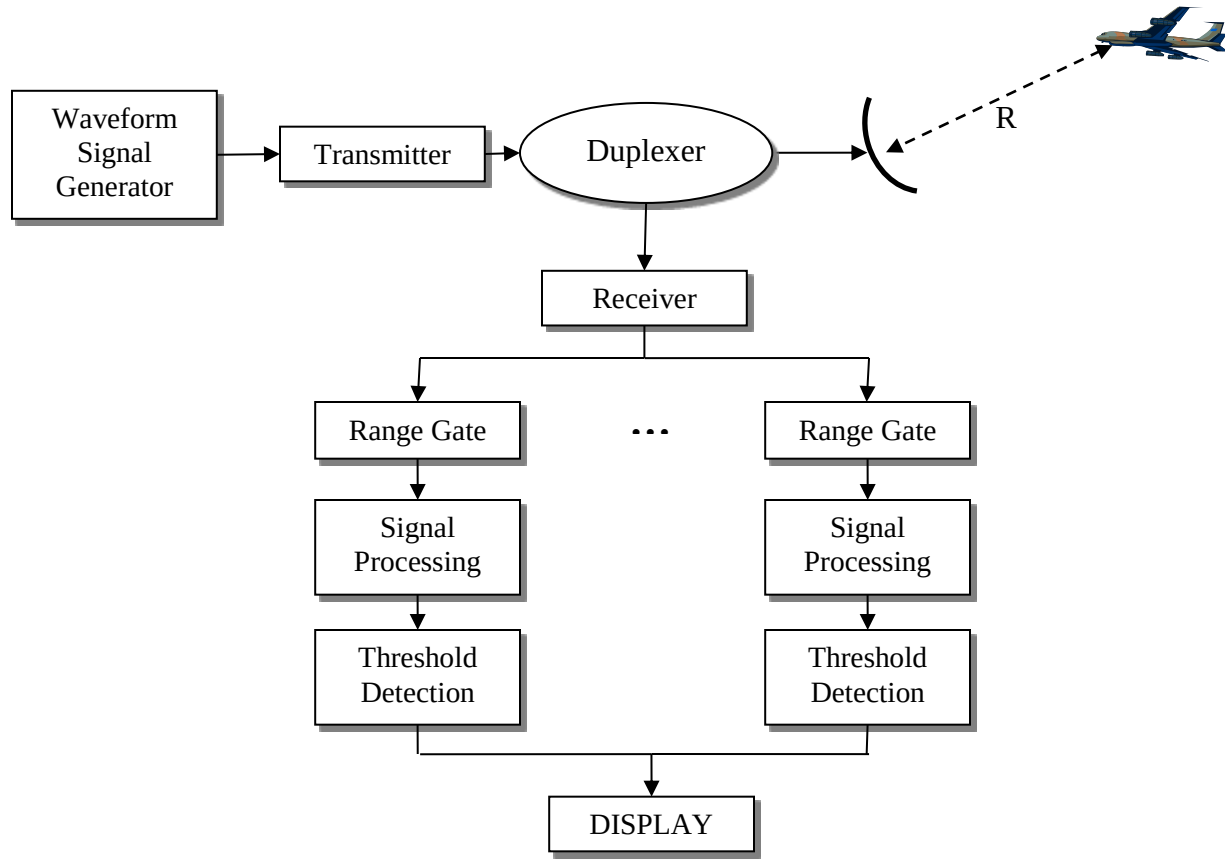


Figure 10.11: Simplified Block Diagram of Monostatic Pulsed Radar System

(Airplane sketch from Microsoft Office Clip art – Used with permission from Microsoft)

Since a single antenna is used to both transmit and receive, the minimum detectable range, $R_{\min}$, for an object given a transmit pulse duration of τ_d would be:

$$R_{\min} = c \cdot \tau_d / 2 \quad (10.13)$$

The maximum range would be determined from the minimum required signal to noise ratio and the antenna and receiver characteristics using the radar equation 10.4. The number of range cells or range bins would simply be $(R_{\max} - R_{\min}) / \Delta R$ where ΔR is the range resolution.

The receiver is a superheterodyne, quadrature receiver which amplifies the weak return signal and demodulates the signal to baseband using a series of mixers, filters, and amplifiers. The range gates basically split up the received signal into sections and each section of the received signal is then processed. Two different processing techniques will be discussed in this section: matched filter or correlation processing and stretch processing. In the case of matched filter processing, the received signal is divided into overlapping sections with time duration, τ_d . Each section corresponds to the return interval from a single range cell or bin. The signal section is processed using a matched filter, and a threshold decision is made based on the magnitude of the

matched filter output at the expected peak to determine whether or not an object is present in the corresponding range cell. Results from each range cell are then used to update the display.

Stretch processing is a specialized technique for dealing with extremely high bandwidth LFM signals where the required sampling rate would be prohibitively high using matched filtering. Initial analog processing converts any LFM chirp signals present in a selected range window to single tone sinusoids with frequencies unique to each range cell or bin within the window. FFT processing is used to determine which tones, if any, are present in the range window, and any frequency bins with FFT magnitudes that exceed a set threshold indicate presence of a target in the corresponding range cell.

We will assume that the transmitted signal is an LFM chirp signal of the form:

$$x(t) = a(t) \sin \left(2\pi F_T t + \pi \frac{\beta}{\tau_d} t^2 \right) \quad (10.14)$$

$a(t)$ is the on/off amplitude modulation
 F_T is the transmit frequency
 τ_d is the pulse burst duration
 β is the frequency sweep range

This signal was described in some detail in the previous section. Assuming there is a target within range and within the beamwidth of the antenna, the transmitted pulse would reflect off the target back to the receiver. This return signal would be considerably weakened (signal energy decreases with the distance traveled) and would be corrupted by noise. The return signal would have the form:

$$r(t) = b(t - t_o) \sin \left(2\pi F_T (t - t_o) + \pi \frac{\beta}{\tau_d} t^2 \right) + n(t) \quad (10.15)$$

$b(t - t_o)$ is the echo amplitude from the target
 t_o is the time delay or travel time of the signal
 $n(t)$ is the noise clutter, jamming, and receiver

Correlation Processing or Matched Filtering

The return signal is processed through the receiver and converted to a baseband, demodulated LFM chirp signal still corrupted with noise. For range determination, the signal processor filters the noisy LFM chirp signal using a matched filter and determines whether or not the peak filter output exceeds a specified threshold value. The threshold value is chosen based on a performance specification for the maximum allowable probability of a false detection or false alarm. If the peak output exceeds the threshold value, then a target is present in the range cell and the target range can be computed based on the peak time, t_{peak} , and the pulse duration, τ_d as follows:

$$R = c \cdot (t_{\text{peak}} - \tau_d) / 2 \quad (10.16)$$

The matched filtering or correlation processing is often implemented using FFTs or fast convolution processing (FCP) as shown in Figure 10.12.

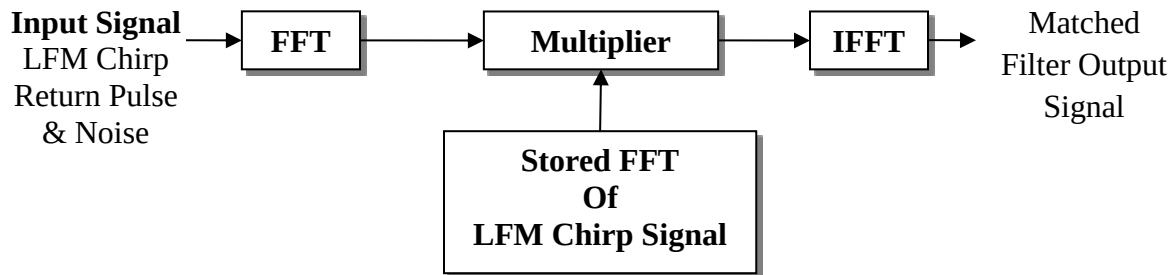


Figure 10.12: Using Fast Convolution Processing to Compute Matched Filter Output

The process for estimating range is illustrated in the following example. However, for simplicity, instead of breaking the receiver signal into sections corresponding to each range cell or bin, the entire signal will be processed.

Example 10.3: Estimating Range to Target

Assume an LFM chirp pulse, transmitted with a pulse duration of $1\mu\text{s}$ and a frequency sweep range of 50 MHz, reflects off an object located within the receive window at a range of 250m. The return pulse is picked up by the antenna, demodulated to a baseband pulse, and then processed through a matched filter using a sampling rate of 250 MHz.

- Assuming that the radar platform and target are both stationary during pulse transmission, calculate the time delay at the receiver for the LFM return signal.
- Using the time delay calculated in part (a) and the characteristics of the LFM chirp signal, plot the pulse signal input to the matched filter and the output of the matched filter. Where does the output peak occur and why does this make sense?
- Add white noise to the return signal and repeat part (b).

Solution

(a) $\text{time\_delay} = 2 \cdot R / c = 2 \cdot 250\text{m} / (3 \cdot 10^8 \text{ m/s}) = 1.6667\mu\text{s}$

(b) The MATLAB m-function file for this part of the example is:

```

function Ex10_3(tau,Beta,Fs,range)

% Usage Ex10_3(tau,Beta,Fs,range)
% This function creates a specified LFM chirp signal
% Calculates and plots the return (received) signal
% based on time delay associated with range,
% Processes the signal through the matched filter
% Plots the filter output
% Inputs: tau=pulse width, Beta=frequency sweep range,

```

```

% Fs = sampling rate, range = distance to object

N = ceil(tau*Fs); %Number of sample points of chirp signal
c = 3e8;
Ts = 1/Fs;
t = 0:Ts:Ts*(N-1);

% Transmitted LFM Chirp signal:
% pulse of duration, tau, sweep frequency range, Beta
x = chirp(t,0,tau,Beta);

% Return signal from object
% calculate time_delay and delay chirp signal
time_delay = 2*range/c;
y = [zeros(1,ceil(time_delay/Ts)) x];
tret = 0:Ts:Ts*(length(y)-1);
figure;
subplot(2,1,1); plot(tret*1e6,y);
title('Return Signal'); grid; xlabel('Time (us)');

% Matched Digital FIR filter to LFM Chirp Signal x(t)
h = ifft(conj(fft(x)));

% Process Received Signal Through Filter
filter_out = conv(h,y);
tout = 0:Ts:Ts*(length(filter_out)-1);
subplot(2,1,2); plot(tout*1e6, filter_out);
title('Output of Matched Filter'); xlabel('Time (us)'); grid

```

The plot shown in Figure 10.13 is created by executing the following MATLAB command:

```
Ex10_3(1e-6, 50e6, 250e6, 250)
```

As shown in Figure 10.13, the output of the matched filter peaks at 2.668 μ s. Using Equation 10.17 to calculate range gives:

$$R = c \cdot \frac{t_{\text{peak}} - \tau_d}{2} = 3 \cdot \frac{10^8 \text{m}}{\text{s}} \cdot \frac{(2.668 - 1) \cdot 10^{-6} \text{s}}{2} = 250.2 \text{ m}$$

In the example, the range to target is 250m. So, is there a problem here? No, because the range accuracy is limited by the sampling interval, T_s . For a 250 MHz sampling rate, the range error can be as high as 0.6m ($3 \cdot 10^8 \cdot T_s/2$); that is, the distance associated with the sampling time

interval, T_s . In other words, the output peak time gets rounded to the nearest sampling instant which will cause a range error.

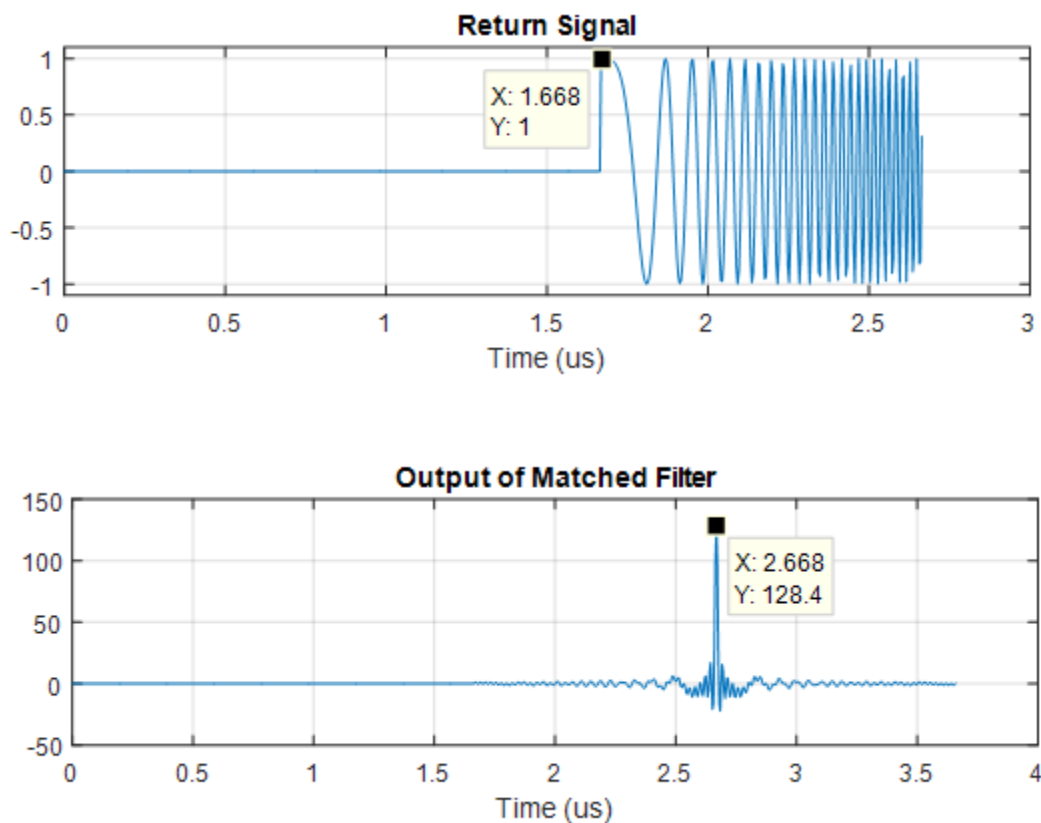


Figure 10.13: Return Signal and Matched Filter Output for Example 10.3(b)

- (c) Noise was added to the chirp signal of part (b) by first modifying a portion of the Ex10_3 m-function file as highlighted below to create a new function Ex10_3c:

```
% Transmitted LFM Chirp signal:
% pulse of duration, tau, sweep frequency range, Beta
x = chirp(t,0,tau,Beta);
xn = x + 2*randn(1,N);

% Return signal from object
% calculate time_delay and delay chirp signal
time_delay = 2*range/c;
y = [zeros(1,ceil(time_delay/Ts)) xn];
```

Running the following MATLAB command produces Figure 10.14:

```
Ex10_3c(1e-6, 50e6, 250e6, 250)
```

The noisy return signal and the resulting matched filter output are shown in Figure 10.14. Although the return signal looks significantly different than the noise-free return signal in Figure 10.13, a distinct peak still appears in the output of the matched filter at $2.668\mu\text{s}$ which means range to target can still be calculated. However, increasing the noise level will eventually bury the output peak. Noise effects will be explored further in the lab exercise at the end of this chapter.

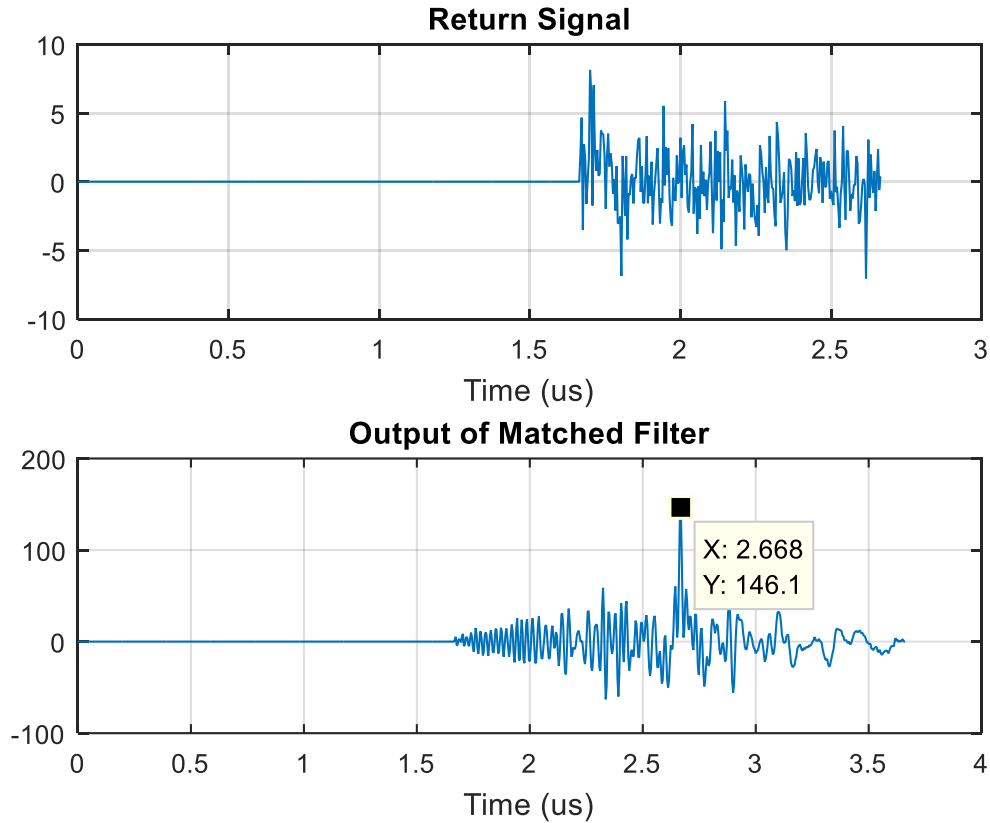


Figure 10.14: Return Signal and Matched Filter Output for Example 10.3(c)

The range resolution, ΔR in meters, using an LFM chirp signal with a frequency sweep range of β (Hz) is given by:

$$\Delta R = \frac{c}{2 \cdot \beta} \quad (10.17)$$

Example 10.4 illustrates range estimation for multiple targets.

Example 10.4: Estimating Range for Multiple Targets

Consider three scatterers located within the receive window at ranges of 200m, 250m, and 260m. Assume an LFM chirp pulse is transmitted with a pulse duration of $1\mu\text{s}$ and a frequency modulation sweep range of 100 MHz. Assume the sampling rate for processing the return signal is 500 MHz. The LFM Chirp Pulse (baseband) is shown in Figure 10.15.

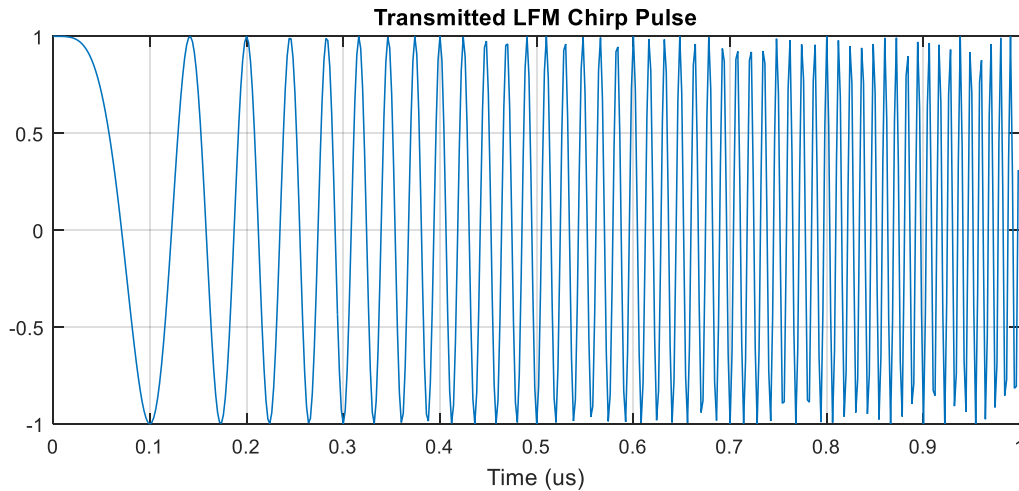


Figure 10.15: LFM Chirp Pulse at Baseband

- Calculate the range resolution.
- Calculate the time delay at the receiver for the return pulses from each of the scatterers.
- Plot each of the return signals and the composite signal (after demodulation, I channel only) assuming that the signal from the scatterers at 200m and 250m return at the same relative strength, but the signal from the scatterer at 260m has half the strength of the other two.
- Process the composite return signal through the matched filter and plot the matched filter output.
- Change the range of the 3rd scatterer to 251m and plot the composite return signal and the matched filter output.

Solution

- The range resolution is: $\Delta R = c/(2 \cdot \beta) = 3 \cdot 10^8 / (2 \cdot 100 \cdot 10^6) = 1.5\text{m}$
- The time delay for each scatterer is: $t_{\text{delay}} = 2 \cdot \text{range}/c$. So, the delay would be $1.333\mu\text{s}$ for the target at 200m, $1.6667\mu\text{s}$ for the target at 250m, and $1.7333\mu\text{s}$ for the target at 260m.
- (c-d) The MATLAB function for completing this example is shown in Figure 10.16.
The command for parts c and d is:

```
Ex10_4(1e-6, 100e6, 500e6, 3, [200 250 260], [1 1 0.5]);
```

```

function Ex10_4(tau,Beta,Fs,nobj,range,RelAttn)

%Usage Ex10_4(tau,Beta,Fs,nobj,range,RelAttn)
% This function creates a specified LFM chirp signal
% Calculates and plots the return (received) signal
% based on range and relative attenuation
% Processes the signal through the matched filter
% Plots the filter output
% Inputs: tau=pulse width, Beta=frequency sweep range,
% Fs = sampling rate, nobj = number of objects,
% range is a vector of the target ranges and
% RelAttn is a vector of the target relative attenuations

Ts = 1/Fs;
N = ceil(tau*Fs); %Number of sample points of input chirp signal
c = 3e8;
t = 0:Ts:Ts*(N-1);

% Transmitted LFM Chirp signal pulse
x = chirp(t,0,tau,Beta);
figure; plot(t*1e6,x);
title('Transmitted LFM Chirp Pulse'); xlabel('Time (us)');grid

% Return signals from scatterers
time_delay = 2*range/c;
y = zeros(nobj,ceil(max(time_delay)/Ts)+N-1);
for m = 1:nobj
    y(m,ceil(time_delay(m)/Ts):ceil(time_delay(m)/Ts)+N-1)=RelAttn(m)*x;
end

% Plot individual return signals
tret = 0:Ts:Ts*(ceil(max(time_delay)/Ts)+N-2);
figure;
for m = 1:nobj
    subplot(nobj,1,m); plot(tret*1e6,y(m,:));
    title('Individual Return Signal');xlabel('Time (us)');grid
end

% Composite return signal
received_signal = ones(1,nobj)*y;
figure; plot(tret*1e6,received_signal);
title('Composite Received Signal'); xlabel('Time (us)');grid

```



```

% Matched Digital FIR filter to LFM Chirp Signal x(t)
h = ifft(conj(fft(x)));

% Process Composite Signal Through Filter
filter_out = conv(h,received_signal);
tout = 0:Ts:Ts*(length(filter_out)-1);
figure; subplot(2,1,1); plot(tout*1e6, filter_out);
title('Output of Matched Filter vs Time');
xlabel('Time (us)');grid

% Change the time scale to distance scale
dist = (tout-tau)*c/2;
subplot(2,1,2); plot(dist, filter_out);
title('Output of Matched Filter vs. Distance');
xlabel('Distance (m)'); grid

```

Figure 10.16: MATLAB function for Example 10.4

The return signal from each scatterer is shown in Figure 10.17. Notice that the time delays match up with those calculated in part (b). Also, noise is being completely ignored in this example so each individual signal matches the original chirp signal.

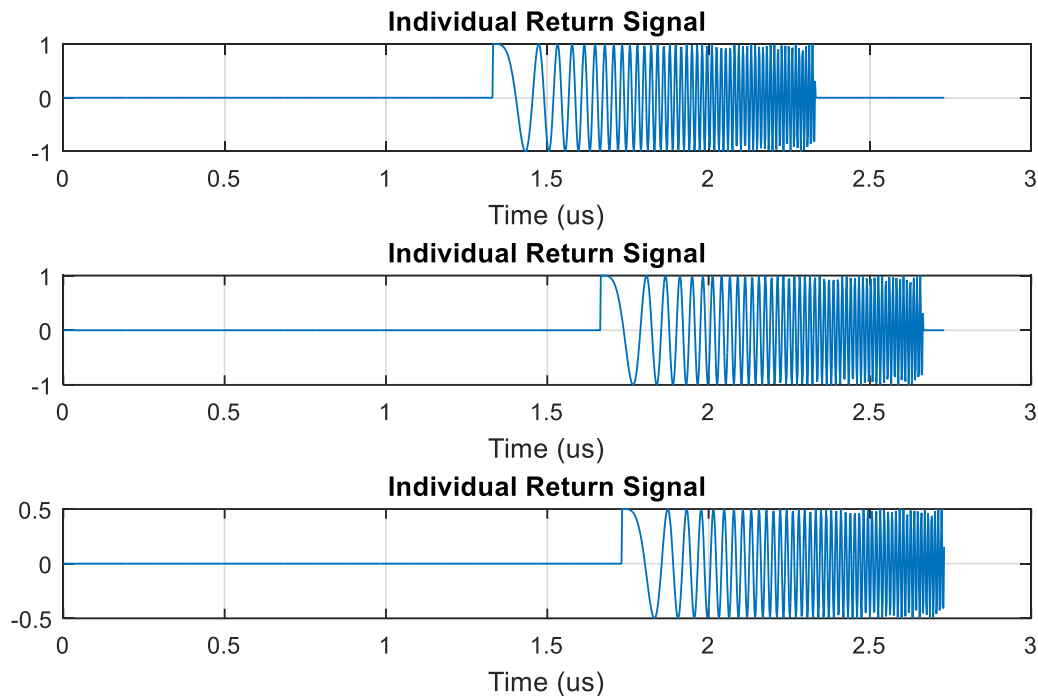


Figure 10.17: Individual Return Signals

The composite signal (I Channel – real part, demodulated to baseband) is the sum of the individual return signals and is shown in Figure 10.18.

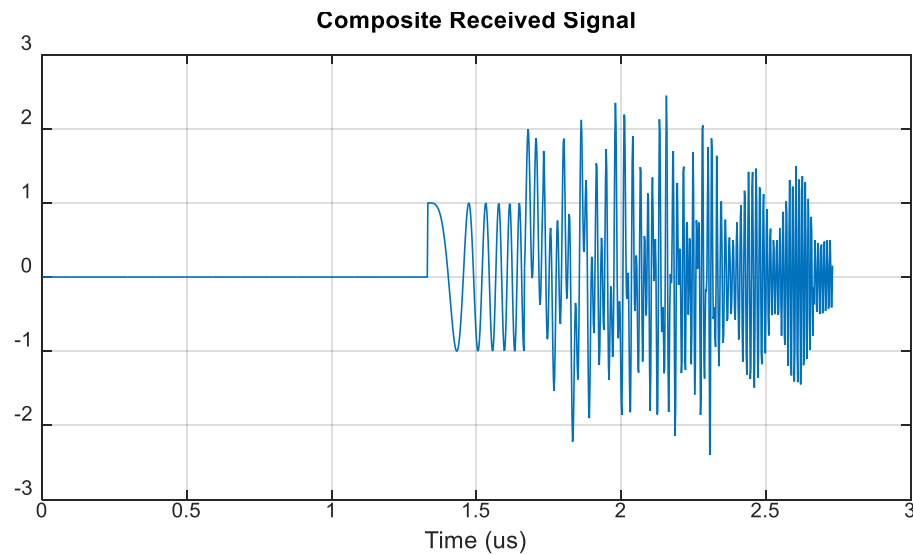


Figure 10.18: Composite Received Signal for Objects at 200, 250, and 260m

Figure 10.19 shows the output of the matched filter. The pulse compression that occurs through the correlation processing on the LFM Chirp pulses makes it possible to distinguish 3 separate targets. The top plot shows the matched filter output vs. time while the lower plot converts the time axis to distance. The three peaks are at the expected ranges of 200, 250, and 260 m.

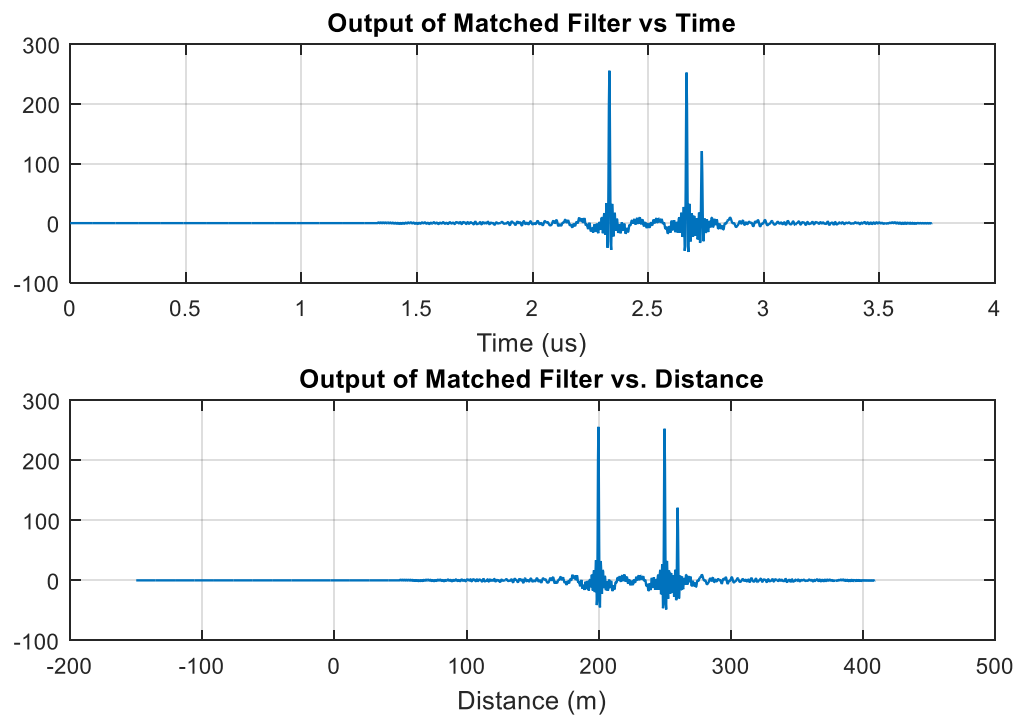


Figure 10.19: Output of Matched Filter for Objects at 200, 250, and 260m

- (e) The same script file shown in Figure 10.16 is used for part e. The only change is the range of the 3rd object so the MATLAB command becomes:

```
Ex10_4(1e-6, 100e6, 500e6, 3, [200 250 251], [1 1 0.5]);
```

The composite signal is shown in Figure 10.20 while the matched filter output is shown in Figure 10.21. Since the range resolution is 1.5m, the objects at 250m and 251m cannot be resolved into two distinct objects but instead appear as a single target.

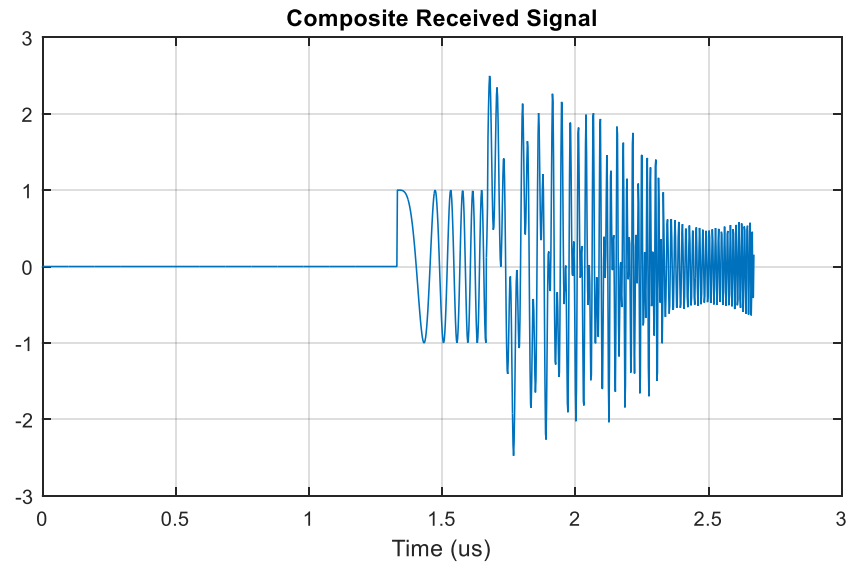


Figure 10.20: Composite Return Signal for Objects at 200, 250, and 251m

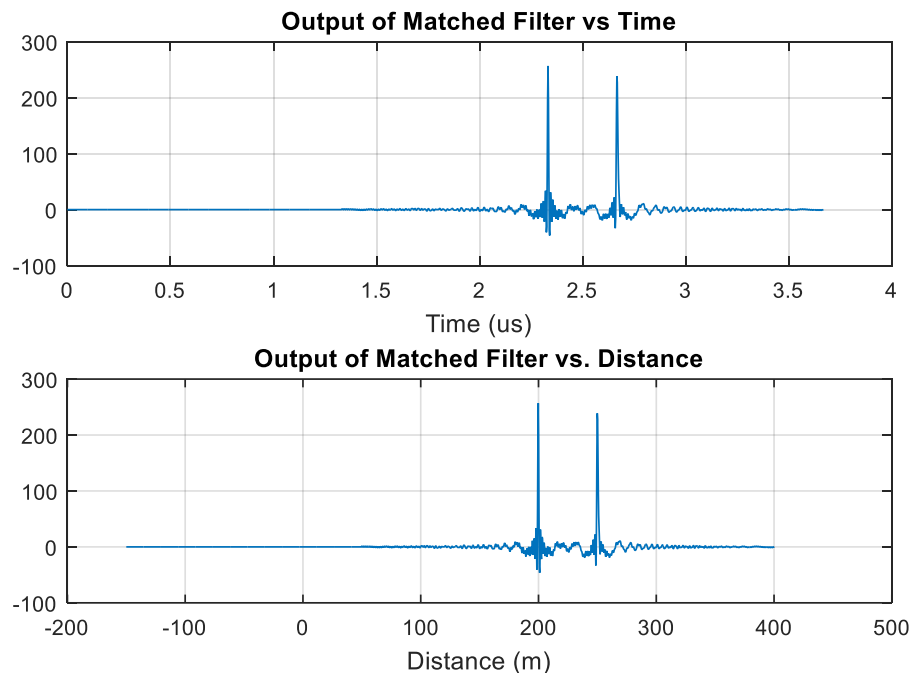


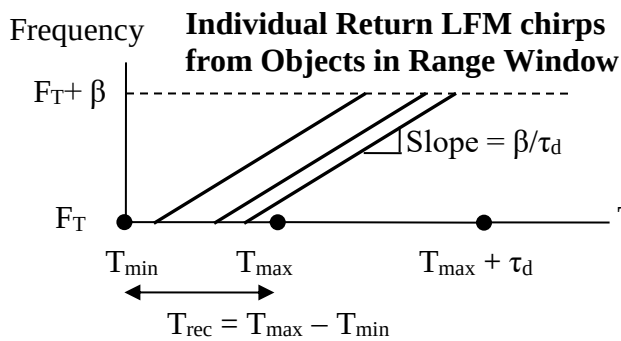
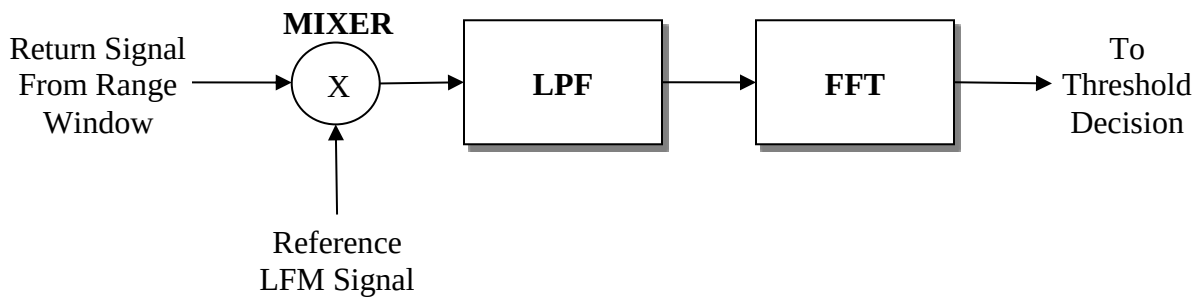
Figure 10.21: Matched Filter Output for Objects at 200, 250, and 251m

Stretch Processing

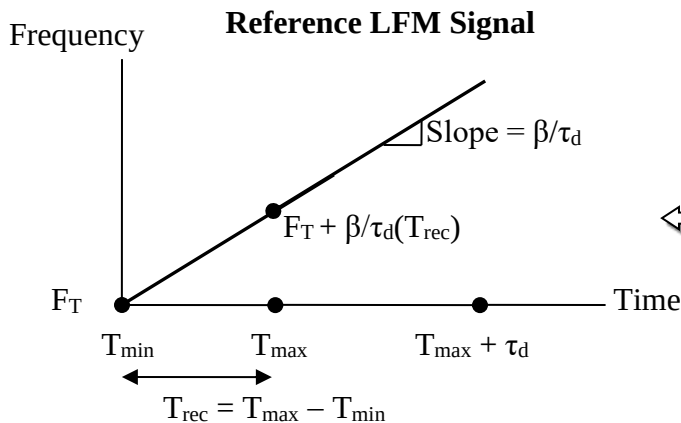
Stretch processing is a specialized technique to deal with exceptionally wideband radar systems where the sample rate for matched filter processing would be prohibitively high. Often, the range window is relatively small but very fine resolution is required within that range window. Since resolution is inversely proportional to the LFM signal bandwidth, β , very fine resolution requires very high bandwidth LFM signals. A simple block diagram for stretch processing is shown in Figure 10.22.

Assume that the range window of interest is $[R_{\min} R_{\max}]$ and the transmitted pulse is an LFM chirp of duration τ_d with a frequency sweep range of β Hz. The shortest possible echo time, $T_{\min}$, is determined from the minimum range and would be equal to $2 \cdot R_{\min}/c$. The longest possible echo time, $T_{\max}$, is determined from the maximum range and would be equal to $2 \cdot R_{\max}/c$. Echo pulses from each object in the range window would arrive at the receiver at times between $T_{\min}$ and $T_{\max}$ and would last for τ_d seconds. The first graph in Figure 10.22 shows return pulses from 3 objects in the range window. The signal at the receiver would be the sum of the three echo pulses plus noise.

As shown in Figure 10.22, the received signal is mixed with an LFM chirp reference signal with the same sweep rate, β/τ_d , as the return pulses. The output of the mixer is the sum and difference frequencies of the two incoming signals. The low-pass filter passes only the difference frequencies. Consider for a moment a single echo pulse arriving at the receiver at some time t_{echo} between $T_{\min}$ and $T_{\max}$. Since this LFM pulse has exactly the same slope as the reference LFM signal, the difference between the two signals will be a constant tone with frequency $(\beta/\tau_d) \cdot (t_{\text{echo}} - T_{\min})$. Objects with ranges close to $R_{\min}$ would produce frequencies close to zero while objects with ranges close to $R_{\max}$ would produce frequencies close to $(\beta/\tau_d) \cdot (T_{\max} - T_{\min})$. Three objects in the range window would produce three different constant tone sinusoids as shown in the graph of the LPF output frequencies in Figure 10.22.



This graph illustrates the return LFM chirps from 3 objects in the range window. Return times vary from $T_{\min}$ to $T_{\max}$ depending on the relative location of the objects in the range window. The signal into the receiver would be the sum of the three returns (plus noise)



The receiver generates an LFM chirp signal with the same sweep rate as the transmitted LFM chirp signal. The sweep begins at $T_{\min}$ (shortest echo time for the given range window) and ends at $T_{\max} + \tau_d$ (longest echo time for given range window plus the LFM pulse duration).

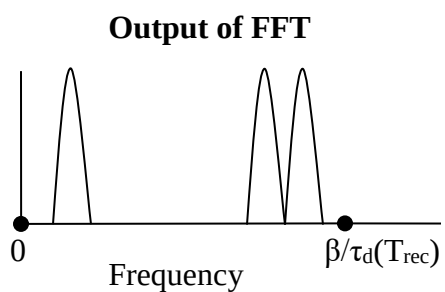
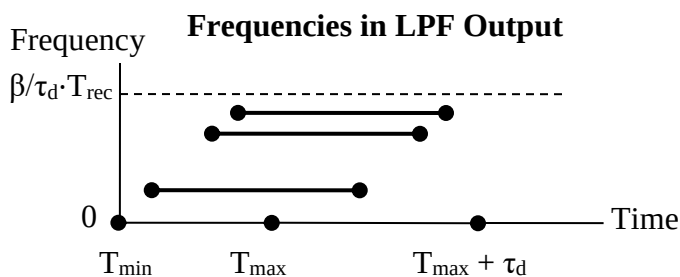


Figure 10.22: Stretch Processing

Performing an FFT on the LPF output identifies the constant tone frequencies in the LPF output signal. A sample FFT output for three objects in the range window is shown in Figure 10.22. Each frequency bin of the FFT represents one range cell in the range window. If the FFT magnitude response exceeds some specified threshold in a particular frequency bin, then an object is presumed present in that particular range cell.

Recall that the range resolution, ΔR , using an LFM chirp signal is $c/(2\beta)$ meters. The difference in return time between objects in two adjacent range cells would be:

$$\Delta t_{\text{adj}} = 2 \cdot \frac{\Delta R}{c} = \frac{1}{\beta} \text{ (seconds)} \quad (10.18)$$

The difference between the constant tone frequencies from objects in two adjacent range cells, which is also the minimum frequency resolution required for the FFT, is given by:

$$f_0 \leq \frac{\beta}{\tau_d} \cdot \Delta t_{\text{adj}} = 1/\tau_d \quad (10.19)$$

The maximum frequency tone in the LPF output signal is:

$$f_{\text{max}} = \frac{\beta}{\tau_d} \cdot (T_{\text{max}} - T_{\text{min}}) \quad (10.20)$$

The minimum sampling frequency, F_s , for the FFT is twice the bandwidth of the LPF output signal:

$$F_s \geq 2 \cdot f_{\text{max}} = 2\beta \cdot (T_{\text{max}} - T_{\text{min}})/\tau_d \quad (10.21)$$

The size of the FFT is:

$$N = F_s/f_0 \quad (10.22)$$

For computational efficiency for the FFT, the value for N computed in equation 10.22 should be rounded up to the nearest power of 2.

If the time interval for the range window of interest, $T_{\text{max}} - T_{\text{min}}$, is smaller than the pulse duration, τ_d , then the required sampling rate for the LFM chirp signal falls below the Nyquist rate of 2β as shown in Equation 10.21. For matched filter processing, the LFM chirp signal must be sampled well above the Nyquist rate. So, for relatively small range window receive times (as compared to the pulse duration) and very fine resolution requirements, stretch processing offers an alternative to the prohibitively high sampling rates required for matched filter processing. The following two examples illustrate some of the principles of stretch processing.

Example 10.5: Estimating Range to Multiple Targets using Stretch Processing

Assume a range window of 100m, LFM chirp pulse duration of 10 μ s, sweep frequency range of 500 MHz, and a transmission frequency of 1 GHz.

- (a) Calculate the range resolution, ΔR .
- (b) Calculate the smallest allowable value for $R_{\min}$ in the 100m range window.
- (c) Calculate the receive window time, $T_{\text{rec}} = T_{\text{max}} - T_{\text{min}}$.
- (d) Plot the mixer and LPF outputs for an object at a relative range of 3m within the range window of interest.
- (e) Plot the mixer and LPF outputs for an object at a relative range of 50m within the range window of interest.

Solution

(a) The range resolution $\Delta R = c/(2 \cdot \beta) = 3 \cdot 10^8/(2 \cdot 500 \cdot 10^6) = 0.3\text{m}$.

(b) The smallest allowable value for $R_{\min}$ in the 100m range window is:

$$R_{\min} \geq c \cdot \frac{T_d}{2} = (3 \cdot 10^8 \text{ m/s}) \cdot 10 \cdot 10^{-6} \text{ s} / 2 = 1500 \text{ m}.$$

So, some valid 100m range windows would be [1500m 1600m] or [1575m 1675m] or [2035m 2135m]. The maximum allowable R_{max} in the range window would be determined from the radar equation (10.4) and the minimum required SNR at the receiver.

(c) The receive time, T_{rec} , is simply the time difference between an echo return from an object at the furthest point in the range window and an echo return from an object at the closest point in the range window. So, $T_{\text{rec}} = 2 \cdot 100\text{m}/(3 \cdot 10^8 \text{ m/s}) = .667\mu\text{s}$.

An m-function, Ex10_5, created for parts (d) and (e) is shown in Figure 10.23.

```
function LPF_output = Ex10_5(relative_range)

% Usage:  mixer_out = Ex10_5(relative_range)
% Takes the range relative to Rmin within a 100m window
% Plots the output of the mixer and LPF using the concept of
% stretch processing.
% Relative_range is Range - Rmin and must be between 0 and 100m
% For this example, pulse length = 10us, frequency sweep = 500
% MHz, and transmission frequency is 1 GHz.

tau=10e-6; Beta = 500e6; Ft = 1e9; c=3e8; Fa = 10*Ft; Ta = 1/Fa;
range_window = 100; trec = 2*range_window/c;

% Create Reference LFM Pulse for Receiver
t=0:Ta:tau+trec; reference=chirp(t,Ft,tau,Ft+Beta);

% Create Transmitted LFM pulse
tret=0:Ta:tau; LFMpulse = chirp(tret,Ft,tau,Ft+Beta);

% Calculate Time Delay and Create return pulse
time_delay = 2*(relative_range)/c;
```

```

delay_intervals = ceil(time_delay/Ta);
return_signal = zeros(1,length(reference));return_signal =
zeros(1,length(reference));
return_signal(delay_intervals:delay_intervals+length(LFMpulse)-
1) = LFMpulse;

% Compute mixer output: product of return LFM and Reference LFM
mixer_out = return_signal.*reference;

% Create LPF with cutoff at twice highest frequency out of mixer
% and filter the mixer output
wn = 2*pi*2*Beta*trec/tau; [b,a]=butter(8,wn,'s');
LPF_tf = tf(b,a); LPF_output = lsim(LPF_tf,mixer_out,t);
subplot(2,1,1); plot(t*1e6,mixer_out);
title('Mixer Output');xlabel('Time(us)');grid
subplot(2,1,2);plot(t*1e6,LPF_output);
title('LPF Output');xlabel('Time(us)');grid
end

```

Figure 10.23: m-function file for Example 10.5 (d), (e)

The low-pass filter design used in the m-function is a very simple analog 8th order Butterworth filter that simply needs to pass all expected difference frequencies within the given range window and reject the sum frequencies out of the mixer. The largest possible difference frequency for the given range window is $\beta/\tau_d \cdot T_{\text{rec}} = 500 \text{ MHz}/(10\mu\text{s}) \cdot 0.667\mu\text{s} = 33.35 \text{ MHz}$. This would correspond to an object located at the maximum end of the range window. The sum frequencies out of the mixer all exceed $2 \cdot F_T$ or 2 GHz for this example. So, the LPF must pass frequencies up to 33.35 MHz and reject frequencies above 2 GHz. In the m-function Ex10_5, the cutoff for the Butterworth filter is chosen (rather arbitrarily) to be twice the required passband or $2 \cdot \beta/\tau_d \cdot T_{\text{rec}} = 66.7 \text{ MHz}$.

(d) Using the m-function file shown in Figure 10.23 and executing the command:

```
Ex10_5(3);
```

produces the mixer output and LPF output shown in Figure 10.24.

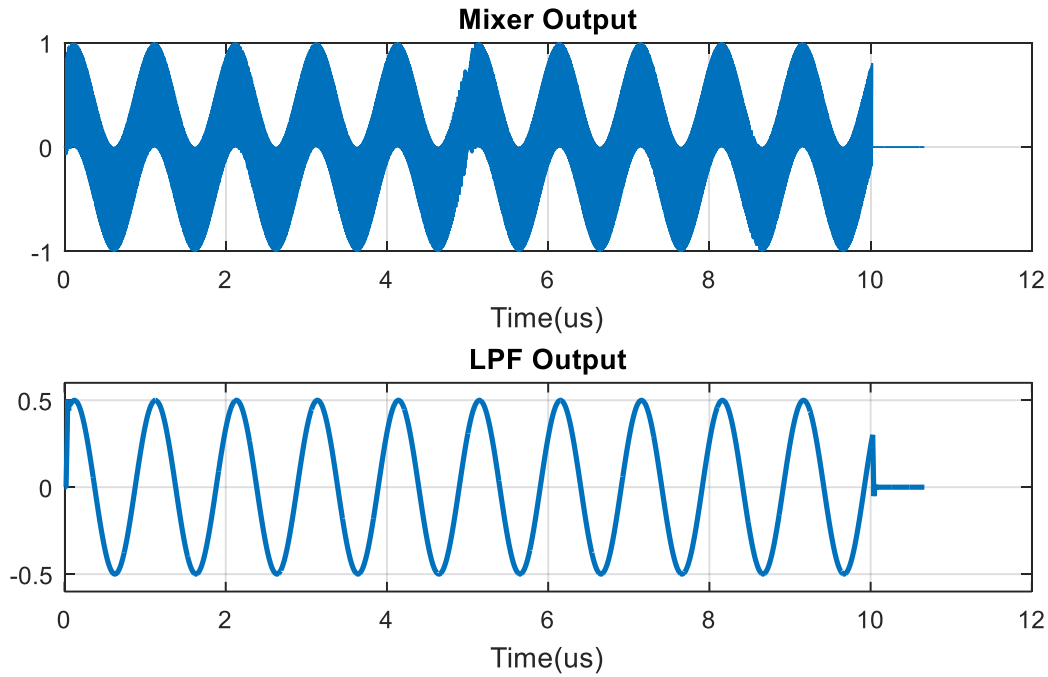


Figure 10.24: Mixer and LPF Output Signals for Relative Range = 3m

The mixer output signal is the product of the reference LFM signal and the return LFM signal and consists of the sum and difference frequencies of the two signals. For a relative range of 3m, the difference frequency should be

$$\frac{\beta}{\tau_d} \cdot (t_{\text{echo}} - t_{\text{min}}) = \frac{\beta}{\tau_d} \cdot \left(2 \cdot \frac{\text{relrange}}{c} \right) = \frac{500 \cdot 10^6 \text{ Hz}}{10 \cdot 10^{-6} \text{ s}} \cdot \left(2 \cdot \frac{3\text{m}}{3 \cdot 10^8 \text{ m/s}} \right) = 1 \text{ MHz}$$

The low-pass filter passes only the difference frequency from the mixer output. As shown in Figure 10.24, the LPF output is indeed a constant tone at a frequency of 1 MHz as expected.

- (e) Using the m-function file shown in Figure 10.23 and executing the command: `Ex10_5(50);` in the MATLAB command window produces the plots shown in Figure 10.25.

Note: the zoom-in feature was used on the plots in Figure 10.25.

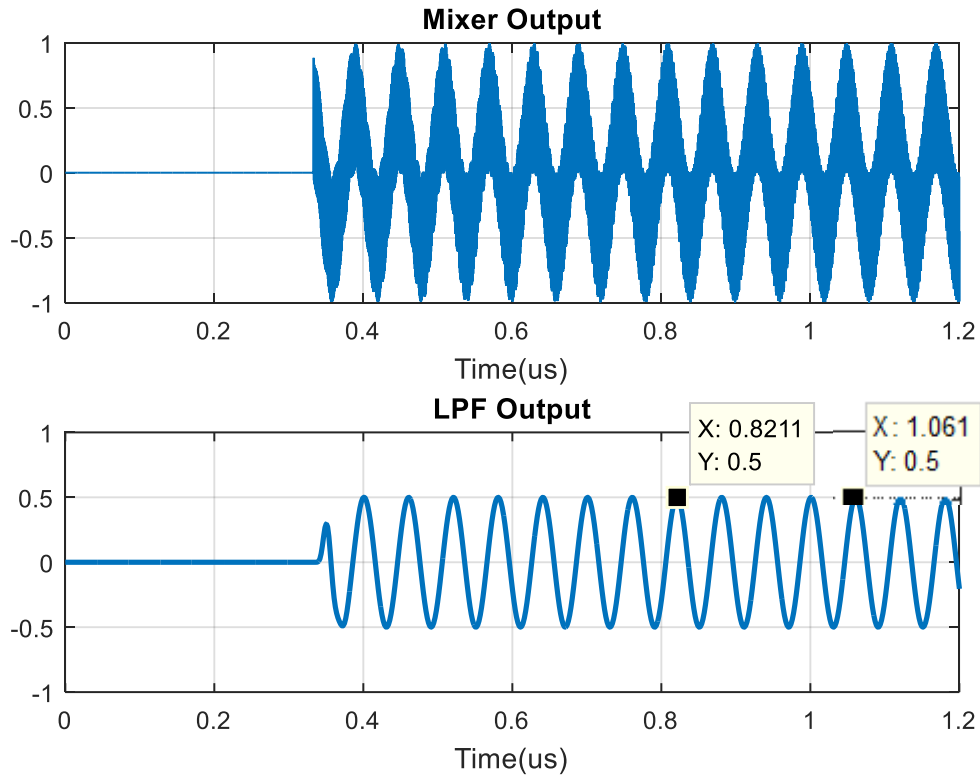


Figure 10.25: Mixer and LPF Output Signals for Relative Range = 50m

Again, the mixer output signal is the product of the reference LFM signal and the return LFM signal and consists of the sum and difference frequencies of the two signals. For a relative range of 50m, the difference frequency should be:

$$\frac{\beta}{\tau_d} \cdot (t_{\text{echo}} - t_{\text{min}}) = \frac{\beta}{\tau_d} \cdot \left(2 \cdot \frac{\text{relrange}}{c} \right) = \frac{500 \cdot 10^6 \text{ Hz}}{10 \cdot 10^{-6} \text{ s}} \cdot \left(2 \cdot \frac{50 \text{ m}}{3 \cdot 10^8 \text{ m/s}} \right) = 16.67 \text{ MHz}$$

As shown in Figure 10.25, the LPF output is indeed a constant tone at a frequency of 16.67 MHz as expected. The expected relative echo time is $(t_{\text{echo}} - t_{\text{min}}) = 2 \cdot \text{relrange}/c = 0.33\mu\text{s}$. The mixer output signal and LPF output signal both begin at this time. Note that for part d, the relative echo time of 20ns is simply too short to see in Figure 10.24 without zooming in significantly.

Example 10.5 illustrates the analog portion of stretch processing and ends with the LPF analog output. Example 10.6 illustrates the digital portion of stretch processing by sampling the LPF analog output and computing an FFT to determine the constant tones present in the LPF output signal. These constant tones easily translate back to relative ranges within the given range window.

Example 10.6: Estimating Range to Multiple Targets using Stretch Processing

Assume all the same parameters for the range window and LFM chirp pulse as described in Example 10.5.

- (a) Determine the sampling frequency, F_s , number of FFT points, N , and frequency resolution, f_o , required to perform an N -pt FFT on the analog LPF output.
- (b) Assuming 3 objects within the range window at relative ranges of 3m, 10m, and 50m, perform an N -pt FFT on the “analog” LPF output signal and plot the results.

Solution

- (a) As indicated in equation (10.21), the minimum sampling frequency, F_s , would be twice the highest expected difference frequency or equivalently the bandwidth of the LPF output signal:

$$F_s \geq 2 \cdot f_{\max} = 2\beta \cdot \frac{T_{\max} - T_{\min}}{\tau_d} = 2 \cdot 500\text{MHz} \cdot \frac{0.667\mu\text{s}}{10\mu\text{s}} = 66.7 \text{ MHz}$$

In a real radar system, the LPF output would be an analog signal. However, in this MATLAB simulation, the LPF output generated by the m-function Ex10_5 is already sampled at $f_a = 10\text{GHz}$. So, for the sake of simplicity, the sampling frequency, F_s , needs to be chosen so that f_a/F_s is an integer. Therefore, the sampling frequency will be set at 100 MHz.

As indicated in Equation (10.19), the minimum frequency resolution, f_o , is

$$f_o \leq \frac{1}{\tau_d} = \frac{1}{10\mu\text{s}} = 0.1 \text{ MHz}$$

The minimum number of points for the FFT would be determined as

$$N \geq \frac{F_s}{f_o} = \frac{100 \text{ MHz}}{0.1 \text{ MHz}} = 1000$$

For computational efficiency, N should be rounded up to a power of 2. So, a 1024-pt FFT will be performed on the LPF output sampled at 100 MHz resulting in a frequency resolution of 0.0977 MHz.

- (b) An m-function, Ex10_6.m, created for part (b) is shown in Figure 10.26. Input arguments for Ex10_6 are the number of FFT points, the sampling frequency, the number of objects in the given range window, and a vector of the relative ranges of the objects within the range window. For each of the objects, the function Ex10_6.m calls the function Ex10_5.m used in the previous example and computes the LPF output signal. The LPF output signals are added together within Ex10_6.m to form the composite LPF “analog” signal. This signal is then sampled at the specified sampling frequency, F_s , and an N -point FFT is computed.

Executing the MATLAB command, `Ex10_6(1024,100e6,3,[3 10 50]);` in the command window launches the process. The resulting output is shown in Figure 10.27. The relative ranges of the three objects are as expected. The m-file could easily be modified to allow windowing of the LPF output signal prior to computing the FFT. Also, noise could be added to the LPF signal using ***randn*** as was done in Example 10.3.

```
function Ex10_6(N,Fs,nobj,relative_range)

% Usage Ex10_6(N,Fs,nobj,relative_range)
% Plots the N-pt FFT of the LPF output from the stretch
% processor LPF output
% Fs is the sampling frequency for the FFT
% Fa/Fs = 10e9/Fs must be an integer.
% nobj is the number of objects in the range window and
% relative_range is a vector with the ranges of the objects
% relative to Rmin.
% This function calls Ex11_5 and uses all the parameters
% associated with Ex11_5.

tau = 10e-6; Beta = 500e6; c = 3e8; Ft=1e9; Fa=10*Ft;

for m = 1:nobj
    x(m,:) = Ex10_5(relative_range(m));
% Mixer output from each object
end

LPF_out = ones(1,nobj)*x;
% Composite LPF output "analog" signal

remainder = mod(Fa,Fs); % Make sure Fs is a valid choice
if remainder ~= 0
    error('Fa/Fs = 10e9/Fs must be an integer')
end

% Sample "analog" LPF output signal at a rate of Fs
R = ceil(Fa/Fs);
LPF_sampled = zeros(1,length(LP_out)/R);
for m = 1:length(LP_out)/R;
    LPF_sampled(m) = LPF_out(R*(m-1)+1);
end
LPF = LPF_sampled(1:N);

% Compute the FFT and Plot
FFT_output = fftshift(fft(LP));
fo = Fs/N; freq_bins = -(N/2)*fo:fo:(N/2-1)*fo;
```

```

subplot(2,1,1);
plot(freq_bins(N/2+1:N)*1e-
6,20*log10(abs(FFT_output(N/2+1:N))));

title('FFT vs Freq'); xlabel('Frequency (MHz)');grid
ylabel('Magnitude (dB)')

range_bins = freq_bins*tau/Beta*c/2;
subplot(2,1,2); plot(range_bins(N/2+1:N),
20*log10(abs(FFT_output(N/2+1:N))));
title('FFT vs Relative Range'); xlabel('Relative Range
(m)');grid;
ylabel('Magnitude (dB)')
end

```

Figure 10.26: m-function file for Example 10.6(b)

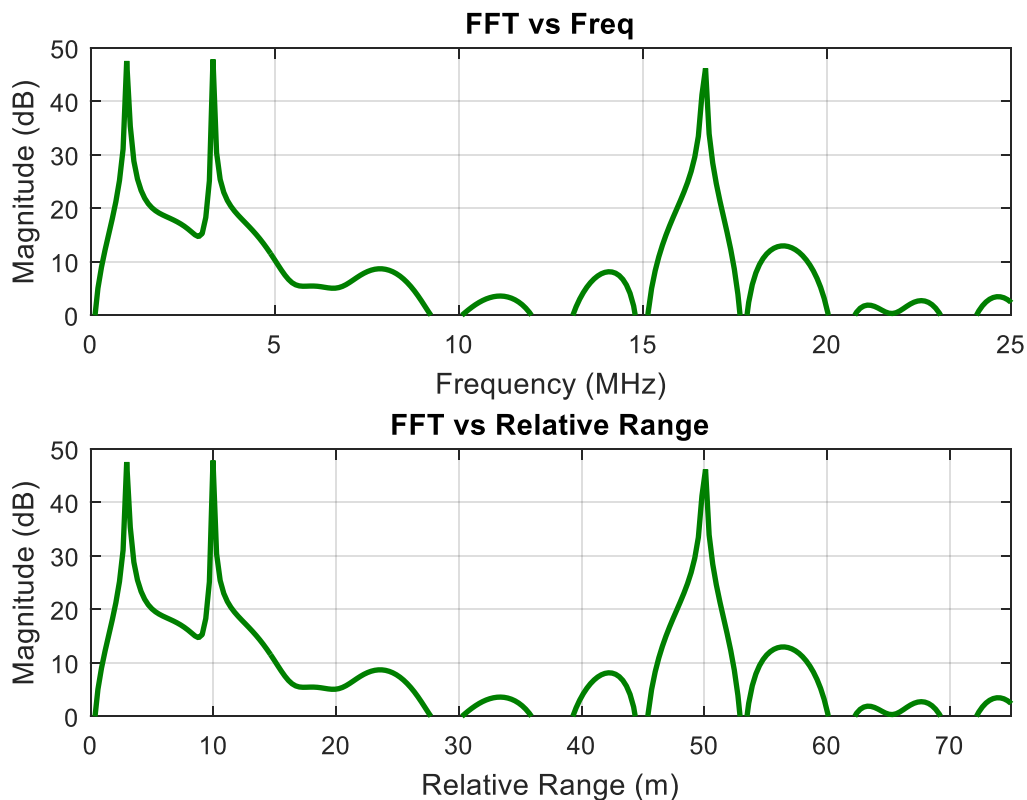


Figure 10.27: FFT Output for Example 10.6(b)

This section is designed to provide an introduction to radar range processing using correlation processing (through matched filtering) or stretch processing. For the interested reader, there are several excellent textbooks offering extensive coverage of radar signal processing. Some of these texts are listed in the references at the end of this text. There are some issues and

considerations that merit mention even though they cannot be covered in sufficient detail in a single introductory chapter.

Signal Conditioning

As mentioned previously, signal to interference ratios are very low for radar systems. Adaptive noise cancellation techniques, similar to those discussed in Chapter 8, can be used prior to processing to increase the signal to noise ratio prior to correlation processing.

Range-Doppler Coupling (Effect of Target Radial Velocity)

The matched filter design and range calculations in this section assumed that both the radar platform and the target were stationary during the pulse/transmission receive time. A moving target changes the expected matched filter output to an LFM pulse; the peak time will shift, the peak output level will be decreased, and the main peak will widen. The shift in peak time results in an error in range estimation proportional to the Doppler shift, F_D , as follows:

$$R_{\text{error}} = -\frac{c \cdot \tau_d \cdot F_D}{2 \cdot \beta} \quad (10.23)$$

The output peak will be reduced by a factor of $(1 - F_D/\beta)$. For a simple (single frequency) pulse waveform, the reduction in output peak is significant, causing targets with Doppler shifts greater than $1/\tau_d$ to remain undetected. For LFM pulses, the output peak remains high enough to allow target detection for a wide range of Doppler shifts making the LFM pulse significantly more *Doppler tolerant* than a simple pulse.

For a moving target with a Doppler shift of F_D , the ideal matched filter would be:

$$H(f) = X^*(f - F_D) \quad (10.24)$$

One possible solution to this problem is to estimate the Doppler shift or equivalently, the radial velocity of the target being tracked first, then adjust the matched filter design accordingly. Another possible solution is to design a bank of matched filters with a variety of expected Doppler shifts and search for an output peak from the bank of filter outputs.

Target Masking (Range Sidelobe Control)

Depending on the position and the radar cross section (RCS) of objects in the range space, there can be a great deal of variability in the signal return strengths from the various objects. Even if two objects are spaced further apart than the range resolution, a weak target return signal can get buried or masked in the filter sidelobe response from a stronger return signal. For radar systems with moderate to high time-bandwidth products, the side lobe will be about 13dB below the output peak. In Chapter 5, windowing was discussed as an effective method for reducing sidelobes in a filter design. A Hamming window can be applied to the transfer function of the matched filter to create a windowed filter with reduced sidelobe gain; that is, $H_w(f) = W(f) \cdot X^*(f)$.

Since this windowed filter will no longer be a matched filter to the transmitted pulse signal, the output peak of the return signal will be reduced. However, the sidelobe level will also be reduced possibly allowing the weaker target to be detected in the presence of the stronger target.

Another effective method for reducing target masking is to use a non-linear FM (NLFM) pulse instead of an LFM pulse. Like windowing, this will reduce the sidelobes of the matched filter response but unlike windowing, the matched filter will remain truly matched to the transmitted NLFM pulse.

10.5 PULSE DOPPLER PROCESSING TO ESTIMATE TARGET VELOCITY

When a target is moving at a different velocity than the radar platform, the received frequency of the echo signal from that target will differ from the transmitted frequency of the pulsed radar waveform as follows:

$$F_r = \frac{(1+\frac{v}{c})}{(1-\frac{v}{c})} F_T \quad (10.25)$$

F_r = received frequency

F_T = transmit frequency

v = relative velocity of target **toward** receiver

If the target is moving toward the radar platform, the received frequency will be higher than the transmitted frequency. If the target is moving away from the receiver, the relative velocity toward the receiver will be negative and the received frequency will be lower than the transmitted frequency. This phenomenon is of course commonly known as Doppler shift. Equation 10.25 can be re-written by using a binomial series on $(1-v/c)^{-1}$ as follows:

$$\begin{aligned} F_r &= \left[1 + \frac{v}{c}\right] \cdot \left[1 - \frac{v}{c}\right]^{-1} \cdot F_T = \left[1 + \frac{v}{c}\right] \cdot \left[1 + \frac{v}{c} + \left(\frac{v}{c}\right)^2 + \left(\frac{v}{c}\right)^3 + \dots\right] F_T \\ &= \left[1 + \frac{2v}{c} + 2\left(\frac{v}{c}\right)^2 + 2\left(\frac{v}{c}\right)^3 + \dots\right] F_T \end{aligned} \quad (10.26)$$

Since the relative radial velocity of the target is significantly smaller than the velocity of the transmitted pulsed radar signal (i.e. $v \ll c$), the received frequency can be approximated as:

$$F_r \approx \left[1 + \frac{2v}{c}\right] F_T \quad (10.27)$$

The Doppler shift, F_D , can therefore be approximated as:

$$F_D \approx \left(\frac{2v}{c}\right) F_T \quad (10.28)$$

If the target is moving toward the radar platform, the Doppler shift will be positive. A target moving away from the radar platform produces a negative Doppler shift. If the Doppler shift can be measured, equation 10.28 can be used to estimate the target radial velocity relative to the radar platform. Table 10.1 shows the range of Doppler shift for an approaching target traveling at a relative radial velocity of 100 m/s for some of the common nominal radar bands.

Table 10.1: Doppler Shift for Target Approaching at 100 m/s

Band	Frequency Range (GHz)	Doppler Shift (kHz)
L	1 – 2	0.667 – 1.33
S	2 – 4	1.33 – 2.67
C	4 – 8	2.67 – 5.33
X	8 – 12	5.33 – 8.00
Ku	12 – 18	8.00 – 12.0
K	18 – 27	12.0 – 18.0
Ka	27 – 40	18.0 – 26.7
mm	40 - 300	26.7 - 200

For range estimation, a single LFM chirp pulse echo was processed through the receiver using matched filtering or stretch processing. A single pulse does not work for Doppler processing because the frequency resolution associated with a single pulse is not sufficiently fine to pick up the expected Doppler shifts. As mentioned in the first section, a typical LFM chirp pulse has a duration of 100 ns to 100 μ s. The frequency resolution of a Discrete Fourier Transform (DFT) is the inverse of the duration of time over which a signal is sampled and would therefore range from 10 kHz to 10 MHz for pulse durations of 100 μ s to 100 ns. Comparing these resolution values to the Doppler shift values in Table 10.1 should convince the reader that a single pulse will not suffice. Transmitting a pulsed radar signal consisting of several LFM pulses will extend the duration of time over which the echo pulsed signal is sampled resulting in much finer frequency resolution for the DFT. A simple block diagram for Doppler pulse processing to estimate velocity is shown in Figure 10.28.

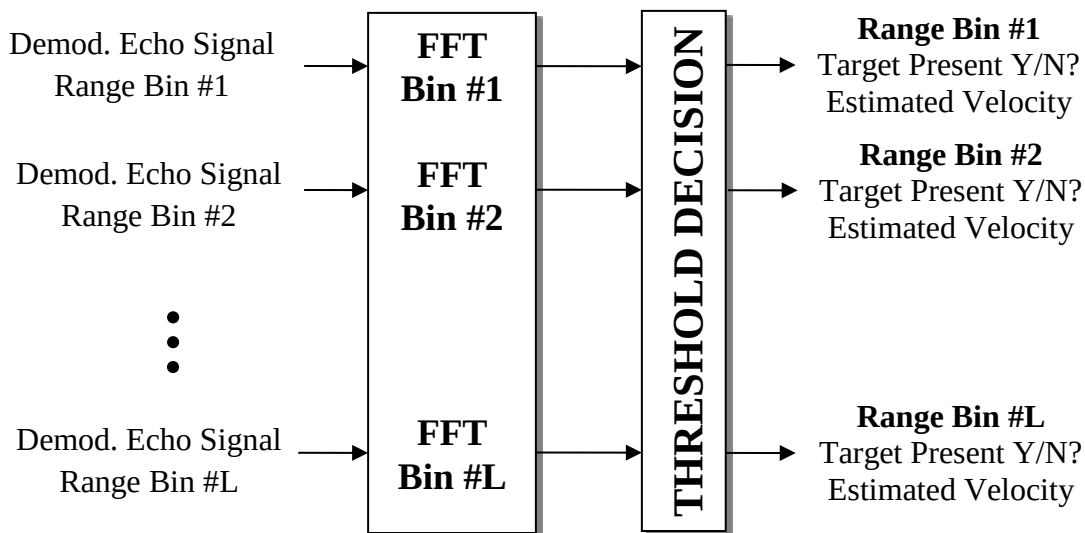


Figure 10.28: Block Diagram for Doppler Pulse Processing

The transmitted signal is a pulsed radar signal consisting of M pulses (LFM chirp) with a duty cycle less than 1% and a pulse repetition frequency (PRF) of several hundred to several thousand pulses per second. Assuming that a target is moving relative to the radar platform, the echo signal from that target would also be a pulsed radar signal with the same PRF; however the chirp pulse frequencies would differ from the transmitted chirp pulse frequencies by the Doppler shift, F_D . Of course, the return signal would also be corrupted by noise and weakened in strength. The receiver demodulates the return signal from each range bin, and if a target is present in the range bin, produces the beat frequency (difference) between the return and transmit chirps:

$$y_{\text{demod.}}(k) = Ae^{j2\pi F_D kT} + \text{noise} \quad k = 0, 1, \dots, M \quad (10.29)$$

$$T = \text{PRI} = 1/\text{PRF}$$

$$M = \text{number of pulses transmitted}$$

Equation 10.29 describes mathematically what the demodulated echo signal from a range bin with a target moving relative to the platform would look like. As indicated in Figure 10.28, a FFT of this demodulated echo signal from each range bin is computed in order to estimate the Doppler shift frequency, F_D . If the magnitude of the DFT exceeds a specified threshold in one of the discrete frequency bins, then a target is assumed present and the target velocity can be estimated from the frequency associated with that frequency bin. Note also that multiple targets within a range bin can be detected as long as they can be sufficiently resolved into separate frequency bins by the DFT.

Since the signal is sampled at the pulse repetition frequency, the spectrum of the DFT will range from $-\text{PRF}/2$ to $+\text{PRF}/2$. Doppler shifts outside this range will alias resulting in errors in estimating target velocity if aliasing is not accounted for. The dwell time, or total duration of time over which the signal is sampled, is $D = M \cdot T$ so the frequency bins of the DFT are separated by:

$$f_0 = \frac{1}{M \cdot T} = \frac{\text{PRF}}{M} \quad (10.30)$$

f_0 = frequency bin resolution (Hz)

M = number of LFM chirp pulses

T = PRI = pulse repetition interval

PRF = pulse repetition frequency

If the Doppler shift is exactly equal to one of the DFT bin frequencies, the peak value of the DFT at the matching bin frequency will be maximized. If the Doppler shift falls between two bin frequencies, the DFT magnitude will drop. This reduction in amplitude is referred to as Doppler *straddle loss*, and the greatest drop in amplitude would occur when the Doppler shift falls exactly in the middle between two frequency bins.

Figure 10.29 shows a typical Doppler spectrum. The center of the spectrum is dominated by clutter making targets in this region impossible to detect using FFT processing. There are advanced processing techniques that use clutter mapping to attempt to detect targets in this region but the topic of clutter mapping is beyond the scope of this text. The entire spectrum is

covered with noise. Return signals outside the clutter region with sufficient signal to noise ratios can be detected and Doppler shift can be approximated as the DFT bin frequency where the peak DFT magnitude occurs. Multiple targets in a range bin can be detected as long as they are sufficiently separated in Doppler shift or relative velocity.

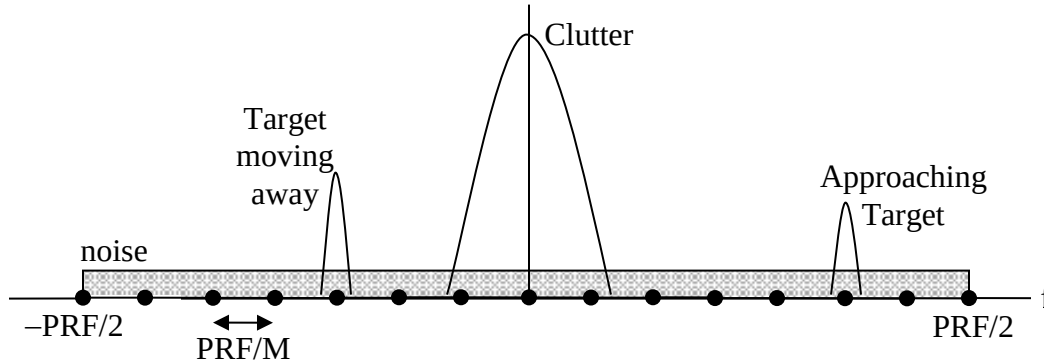


Figure 10.29: Sketch of DFT Spectrum

Example 10.7: Estimating Target Velocity using Pulse Doppler Processing

A pulsed radar signal consisting of 20 LFM chirp pulses with a PRF of 10 kHz and a transmit frequency of 3 GHz echoes off a target traveling toward the radar platform at a relative velocity of 150 m/s.

- Compute the Doppler shift, F_D , for the echo signal.
- Compute the frequency resolution and frequency range for the DFT of the echo signal. Is the Doppler shift within the frequency range for the DFT? Is the Doppler shift exactly equal to one of the DFT bin frequencies or will it straddle between two bins?
- Using MATLAB, create the demodulated echo signal (equation 10.29) and plot the DFT of the signal. What is the estimated Doppler shift based on the DFT?
- Repeat part (c) using a Doppler shift of 2750 Hz (target velocity 137.5 m/s). What happens to the DFT spectrum? What is the estimated target velocity?
- Repeat part (c) using a Doppler shift of 6000 Hz? What happens? Why?
- Assuming the target produces a detectable peak in the DFT spectrum and the Doppler shift is within the DFT frequency range (no aliasing), what is the largest possible estimation error for target velocity?

Solution

(a) $F_D = (2v/c) \cdot F_T = (2 \cdot 150 \text{ m/s}) / (3 \cdot 10^8 \text{ m/s}) \cdot (3 \cdot 10^9 \text{ Hz}) = 3000 \text{ Hz}$

- (b) $f_0 = \text{PRF}/M = 10\text{kHz}/20 = 500 \text{ Hz}$. The frequency range is $-\text{PRF}/2$ to $+\text{PRF}/2$ or -5000Hz to $+5000 \text{ Hz}$ so the Doppler shift is within range. Also, the Doppler shift is an integer multiple of the frequency resolution so it does equal one of the DFT bin frequencies.

- (c) The MATLAB function used to create the demodulated echo signal (with a bit of noise added in) and to plot the DFT spectrum is shown in Figure 10.30. Executing the MATLAB command: `Ex10_7(20,10000,3000,0,0)`; produces the DFT spectrum shown in Figure 10.31. Since the Doppler shift equals one of the bin frequencies, the estimated Doppler shift is equal to the actual Doppler shift of 3000 Hz.

```
function Ex10_7(M,PRF,Fd,win,num_zeros)

% Usage: Ex10_7(M,PRF,Fd,win,num_zeros)
% Plots the FFT of pulsed radar Echo Signal from moving target
% M is number of pulses, PRF is pulse repetition frequency, Fd
is Doppler
% shift, win = 1 uses Hamming window and any other entry does
% nothing, num_zeros is the number of desired zeros to be added
to the
% echo signal (zero stuffing) to improve resolution of the DFT

% Create demodulated pulsed echo signal
PRI = 1/PRF;
t = 0:PRI:PRI*(M-1);
echo_signal = exp(1i*2*pi*Fd*t)+0.5*randn(1,M);

% Window the data if hamming window if win = 1
if win ==1;
    echo_signal = echo_signal.*hamming(M)';
end

% Add zeros if zero stuffing option is selected
echo_signal = [echo_signal zeros(1,num_zeros)];
% Compute the DFT
N = M + num_zeros;
Yf = fftshift(fft(echo_signal));
fo = PRF/N;
f = ceil(-N/2)*fo:fo:floor((N-1)/2)*fo;
figure;
plot(f,abs(Yf));
xlabel('Doppler Shift (Hz)');ylabel('Magnitude');grid
hold on; stem(f,abs(Yf));
title('DFT of Echo Signal');hold off;
end
```

Figure 10.30: MATLAB m-function for Examples 10.7 and 10.8

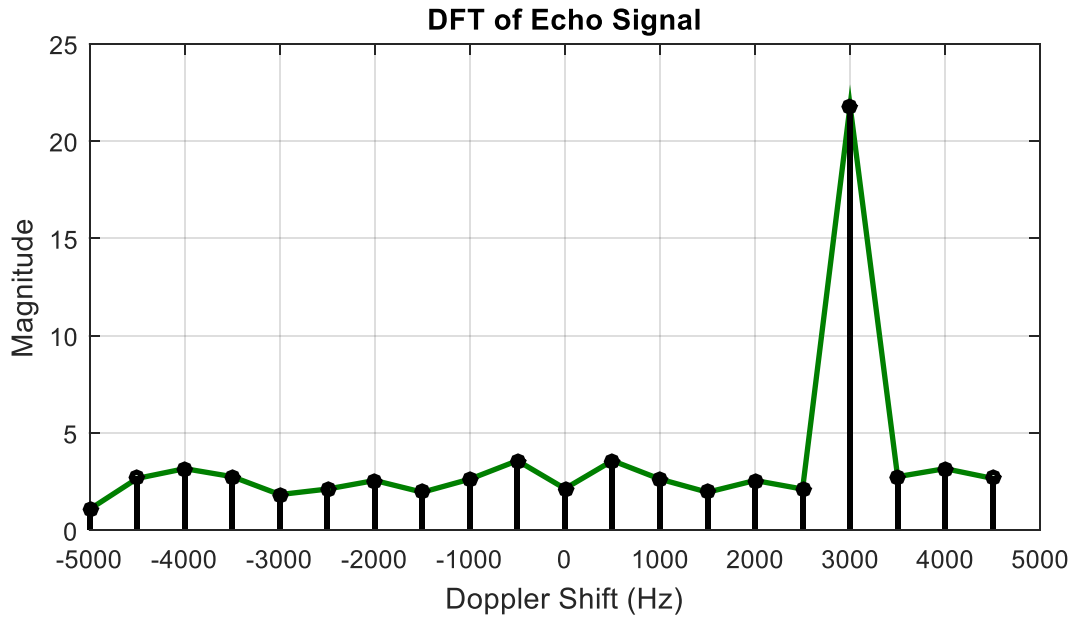


Figure 10.31: Doppler Shift = 3000 Hz (Example 10.7c)

- (d) Executing the MATLAB command: `Ex10_7(20, 10000, 2750, 0, 0)`; produces the DFT spectrum shown in Figure 10.32. Notice that since the Doppler shift of 2750 Hz is not one of the DFT bin frequencies, the DFT spectrum shows significant magnitude in the two surrounding bin frequencies of 2500 Hz and 3000 Hz. Note also that the peak magnitude has dropped considerably compared to the peak magnitude in Figure 10.31. Using Figure 10.32, the estimated Doppler shift would be 2500 Hz so the estimated target velocity would be 125 m/s, an error of 12.5 m/s.

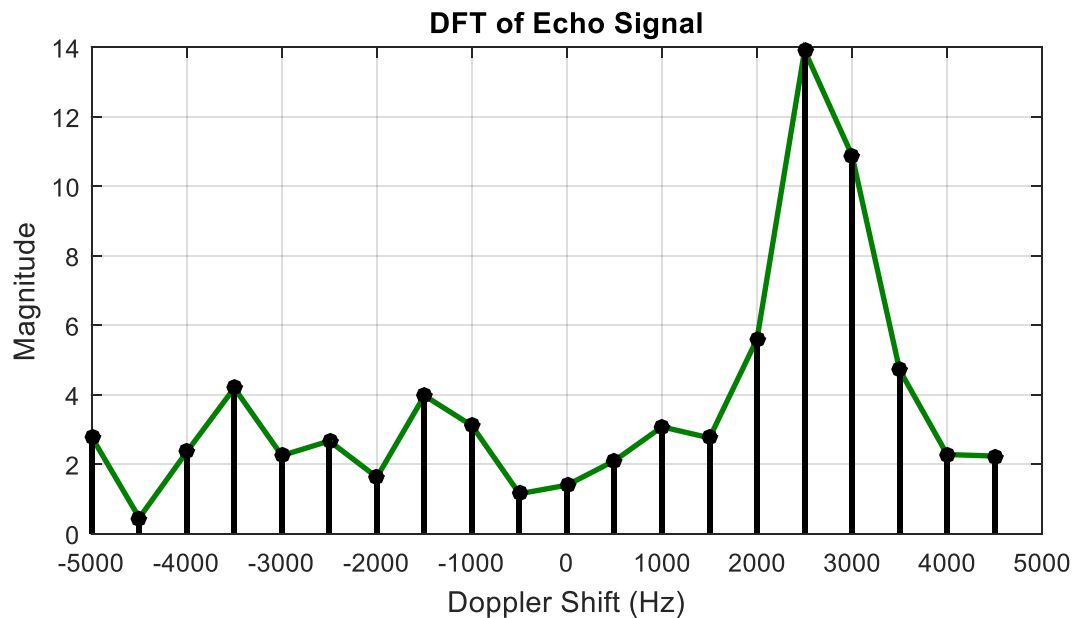


Figure 10.32: Doppler Shift = 2750 Hz (Example 10.7d)

- (e) Executing the MATLAB command: `Ex10_7(20,10000,6000,0,0)`; produces the DFT spectrum shown in Figure 10.33. Since the Doppler shift of 6000 Hz is outside the frequency range of the DFT, it aliases to -4000 Hz which would incorrectly identify a target moving away at a relative velocity of 200 m/s instead of an approaching target at 600 m/s.

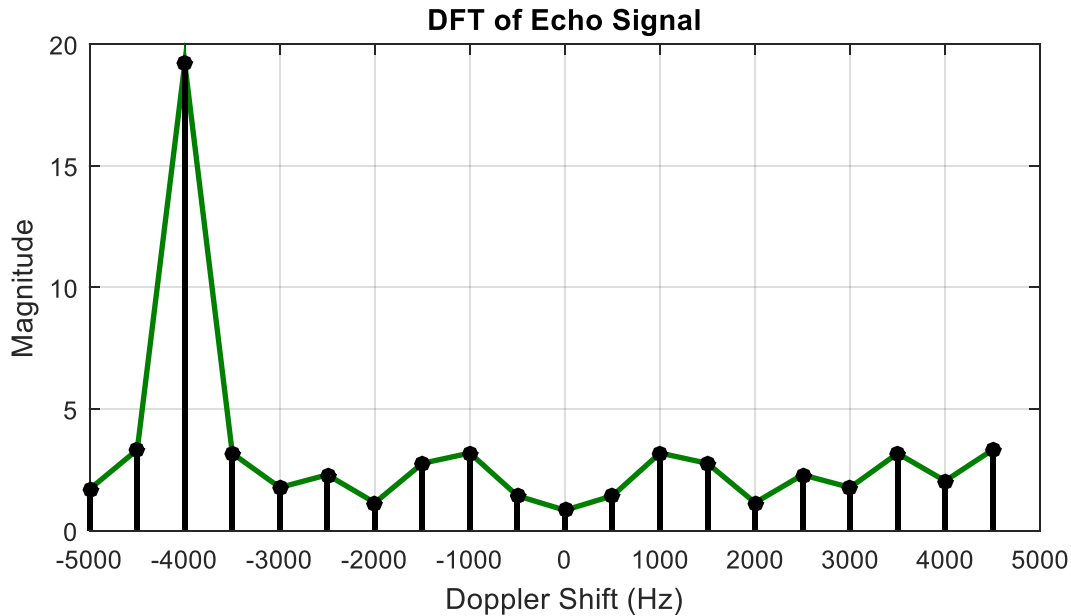


Figure 10.33 Doppler Shift = 6000 Hz (Example 10.7e)

- (f) The largest estimation error for Doppler shift would be half the frequency resolution or $\text{PRF}/(2 \cdot M) = 250$ Hz. For a transmit frequency of 3 GHz, this would correspond to a maximum relative velocity error of $(250 \text{ Hz} \cdot c) / (2 \cdot F_T) = 12.5$ m/s.

The echo signal data can be windowed prior to computing the DFT spectrum. Windowing was first introduced in Chapter 5 for FIR filter design and later applied to spectral analysis in Chapters 7 and 8. Windowing will reduce the sidelobe levels, widen the mainlobe, and reduce the mainlobe peak. So windowing the data can be advantageous when trying to resolve multiple targets within a range bin. The frequency resolution of the DFT spectrum can be improved by adding zeros to the data sequence prior to computing the DFT spectrum. If windowing is also being used, the data should be windowed first then stuffed with zeros. Example 10.8 shows the effects of windowing and zero stuffing on the DFT spectrum.

Example 10.8: Effect of Windowing and Zero Stuffing on Target Velocity Estimate

A pulsed radar signal consisting of 20 LFM chirp pulses with a PRF of 10 kHz and a transmit frequency of 3 GHz echoes off a target traveling toward the radar platform at a relative velocity of 137.5 m/s (Doppler shift = 2750 Hz).

- (a) Create the demodulated echo signal then window the data using a Hamming window. Plot the DFT spectrum. How does the spectrum of the windowed signal compare to the non-windowed signal?

- (b) Add 10 zeros (zero stuffing), to the end of the windowed echo signal and plot the DFT. What is the estimated Doppler shift? What is the estimated relative velocity of the target?

Solution

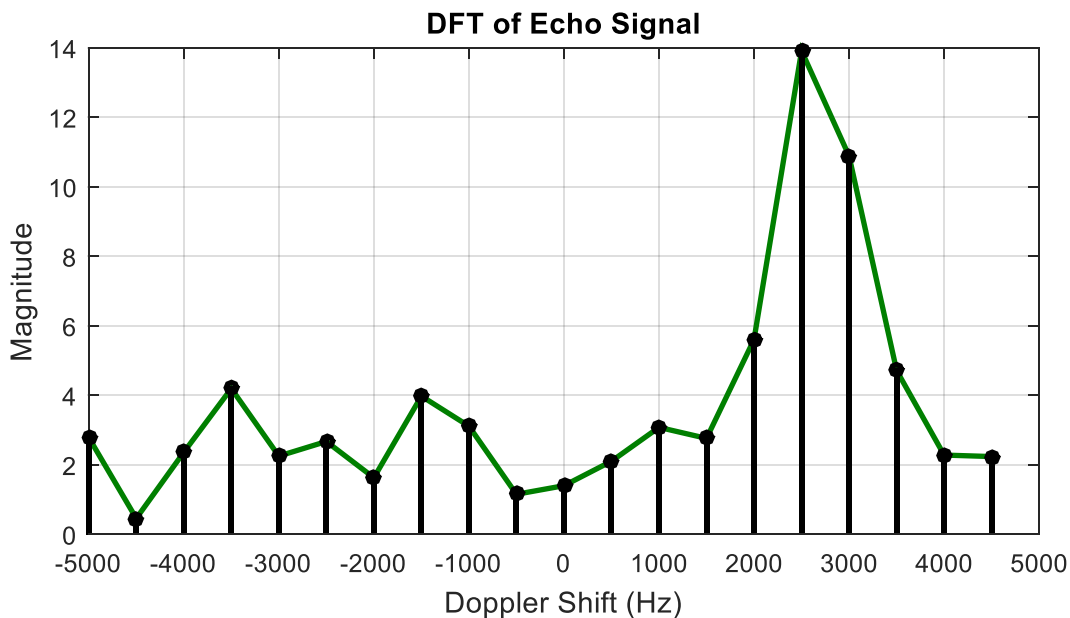
- (a) The MATLAB function used for this example is the same function used for Example 10.7 and is shown in Figure 10.30. Executing the command: `Ex10_7(20,10000,2750,1,0);` produces the DFT spectrum for the windowed data shown in Figure 10.34(b). For comparison purposes, the DFT spectrum of the non-windowed data is shown in Figure 10.34(a). Notice that the Hamming window does indeed reduce the level of the sidelobes but also reduces the mainlobe peak.
- (b) Executing the MATLAB command: `Ex10_7(20,10000,2750,1,10);` will first window the data as in part (a) then add 10 zeros prior to computing and plotting the DFT spectrum. The resulting spectrum is shown in Figure 10.34(c). Notice that the frequency resolution is now finer. The new resolution is:

$$f_0 = \text{PRF}/(M + \text{num}_{\text{zeros}}) = 10000\text{Hz}/(20 + 10) = 333.3\text{Hz}.$$

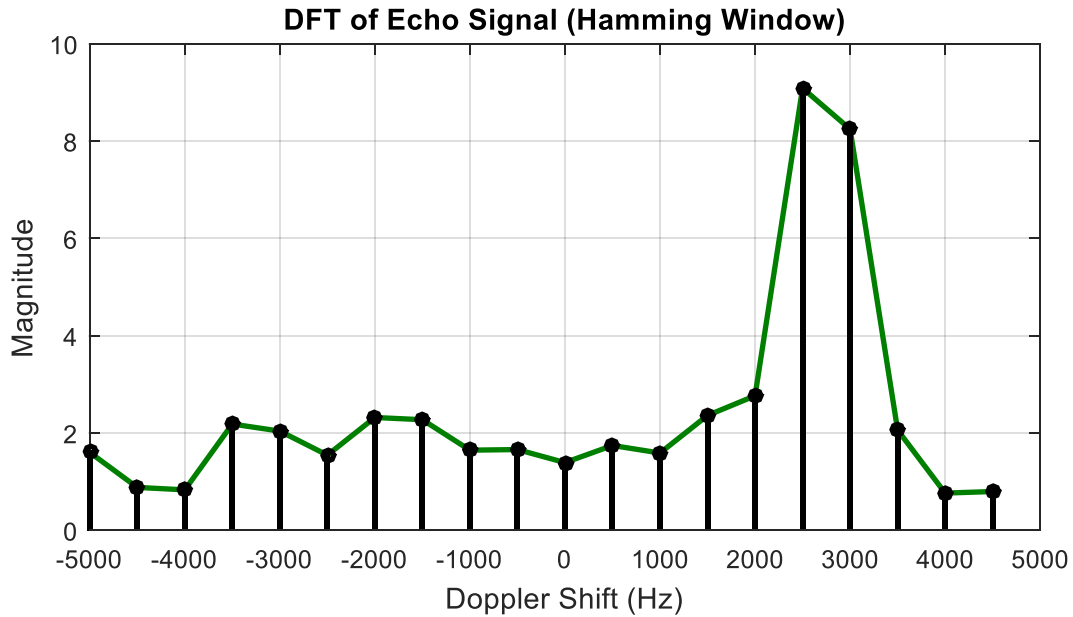
The estimated Doppler shift becomes 2666.7 Hz so the estimated relative velocity is

$$v = (F_D \cdot c)/(2 \cdot F_T) = (2666.7\text{Hz} \cdot 3 \cdot 10^8 \text{ m/s})/(2 \cdot 3 \cdot 10^9\text{Hz}) = 133.3 \text{ m/s}$$

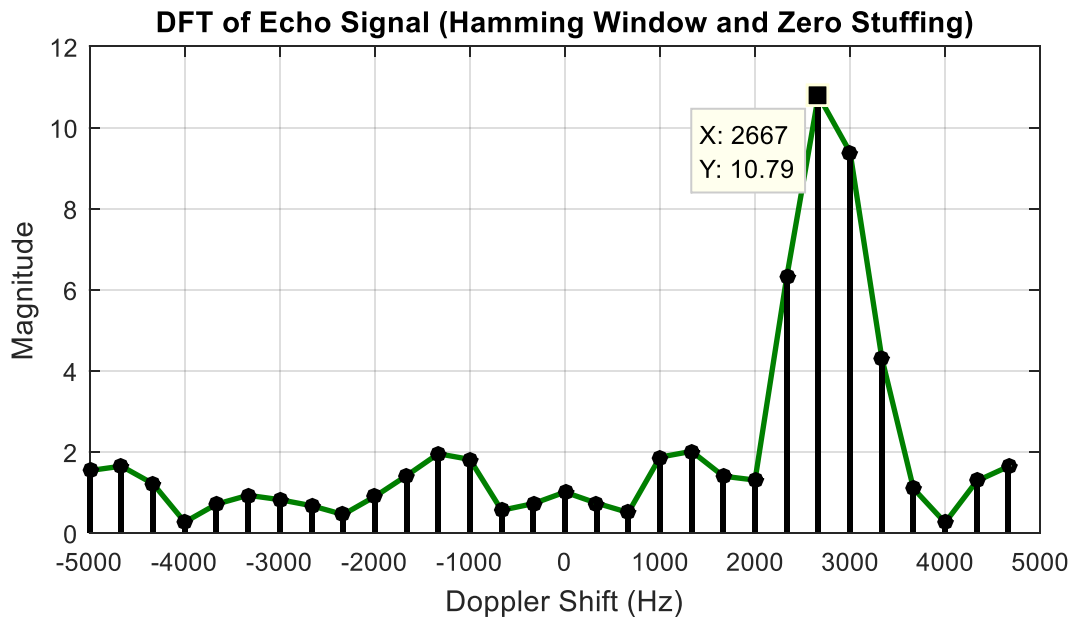
which reduces the estimated velocity error to 4.2 m/s.



(a) Doppler Shift = 2750 Hz (No Windowing)



(b) Doppler Shift = 2750 Hz (Hamming Window)



(c) Doppler Shift = 2750 Hz (Hamming Window and Zero Stuffing)

Figure 10.34: Effect of Windowing and Zero Stuffing on DFT Spectrum (Example 10.8)

This section is designed to provide an introduction to Doppler processing. As mentioned previously, there are several excellent textbooks offering extensive coverage of radar signal processing and some of these texts are listed in the references at the end of this text. There are

some issues and considerations that merit mention even though they cannot be covered in sufficient detail in a single introductory chapter.

Doppler Ambiguity and Range Ambiguity

The Doppler ambiguity is $PRF/2$, so any targets which cause a Doppler shift outside the range of $-PRF/2$ to $+PRF/2$ will alias into the DFT range. Also, any targets with a Doppler shift that is an integral multiple of the PRF will alias to D.C. and get lost in the clutter region of the DFT spectrum. Integral multiples of PRF are known as *blind speeds*. Clearly, increasing the PRF will extend the DFT spectrum and the corresponding range of Doppler shifts. However, increasing the PRF will also reduce the range ambiguity which is equal to $c/(2 \cdot PRF)$. As discussed earlier, the range ambiguity is the distance between targets such that both targets will eventually echo pulses back at the same time; that is, earlier echo pulses from the further target will coincide with later echo pulses from the nearer target. It may not be possible to settle on a single PRF that will satisfy both Doppler and range ambiguity requirements. In this case, radar systems use multiple PRFs to satisfy specifications. The PRF can be varied either within the set of M transmitted pulses or between sets of M transmitted pulses.

Matched Filter Banks

Pulse Doppler processing involves taking the DFT of the demodulated echo signal. Taking this DFT is exactly equivalent to processing the demodulated echo signal through M matched filters where each filter is matched to one of the discrete frequency bins. So, the DFT spectrum displays how well the demodulated echo signal correlates with each of the discrete frequencies in the spectrum. Another way to visualize pulse Doppler processing is as a bank of M band-pass filters each centered at one of the discrete bin frequencies. Windowing the data simply re-shapes the response of the band-pass filters.

Moving Target Indication

Another type of Doppler processing is called Moving Target Indication (MTI). MTI processing is done in the time domain and only determines the presence of a moving target within a range bin. It is not possible to determine velocity of the target or even whether there are multiple targets within the range bin. However, MTI processing is considerably simpler, computation-wise, than pulse Doppler processing. MTI processors use a high-pass filter to eliminate the clutter component of the signal spectrum followed by a threshold decision to find targets that are too large to simply be noise.

Chapter 10 Problems

Problem 10.1: A simple (single frequency) pulsed radar signal has a pulse repetition frequency (PRF) of 1200 pulses per second and a duty cycle of 0.5%.

- (a) Calculate the pulse duration, τ_d .
- (b) Calculate the range resolution.
- (c) Calculate the range ambiguity.
- (d) Explain the difference between range resolution and range ambiguity.

Problem 10.2: An LFM chirp pulsed radar signal has a pulse repetition frequency (PRF) of 1200 pulses per second, a duty cycle of 0.5%, and a frequency sweep range of 10 MHz.

- (a) Calculate the pulse duration, τ_d .
- (b) Calculate the range resolution.
- (c) Calculate the range ambiguity.
- (d) Compare the results with those in Problem 10.1.

Problem 10.3: An LFM chirp pulsed radar signal with a pulse duration equal to 4 μs is transmitted at a frequency of 2 GHz.

- (a) Calculate the range to target if the echo signal is processed through a matched filter and produces a peak exceeding the minimum threshold at 9 μs .
- (b) Calculate the target velocity in both m/s and mph if Doppler processing of the echo pulses indicates a Doppler shift of $F_D = 1$ kHz.

Problem 10.4: A certain C-band radar system has a peak transmitted power of 2 MW, an antenna gain of 40 dB, a transmission frequency of 4.2 GHz, a pulse duration of 100 ns, and a receiver noise figure of 3.2 dB. Assume that the minimum required SNR at the receiver output prior to processing is 12 dB and that the radar losses are negligible ($L = 0$ dB).

- (a) Using Equation 10.4, calculate the maximum range for this system assuming that the radar cross section (RCS) of a target will be at least 0.2 m^2 .
- (b) Calculate the minimum range for this system using Equation 10.13.

Problem 10.5:

- (a) Consider an LFM chirp signal with a pulse duration of $\tau_d = 1.0$ μs , a sweep frequency range of $\beta = 100$ MHz, and a sampling rate of $F_s = 500$ MHz. Using MATLAB, plot the chirp signal, find the matched filter coefficients for the chirp signal, and then plot the matched filter output response to the LFM chirp signal.
- (b) Repeat (a) for $\tau_d = 2.0$ μs , $\beta = 100$ MHz, and $F_s = 500$ MHz.
- (c) Repeat (a) for $\tau_d = 2.0$ μs , $\beta = 10$ MHz, and $F_s = 500$ MHz.
- (d) Comment on the results.

Problem 10.6: Suppose two scatterers are located within the receive window at relative ranges of 100m and 102m. Assume an LFM chirp pulse is transmitted with a pulse duration of 500 ns and a frequency modulation sweep range of 250 MHz. Assume the sampling rate for processing the return signal is 1000 MHz. Use the *Ex10_4.m* file to complete parts (c)-(e) of this problem.

- Calculate the range resolution.
- Calculate the time delay at the receiver for the return pulses from each of the scatterers.
- Plot each of the return signals and the composite signal assuming that the signal from the scatterer at 102m has one-third the strength of the return signal from the closer target.
- Process the composite return signal through the matched filter and plot the matched filter output.
- How accurate are the range estimates? What is the largest expected range error for this system?
- Modify the *Ex10_4.m* file to add some random noise to the composite return signal using *randn*. Experiment with the amount of noise and plot some representative results.

Problem 10.7: Consider a radar system using stretch processing to determine range. Assume a range window of 50m, LFM chirp pulse duration of 1.0 μ s, sweep frequency range of 750 MHz, and a transmission frequency of 1 GHz.

- Calculate the range resolution, ΔR .
- Calculate the smallest allowable value for $R_{\min}$ for this radar system.
- Calculate the receive window time, $T_{\text{rec}} = T_{\text{max}} - T_{\text{min}}$.
- Suppose two objects are located in a receive window at relative ranges of 2m and 10m. Compute the difference frequency for each of these objects at the output of the LPF in the stretch processor.

Problem 10.8: Consider a radar system using stretch processing to determine range. Assume all the same parameters for the range window and LFM chirp pulse as described in Problem 10.7.

- Determine the sampling frequency, F_s , number of FFT points, N , and frequency resolution, f_o , required to perform an N -pt FFT on the analog LPF output.
- Are the difference frequencies for the two objects described in Problem 10.7 equal to one of the bin frequencies for the FFT? If not, what will be the consequences of the mismatch in terms of estimating range?
- Based on the frequency resolution of the FFT, what is the largest possible error in estimating relative range? How does this compare to the range resolution, ΔR , calculated in Problem 10.7?
- Compare the sampling frequency required for stretch processing computed in part (a) of this problem to the sampling frequency that would be required to use matched filtering to compute range instead of stretch processing?

Problem 10.9: A pulsed radar signal consisting of 20 LFM chirp pulses with a PRF of 20 kHz and a transmit frequency of 5.6 GHz echoes off a target traveling away from the radar platform at a relative velocity of 250 m/s.

- (a) Compute the Doppler shift, F_D , for the echo signal.
- (b) Compute the frequency resolution and frequency range for the DFT of the echo signal. Is the Doppler shift within the frequency range for the DFT? Is the Doppler shift exactly equal to one of the DFT bin frequencies or will it straddle between two bins?
- (c) Using MATLAB and the m-file created for example 10.7, create the demodulated return echo signal and plot the DFT of the signal. What is the estimated Doppler shift based on the DFT? What is the estimated target velocity.
- (d) Assuming the target produces a detectable peak in the DFT spectrum and the Doppler shift is within the DFT frequency range (no aliasing), what is the largest possible estimation error for target velocity?
- (e) Window the demodulated echo signal using a Hamming window, stuff 10 zeros at the end of the data stream, and then compute and plot the DFT of the modified return signal. What is the estimated Doppler shift? What is the estimated target velocity? How does the DFT of the windowed and zero-stuffed signal compare to the DFT in part (c)?

CHAPTER 10 LAB EXERCISE

Radar Range Processing

Objectives

1. Investigate properties of chirp signals both in the time and frequency domain.
2. Design a matched filter capable of detecting a chirp signal buried in noise.
3. Use Simulink[®] to simulate a radar detector that computes the range to a target based on an echo return chirp signal buried in noise.

Procedure

A. Chirp Signal Properties and FFTs of Chirp Signals

1. Using the MATLAB function ***chirp.m***, create a chirp signal with the following properties:

A linear frequency sweep rate of 0 to β .

A pulse bandwidth product $\tau_d \cdot \beta \geq 100$

An sampling rate of 5β to 10β .

Record the variables for your chirp signal

$\tau_d =$ _____ $\beta =$ _____ $\tau_d \cdot \beta =$ _____ $F_s =$ _____

2. Plot the chirp signal and the DFT of the chirp signal.

3. Decrease the pulse bandwidth product by decreasing pulse width, τ_d . Discuss the effect on the chirp signal and its FFT. Print a few graphs to support your arguments.

B. Matched filters

1. Recreate your original chirp signal from step 1 of Part A.
2. Design a matched filter using the following MATLAB statement

```
h = ifft(conj(fft(y)));
```

3. Send your chirp signal, y , through the matched filter using the MATLAB command

```
output = conv(h,y);
```

4. Plot the filter output vs time. Remember to set-up your time increments using the sampling frequency for your particular chirp signal. Print the output.
5. This filter compresses the original chirp signal in time. At what time does your peak value occur for the filter output? How does this relate to the pulse width, τ_d , of your chirp signal?

6. Bury your chirp signal in noise using the following MATLAB command

```
y_noise = y + 2*randn(1,length(y));
```

7. Use subplot(2,1,1) and plot the noisy signal vs. time.
8. Now filter the noisy chirp signal using the same matched filter from step 2. Using subplot(2,1,2), plot the filter output for the noisy input then print your graph. How does this output compare to the filter output for the original chirp signal?

C. Radar Range Processing using Chirp Signals

1. Build the following system shown in Figure 1 in Simulink®.
2. The chirp signal, random source and digital filter are in the signal processing toolbox.
3. Set the properties in the chirp signal block to match those of your chirp signal from Part A Step 1.

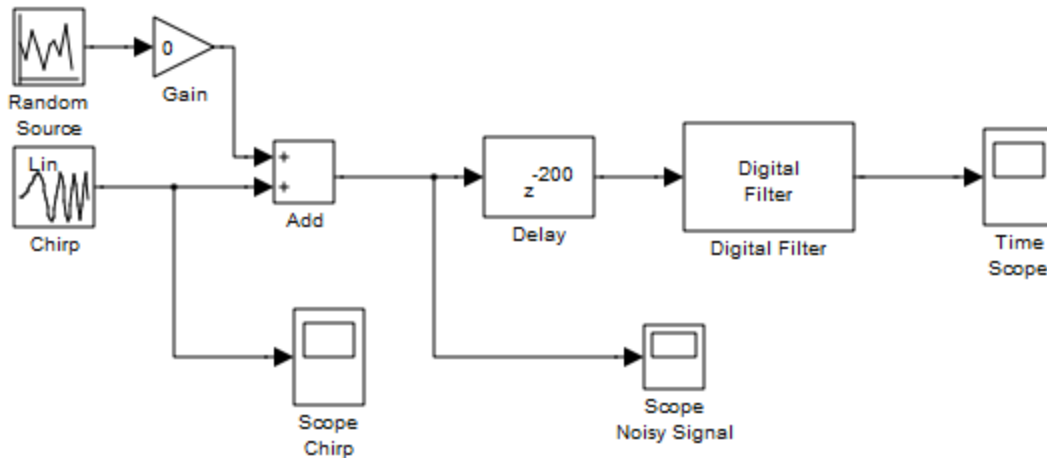


Figure 1: Radar Range Processing Simulation

4. Set the properties in the chirp signal block to match those of your chirp signal from Part A Step 1.
5. Set up the numerator coefficients of your Digital Filter block to h if that is how you defined your matching filter in Part B.
6. Set the sample time in the Random Source block to be the same as the sample time for the chirp signal.

7. Set the delay to 200 samples. Calculate the expected delay or travel time:

$$T_d = 200 * (1/F_s) = \underline{\hspace{2cm}}$$

8. Set up some reasonable stop time for your simulation.
9. Run the simulation. Determine at what time the filter output peaks. Subtract the pulse width of your chirp to determine delay time:

$$T_d = \text{Peak time} - \tau_d = \underline{\hspace{2cm}}. \quad \text{How does this compare to Step 6?}$$

10. Calculate the range to the target in meters. $R = \underline{\hspace{2cm}}$

11. Now add noise by adjusting the gain of the gain block following the random source. At what value of gain do you lose the ability to calculate range?

Appendix

Table A.1: Fourier Series

Table A.2: Fourier Transforms

Table A.3: Fourier Transform Properties

Table A.4: Z-Transform Properties

Table A.5: Z-Transforms

Table A.1: Fourier Series

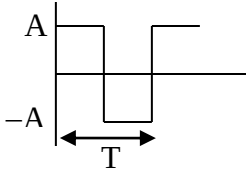
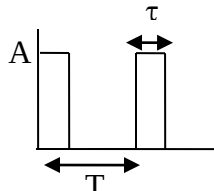
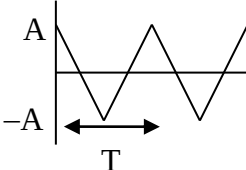
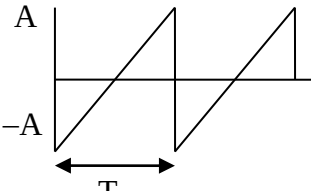
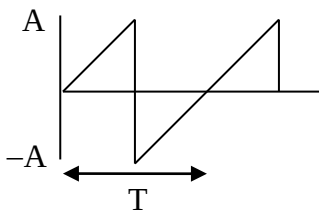
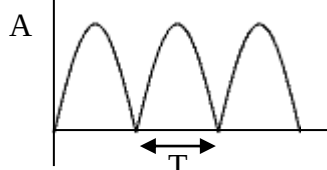
PERIODIC WAVEFORM	FOURIER SERIES EXPRESSION
	$\frac{4A}{\pi} \left[\sin(2\pi f_0 t) + \frac{1}{3} \sin(2\pi(3f_0)t) + \frac{1}{5} \sin(2\pi(5f_0)t) + \dots \right]$
	$Ad + \frac{A}{\pi} \{ [\sin(2\pi d) \cos(2\pi f_0 t) + 2 \sin^2(\pi d) \sin(2\pi f_0 t)]$ $+ \frac{1}{2} [\sin(2\pi(2d)) \cos(2\pi(2f_0)t) + 2 \sin^2(\pi(2d)) \sin(2\pi(2f_0)t)]$ $+ \frac{1}{3} [\sin(2\pi(3d)) \cos(2\pi(3f_0)t) + 2 \sin^2(\pi(3d)) \sin(2\pi(3f_0)t)]$ $+ \dots \} \quad \text{where } d = \tau/T$
	$\frac{8A}{\pi^2} \left[\cos(2\pi f_0 t) + \frac{1}{3^2} \cos(2\pi(3f_0)t) + \frac{1}{5^2} \cos(2\pi(5f_0)t) + \dots \right]$
	$-\frac{2A}{\pi} \left[\sin(2\pi f_0 t) + \frac{1}{2} \sin(2\pi(2f_0)t) + \frac{1}{3} \sin(2\pi(3f_0)t) + \dots \right]$
	$\frac{2A}{\pi} \left[\sin(2\pi f_0 t) - \frac{1}{2} \sin(2\pi(2f_0)t) + \frac{1}{3} \sin(2\pi(3f_0)t) - \dots \right]$
	$\frac{2A}{\pi} - \frac{4A}{\pi} \left[\frac{1}{4 \cdot 1^2 - 1} \cos(2\pi(f_0)t) + \frac{1}{4 \cdot 2^2 - 1} \cos(2\pi(2f_0)t) \right.$ $\left. + \frac{1}{4 \cdot 3^2 - 1} \cos(2\pi(3f_0)t) + \dots \right]$

Table A.2: Fourier Transforms

Time Signal: $f(t)$	$F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-j\omega t} dt = \int_{-\infty}^{\infty} f(t)e^{-j2\pi f t} dt$
$\delta(t)$	1
$\delta(t - \tau)$	$e^{-j\omega\tau}(1) = e^{-j2\pi f\tau}(1)$
$u(t) = \begin{cases} 0 & t < 0 \\ 1 & t > 0 \end{cases}$	$\pi\delta(\omega) + \frac{1}{j\omega} = \pi\delta(2\pi f) + \frac{1}{j2\pi f}$
$u(t - \tau)$	$e^{-j\omega\tau} \left[\pi\delta(\omega) + \frac{1}{j\omega} \right] = e^{-j2\pi f\tau} \left[\pi\delta(2\pi f) + \frac{1}{j2\pi f} \right]$
$e^{-at}u(t) \quad a > 0$	$\frac{1}{j\omega + a} = \frac{1}{j2\pi f + a}$
$te^{-at}u(t) \quad a > 0$	$\frac{1}{(j\omega + a)^2} = \frac{1}{(j2\pi f + a)^2}$
$rect\left(\frac{t}{\tau}\right) = \begin{cases} 1 & t \leq \frac{\tau}{2} \\ 0 & t \geq \frac{\tau}{2} \end{cases}$	$\tau \cdot sinc\left(\frac{\tau\omega}{2\pi}\right) = \tau \cdot sinc(\tau f)$
$sinc(t) = \frac{\sin(\pi t)}{\pi t}$	$rect\left(\frac{\omega}{2\pi}\right) = rect(f)$
$\cos(\omega_0 t) = \cos(2\pi f_0 t)$	$\pi[\delta(\omega + \omega_0) + \delta(\omega - \omega_0)] = \pi[\delta(f + f_0) + \delta(f - f_0)]$
$\sin(\omega_0 t) = \sin(2\pi f_0 t)$	$j\pi[\delta(\omega + \omega_0) - \delta(\omega - \omega_0)] = j\pi[\delta(f + f_0) - \delta(f - f_0)]$
$\cos(\omega_0 t + \theta)$	$\pi[e^{-j\theta}\delta(\omega + \omega_0) + e^{j\theta}\delta(\omega - \omega_0)]$

Table A.3: Fourier Transform Properties

Property	Function	Fourier Transform
Linearity	$a_1 f_1(t) + a_2 f_2(t)$	$a_1 F_1(\omega) + a_2 F_2(\omega)$
Time Delay	$f(t - \tau)$	$e^{-j\omega\tau} F(\omega)$
Convolution	$f(t) * g(t) = \int_{-\infty}^{\infty} f(\tau)g(t - \tau)d\tau$	$F(\omega)G(\omega)$
Time Scale	$f(at)$	$\frac{1}{ a } F\left(\frac{\omega}{a}\right)$
Duality	$F(t)$	$2\pi \cdot f(-\omega)$
Frequency Shift	$e^{j\omega_0 t} f(t)$	$F(\omega - \omega_0)$
Correlation	$\int_{-\infty}^{\infty} f(\tau - t)g(\tau)d\tau$	$F(-\omega)G(\omega)$

Table A.4: Z-Transform Properties

Property	Function	Z-Transform
Linearity	$a_1 f_1(k) + a_2 f_2(k)$	$a_1 F_1(z) + a_2 F_2(z)$
Time Delay	$f(k - n)$	$z^{-n} F(z)$
Convolution	$f(k) * g(k) = \sum_{n=-\infty}^{\infty} f(n)g(k - n)$	$F(z)G(z)$
Initial Value Theorem	$f(0)$	$\lim_{z \rightarrow \infty} F(z)$
Final Value Theorem	$\lim_{k \rightarrow \infty} f(k)$	$\lim_{z \rightarrow 1} (z - 1)F(z)$

Table A.5: Z-Transforms

Discrete Time Signal: $f(k)$		$F(z) = \sum_{k=-\infty}^{\infty} f(k)z^{-k}$
Impulse	$\delta(k)$	1
Time-Delayed Impulse	$\delta(k - n)$	$z^{-n} \cdot 1$
Unit Step	$u(k)$	$\frac{z}{z - 1}$
Delayed Unit Step	$u(k - n)$	$z^{-n} \cdot \frac{z}{z - 1}$
Ramp	$ku(k)$	$\frac{z}{(z - 1)^2}$
Exponential	$a^k u(k)$	$\frac{z}{z - a}$
Time-multiplied Exponential	$ka^k u(k)$	$\frac{z}{(z - a)^2}$
Time-multiplied Exponential (General)	$\frac{1}{(n - 1)!} \left\{ \prod_{i=0}^{n-2} (k - i) \right\} \cdot a^k u(k)$	$\frac{z}{(z - a)^n}$
Cosine	$\cos(\omega_0 k)$	$\frac{z(z - \cos(\omega_0))}{z^2 - (2 \cos(\omega_0))z + 1}$
Sine	$\sin(\omega_0 k)$	$\frac{z \sin(\omega_0)}{z^2 - (2 \cos(\omega_0))z + 1}$
Exponentially Modulated Phase-Shifted Cosine	$2 \cdot C \cdot a^k \cos(\omega_0 k + \theta)$	$\frac{(C \angle \theta)z}{z - (a \angle \omega_0)} + \frac{(C \angle -\theta)z}{z - (a \angle -\omega_0)}$

Bibliography

Textbooks

Ambardar, A., *Analog and Digital Signal Processing*, 2nd edition, Brooks/Cole Publishing Co., Pacific Grove, CA., 1999.

Attaway, S., *MATLAB® A Practical Introduction to Programming and Problem Solving*, 2nd edition, Elsevier, Inc., Waltham, MA, 2012.

Bracewell, R.N., *The Fourier Transform and its Applications*, 2nd edition, McGraw-Hill, New York, N.Y., 1978.

Chassaing, R. and Reay, D., *Digital Signal Processing and Applications with the TMS320C6713 and TMS320C6416 DSK*, John Wiley and Sons, Hoboken, N.J., 2008.

Dodge, C. and Jerse, T.A., *Computer Music: Synthesis, Composition, and Performance*, 2nd edition, Thompson Learning, Florence, KY, 1997.

Franklin, G.F. and Powell, J.D., *Digital Control of Dynamic Systems*, Addison-Wesley, Reading, MA, 1980.

Ifeachor, E.C. and Jervis, B.W., *Digital Signal Processing: A Practical Approach*, 2nd edition, Prentice-Hall, Upper Saddle River, N. J., 2001.

Lyons, R.G., *Understanding Digital Signal Processing*, 2nd edition, Pearson Education, Upper Saddle River, N. J., 2004.

Mahafza, B.R., *Radar Systems: Analysis and Design Using MATLAB®*, 2nd edition, Chapman & Hall/CRC, Boca Raton, FL., 2005.

McClellan, J.H., Schafer, R.W., and Yoder, M.A., *DSP FIRST: A Multimedia Approach*, Prentice-Hall, Inc., Upper Saddle River, N. J., 1998.

Mitra, S.K., *Digital Signal Processing: A Computer-Based Approach*, 2nd edition, McGraw-Hill, New York, N.Y., 2001.

O'Flynn, M. and Moriarty, E., *Linear Systems: Time Domain and Transform Analysis*, John Wiley and Sons, Inc., Hoboken, N. J., 1987.

Oppenheim, A.V., Schafer, R.W., and Buck, J.R., *Discrete-Time Signal Processing*, 2nd edition, Prentice-Hall, Upper Saddle River, N.J., 1999.

Richards, M.A., *Fundamentals of Radar Signal Processing*, McGraw-Hill, New York, N.Y., 2005.

Smith III, J. O., *Physical Audio Signal Processing For Virtual Musical Instruments and Audio Effects*, on-line textbook available at <http://www.dsprelated.com/dspbooks/pasp/>.

Smith, S.W., *Digital Signal Processing: A Practical Guide for Engineers and Scientists*, Newnes (Imprint of Elsevier Science), New York, 2003.

Papers and On-line Articles

Banks, K., *The Goertzel Algorithm*, EE Times, August 28, 2002.

<http://www.eetimes.com/design/embedded/4024443/The-Goertzel-Algorithm>

Bradford, R., Dobson, R. and ffitch, J., *Sliding is Smoother than Jumping*, International Computer Music Conference (ICMC 2005), Barcelona, Spain, September 2005.

Chowning, John. *The Synthesis of Complex Audio Spectra by Means of Frequency Modulation*. Journal of the Audio Engineering society, 21(7), 1973.

Cofer, R.C., and Harding, B., Avnet, *Fixed-Point DSP and Algorithm Implementation*, EE Times, October 25, 2006.

<http://www.dspdesignline.com/howto/showArticle.jhtml?articleID=193402220>

Kester W. and Bryant, J., *ADCs for DSP, part 1*, EE Times, October 4, 2007

<http://www.eetimes.com/design/signal-processing-dsp/4017542/ADCs-for-DSP-part-1>

Klein, B., *Signal Chain Basics (Part 8): Flash-and Pipeline-Converter Operation Explored*, EE Times, February 16, 2008.

<http://www.eetimes.com/design/automotive-design/4010021/SIGNAL-CHAIN-BASICS-Part-8--Flash-and-Pipeline-Converter-Operation-Explored>

LDS, DACTRON, *Understanding FFT Windows*, Application Note AN014, 2003.

Lerner, B., Analog Devices, *Application Dictates Choice: Fixed or Floating Point DSP?*, EDN, October 2006.

<http://www.edn.com/index.asp?layout=article&articleid=CA6382654&ref=nbra>

Lerner, B., Analog Devices, *Fixed vs. floating point: a surprisingly hard choice*, EE Times February 6, 2007.

<http://www.eetimes.com/design/signal-processing-dsp/4017010/Fixed-vs-floating-point-a-surprisingly-hard-choice>

Maxim, *Understanding Pipelined ADCs*, Application Note 1023, October 2001

<http://www.maxim-ic.com/app-notes/index.mvp/id/1023>

Risset, Jean-Claude. *Introductory Catalogue of Computer-Synthesized Sounds*. Murray Hill, N.J.: Bell Telephone Laboratories, 1969.

Staller, L. *Understanding Analog to Digital Converter Specifications*, EE Times, February 24, 2005.

<http://www.eetimes.com/design/embedded/4025078/Understanding-analog-to-digital-converter-specifications>

Turner, J. and Hoffbeck, J. P. “Putting Theory into Practice with Simulink” Proceedings of the 2005 ASEE Conference, Portland, OR, 2005.

Watson, A.B., *Image Compression Using the Discrete Cosine Transform*, Mathematica Journal, 4(1), PP 81-88, 1994.

Websites (site availability could change)

Analog Devices: Sigma-Delta ADC Tutorial

<http://designtools.analog.com/dt/sdtutorial/sdtutorial.html>

Bores: Introduction to DSP – collection of tutorials

<http://www.bores.com/courses/intro/index.htm>

Connexions: good source of DSP information (and lots of other topics as well)

<http://cnx.org/>

Ditech Networks: Echo Basics Tutorial

<http://www.ditechnetworks.com/learningCenter/echoBasics.html>

dspGuru by Iowegian International: provides a variety of DSP information

<http://www.dsprelated.com/dspbooks/pasp/>

DSPRelated.com: Forums/Discussion groups, on-line books and articles

<http://www.dsprelated.com/>

thefreesound project: nice source for different types of free sound files to experiment with

<http://www.freesound.org/index.php>

Impulse Adventure: Good references on jpeg, quantization matrices, and Huffman coding

<http://www.impulseadventure.com/photo/jpeg-quantization.html>

<http://www.impulseadventure.com/photo/jpeg-huffman-coding.html>

MATLAB® Central File Exchange: a website for sharing MATLAB® applications

<http://www.mathworks.com/matlabcentral/fileexchange/>

Texas Instruments: FilterPro – free analog filter design software

<http://focus.ti.com/docs/toolsw/folders/print/filterpro.html>

Index

A

- Adaptive Filtering, 342-346
- Additive Synthesis, 35
- Aliasing, 69-73
- All-pass Filters, 231-233
- Analog Signals, 11
- Analog-to-Digital Converters
 - Flash (Parallel) ADC, 62
 - Pipeline ADC, 63-64
 - Quantization Error, 59
 - Quantization Levels, 59
 - Resolution, 59
 - Sigma-Delta ADC, 64-66
 - Signal-to-Noise Ratio (SNR), 59
 - Step Size, 59
 - Successive Approximation ADC, 60-61
- Anti-Aliasing Filter Design, 74-77
- Arithmetic Overflow, 161-163, 226-229
- Audio Effects
 - Chorus, 164-165
 - Digital Graphic Equalizer, 243-246
 - Flanging, 165-167
 - Phasing, 233-235
 - Reverb, 236-243
 - Ring Modulation, 42-44
 - Tremolo, 41-44

B

- Band-limited Signal, 26-28
- Bandwidth, 26-28
- Bessel Filter, 77-78
- Bessel Functions, 37-38
- Butterfly Diagrams for FFT, 277-278
- Butterworth Filters, 75-77

C

- Cascade Realization IIR Filters, 216-218
- Chebyshev Filters, 77-78
- Chirp Signals, 426-427
- Circular Addressing, 313-314

C

- Coefficient Quantization FIR Filters, 156-158
- Coefficient Quantization IIR Filters, 218-226
- Color Depth, 357
- Comb Filters (Feedback, IIR), 237
- Comb Filters (Feedforward, FIR), 163-165
- Comparing FIR and IIR Filters, 205
- Convolution
 - Computing Convolution using DFT, 286
 - Definition, 113-115
 - Over-lap Add Algorithm, 314-315
 - Over-lap Save Algorithm, 316-317
 - Periodic Convolution, 287-289
 - Zero Padding, 288
- Correlation
 - Computing Correlation using DFT, 290
 - Definition, 120-123

D

- Decimation, 177-179
- Difference Equation, 105
- Digital Graphic Equalizer, 243-246
- Digital Signals, 13
- Direct Form - Symmetric, 159
- Direct Form - Transposed, 159
- Direct Form, 159
- Discrete Cosine Transform (DCT), 379-387
- Discrete Fourier Transform (DFT)
 - Definition, 258-260
 - Examples, 260-268
 - Key Facts about DFTs, 269
 - Leakage, 264-272
 - Windowing Data, 270-272
- Discrete Signals, 12
- Discrete Systems, 105-112
 - Impulse Response, 106-107
 - Input/Output Difference Equation, 105
 - Stability, 107, 111-112
 - Transfer Function, 109-111
- Doppler Shift, 455

E

Echo Cancellation, 342-343
Echo Filters, 230

F

Fast Convolution, 286
Fast Fourier Transform (FFT)
 Butterfly Notation, 277-278
 Flow Graphs Radix-2 for DIT FFT, 279-282
 Radix-2 Decimation in Time FFT, 275-276
 Real-time Processing of Data, 318-329
 Twiddle Factors, 272-275
Filter Design and Analysis Tool (FDAT)
 FIR Design 145-155
 IIR Design 207-214
Filters (see FIR, IIR, or Anti-aliasing Filters)
Finite Impulse Response (FIR) Filters
 Arithmetic Overflow (Scaling), 161-163
 Decimation, 177-179
 Filter Design and Analysis Tool, 145-155
 High-Pass, Band-Pass, and Notch Design, 141-145
 Impulse Response, 128
 Interpolation, 171-174
 Low-Pass Design, 130-141
 Output Response of Filter, 129
 Realizations, 159-160
 Roundoff Errors in Arithmetic Operations, 161
 Special Types of FIR Filters, 163-174
 Stability, 128
 Transfer Function, 128
 Wordlength Effects, 156-158
Fixed Point Format, 95-101
Flanging, 165-167
Flash (Parallel) ADC, 62
Floating Point Format, 102-105
FM Synthesis of Instruments, 36-41
Fold-over Frequency, 73
Fourier Series Design of FIR Filters, 130-145
Fourier Series, 20-23
Fourier Transform, 24-26
Frequency Response Ideal Low-Pass Filter, 130
Frequency Sampling for FIR Filter Design, 141

G

Gaussian Filters, 169-170
Goertzel Algorithm, 298-302

H

Huffman Code, 397-404

I

Images
 Blurring, 365-368
 Brighten, 362-364
 Contrast Adjustment, 361-364
 Darken, 362-364
 De-noising, 374-377
 Displaying Images, 360
 Edge Detection, 371-373
 Exporting Images, 360
 Importing Images, 360
 Lossless and Lossy Compression, 358
 Sharpening, 369-371
Impulse Response, 106-107, 128, 202
Infinite Impulse Response (IIR) Filters
 Arithmetic Overflow (Scaling), 226-230
 Design (Direct), 206
 Design (Indirect), 205-206
 Difference Equation, 202
 Filter Design and Analysis Tool, 207-214
 Impulse Response, 202
 Output Response, 203
 Realizations, 215-218
 Roundoff Errors in Arithmetic Operations, 226-230
 Special Types of IIR Filters, 230-246
 Stability, 203
 Transfer Function, 202
 Wordlength Effects, 218-226
Interpolation, 80-82
Interpolation Filters (FIR) 171-174
Inverse Z-Transforms, 117-119

J

JPEG Image Compression Format, 377-404

L

Leakage, 264-272
Least Mean Squares Algorithm, 345-346
Lossless and Lossy Image Compression, 358

M

Matched Filters for Radar Applications, 420-426, 428-432
Matched Filters, 291-294
Median Filters, 374-377
Moving Average (MA) Filters, 167-168
Multi-Rate Processing and Rate Conversion, 175-184

N

Noise Cancellation, 344
Non-periodic Signals, 24
Nyquist-Shannon Sampling Theorem, 66-68

O

Optimal Design of FIR Filters, 140
Over-lap Add Algorithm, 314-315
Over-lap Save Algorithm, 316-317

P

Parallel Realization IIR Filters, 217-218
Periodic Signals, 14-18
Phasing, 233-235
Pipeline ADC, 63-64
Pixel, 357
Power Spectral Density, 283-285
Pulse Compression using Matched Filters, 428-432
Pulse Doppler Processing, 455-464
Pulse Repetition Interval (PRI), 415
Pulsed Radar Signal, 415

Q

Q15 Format, 99-102
Quantization Matrices for Image Compression, 387-396

R

Radar Performance Metrics, 416-420
Radix-2 Decimation in Time (DIT) FFT Algorithm, 275-276
Range to Target Determination, 432-454
Range-Doppler Coupling, 454
Realizations for FIR Filters, 159-160
Realizations for IIR Filters, 215-218
Real-time Processing of Data for Spectral (FFT) Analysis, 318-328
Reconstruction, 80-82
Reverb, 236-243
Ring Modulation, 42-44

S

Sample Rate Conversion, 175-184
Sampling Frequency, 58
Sampling Theorem, 66-68
Scaling FIR Filter Coefficients, 161-163
Scaling IIR Filter Coefficients, 226-230
Sigma-Delta ADC, 64-66
Signal Conditioning, 78-80
Signals
 Analog, 11
 Digital, 13
 Discrete, 12
 Periodic, 14
 Non-Periodic, 19
Sliding DFT, 329-341
Spectrogram, 28-30
Spectrum of Analog Signals, 20-26
Spectrum of Sampled Signals, 67-69
Stability of Discrete Systems, 107, 111-112
Straddle Loss, 457
Stretch Processing, 444-454
Successive Approximation ADC, 60-61
Synthesis of Musical Instruments, 30-41

T

Target Masking, 454
Transfer Function of Discrete System, 109-111
Tremolo, 41-44
Twiddle Factors, 272-275
Twos Complement, 95-99

V

Velocity of Target, 455-464

W

Wiener Filters, 374-377
Windowing Data Prior to Spectral Analysis,
270-272
Windows, 136-140
Windows-Based Design of FIR Filters using
Fourier Series, 130-140
Word Length Effects FIR Filters, 156-158
Word Length Effects IIR Filters, 218-226

Z

Zero Stuffing, 461-463
Zero-Padding
 Convolution, 228
 Correlation, 290-291
Zoom FFT, 295-297
Z-Transforms, 107-112