# Digital Signal Processing
## *Using MATLAB® and Wavelets*

*Michael Weeks*

Electrical Engineering Series

# Digital Signal Processing

*Using MATLAB® and Wavelets*

License, Disclaimer of Liability, and Limited Warranty

The CD-ROM that accompanies this book may only be used on a single PC. This license does not permit its use on the Internet or on a network (of any kind). By purchasing or using this book/CD-ROM package (the "Work"), you agree that this license grants permission to use the products contained herein, but does not give you the right of ownership to any of the textual content in the book or ownership to any of the information or products contained on the CD-ROM. Use of third party software contained herein is limited to and subject to licensing terms for the respective products, and permission must be obtained from the publisher or the owner of the software in order to reproduce or network any portion of the textual material or software (in any media) that is contained in the Work.

Infinity Science Press LLC ("ISP" or "the Publisher") and anyone involved in the creation, writing or production of the accompanying algorithms, code, or computer programs ("the software") or any of the third party software contained on the CD-ROM or any of the textual material in the book, cannot and do not warrant the performance or results that might be obtained by using the software or contents of the book. The authors, developers, and the publisher have used their best efforts to insure the accuracy and functionality of the textual material and programs contained in this package; we, however, make no warranty of any kind, express or implied, regarding the performance of these contents or programs. The Work is sold "as is" without warranty (except for defective materials used in manufacturing the disc or due to faulty workmanship);

The authors, developers, and the publisher of any third party software, and anyone involved in the composition, production, and manufacturing of this work will not be liable for damages of any kind arising out of the use of (or the inability to use) the algorithms, source code, computer programs, or textual material contained in this publication. This includes, but is not limited to, loss of revenue or profit, or other incidental, physical, or consequential damages arising out of the use of this Work.

The sole remedy in the event of a claim of any kind is expressly limited to replacement of the book and/or the CD-ROM, and only at the discretion of the Publisher.

The use of "implied warranty" and certain "exclusions" vary from state to state, and might not apply to the purchaser of this product.

# Digital Signal Processing
*Using MATLAB® and Wavelets*

Michael Weeks
Georgia State University

Infinity Science Press LLC
Hingham, Massachusetts

Copyright 2007 by Infinity Science Press LLC
All rights reserved.

*This publication, portions of it, or any accompanying software may not be reproduced in any way, stored in a retrieval system of any type, or transmitted by any means or media, electronic or mechanical, including, but not limited to, photocopy, recording, Internet postings or scanning, without prior permission in writing from the publisher.*

Publisher: David F. Pallai

Infinity Science Press LLC
11 Leavitt Street
Hingham, MA 02043
Tel. 877-266-5796 (toll free)
Fax 781-740-1677
info@infinitysciencepress.com
*www.infinitysciencepress.com*

This book is printed on acid-free paper.

Michael Weeks. *Digital Signal Processing Using MATLAB and Wavelets.*
ISBN: 0-9778582-0-0

The publisher recognizes and respects all marks used by companies, manufacturers, and developers as a means to distinguish their products. All brand names and product names mentioned in this book are trademarks or service marks of their respective companies. Any omission or misuse (of any kind) of service marks or trademarks, etc. is not an attempt to infringe on the property of others.

Library of Congress Cataloging-in-Publication Data

Weeks, Michael.
Digital signal processing using MATLAB and Wavelets / Michael Weeks.
p. cm.
Includes index.
ISBN 0-9778582-0-0 (hardcover with cd-rom : alk. paper)
1. Signal processing–Digital techniques–Mathematics. 2. MATLAB. 3. Wavelets (Mathematics) I. Title.
TK5102.9.W433 2006
621.382'2–dc22
2006021318

6 7 8 9 5 4 3 2 1

Our titles are available for adoption, license or bulk purchase by institutions, corporations, etc. For additional information, please contact the Customer Service Dept. at 877-266-5796 (toll free).

Requests for replacement of a defective CD-ROM must be accompanied by the original disc, your mailing address, telephone number, date of purchase and purchase price. Please state the nature of the problem, and send the information to Infinity Science Press, 11 Leavitt Street, Hingham, MA 02043.

The sole obligation of Infinity Science Press to the purchaser is to replace the disc, based on defective materials or faulty workmanship, but not based on the operation or functionality of the product.

I dedicate this book to my wife Sophie. *Je t'aimerai pour toujours.*

# Contents

**Preface** **xxi**

**1 Introduction** **1**
1.1 Numbers . . . 1
1.1.1 Why Do We Use a Base 10 Number System? . . . 2
1.1.2 Why Do Computers Use Binary? . . . 2
1.1.3 Why Do Programmers Sometimes Use Base 16 (Hexadecimal)? . . . 3
1.1.4 Other Number Concepts . . . 4
1.1.5 Complex Numbers . . . 5
1.2 What Is a Signal? . . . 10
1.3 Analog Versus Digital . . . 14
1.4 What Is a System? . . . 19
1.5 What Is a Transform? . . . 20
1.6 Why Do We Study Sinusoids? . . . 22
1.7 Sinusoids and Frequency Plots . . . 24
1.8 Summations . . . 26
1.9 Summary . . . 27
1.10 Review Questions . . . 27

**2 MATLAB** **29**
2.1 Working with Variables . . . 30
2.2 Getting Help and Writing Comments . . . 31
2.3 MATLAB Programming Basics . . . 32
2.3.1 Scalars, Vectors, and Matrices . . . 33
2.3.2 Number Ranges . . . 35
2.3.3 Output . . . 35
2.3.4 Conditional Statements (if) . . . 36
2.3.5 Loops . . . 39

2.3.6 Continuing a Line . . . 39
2.4 Arithmetic Examples . . . 39
2.5 Functions . . . 52
2.6 How NOT to Plot a Sinusoid . . . 53
2.7 Plotting a Sinusoid . . . 56
2.8 Plotting Sinusoids a Little at a Time . . . 60
2.9 Calculating Error . . . 63
2.10 Sometimes 0 Is Not Exactly 0 . . . 64
2.10.1 Comparing Numbers with a Tolerance . . . 65
2.10.2 Rounding and Truncating . . . 69
2.11 MATLAB Programming Tips . . . 70
2.12 MATLAB Programming Exercises . . . 71
2.13 Other Useful MATLAB Commands . . . 81
2.14 Summary . . . 83
2.15 Review Questions . . . 83

**3 Filters** **85**
3.1 Parts of a Filter . . . 89
3.2 FIR Filter Structures . . . 91
3.3 Causality, Linearity, and Time-Invariance . . . 98
3.4 Multiply Accumulate Cells . . . 103
3.5 Frequency Response of Filters . . . 104
3.6 IIR Filters . . . 111
3.7 Trends of a Simple IIR Filter . . . 113
3.8 Correlation . . . 115
3.9 Summary . . . 128
3.10 Review Questions . . . 130

**4 Sinusoids** **133**
4.1 Review of Geometry and Trigonometry . . . 133
4.2 The Number $\pi$ . . . 134
4.3 Unit Circles . . . 136
4.4 Principal Value of the Phase Shift . . . 138
4.5 Amplitudes . . . 139
4.6 Harmonic Signals . . . 140
4.7 Representing a Digital Signal as a Sum of Sinusoids . . . 145
4.8 Spectrum . . . 152
4.9 Summary . . . 156

4.10 Review Questions . . . 156

**5 Sampling** **159**
5.1 Sampling . . . 160
5.2 Reconstruction . . . 162
5.3 Sampling and High-Frequency Noise . . . 162
5.4 Aliasing . . . 164
5.4.1 Aliasing Example . . . 165
5.4.2 Folding . . . 168
5.4.3 Locations of Replications After Sampling . . . 171
5.5 Nyquist Rate . . . 175
5.6 Bandpass Sampling . . . 176
5.7 Summary . . . 182
5.8 Review Questions . . . 183

**6 The Fourier Transform** **187**
6.1 Fast Fourier Transform Versus the Discrete Fourier Transform . . . 190
6.2 The Discrete Fourier Transform . . . 191
6.3 Plotting the Spectrum . . . 196
6.4 Zero Padding . . . 202
6.5 DFT Shifting Theory . . . 203
6.6 The Inverse Discrete Fourier Transform . . . 204
6.7 Forward and Inverse DFT . . . 207
6.8 Leakage . . . 212
6.9 Harmonics and Fourier Transform . . . 214
6.10 Sampling Frequency and the Spectrum . . . 219
6.11 Summary . . . 221
6.12 Review Questions . . . 221

**7 The Number $e$** **225**
7.1 Reviewing Complex Numbers . . . 225
7.2 Some Interesting Properties of $j$ . . . 228
7.2.1 Rotating Counterclockwise . . . 228
7.2.2 Rotating Clockwise . . . 229
7.2.3 Removing $j$ from $\sqrt{-a}$ . . . 230
7.3 Where Does $e$ Come from? . . . 230
7.4 Euler's Formula . . . 233
7.5 Alternate Form of Euler's Equation . . . 235
7.6 Euler's Inverse Formula . . . 236
7.7 Manipulating Vectors . . . 238

7.7.1 Adding Two Vectors . . . . . . . . . . . . . . . . . . . . . . . 238
7.7.2 Adding Vectors in General . . . . . . . . . . . . . . . . . . . . 239
7.7.3 Adding Rotating Phasors . . . . . . . . . . . . . . . . . . . . . 240
7.7.4 Adding Sinusoids of the Same Frequency . . . . . . . . . . . 240
7.7.5 Multiplying Complex Numbers . . . . . . . . . . . . . . . . . 240
7.8 Adding Rotating Phasors: an Example . . . . . . . . . . . . . . . . . 243
7.9 Multiplying Phasors . . . . . . . . . . . . . . . . . . . . . . . . . . 249
7.10 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 250
7.11 Review Questions . . . . . . . . . . . . . . . . . . . . . . . . . . . 250

**8 The $z$-Transform 253**
8.1 The $z$-Transform . . . . . . . . . . . . . . . . . . . . . . . . . . . 254
8.2 Replacing Two FIR Filters in Series . . . . . . . . . . . . . . . . . . 255
8.3 Revisiting Sequential Filter Combination with $z$ . . . . . . . . . . . . 257
8.4 Why Is $z^{-1}$ the Same as a Delay by One Unit? . . . . . . . . . . . 259
8.5 What Is $z$? . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 260
8.6 How the $z$-Transform Reduces to the Fourier Transform . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 261
8.7 Powers of $-z$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 261
8.8 Showing that $x[n] * h[n] \leftrightarrow X(z)H(z)$ . . . . . . . . . . . . . . . . 262
8.9 Frequency Response of Filters . . . . . . . . . . . . . . . . . . . . . 263
8.10 Trends of a Simple IIR Filter, Part II . . . . . . . . . . . . . . . . . 271
8.11 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 271
8.12 Review Questions . . . . . . . . . . . . . . . . . . . . . . . . . . . 272

**9 The Wavelet Transform 275**
9.1 The Two-Channel Filter Bank . . . . . . . . . . . . . . . . . . . . . 277
9.2 Quadrature Mirror Filters and Conjugate Quadrature Filters . . . . . . . . . . . . . . . . . . . . . . . . . . 279
9.3 How the Haar Transform Is a 45-Degree Rotation . . . . . . . . . . . 280
9.3.1 How The Haar Transform Affects a Point's Radius . . . . . . 281
9.3.2 How The Haar Transform Affects a Point's Angle . . . . . . . 282
9.4 Daubechies Four-Coefficient Wavelet . . . . . . . . . . . . . . . . . 284
9.5 Down-Sampling and Up-Sampling . . . . . . . . . . . . . . . . . . . 288
9.5.1 Example Using Down/Up-Samplers . . . . . . . . . . . . . . 288
9.5.2 Down-Sampling and Up-Sampling with 2 Coefficients . . . . . 290
9.5.3 Down-Sampling and Up-Sampling with Daubechies 4 . . . . . 292
9.6 Breaking a Signal Into Waves . . . . . . . . . . . . . . . . . . . . . . 295

9.7 Wavelet Filter Design—Filters with Four Coefficients . . . 309
9.8 Orthonormal Bases . . . 311
9.9 Multiresolution . . . 314
9.10 Biorthogonal Wavelets . . . 320
9.11 Wavelet Transform Theory . . . 324
9.12 Summary . . . 336
9.13 Review Questions . . . 336

**10 Applications 339**
10.1 Examples Working with Sound . . . 339
10.2 Examples Working with Images . . . 342
10.3 Performing the 2D Discrete Wavelet Transform on an Image . . . 344
10.3.1 2D DWT of a Grayscale Image . . . 347
10.3.2 2D DWT of a Color Image . . . 348
10.4 The Plus/Minus Transform . . . 350
10.5 Doing and Undoing the Discrete Wavelet Transform . . . 352
10.6 Wavelet Transform with Matrices . . . 357
10.7 Recursively Solving a *Su Doku* Puzzle . . . 361
10.8 Converting Decimal to Binary . . . 369
10.9 Frequency Magnitude Response of Sound . . . 373
10.10 Filter Design . . . 376
10.10.1 Windowing Methods . . . 376
10.10.2 Designing an FIR Filter . . . 380
10.11 Compression . . . 385
10.11.1 Experimenting with Compression . . . 387
10.11.2 Compressing an Image Ourselves . . . 393
10.12 Summary . . . 396
10.13 Review Questions . . . 396

**A Constants and Variables Used in This Book 399**
A.1 Constants . . . 399
A.2 Variables . . . 399
A.3 Symbols Common in DSP Literature . . . 401

**B Equations 403**
B.1 Euler's Formula . . . 403
B.2 Trigonometric Identities and Other Math Notes . . . 404
B.3 Sampling . . . 405

B.4 Fourier Transform (FT) . . . . . . . . . . . . . . . . . . . . . . . . . 405
B.5 Convolution . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 407
B.6 Statistics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 407
B.7 Wavelet Transform . . . . . . . . . . . . . . . . . . . . . . . . . . . . 409
B.8 $z$-Transform . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 409

**C DSP Project Ideas** **411**

**D About the CD-ROM** **415**

**E Answers to Selected Review Questions** **417**

**F Glossary** **439**

**Bibliography** **445**

**Index** **449**

# List of Figures


1.1 An example vector. . . . . 6
1.2 Calculating $\theta = \arctan(b/a)$ leads to a problem when $a$ is negative. . 8
1.3 A sound signal with a tape analog. . . . . 15
1.4 Sampling a continuous signal. . . . . 17
1.5 Three ways of viewing a signal. . . . . 19
1.6 An example system. . . . . 20
1.7 Three glasses of water. . . . . 21

2.1 A 200 Hz sinusoid produced by example MATLAB code. . . . . 57
2.2 Using the "plotsinusoids" function. . . . . 59
2.3 This signal repeats itself every second. . . . . 61
2.4 A close-up view of two sinusoids from 0.9 to 1 second. . . . . 62

3.1 An example signal, filtered. . . . . 86
3.2 The frequency content of the example signal, and low/highpass filters. 87
3.3 A digital signal, delayed, appears as a time-shifted version of itself. . 88
3.4 An adder with two signals as inputs. . . . . 89
3.5 A multiplier with two signals as inputs. . . . . 89
3.6 An example FIR filter with coefficients 0.5 and -0.5. . . . . 90
3.7 Signal $y$ is a delayed version of $x$. . . . . 91
3.8 FIR filter with coefficients {0.5, 0.5}. . . . . 91
3.9 An example FIR filter with coefficients 0.6 and 0.2. . . . . 92
3.10 General form of the FIR filter. . . . . 94
3.11 An example FIR filter. . . . . 97
3.12 A representation of an FIR filter. . . . . 97
3.13 Linear condition 1: scaling property. . . . . 100
3.14 Linear condition 2: additive property. . . . . 100
3.15 Multiply accumulate cell. . . . . 103
3.16 Multiply accumulate cells as a filter. . . . . 104
3.17 Frequency magnitude response for a lowpass filter. . . . . 105

3.18 Frequency magnitude response for a highpass filter. . . . . . . . . . . 105
3.19 Passband, transition band, and stopband, shown with ripples. . . . . 107
3.20 Frequency magnitude response for a bandpass filter. . . . . . . . . . 108
3.21 Frequency magnitude response for a bandstop filter. . . . . . . . . . 108
3.22 A notch filter. . . . . . . . . . . . . . . . . . . . . . . . . . . . 109
3.23 Frequency magnitude response for a bandpass filter with two passbands. 109
3.24 A filter with feed-back. . . . . . . . . . . . . . . . . . . . . . . . 111
3.25 Another filter with feed-back. . . . . . . . . . . . . . . . . . . . . 112
3.26 A third filter with feed-back. . . . . . . . . . . . . . . . . . . . . 113
3.27 General form of the IIR filter. . . . . . . . . . . . . . . . . . . . . 114
3.28 A simple IIR filter. . . . . . . . . . . . . . . . . . . . . . . . . . 114
3.29 Output from a simple IIR filter. . . . . . . . . . . . . . . . . . . . 116
3.30 Two rectangles of different size (scale) and rotation. . . . . . . . . . 126
3.31 Two rectangles represented as the distance from their centers to their edges. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 127
3.32 A rectangle and a triangle. . . . . . . . . . . . . . . . . . . . . . . 128
3.33 A rectangle and triangle represented as the distance from their centers to their edges. . . . . . . . . . . . . . . . . . . . . . . . . . . . 129

4.1 A right triangle. . . . . . . . . . . . . . . . . . . . . . . . . . . . 134
4.2 An example circle. . . . . . . . . . . . . . . . . . . . . . . . . . . 134
4.3 An angle specified in radians. . . . . . . . . . . . . . . . . . . . . 136
4.4 An angle specified in radians. . . . . . . . . . . . . . . . . . . . . 137
4.5 A 60 Hz sinusoid. . . . . . . . . . . . . . . . . . . . . . . . . . . 139
4.6 A vector of $-a$ at angle $\theta = a$ at angle $(\theta + \pi)$. . . . . . . . . . . . . 139
4.7 A harmonic signal. . . . . . . . . . . . . . . . . . . . . . . . . . . 142
4.8 A short signal. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 144
4.9 The short signal, repeated. . . . . . . . . . . . . . . . . . . . . . . 145
4.10 A digital signal (top) and its sum of sinusoids representation (bottom). 146
4.11 The first four sinusoids in the composite signal. . . . . . . . . . . . . 147
4.12 The last four sinusoids in the composite signal. . . . . . . . . . . . . 148
4.13 Frequency magnitude spectrum and phase angles. . . . . . . . . . . . 153
4.14 Spectrum plot: magnitude of $x(t) = 2 + 2\cos(2\pi(200)t)$. . . . . . . . 154
4.15 Spectrum plot: magnitudes. . . . . . . . . . . . . . . . . . . . . . . 155
4.16 Spectrum plot: phase angles. . . . . . . . . . . . . . . . . . . . . . 155

5.1 Sampling a noisy signal. . . . . . . . . . . . . . . . . . . . . . . . 163
5.2 Aliasing demonstrated. . . . . . . . . . . . . . . . . . . . . . . . . 167
5.3 $\cos(-2\pi 10t - \pi/3)$ and $\cos(2\pi 10t + \pi/3)$ produce the same result. . 170
5.4 Replications for $f_1 = 1$ Hz and $f_s = 4$ samples/second. . . . . . . . . 172

5.5 Replications for $f_1 = 2.5$ Hz and $f_s = 8$ samples/second. . . . . . . . 173
5.6 Replications for $f_1 = 3$ Hz and $f_s = 4$ samples/second. . . . . . . . . 173
5.7 Replications for $f_1 = 5$ Hz and $f_s = 4$ samples/second. . . . . . . . . 174
5.8 Replications for $f_1 = 3$ or 5 Hz and $f_s = 4$ samples/second. . . . . . 175
5.9 Example signal sampled at 2081 Hz. . . . . . . . . . . . . . . . . . . 177
5.10 Example signal sampled at 100 Hz. . . . . . . . . . . . . . . . . . . . 177
5.11 Example signal sampled at 103 Hz. . . . . . . . . . . . . . . . . . . . 179
5.12 Example signal sampled at 105 Hz. . . . . . . . . . . . . . . . . . . . 181
5.13 Example signal sampled at 110 Hz. . . . . . . . . . . . . . . . . . . . 181

6.1 A person vocalizing the "ee" sound. . . . . . . . . . . . . . . . . . . 188
6.2 J.S. Bach's *Adagio from Toccata and Fuge in C*—frequency magnitude response. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 189
6.3 A sustained note from a flute. . . . . . . . . . . . . . . . . . . . . . . 190
6.4 Comparing $Nlog_2(N)$ (line) versus $N^2$ (asterisks). . . . . . . . . . . 191
6.5 Spectrum for an example signal. . . . . . . . . . . . . . . . . . . . . 198
6.6 Improved spectrum for an example signal. . . . . . . . . . . . . . . . 200
6.7 Example output of DFT-shift program. . . . . . . . . . . . . . . . . 205
6.8 Frequency content appears at exact analysis frequencies. . . . . . . . 213
6.9 Frequency content appears spread out over analysis frequencies. . . . 213
6.10 Approximating a triangle wave with sinusoids. . . . . . . . . . . . . . 215
6.11 Approximating a saw-tooth wave with sinusoids. . . . . . . . . . . . 217
6.12 Approximating a square wave with sinusoids. . . . . . . . . . . . . . 218
6.13 Approximating a saw-tooth square wave with sinusoids. . . . . . . . 218
6.14 Frequency response of a lowpass filter. . . . . . . . . . . . . . . . . . 220
6.15 Frequency response of a highpass filter. . . . . . . . . . . . . . . . . 220

7.1 A complex number can be shown as a point or a 2D vector. . . . . . 226
7.2 A vector forms a right triangle with the x-axis. . . . . . . . . . . . . 226
7.3 A rotating vector. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 227
7.4 A vector rotates clockwise by $-\pi/2$ when multiplied by $-j$. . . . . . 230
7.5 Simple versus compounded interest. . . . . . . . . . . . . . . . . . . 232
7.6 Phasors and their complex conjugates. . . . . . . . . . . . . . . . . . 236
7.7 Graph of $x(0)$ where $x(t) = 3e^{j\pi/6}e^{j2\pi 1000t}$. . . . . . . . . . . . . . . 237
7.8 Two example vectors. . . . . . . . . . . . . . . . . . . . . . . . . . . 241
7.9 Adding two example vectors. . . . . . . . . . . . . . . . . . . . . . . 241
7.10 Adding and multiplying two example vectors. . . . . . . . . . . . . . 242
7.11 Two sinusoids of the same frequency added point-for-point and analytically. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 245
7.12 A graphic representation of adding 2 phasors of the same frequency. 247

7.13 A graphic representation of adding 2 sinusoids of the same frequency. 248

8.1 FIR filters in series can be combined. . . . . . . . . . . . . . . . . . . . 256
8.2 Two trivial FIR filters. . . . . . . . . . . . . . . . . . . . . . . . . . . 259
8.3 Two trivial FIR filters, reduced. . . . . . . . . . . . . . . . . . . . . . . 260
8.4 Example plot of zeros and poles. . . . . . . . . . . . . . . . . . . . . . 270

9.1 Analysis filters. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 276
9.2 Synthesis filters. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 276
9.3 A two-channel filter bank. . . . . . . . . . . . . . . . . . . . . . . . . . 277
9.4 A quadrature mirror filter for the Haar transform. . . . . . . . . . . . 280
9.5 A conjugate quadrature filter for the Haar transform. . . . . . . . . . . 280
9.6 A two-channel filter bank with four coefficients. . . . . . . . . . . . . . 284
9.7 Different ways to indicate down-samplers and up-samplers. . . . . . 288
9.8 A simple filter bank demonstrating down/up-sampling. . . . . . . . . 288
9.9 A simple filter bank demonstrating down/up-sampling, reduced. . . 289
9.10 Tracing input to output of a simple filter bank. . . . . . . . . . . . . . 289
9.11 A two-channel filter bank with down/up-samplers. . . . . . . . . . . . 290
9.12 A filter bank with four taps per filter. . . . . . . . . . . . . . . . . . . 292
9.13 Wavelet analysis and reconstruction. . . . . . . . . . . . . . . . . . . . 296
9.14 Alternate wavelet reconstruction. . . . . . . . . . . . . . . . . . . . . . 296
9.15 Impulse function analyzed with Haar. . . . . . . . . . . . . . . . . . . 298
9.16 Impulse function analyzed with Daubechies-2. . . . . . . . . . . . . . . 299
9.17 Impulse function (original) and its reconstruction. . . . . . . . . . . . 300
9.18 Example function broken down into 3 details and approximation. . . 301
9.19 Impulse function in Fourier-domain. . . . . . . . . . . . . . . . . . . . 304
9.20 Impulse function in Wavelet-domain. . . . . . . . . . . . . . . . . . . . 305
9.21 Designing a filter bank with four taps each. . . . . . . . . . . . . . . . 310
9.22 Two levels of resolution. . . . . . . . . . . . . . . . . . . . . . . . . . . 315
9.23 Biorthogonal wavelet transform. . . . . . . . . . . . . . . . . . . . . . . 320
9.24 Biorthogonal wavelet transform. . . . . . . . . . . . . . . . . . . . . . . 322
9.25 One channel of the analysis side of a filter bank. . . . . . . . . . . . . 329
9.26 The first graph of the "show_wavelet" program. . . . . . . . . . . . . 334
9.27 The second graph of the "show_wavelet" program. . . . . . . . . . . . 335
9.28 Analysis filters. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 337
9.29 Synthesis filters. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 337

10.1 Two-dimensional DWT on an image. . . . . . . . . . . . . . . . . . . . 345
10.2 Results of "DWT undo" for a 1D signal: original and approximation. 354
10.3 Results of "DWT undo" for a 1D signal: details for octaves 1–3. . . 355

10.4 Three octaves of the one-dimensional discrete wavelet transform. . . . 362
10.5 Example of the "show_sound" program. . . . . . . . . . . . . . . . . . 377
10.6 Two similar filters, with and without a gradual transition. . . . . . . . 379
10.7 Using nonwindowed filter coefficients. . . . . . . . . . . . . . . . . . . . . 382
10.8 Using windowed filter coefficients. . . . . . . . . . . . . . . . . . . . . . . 383
10.9 Windowed filter coefficients and those generated by `fir1`. . . . . . . . 384
10.10 Alternating flip for the windowed filter coefficients. . . . . . . . . . . 386

# List of Tables


1.1 Example signals. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

5.1 Frequencies detected with bandpass sampling. . . . . . . . . . . . . . 182

6.1 Example DFT calculations. . . . . . . . . . . . . . . . . . . . . . . . . 194
6.2 Example DFT calculations (rectangular form). . . . . . . . . . . . . 195
6.3 Sinusoids that simplify things for us. . . . . . . . . . . . . . . . . . . 211

7.1 Compound interest on \$1000 approximates \$1000 $\times e$. . . . . . . . . . 232

8.1 Convolution of $x$ and $h$. . . . . . . . . . . . . . . . . . . . . . . . . 263

9.1 Impulse function analyzed with Haar. . . . . . . . . . . . . . . . . . 298

10.1 An example *Su Doku* puzzle. . . . . . . . . . . . . . . . . . . . . . . 363
10.2 The example *Su Doku* puzzle's solution. . . . . . . . . . . . . . . . . 364
10.3 Converting from decimal to binary. . . . . . . . . . . . . . . . . . . . 370
10.4 Converting from a decimal fraction to fixed-point binary. . . . . . . . 371
10.5 Hexadecimal to binary chart. . . . . . . . . . . . . . . . . . . . . . . 374

A.1 Greek alphabet. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 402

# Preface

Digital signal processing is an important and growing field. There have been many books written in this area, however, I was motivated to write this manuscript because none of the existing books address the needs of the computer scientist. This work attempts to fill this void, and to bridge the disciplines from which this field originates: mathematics, electrical engineering, physics, and engineering mechanics. Herein, it is hoped that the reader will find sensible explanations of DSP concepts, along with the analytical tools on which DSP is based.

### DSP: Now and Then

Recently, I was asked if a student who took a course in DSP at another university should be allowed to transfer the credit to our university. The question was an interesting one, especially when I noticed that the student had taken the course 10 years earlier. For computer science, 10 years is a long time. But how much has DSP changed in the last decade? It occurred to me that there were several important changes, but theoretically, the core information has not changed. The curriculum of the DSP course in question contained all of the same topics that I covered in my DSP class that summer, with only two exceptions: MATLAB® and Wavelets.

### MATLAB

MATLAB is one of several tools for working with mathematics. As an experienced programmer, I was skeptical at first. Why should I learn another programming language when I could do the work in C/C++? The answer was simple: working with MATLAB is easier! Yes, there are some instances when one would want to use another language. A compiled program, such as one written in C++, will run faster than an interpreted MATLAB program (where each line is translated by the computer when the program is run). For hardware design, one might want to write a program in Verilog or VHDL, so that the program can be converted to a circuit. If you already know a programming language such as C, C++, Java, FORTRAN, etc.,

you should be able to pick up a MATLAB program and understand what it does. If you are new to programming, you will find MATLAB to be a forgiving environment where you can test out commands and correct them as needed.

### Wavelets

The wavelet transform is an analysis tool that has a relatively short history. It was not until 1989 that Stéphane Mallat (with some help from Meyers) published a revolutionary paper on wavelet theory, tying together related techniques that existed in several disciplines [1]. Other people have made significant contributions to this theory, such as Ingrid Daubechies [2], Strang and Nguyen [3], and Coifman and Wickerhauser [4]. It is a daunting task to write about the people who have contributed to wavelets, since anyone who is left out could be reading this now!

The wavelet transform is an important tool with many applications, such as compression. I have no doubt that future generations of DSP teachers will rank it second only to the Fourier transform in terms of importance.

### Other Developments

Other recent changes in DSP include embedded systems, the audio format MP3, and public awareness. Advertising by cellular phone marketers, which tries to explain to a nontechnical audience that digital signal processing is better than analog, is an example of the growing public awareness. Actually, the ads do not try to say *how* digital is better than analog, but they do point out problems with wireless analog phones.

## Acknowledgments

This book was only possible with the support of many people. I would like to thank Dr. Rammohan Ragade from the University of Louisville, who guided my research when I first started out. I would also like to thank Dr. Magdy Bayoumi at the University of Louisiana, who taught me much about research as well as how to present research results to an audience. Dr. Marty Fraser, who recently retired from Georgia State University, was a great mentor in my first years in academia.

My students over the years have been quite helpful in pointing out any problems within the text. I would especially like to thank Evelyn Brannock and Ferrol Blackmon for their help reviewing the material. I would also like to acknowledge Drs.

Kim King and Raj Sunderraman for their help. Finally, I could not have written this book without the support and understanding of my wife.

-M.C.W., Atlanta, Georgia, 2006

# Chapter 1

# Introduction

Digital Signal Processing (DSP) is an important field of study that has come about due to advances in communication theory, digital (computer) technology, and consumer devices. There is always a driving need to make things better and DSP provides many techniques for doing this. For example, people enjoy music and like to download new songs. However, with slow Internet connection speeds (typically 56 kilobits per second for a dial-up modem), downloading a song could take hours. With MP3 compression software, though, the size of the song is reduced by as much as 90%, and can be downloaded in a matter of minutes. The MP3 version of the song is not the same as the original, but is a "good enough" approximation that most users cannot distinguish from the original. How is this possible? First, it is important to know about the original song (a signal), and how it is represented digitally. This knowledge leads to an algorithm to remove data that the user will not miss. All of this is part of Digital Signal Processing.

To understand how to manipulate (process) a signal, we must first know a bit about the values that make up a signal.

## 1.1 Numbers

Consider our concepts of numbers. In the ancient world, people used numbers to count things. In fact, the letter "A" was originally written upside down, and represented an ox [5]. (It looks a bit like an ox's head if you write it upside down.) Ratios soon followed, since some things were more valuable than others. If you were an ancient pig farmer, you might want a loaf of bread, but would you be willing to trade a pig for a loaf of bread? Maybe you would be willing to trade a pig for a loaf of bread and three ducks. The point is, ratios developed as a way to compare dissimilar things. With ratios, come fractions. A duck may be worth three loaves of

bread, and if you traded two ducks you would expect six loaves of bread in return. This could work the other way, too. If you had a roasted duck, you might divide it into 3 equal sections, and trade one of the sections for a bread loaf. Ratios lead the way to fractions, and our use of the decimal point to separate the whole part of the number from the fractional part.

Zero is a useful number whose invention is often credited to the Arabs, though there is much debate on this point. Its origins may not ever be conclusive. It is one of the symbols we use in our base 10 system, along with the numerals 1 through 9. (Imagine if we used Roman numerals instead!) The concept of zero must have been revolutionary in its time, because it is an abstract idea. A person can see 1 duck or 3 loaves of bread, but how do you see 0 of something? Still, it is useful, even if just a counting system is used. We also use it as a placeholder, to differentiate 10 from 1 or 100.

Ten is a significant number that we use as the basis of our decimal number system. There is no symbol for ten like there is for 0–9 (at least, not in the decimal number system). Instead, we use a combination of symbols, i.e., 10. We understand that the 1 is for the ten's digit, and the 0 is the one's digit, so that placement is important. In any number system, we have this idea of placement, where the digits on the left are greater in magnitude than the digits on the right. This is also true in number systems that computers use, i.e., binary.

### 1.1.1 Why Do We Use a Base 10 Number System?

It's most likely that we use a base 10 number system because humans have 10 fingers and used them to count with, as children still do today.

### 1.1.2 Why Do Computers Use Binary?

Computers use base 2, binary, internally. This number system works well with the computer's internal parts; namely, transistors. A transistor works as a switch, where electricity flows easily when a certain value is applied to the gate, and the flow of electricity is greatly impeded when another value is applied. By viewing the two values that work with these transistors as a logical 0 (ground) and a logical 1 (e.g., 3.3 volts, depending on the system), we have a number system of two values. Either a value is "on," or it is "off." One of the strengths of using a binary system is that correction can take place. For example, if a binary value appears as 0.3 volts, a person would conclude that this value is supposed to be 0.0 volts, or logic 0. To the computer, such a value given to a digital logical circuit would be close enough to 0.0 volts to work the same, but would be passed on by the circuit as a corrected value.

### 1.1.3 Why Do Programmers Sometimes Use Base 16 (Hexadecimal)?

Computers use binary, which is great for a computer, but difficult for a human. For example, which is easier for you to remember, 0011001000010101 or 3215? Actually, these two numbers are the same, the first is in binary and the second is in hexadecimal. As you can see, hexadecimal provides an easier way for people to work with computers, and it translates to and from binary very easily. In fact, one hexadecimal digit represents four bits. With a group of four bits, the only possibilities are: 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, and 1111. These 16 possible combinations of 4 bits map to the numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. At this point, we have a problem, since we would be tempted to make the next number "10," as we would in decimal. The problem with this is that it is ambiguous: would "210" mean 2, 1, 0, or would it mean 2, 10? Since we have 16 different values in a group of four binary digits, we need 16 different symbols, one symbol for each value. This is why characters from the alphabet are used next: A, B, C, D, E, and F.

It is inefficient to use 1-bit registers, so computers group bits together into a word. The *word size* is dependent on the architecture (e.g., the "64" in the Nintendo 64 stands for the 64-bit word size of its processor). Typically, the word size is a multiple of a byte (8 bits), and hexadecimal numbers work nicely with such machines. For example, a machine with a 1-byte word size would work with data sizes of 2 hexadecimal digits at a time. A 16-bit word size means 4 hexadecimal digits are used.

Here is an example demonstrating word size. Suppose we have two multiplications,

$$9 \times 3 \qquad\qquad 4632 \times 9187.$$

We can do both of the above multiplications, but most people have an immediate answer for the one on the left, but need a minute or two for the one on the right. Why? While they might have the single-digit multiplication answers memorized, they use an algorithm for the multiple digit multiplications, i.e., multiply the rightmost digits, write the result, write the carry, move one digit to the left, for the "top" number, and repeat the process. In a sense, we have a 1-digit word size when performing this calculation. Similarly, a computer will multiply (or add, subtract, etc.) in terms of its word size. For example, suppose that a computer has a word size of 8 bits. If it needed to increment a 16-bit number, it would add one to the low 8 bits, then add the carry to the high 8 bits.

### 1.1.4 Other Number Concepts

Negative numbers are another useful concept that comes about due to borrowing. It is not difficult to imagine someone owing you a quantity of some item. How would you represent this? A negative amount is what we use today, e.g., in a checking account, to denote that not only do you have zero of something, but that you owe a quantity to someone else.

Infinity is another strange but useful idea. With our number system, no matter what extremely large number someone comes up with, you could still add 1 to it. So we use infinity to fill in for "the largest possible number." We can talk about all numbers by including negative infinity and infinity as the limits of our range. This is often seen in formulas as a way of covering all numbers.

Most of the discussion will be centered on fixed-point numbers. This means exactly what the name implies—the *radix point* is fixed. A radix point is the name given to the symbol separating the whole part from the fractional part—a period in the U.S., while Europeans use a comma. It is not exactly a "decimal point," since this term implies that base 10 (decimal) is being used. A fixed-point number can be converted to decimal (and back) in the same way that a binary number is converted.

Recall that to convert a binary number to decimal, one would multiply each bit in sequence by a multiple of 2, working from the right to the left. Thus, for the binary number 01101001, the decimal equivalent is $0 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 1 \times 64 + 1 \times 32 + 1 \times 8 + 1 \times 1 = 105$.

For a fixed-point number, this idea holds, except that the bit to the left of the radix point corresponds to $2^0$. One can start here and work left, then start again at the radix point and work right. For example, converting the fixed-point number 01010.111 would be $0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} = 1 \times 8 + 1 \times 2 + 1 \times (1/2) + 1 \times (1/4) + 1 \times (1/8) = 10.875$ in decimal.

To find a decimal number in binary, we separate the whole part from the fractional part. We take the whole part, divide it by 2, and keep track of the result and its remainder. Then we repeat this with the result until the result becomes zero. Read the remainders back, from bottom to top, and that is our binary number. For example, suppose we want to convert the decimal number 4 to binary:

$4/2 = 2$ remainder 0
$2/2 = 1$ remainder 0
$1/2 = 0$ remainder 1.

Therefore, decimal 4 equals 100 in binary. We can precede our answer by 0 to avoid confusion with a negative binary number, i.e., 0100 binary.

For a fractional decimal number, multiply by 2 and keep the whole part. Then repeat with the remaining fractional part until it becomes zero (though it is possible for it to repeat forever, just as 1/3 does in decimal). When finished, read the whole parts back from the top down. For example, say we want to convert .375 to binary:

$.375 \times 2 = 0$ plus .75
$.75 \times 2 = 1$ plus .5
$.5 \times 2 = 1$ plus 0.

Our answer for this is .011 binary. We can put the answers together and conclude that 4.375 equals 0100.011 in binary.

### 1.1.5 Complex Numbers

Complex numbers are important to digital signal processing. For example, functions like the Fourier transform (`fft` and `ifft`) return their results as complex numbers. This topic is covered in detail later in this book, but for the moment you can think of the Fourier transform as a function that converts data to an interesting form. Complex numbers provide a convenient way to store two pieces of information, either $x$ and $y$ coordinates, or a magnitude (length of a 2D vector) and an angle (how much to rotate the vector). This information has a physical interpretation in some contexts, such as corresponding to the magnitude and phase angle of a sinusoid for a given frequency, when returned from the Fourier transform.

Most likely, you first ran into complex numbers in a math class, factoring roots with the quadratic formula. That is, an equation such as

$$2x^2 + 4x - 30 = 0$$

could be factored as $(x - root_1)(x - root_2)$, where

$$root_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

If the numbers work out nicely, then the quadratic formula results in a couple of real roots. In this case, it would be $3, -5$, to make $(x-3)(x+5) = 0$. A problem arises when $b^2 - 4ac$ is negative because the square root of a negative quantity does not exist. The square root operator is supposed to return the positive root. How can we take a positive number, multiply it by itself, and end up with a negative number? For every "real" value, even the negative ones, the square is positive. To deal with this, we have the *imaginary* quantity $j$. (Some people prefer $i$ instead.)

This imaginary quantity is defined:

$$j = \sqrt{-1}.$$

Numbers that have $j$ are called "imaginary" numbers, as well as *complex* numbers. A *complex number* has a real part and an imaginary part, in the form $real + imaginary \times j$. For example, $x = 3 + 4j$. To specify the real or imaginary part, we use functions of the same name, i.e., $Real(x) = 3$, and $Imaginary(x) = 4$, or simply $\Re(x) = 3$, and $\Im(x) = 4$.

The best way to think of complex numbers is as a two-dimensional generalization of numbers; to think of real numbers as having an imaginary component of zero. Thus, real numbers lie on the x-axis.

Complex numbers can be used to hold two pieces of information. For example, you probably remember converting numbers with polar coordinates to Cartesian[1] ones, which are two ways of expressing the same thing. A complex number can represent a number in polar coordinates. The best way to show this is graphically. When we want to draw a point in two-dimensional space, we need to specify an $X$ coordinate and a $Y$ coordinate. This set of two numbers tells us where the point is located. Polar coordinates do this too, but the point is specified by a length and an angle. That is, the length specifies a line segment starting at point 0 and going to the right, and the angle says how much to rotate the line segment counterclockwise. In a similar way, a complex number represents a point on this plane, where the real part and imaginary part combine to specify the point. Figure 1.1 shows a 2D "vector" $r$ units long, rotated at an angle of $\theta$.

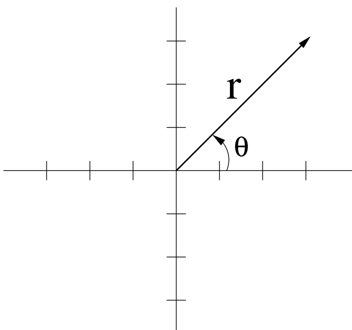



**Figure 1.1:** An example vector.

[1]Named for Rene Descartes, the same man famous for the saying "cogito ergo sum," or "I think therefore I am."

To convert from polar coordinates to complex Cartesian ones [6]:

$$x = r\ \cos(\theta)$$

$$y = r\ \sin(\theta).$$

The MATLAB® function below converts polar coordinates to complex coordinates. MATLAB will be explained in Chapter 2, "MATLAB," but anyone with some programming experience should be able to get the idea of the code below.

```
%  Convert from magnitude,angle form to x+jy form
%
%  usage:    [X] = polar2complex(magnitudes, angles)

function [X] = polar2complex(mag, angles)

% Find x and y coordinates as a complex number
X = mag.*cos(angles) + j*mag.*sin(angles);
```

To convert from Cartesian coordinates to polar ones:

$$r = |x + jy| = \sqrt{x^2 + y^2}$$

$$\theta = \arctan(y/x).$$

There is a problem with the equation used to convert to polar coordinates. If both real and imaginary parts ($x$ and $y$) are negative, then the signs cancel, and the *arctan* function returns the same value as if they both were positive. In other words, converting $2 + j2$ to polar coordinates gives the exact same answer as converting $-2 - j2$. In a similar fashion, the *arctan* function returns the same value for $\arctan(y/-x)$ as it does for $\arctan(-y/x)$. To fix this problem, examine the real part ($x$), and if it is negative, add $\pi$ (or $180°$) to the angle. Note that subtracting $\pi$ also works, since the angle is off by $\pi$, and the functions are $2\pi$ periodic. That is, the cosine and sine functions return the same results for an argument of $\theta + \pi - 2\pi$, as they do for an argument of $\theta + \pi$, and $\theta + \pi - 2\pi = \theta - \pi$.

Figure 1.2 demonstrates this. A vector is shown in all four quadrants. The radius $r$ equals $\sqrt{x^2 + y^2}$, and since $x$ is always positive or negative $a$, and $y$ is always positive or negative $b$, all four values of $r$ are the same, where $a$ and $b$ are

the distances from the origin along the real and imaginary axes, respectively. Let $a = 3$ and $b = 4$:

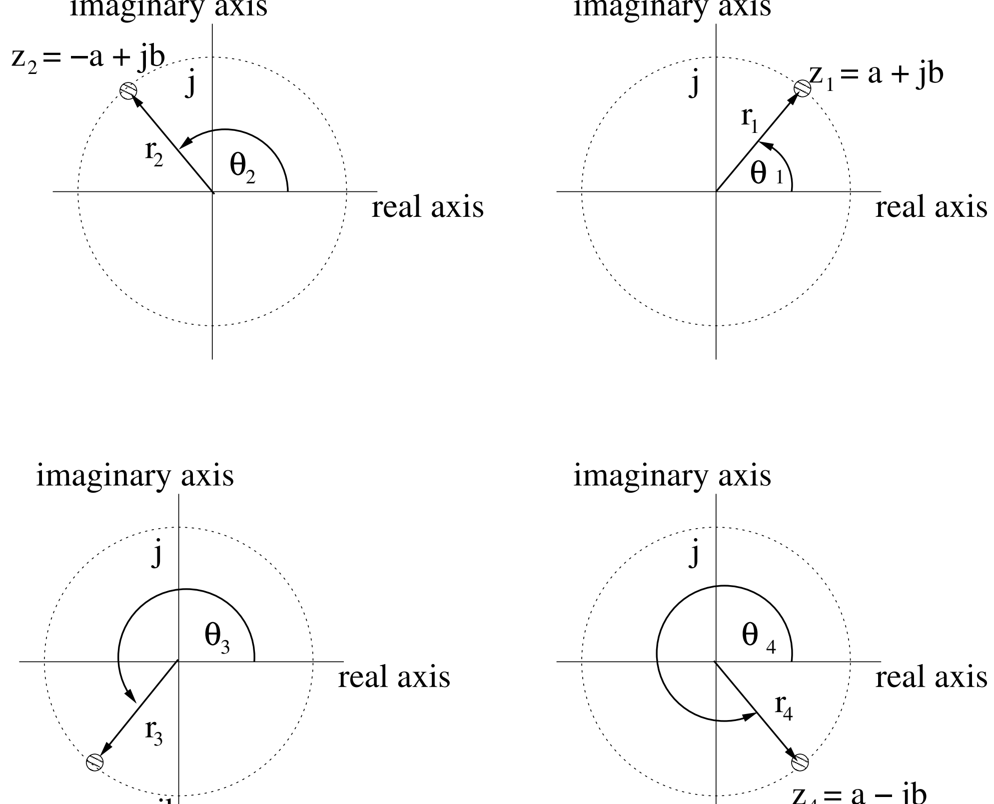



**Figure 1.2:** Calculating $\theta = \arctan(b/a)$ leads to a problem when $a$ is negative.

$$\begin{aligned}
r_1 &= \sqrt{a^2 + b^2} = \sqrt{3^2 + 4^2} = 5 \\
r_2 &= \sqrt{(-a)^2 + (b)^2} = \sqrt{(-3)^2 + 4^2} = 5 \\
r_3 &= \sqrt{(-a)^2 + (-b)^2} = \sqrt{(-3)^2 + (-4)^2} = 5 \\
r_4 &= \sqrt{(a)^2 + (-b)^2} = \sqrt{(-3)^2 + (-4)^2} = 5.
\end{aligned}$$

Here are the calculated angles:

$$\begin{aligned}
\theta_1 &= \arctan(b/a) = \arctan(4/3) = 0.9273\ rad \\
\theta_2 &= \arctan(b/-a) = \arctan(-4/3) = -0.9273\ rad \\
\theta_3 &= \arctan(-b/-a) = \arctan(4/3) = 0.9273\ rad \\
\theta_4 &= \arctan(-b/a) = \arctan(-4/3) = -0.9273\ rad.
\end{aligned}$$

Clearly, $\theta_2$ and $\theta_3$ are not correct since they lie in the second and third quadrants, respectively. Therefore, their angles should measure between $\pi/2$ ($\approx 1.57$) and $\pi$ ($\approx 3.14$) for $\theta_2$ and between $-\pi/2$ and $-\pi$ for $\theta_3$. Adding $\pi$ to $\theta_2$ and $-\pi$ to $\theta_3$ fixes the problem. $\theta_4$ is fine, even though the *arctan* function returns a negative value for it. Here are the corrected angles:

$$\begin{aligned}
\theta_1 &= \arctan(b/a) = \arctan(4/3) = 0.9273\ rad \\
\theta_2 &= \arctan(b/-a) + \pi = -0.9273 + \pi = 2.2143\ rad \\
\theta_3 &= \arctan(-b/-a) - \pi = 0.9273 - \pi = -2.2143\ rad \\
\theta_4 &= \arctan(-b/a) = \arctan(-4/3) = -0.9273\ rad.
\end{aligned}$$

The function below converts from complex numbers like $x + jy$ to polar coordinates. This is not as efficient as using *abs* and *angle*, but it demonstrates how to implement the equations.

```
%
%  Convert from complex form (x+jy)
%  to polar form (magnitude,angle)
%
%  usage:    [r, theta] = complex2polar(X)
%
function [mag, phase] = complex2polar(X)

% Find magnitudes
mag = sqrt(real(X).*real(X) + imag(X).*imag(X));

% Find phase angles
% Note that parameters for tan and atan are in radians
```

```
for i=1:length(X)
    if (real(X(i)) > 0)
        phase(i) = atan(imag(X(i)) / real(X(i)));
    elseif (real(X(i)) < 0)
        % Add to +/- pi, depending on quadrant of the point
        if (imag(X(i)) < 0)  % then we are in quadrant 3
            phase(i) = -pi + atan(imag(X(i)) / real(X(i)));
        else                 % we are in quadrant 2
            phase(i) =  pi + atan(imag(X(i)) / real(X(i)));
        end
    else
    % If real part is 0, then it lies on the + or - y-axis
    % depending on the sign of the imaginary part
        phase(i) = sign(imag(X(i)))*pi/2;
    end
end
```

## 1.2 What Is a Signal?

A *signal* is a varying phenomenon that can be measured. It is often a physical quantity that varies with time, though it could vary with another parameter, such as space. Examples include sound (or more precisely, acoustical pressure), a voltage (such as the voltage differences produced by a microphone), radar, and images transmitted by a video camera. Temperature is another example of a signal. Measured every hour, the temperature will fluctuate, typically going from a cold value (in the early morning) to a warmer one (late morning), to an even warmer one (afternoon), to a cooler one (evening) and finally a cold value again at night. Often, we must examine the signal over a period of time. For example, if you are planning to travel to a distant city, the city's average temperature may give you a rough idea of what clothes to pack. But if you look at how the temperature changes over a day, it will let you know whether or not you need to bring a jacket.

Signals may include error due to limitations of the measuring device, or due to the environment. For example, a temperature sensor may be affected by wind chill. At best, signals represented by a computer are good approximations of the original physical processes.

Some real signals can be measured continuously, such as the temperature. No matter what time you look at a thermometer, it will give a reading, even if the time between the readings is arbitrarily small. We can record the temperature at intervals of every second, every minute, every hour, etc. Once we have recorded these

measurements, we understand intuitively that the temperature has values *between* readings, and we do not know what values these would be. If a cold wind blows, the temperature goes down, and if the sun shines through the clouds, then it goes up. For example, suppose we measure the temperature every hour. By doing this, we are choosing to ignore the temperature for all time except for the hourly readings. This is an important idea: the signal may vary over time, but when we take periodic readings of the signal, we are left with only a *representation* of the signal.

A signal can be thought of as a (continuous or discrete) sequence of (continuous or discrete) values. That is, a continuous signal may have values at any arbitrary index value (you can measure the temperature at noon, or, if you like, you can measure it at 0.0000000003 seconds after noon). A discrete signal, however, has restrictions on the index, typically that it must be an integer. For example, the mass of each planet in our solar system could be recorded, numbering the planets according to their relative positions from the sun. For simplicity, a discrete signal is assumed to have an integer index, and the relationship between the index and time (or whatever parameter) must be given. Likewise, the *values* for the signal can be with an arbitrary precision (continuous), or with a restricted precision (discrete). That is, you could record the temperature out to millionths of a degree, or you could restrict the values to something reasonable like one digit past the decimal. Discrete does not mean integer, but rather that the values could be stored as a rational number (an integer divided by another integer). For example, 72.3 degrees Fahrenheit could be thought of as 723/10. What this implies is that irrational numbers cannot be stored in a computer, but only approximated. $\pi$ is a good example. You might write 3.14 for $\pi$, but this is merely an approximation. If you wrote 3.141592654 to represent $\pi$, this is still only an approximation. In fact, you could write $\pi$ out to 50 million digits, but it would still be only an approximation!

It is possible to consider a signal whose index is continuous and whose values are discrete, such as the number of people who are in a building at any given time. The index (time) may be measured in fractions of a second, while the number of people is always a whole number. It is also possible to have a signal where the index is discrete, and the values are continuous; for example, the time of birth of every person in a city. Person #4 might have been born only 1 microsecond before person #5, but they technically were not born at the same time. That does not mean that two people cannot have the exact same birth time, but that we can be as precise as we want with this time. Table 1.1 gives a few example signals, with continuous as well as discrete indices and quantities measured.

For the most part, we will concentrate on continuous signals (which have a continuous index and a continuous value), and discrete signals (with an integer index and a discrete value). Most signals in nature are continuous, but signals represented

**Table 1.1:** Example signals.

| quantity measured | index | |
|---|---|---|
| | **discrete** | **continuous** |
| **discrete** | mass (planet number) | people in a building (time) |
| **continuous** | birth time (person) | temperature (time) |

inside a computer are discrete. A discrete signal is often an approximation of a continuous signal. One notational convention that we adopt here is to use $x[n]$ for a discrete signal, and $x(t)$ for a continuous signal. This is useful since you are probably already familiar with seeing the parentheses for mathematical functions, and using square brackets for arrays in many computer languages.

Therefore, there are 4 kinds of signals:

- A signal can have a continuous value for a continuous index. The "real world" is full of such signals, but we must approximate them if we want a digital computer to work with them. Mathematical functions are also continuous/continuous.
- A signal can have a continuous value for a discrete index.
- A signal can have a discrete value for a continuous index.
- A signal can have a discrete value for a discrete index. This is normally the case in a computer, since it can only deal with numbers that are limited in range. A computer could calculate the value of a function (e.g., $\sin(x)$), or it could store a signal in an array (indexed by a positive integer). Technically, both of these are discrete/discrete signals.

We will concentrate on the continuous/continuous signals, and the discrete/discrete signals, since these are what we have in the "real world" and the "computer world," respectively. We will hereafter refer to these signals as "analog" and "digital," respectively.

In the digital realm, a signal is nothing more than an array of numbers. It can be thought of in terms of a vector, a one-dimensional matrix. Of course, there are multidimensional signals, such as images, which are simply two-dimensional matrices. Thinking of signals as matrices is an important analytical step, since it allows us to use linear algebra with our signals. The meaning of these numbers depends on the application as well as the time difference between values. That

is, one array of numbers could be the changes of acoustic pressure measured at 1 millisecond intervals, or it could be the temperature in degrees centigrade measured every hour. This essential information is not stored within the signal, but must be known in order to make sense of the signal. Signals are often studied in terms of time and amplitude. Amplitude is used as a general way of labeling the units of a signal, without being limited by the specific signal. When speaking of the amplitude of a value in a signal, it does not matter if the value is measured in degrees centigrade, pressure, or voltage.

Usually, in this text, parentheses will be used for analog signals, and square brackets will be used for digital signals. Also, the variable $t$ will normally be a continuous variable denoting time, while $n$ will be discrete and will represent sample number. It can be thought of as an index, or array offset. Therefore, $x(t)$ will represent a continuous signal, but $x[n]$ will represent a discrete signal. We could let $t$ have values such as 1.0, 2.345678, and 6.00004, but $n$ would be limited to integer values like 1, 2, and 6. Theoretically, $n$ could be negative, though we will limit it to positive integers {0, 1, 2, 3, etc. } unless there is a compelling reason to use negatives, too. The programming examples present a case where these conventions will not be followed. In MATLAB, as in other programming languages, the square brackets and the parentheses have different meanings, and cannot be used interchangeably.

Any reader who understands programming should have no trouble with signals. Typically, when we talk about a signal, we mean something concrete, such as *mySignal* in the following C++ program.

```
#include <iostream>

using namespace std;

int main()
   float mySignal[] =  4.0, 4.6, 5.1, 0.6, 6.0 ;
   int n, N=5;

   // Print the signal values
   for (n=0; n<N; n++)
      cout << mySignal[n] << ' ';
   cout << endl;

   return  0;
```

For completeness, we also include below a MATLAB program that does the same

thing (assign values to an array, and print them to the screen).

```
>> mySignal = [ 4.0, 4.6, 5.1, 0.6, 6.0 ];
>> mySignal

mySignal =

    4.0000    4.6000    5.1000    0.6000    6.0000
```

Notice that the MATLAB version is much more compact, without the need to declare variables first. For simplicity, most of the code in this book is done in MATLAB.

## 1.3 Analog Versus Digital

As mentioned earlier, there are two kinds of signals: analog and digital. The word analog is related to the word analogy; a continuous ("real world") signal can be converted to a different form, such as the analog copy seen in Figure 1.3. Here we see a representation of air pressure sensed by a microphone in time. A cassette tape recorder connected to the microphone makes a copy of this signal by adjusting the magnetization on the tape medium as it passes under the read/write head. Thus, we get a copy of the original signal (air pressure varying with time) as magnetization along the length of a tape. Of course, we can later read this cassette tape, where the magnetism of the moving tape affects the electricity passing through the read/write head, which can be amplified and passed on to speakers that convert the electrical variations to air pressure variations, reproducing the sound.

An analog signal is one that has continuous values, that is, a measurement can be taken at any arbitrary time, and for as much precision as desired (or at least as much as our measuring device allows). Analog signals can be expressed in terms of functions. A well-understood signal might be expressed exactly with a mathematical function, or approximately with such a function if the approximation is good enough for the application. A mathematical function is very compact and easy to work with. When referring to analog signals, the notation $x(t)$ will be used. The time variable $t$ is understood to be continuous, that is, it can be any value: $t = -1$ is valid, as is $t = 1.23456789$.

A digital signal, on the other hand, has discrete values. Getting a digital signal from an analog one is achieved through a process known as *sampling*, where values are measured (sampled) at regular intervals, and stored. For a digital signal, the values are accessed through an index, normally an integer value. To denote a digital

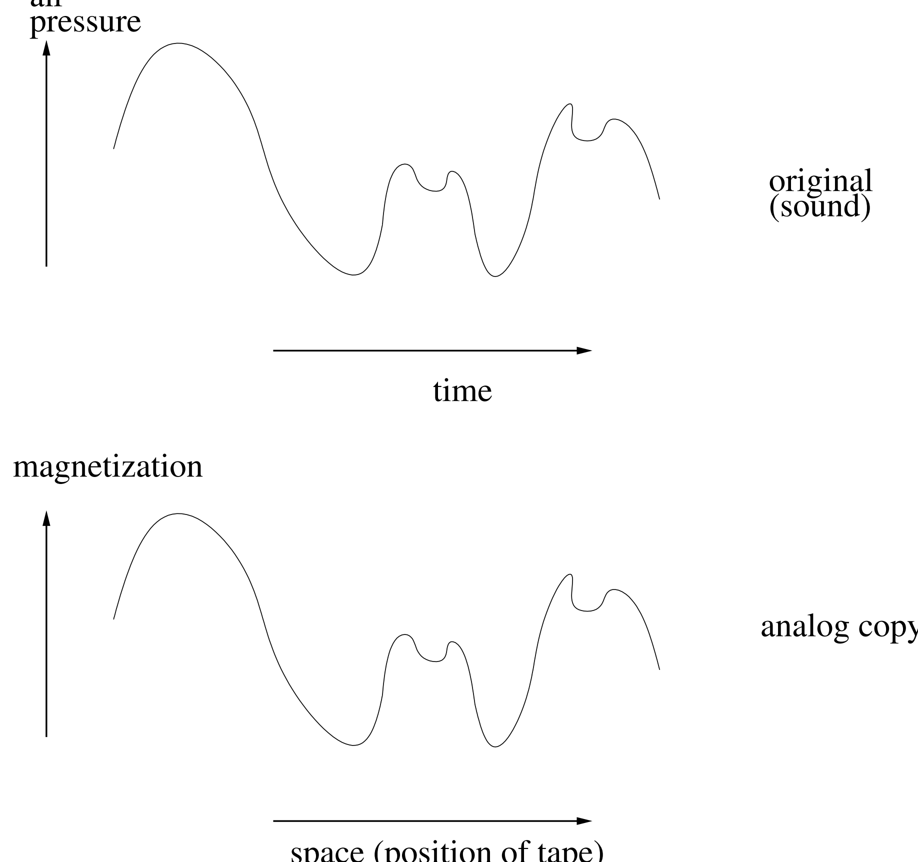



**Figure 1.3:** A sound signal with a tape analog.

signal, $x[n]$ will be used. The variable $n$ is related to $t$ with the equation $t = nT_s$, where $T_s$ is the sampling time. If you measure the outdoor temperature every hour, then your sampling time $T_s = 1$ hour, and you would take a measurement at $n = 0$ (0 hours, the start time), then again at $n = 1$ (1 hour), then at $n = 2$ (2 hours), then at $n = 3$ (3 hours), etc. In this way, the signal is *quantized* in time, meaning that we have values for the signal only at specific times.

The sampling time does not need to be a whole value, in fact it is quite common to have signals measured in milliseconds, such as $T_s = 0.001$ seconds. With this sampling time, the signal will be measured every $nT_s$ seconds: 0 seconds, 0.001 seconds, 0.002 seconds, 0.003 seconds, etc. Notice that $n$, our index, is still an integer, having values of 0, 1, 2, 3, and so forth. A signal measured at $T_s = 0.001$ seconds is still quantized in time. Even though we have measurements at 0.001 seconds and 0.002 seconds, we do not have a measurement at 0.0011 seconds.

Figure 1.4 shows an example of sampling. Here we have a (simulated) continuous curve shown in time, top plot. Next we have the sampling operation, which is like multiplying the curve by a set of impulses which are one at intervals of every 0.02 seconds, shown in the middle plot. The bottom plot shows our resulting digital signal, in terms of sample number. Of course, the sample number directly relates to time (in this example), but we have to remember the time between intervals for this to have meaning. In this text, we will start the sample number at zero, just like we would index an array in C/C++. However, we will add one to the index in MATLAB code, since MATLAB indexes arrays starting at 1.

Suppose the digital signal $x$ has values $x[1] = 2$, and $x[2] = 4$. Can we conclude that $x[1.5] = 3$? This is a problem, because there is no value for $x[n]$ when $n = 1.5$. Any interpolation done on this signal must be done very carefully! While $x[1.5] = 3$ may be a good guess, we cannot conclude that it is correct (at the very least, we need more information). We simply do not have a measurement taken at that time.

Digital signals are *quantized* in amplitude as well. When a signal is sampled, we store the values in memory. Each memory location has a finite amount of precision. If the number to be stored is too big, or too small, to fit in the memory location, then a truncated value will be stored instead. As an analogy, consider a gas pump. It may display a total of 5 digits for the cost of the gasoline; 3 digits for the dollar amount and 2 digits for the cents. If you had a huge truck, and pumped in $999.99 worth of gas, then pumped in a little bit more (say 2 cents worth), the gas pump will likely read $000.01 since the cost is too large a number for it to display. Similarly, if you went to a gas station and pumped in a fraction of a cent's worth of gas, the cost would be too small a number to display on the pump, and it would likely read $000.00. Like the display of a gas pump, the memory of a digital device has a finite amount of precision. When a value is stored in memory, it must not be too large

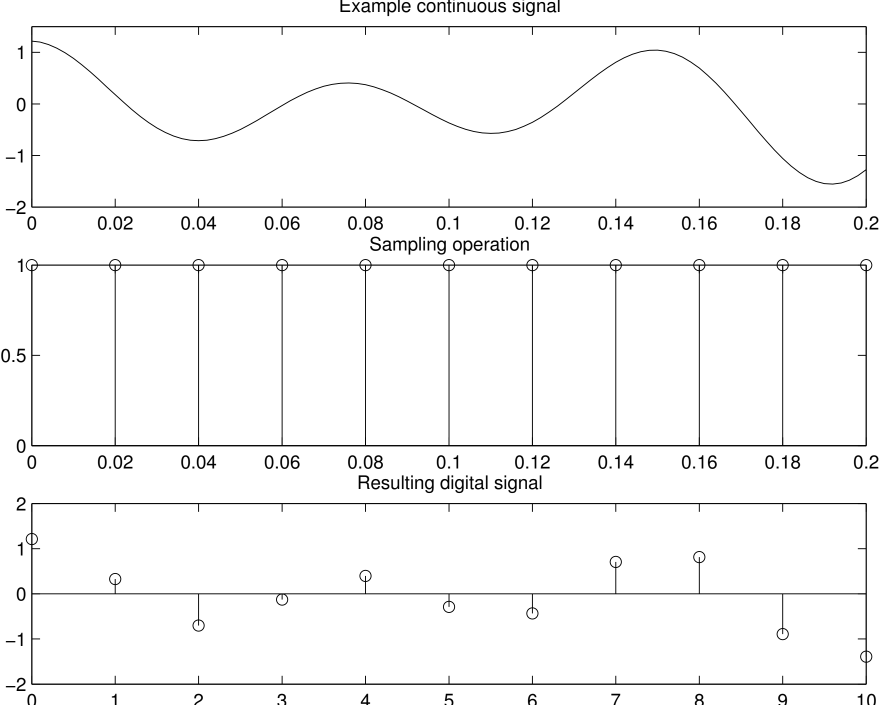



**Figure 1.4:** Sampling a continuous signal.

nor too small, or the amount that is actually stored will be cut to fit the memory's capacity. This is what is meant by quantization in amplitude. Incidentally, since "too small" in this context means a fractional amount below the storage ability of the memory, a number could be both too large and "too small" at the same time. For example, the gas pump would show \$000.00 even if \$1000.004 were the actual cost of the gas.

Consider the following C/C++ code, which illustrates a point about precision.

```
unsigned int i;
i=1;
while (i != 0) {
    /* some other code */
    i++;
}
```

This code appears to be an infinite loop, since variable $i$ starts out greater than 0 and always increases. If we run this code, however, it will end. To see why this happens, we will look at internal representation of the variable $i$. Suppose our computer has a word size of only 3 bits, and that the compiler uses this word size for integers. This is not realistic, but the idea holds even if we had 32, 64, or 128 bits for the word size instead. Variable $i$ starts out at the decimal value 1, which, of course, is 001 in binary. As it increases, it becomes 010, then 011, 100, 101, 110, and finally 111. Adding 1 results in a 1000 in binary, but since we only use 3 bits for integers the leading 1 is an overflow, and the 000 will be stored in $i$. Thus, the condition to terminate the `while` loop is met. Notice that this would also be the case if we remove the `unsigned` keywords, since the above values for $i$ would be the same, only our interpretation of the decimal number that it represents will change.

Figure 1.5 shows how a signal can appear in three different ways; as an analog signal, a digital signal, and as an analog signal based upon the digital version. That is, we may have a "real world" signal that we digitize and process in a computer, then convert back to the "real world." At the top, this figure shows how a simulated analog signal may appear if we were to view it on an oscilloscope. If we had an abstract representation for it, we would call this analog signal $x(t)$. In the middle graph of this figure, we see how the signal would appear if we measured it once every 10 milliseconds. The digital signal consists of a collection of points, indexed by the sample number $n$. Thus, we could denote this signal as $x[n]$. At the bottom of this figure, we see a reconstructed version of the signal, based on the digital points. We need a new label for this signal, perhaps $x'(t)$, $\tilde{x}(t)$, or $\hat{x}(t)$, but these symbols have other meanings in different contexts. To avoid confusion, we can give it a new name altogether, such as $x_2(t)$, $x_{reconstructed}(t)$ or $new_x(t)$. While the reconstructed

signal has roughly the same shape as the original (analog) signal, differences are noticeable. Here we simply connected the points, though other (better) ways do exist to reconstruct the signal.

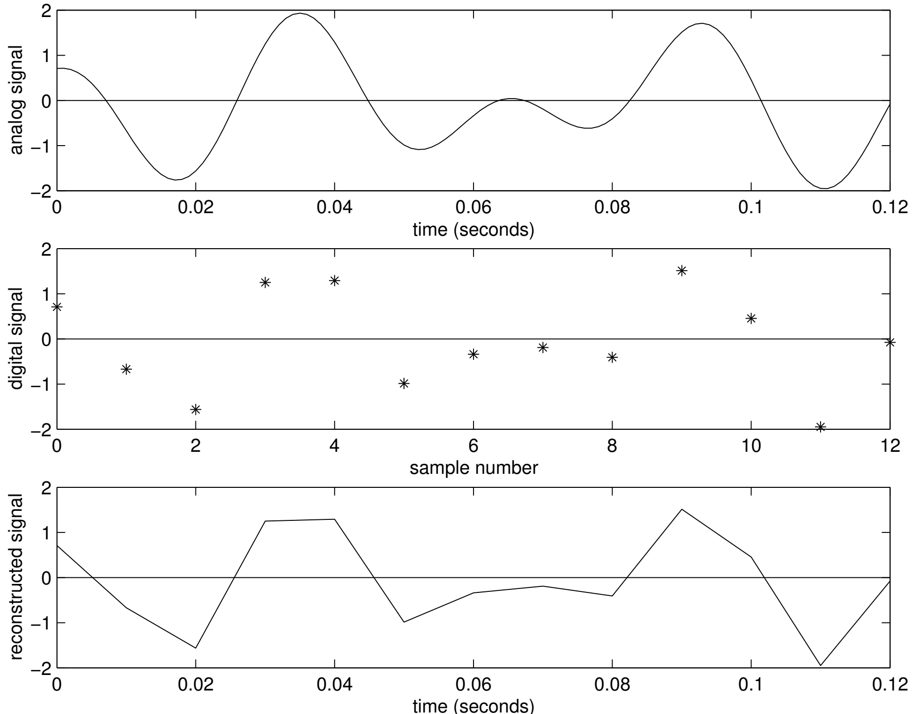



**Figure 1.5:** Three ways of viewing a signal.

It is possible to convert from analog to digital, and digital to analog. Analog signal processing uses electronics parts such as resistors, capacitors, operational amplifiers, etc. Analog signal processing is cheaper to implement, because these parts tend to be inexpensive. In DSP, we use multipliers, adders, and delay elements (registers) to process the signal. DSP is more flexible. For example, error detection and error correction can easily be implemented in DSP.

## 1.4 What Is a System?

A *system* is something that performs an operation (or a transform) on a signal. A system may be a physical device (hardware), or software. Figure 1.6 shows an abstraction of a system, simply a "black box" that produces output $y[n]$ based upon

input $x[n]$. A simple example system is an incrementer that adds 1 to each value of the signal. That is, suppose the signal $x[n]$ is $\{1, 2, 5, 3\}$. If $y[n] = x[n] + 1$, then $y[n]$ would be $\{2, 3, 6, 4\}$.

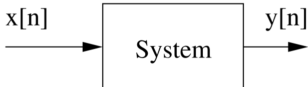



**Figure 1.6:** An example system.

Examples of systems include forward/inverse Fourier transformers (Chapter 6, "The Fourier Transform"), and filters such as the Finite Impulse Response filter and the Infinite Impulse Response filter (see Chapter 3, "Filters").

## 1.5 What Is a Transform?

A *transform* is the operation that a system performs. Therefore, systems and transforms are tightly linked. A transform can have an inverse, which restores the original values.

For example, there is a cliché that an optimist sees a glass of water as half full while a pessimist sees it as half empty. The amount of water in the glass does not change, only the way it's represented. Suppose a pessimist records the amount of water in 3 glasses, each of 1 liter capacity, as 50% empty, 75% empty, and 63% empty. How much water is there in total? To answer this question, it would be easy to add the amount of water in each glass, but we do not have this information. Instead, we have measures of how much capacity is left in each. If we put the data through the "optimist's transform," $y[n] = 1 - x[n]$, the data becomes 50%, 25%, and 37%. Adding these number together results in 112% of a liter, or 1.12 liters of water.

A transform can be thought of a different way of representing the same information.

How full are the glasses of water in Figure 1.7? An optimist would say that they are half full, one-quarter full, and one-tenth full, respectively.

A pessimist would say the water is 1−(optimist) empty, e.g., .5 empty, .75 empty, and .90 empty. So we can convert back and forth between the two ways of specifying the same information. Sometimes one way of looking at the information is better than another. For example, if you want to know the total amount in all three glasses, the optimist's way of seeing things would be easier to work with. However, if you needed how much more to pour in each glass to fill them, the pessimist's way of measuring the amounts is better.

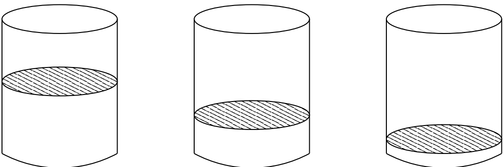


**Figure 1.7:** Three glasses of water.

Suppose that we have a simple incrementer system that adds 1 to every input. We will use $x_1[n]$ to represent the input signal, and $y_1[n]$ to represent the output. We can write the output in terms of the input, i.e., $y_1[n] = x_1[n]+1$. If $x_1$ is the sequence $\{1, 3, 7\}$, the output $y_1$ would be the sequence $\{2, 4, 8\}$. Now suppose we want to get the original values of $x_1$ back. To do this, we need to "undo" the transform, which we can do with the following "inverse" transform. Since we have given a name to the reverse transform, we will call the first transform ($y_1[n] = x_1[n] + 1$) the "forward" transform.

The inverse transform for the incrementer system would be a decrementer, that simply subtracts 1 from each value, i.e., $y_2[n] = x_2[n] - 1$. If we hooked these two systems up in series, we would have the output of the first system as the input to the second, effectively assigning $x_2[n] = y_1[n]$. So $y_2[n] = x_2[n] - 1$, $y_2[n] = y_1[n] - 1$, or $y_2[n] = (x_1[n] + 1) - 1$. It is easy to see that the two systems cancel each other out, and the final output ($y_2[n]$) is what we started with originally ($x_1[n]$).

Typically, we do not study systems that add or subtract constants.

A natural question is, "Why would we do this?" One answer is so that we can analyze the transformed signal, or compress it. This does not follow from such a simple example, but more complicated transforms (such as the discrete cosine transform) have been used effectively to allow a compression program to automatically alter the signal to store it in a compact manner.

Sometimes transforms are performed because things are easier to do in the transformed domain. For example, suppose you wanted to find the number of years between the movies *Star Wars* (copyright MCMLXXVII) and *Star Wars: Episode II—Attack of the Clones* (copyright MMII). Subtracting one number from another is not a problem, but Roman numerals are difficult to work with [7]! The algorithm we use of subtracting the right-most column digits first, writing the difference and borrowing from the left, does not work. Instead, the easiest thing to do is convert the Roman numerals to decimal numbers, perform the subtraction, and convert the result back to Roman numerals. In this case, the transform would convert MCMLXXVII to 1977, and MMII to 2002. Subtracting 1977 from 2002 gives us 25, which is the inverse-transformed to XXV. Performing the transform and later the inverse-

transform may seem like a waste of effort, but not if it greatly simplifies the steps in between.

If you have had a course in differential equations, you may have seen the Laplace transform used for just this purpose: to provide an easy way to solve these equations. After the equation is set up, you apply the Laplace transform (often by looking it up in a table), then the transformed equation can be manipulated and solved using algebra, and finally the result is inverse-transformed back to the time-domain (again, often by looking at a table). The Laplace transform also has a use in determining the stability of systems, and this is something we will return to in a later chapter.

## 1.6 Why Do We Study Sinusoids?

We look at sinusoidal functions (sine and cosine) because they are interesting functions that often appear in the "analog" world. Examining a single cosine function is easy to do, and what applies to one cosine function also applies to a signal composed of several sinusoids. Since sinusoidal functions occur frequently, it is nice to be able to use a few pieces of data to represent a sinusoid. Almost every sinusoid in this text will be of the following format:

$$\text{amplitude} \times \cos(2\pi \times \text{ frequency } \times \text{ t } + \text{ phase}).$$

That is all! When the amplitude, frequency, and phase are known, we have all the information we need to find the value of this sinusoid for *any* value of time ($t$). This is a very compact way of representing a signal. If the signal is more complex, say it is composed of two sinusoids, we just need the amplitude, frequency, and phase information of the two sinusoids. In this way, we can represent the signal, then later remake it from this information.

Think of how the earth rotates around the sun in a year's time. The earth gets the same amount of sunlight every day, but the earth's tilt means that one hemisphere or the other receives more sun. Starting at the December solstice (around December 22), the earth's Northern hemisphere receives the least amount of sunlight [8]. As the earth rotates around the sun, this hemisphere gets an increasing amount of sunlight (spring time), until it reaches the maximum (summer), then the days get shorter (fall), and it finally returns to the low point again in winter. The seasons vary with time. In two dimensions, one could diagram this with the sun in the center, and the earth following a circular path around it (approximately). Another way of viewing this information is to graph the distance with time as the x-axis. For the y-axis, consider the horizontal distance between the two. This graph looks like a sine wave. The seasons change with time, but they always repeat. Therefore, this signal is periodic.

Examples of sinusoidal signals are all around us. The rotation of the moon around the earth, the temperature of a city over a day's time, and the acoustic waves made by human speech are all examples of sinusoidal-like signals. Human speech is interesting, because it may be a single sinusoid (such as a vowel sound), or a composite signal made up of several sinusoids.

Composite signals are important in theory as well as practice. Any function that is not truly random can be approximated with a sum of sinusoids. In other words, a signal can be broken down into a combination of simpler signals. As a mathematical abstraction, the composite signal

$$x(t) = \cos(2\pi 100t) + \cos(2\pi 200t)$$

can easily be plotted in MATLAB. The example below will work, though it is inefficient.

```
Ts = 0.0002;    % sampling time, Ts, is .2 ms
for n = 1:100   % we will have 100 samples
    x(n) = cos(2*pi*100*n*Ts) + cos(2*pi*200*n*Ts);
end
plot(x);
```

A better way is to eliminate the `for` loop, as shown below.

```
Ts = 0.0002;  % sampling time, Ts, is .2 ms
n = 1:100;    % we will have 100 samples
x = cos(2*pi*100*n*Ts) + cos(2*pi*200*n*Ts);
plot(n, x, 'b');
```

The code above adds the two cosine signals together, point by point, and stores the results in a vector $x$ [6] [9] [10]. We can use $n$ in place of `1:length(x)` for this particular example, if we want. Also, using $x$ would work here just as well as `x(n)`.

Notice that the MATLAB example above does not actually plot $x(t)$. It plots $x(t = nT_s)$, where $n$ is an integer 1..1000. $T_s$ is the sampling time, which is an important value that we will look into more closely in a later chapter. We could simulate $x(t)$, however, it would still be a digital signal since variable $t$ has a limited set of values. Example code appears below.

```
% simulated x(t)
t = 0:0.0002:0.02;
x = cos(2*pi*100*t) + cos(2*pi*200*t);
plot(1:length(x), x, 'b');
```

The `1:length(x)` parameter specifies an array that has the same number of elements as $x$ does, which can be seen along the x-axis. A final example of the plot command follows:

```
plot(t, x, 'b');
```

Notice how the x-axis now reflects time instead of sample number.

When several sinusoid components are in the same signal, and each of the frequencies is an integer multiple of a common frequency, we say that the frequencies are harmonically related.

A "truly random" function cannot be represented well with a sum of sinusoids. Such a function would not repeat itself, but would go on forever as an unpredictable value for any time value. This is similar to an irrational number. Most values are rational, that is, can be represented as a division of one integer number by another, for example, the quantity 1.5 is a rational number since it is the same as 3/2. It could also be represented as 15/10, if desired, just as 1.4 could be represented as 14/10, and 1.3 could be represented as 13/10. In fact, any number that has only 1 digit past the decimal point could be written as an integer over 10. In a similar line of reasoning, any number with 2 digits past the decimal could be written as an integer over 100, e.g., $1.23 = 123/100$. We can keep going with this, even with an extreme case like $1.23456789 = 123456789/10000000$. Since our denominator (the number on the bottom) can be as large as we like, it may seem like EVERY number can be represented in such a way, and is therefore rational. But there are a few numbers that are irrational, such as $\pi$. You can write as many digits for $\pi$ as you like, and someone could come along after you and add another digit. It does not repeat, as far as we know, even when computed out to millions of digits. Truly random functions are like this—they do exist, but we can deal with them the way we deal with $\pi$, by approximation. One could represent $\pi$ as the rational number 3.14159. This is not *TRULY* $\pi$, but it is close enough for most applications. In a similar fashion, one could find an approximation for a truly random signal that would be judged "good enough" for the application.

## 1.7 Sinusoids and Frequency Plots

Suppose you had the following sequence of numbers, with little information about them: {59, 65, 68, 70, 61, 61, 63, 66, 66, 67, 70, 71, 71, 70, 70, 70}. You could calculate the average, the variance, and what the minimum and maximum values are. If you had to guess what the signal is likely to be, measured at an arbitrary time, 66 seems to be a good guess. Now suppose you get the information that these

values correspond to temperature readings in degrees Fahrenheit. These numbers could be used to decide how to dress for the climate, and you might conclude that long-sleeved garments would be warm enough, perhaps with a sweater. Now suppose that these readings were taken in Atlanta during the month of August. How can this be? Atlanta is known to have a hot climate, especially in August. There is one key piece of information that is missing: that these temperatures were taken every day at 2 a.m. Now it all makes sense, that the temperatures were read at a time when they are likely to be their lowest.

Taking samples too infrequently leads to poor conclusions. Here one might assume that the temperatures fluctuate by only a few degrees. Or one could assume that the temperatures above are indicative of the climate, instead of being an extreme. We expect that temperature will rise in the afternoon and fall in the nighttime, not unlike a sine function. But this example shows how we can miss the overall rise and fall trend by sampling at regular intervals that happen to coincide with the low points. If we were to sample a bit more frequently, say every 23 hours starting with an initial read at 2 a.m., this new set of readings would not be better. They would make it appear as if the temperature gradually gets warmer (as the sampling time becomes earlier: first 2 a.m., then 1 a.m., then midnight, then 11 p.m., etc., eventually reaching the afternoon), then gradually gets cooler (when the samples get earlier and earlier in the morning). A better sampling rate would be every 12 hours or less, since we would get a much better idea of how the temperature fluctuates. At worst, we would read temperatures right in the middle of the extremes, but we would still have a good idea of the average.

We use both time and frequency in language to view the same information in different ways. For example, you may ask "how long is it before I have to change the oil in my car?" in which case the answer would be in terms of time (months). A related question, "how many times a year do I need to change the oil in my car?" results in a frequency-based answer (oil changes per year). Time and frequency have an inverse relationship in language as well as in DSP.

We are used to thinking about frequency in some contexts. When you move to a new town, you are likely to set up your stereo and scan the channels. Moving the dial as far as it will go to the left, you slowly turn it until you hear something. If you like what you hear, you make note of the dial's location, and move it to the right. When finished, you have essentially made a graph of the spectrum: where you found nonzero energy for the dial's location, you marked the frequency.

You may wonder why radio stations often have numbers past the decimal point, such as 88.5 instead of just 88 or 89. These numbers are in terms of MHz, or millions of Hertz, and the extra digit helps you tune in a good signal. After you graduate and get a great job, and see that you have $1.3M in your bank account, think back

to this example, and how the .3 is significant to you! You may have noticed how FM radio stations end in odd numbers, at least in the United States. The U.S. allocated the FM spectrum in 200 kHz chunks, starting on an odd boundary. So you might have a local 88.5 MHz station, but not one at 88.6 MHz.

Now imagine instead that you have a measure of all the energy of the radio spectrum for an instant in time. How could you use this to tell which radio station to listen to? You cannot use it for this purpose. Though you are used to signals in terms of time, there are some cases in which the frequency information makes things much easier.

## 1.8 Summations

In Digital Signal Processing, you are likely to run into formulas containing summations and other seemingly complicated things. Summations are nothing to be afraid of. For example, the following formula and the C/C++ code below it accomplish the same task. This example summation is just a compact way of expressing an idea: for each number in $x$, multiply by two and subtract one, add all the results together, and store the final answer in the variable $s$. Let $x = \{1.4, 5.7, 3.0, 9.2, 6.8\}$, and let $N$ be the number of values in $x$.

$$s = \sum_{i=0}^{N-1} 2x[i] - 1$$

Here is part of a program in C/C++ to do the same thing:

```
int i;
float x[] = {1.4, 5.7, 3.0, 9.2, 6.8};
int N=5;  // the number of values in x[]
float s = 0;

for (i=0; i<N; i++)
   s = s + 2*x[i] - 1;
```

If you find the mathematical notation confusing, then practice a few examples of converting sums to programs. After a while, you should be comfortable with both ways.

In MATLAB, we could do the following:

```
x = [1.4, 5.7, 3.0, 9.2, 6.8];
```

```
s = 0;
for i=1:length(x)
   s = s + 2*x(i) - 1;
end
```

Even more compactly, we could accomplish the summation in MATLAB with:

```
x = [1.4, 5.7, 3.0, 9.2, 6.8];
s = sum(2*x - 1);
```

MATLAB will be covered in more detail in Chapter 2.

## 1.9 Summary

This chapter covers the concepts of numbers, from counting to a number line, to negative numbers, infinity and negative infinity, fractional numbers, irrational numbers, and finally complex numbers ($\sqrt{-1}$ being the y-axis). Topics also covered include polar coordinates, complex exponentials, continuous versus discrete, digital versus analog, signals, systems, transforms, sinusoids, and composite signals. Coverage of these topics will be expanded in the following chapters.

Signals are the data that we work with, and digital signals can be thought of as an array (or matrix). Online systems would have data in streams, but we typically do not discuss these to keep things simple. Transforms can be used for analysis, and are often used in compression applications.

## 1.10 Review Questions

1. What is a signal?

2. What is a system?

3. What is a transform?

4. What does $x[n]$ imply?

5. What does $x(t)$ imply?

6. What is the decimal value of the binary fixed-point number 0101.01? (show work)

7. Represent 75.625 as a fixed-point binary number.

8. Represent 36.1 as a fixed-point binary number.

9. Convert the Cartesian point (2, 5) to polar coordinates.

10. Convert the polar coordinates 4, 25 degrees to Cartesian coordinates.

11. Explain what we mean by quantization in time, and quantization in amplitude.

12. Why are sinusoids useful in digital signals?

13. Why is there a problem using $\theta = \arctan(b/a)$ for conversion to polar coordinates?

14. What are the differences between analog and digital?

# Chapter 2

# MATLAB

MATLAB is a programming environment which has gained popularity due to its ease of use. The purpose of this chapter is not to teach MATLAB programming, but it is intended to provide a quick reference to MATLAB. It is easy to understand if you already know a programming language; if you do not, the following examples should help. Whether you are an experienced programmer or a beginner, the best thing to do is try the examples on your computer. MATLAB provides an interactive environment that makes this easy: you type a command, and it executes it. If there is a problem, it will let you know, and you can try again. If a command works well, you can copy and paste it into a file, creating a complex program one step at a time.

In fact, we will consider any group of commands that are stored in a file to be a program. The only distinction between a program and a function is the "`function`" keyword used in the first (noncomment) line in a file. This line specifies the inputs and outputs, as well as the function name. It is a good idea to give the function the same name as the file that it is stored in. The filename should also have ".m" as the extension. An example function will follow shortly.

MATLAB provides an editor, which will color text differently based on its functionality. Under this editor, comments appear in green, strings appear in red, and keywords (such as `if`, `end`, `while`) appear in blue. This feature helps the programmer spot errors with a command, such as where an unterminated string begins. This editor is not required, however, because any text editor will work. Here is one frustrating mistake that programmers occasionally make: if a program or function is changed in the editor, but not saved to disk, then the *old* version of it will be used when it is executed. Make sure to save your work before trying to run it!

## 2.1 Working with Variables

A common programming mistake is to use a variable without declaring it first. This is not a problem in MATLAB. In fact, MATLAB is very flexible about variables and data types. If an undefined variable is assigned a value, for example:

```
a = 1;
```

MATLAB will assume that the variable $a$ should be created, and that the value 1 should be stored in it. If the variable already exists, then it will have the number 1 replace its previous value.

This works with scalars, vectors, and matrices. The numbers stored in variables can be integers (e.g., 3), floating-points (e.g., 3.1), or even complex (e.g., $3.1+5.2j$).

```
a = 3;
b = 3.1;
c = 3.1+5.2j;
```

A semicolon ";" is often put at the end of a command. This has the effect of suppressing the output of the resulting values. Leaving off the semicolon does not affect the program, just what is printed to the screen, though printing to the screen can make a program run slowly. If a variable name is typed on a line by itself, then the value(s) stored for that variable will be printed. This also works inside a function or a program. A variable on a line by itself (without a semicolon) will be displayed.

```
>> a

a =

     3

>> b

b =

    3.1000

>> c
```

```
c =

   3.1000 + 5.2000i
```

MATLAB shows the complex value stored in $c$ with $i$ representing the complex part (the square root of $-1$). Notice, though, that it also understands $j$, as the assignment statement for $c$ shows. Both $i$ and $j$ can be used as variables, just as they are in other languages. It is advisable to use $k$ instead, or some other variable name, to avoid confusion.

It is a good idea to use uppercase variable names for matrices, and lowercase names for scalars. This is not a requirement; MATLAB does not care about this. But it is case sensitive; that is, the variable $x$ is considered to be different from the variable $X$.

One final note about functions is that the variables used in a function are not accessible outside of that function. While this helps the programmer maintain local variables, it also means that, if the function produces an error, the programmer will not be able to simply type a variable's name at the prompt to find its value. A quick fix for this is to comment out the function's declaration line, then call it again. Alternatively, MATLAB provides breakpoints and other programming interface features that you would expect from a software development kit.

## 2.2 Getting Help and Writing Comments

The `help` command is very useful. Often, a programmer will remember the function he or she wants to use, but first want to verify the name, and then figure out what the parameters are, and the order of the parameters. The `help` command is useful for this purpose. Just type `help` at the prompt, followed by the function name, for example, `help conv` gives information on the convolution function (`conv`).

When creating a function or program, it is important to document the code. That is, make sure to include enough comments so that someone else can figure out what the code does. This "someone else" could even be yourself, if you make a program and enough time passes before you need to use it again. Comments in MATLAB start with a percent sign "%," and the comment lasts until the end of the line, just like the C++ style comment (of two forward slashes). A comment can appear on the same line as a command, as long as the command is first (otherwise, it would be treated as part of the comment).

It is a good idea to put several lines of comments at the top of your programs and functions. Include information like the order of the inputs, the order of the outputs,

and a brief description of what the code does. The nice thing about MATLAB is that the comments at the top of your file will be returned by the help function.

Try this: Type `help xyz` at the MATLAB prompt. It should tell you that no such function exists.

```
>> help xyz

xyz.m not found.
```

Type the function below, and save it as "xyz.m":

```
%
% This is my function. It returns the input as the output.
%
%   Usage:   a = xyz(b)
%
function [a] = xyz(b)

a=b;
```

Now try `help xyz` again. You should see the first few lines printed to the screen. Be sure to put the percent sign in the left-most column if you want these comments to be found by the help facility.

```
>> help xyz


  This is my function. It returns the input as the output.

    Usage:   a = xyz(b)
```

## 2.3 MATLAB Programming Basics

This section covers the main things that you need to know to work with MATLAB. The command `clc` clears the command window, and is similar to the `clear` command in the Unix environment. The command `clear`, by the way, tells MATLAB

to get rid of all variables, or can be used to get rid of a specific variable, such as `clear A`. Clearing a variable is a good idea when changing the number of elements in an array variable. To see a list of the variables in memory, type `who`. The similar command `whos` gives the same list, but with more information.

```
>> A = 3;
>> who

Your variables are:

A

>> whos
  Name      Size                    Bytes  Class

  A         1x1                         8  double array

Grand total is 1 element using 8 bytes

>> clear
>> who
>>
```

In the example above, we see that the variable $A$ is defined, and the difference between `who` and `whos`. After the `clear` statement, the `who` command returns nothing since we got rid of the variables.

### 2.3.1 Scalars, Vectors, and Matrices

A variable can be given a "normal" (scalar) value, such as `b=3` or `b=3.1`, but it is also easy to give a variable several values at once, making it a vector, or even a matrix. For example, `c=[1 2]` stores both numbers in variable $c$. To make a matrix, use a semicolon (";") to separate the rows, such as: `d=[1 2; 3 4; 5 6]`. This makes $d$ a matrix of 3 rows and 2 columns. To access one of the values, use two indices, e.g., `d(2,2)` returns the value 4.

```
>> d=[1 2; 3 4; 5 6]

d =
```

```
     1     2
     3     4
     5     6
```

What if you want to add another row onto $d$? This is easy, just give it a value: `d(4,:)=[7 8]`. Notice the colon ":" in the place of the column specifier. This is intentional, it is a little trick to specify the whole range. We could have used the command `d(4,1:2)=[7 8]` to say the same thing, but the first version is easier (and less typing).

```
>> d(4,:)=[7 8]

d =

     1     2
     3     4
     5     6
     7     8
```

What if we want to select column 1 only? The following code does this.

```
>> d(:,1)

ans =

     1
     3
     5
     7
```

Compare the previous output to the following command.

```
>> d(1,:)

ans =

     1     2
```

The last two examples show how to access column 1 and row 1, respectively.

### 2.3.2 Number Ranges

A variable can hold a range of numbers. For example, imagine that variable $N$ should hold all numbers 1, 2, 3, 4, etc., up to the number 100. Here is one way to create $N$:

```
for index=1:100
   N(index)=index;
end
```

This does the job, but there is a faster and simpler way to say the same thing. Below, we set $N$ equal to a range, specified with a colon ":" symbol between the starting and ending values.

```
N=1:100;
```

If a third number is specified, then the second one will be used as the increment. For example, `M=0:0.4:2` would assign the sequence $\{0, 0.4, 0.8, 1.2, 1.6, 2\}$ to variable $M$. The following example shows all numbers between 8 and 20, with an increment of 2.

```
>> 8:2:20

ans =

     8    10    12    14    16    18    20
```

### 2.3.3 Output

Often, during a program's run, the program needs to convey information to the user. To print a string or a number to the screen, the `sprintf` command is quick and simple.

```
>> sprintf('The value of pi is %5.3f', pi)

ans =

The value of pi is 3.142
```

Notice that there is no semicolon at the end of the above command. The reason for this is that it would suppress the output, but here we want the output to go to the user's screen. Of course, `sprintf` can also be used to create a string to be stored in a variable.

Like all other programming languages, there is always more than one way to do the same thing. The `disp` command can also display information to the screen. It does not matter if the `disp` command is followed by a semicolon. It can be used in conjunction with `sprintf`, as the following example shows.

```
>> disp('this is a string')
this is a string
>> disp('this is a string');
this is a string
>> disp(sprintf('The value of pi is %5.3f',pi))
The value of pi is 3.142
```

The single quotes specify a string. Notice how the following output changes depending on the presence of the quotes.

```
>> hello = 5;
>> disp('hello')
hello
>> disp(hello)
     5
```

### 2.3.4 Conditional Statements (if)

Like most programming languages, the way to execute code conditionally is with the `if` statement. In the following example, the division is done only when the denominator does not equal zero. Dividing by zero is one common mistake that can stop a program.

```
% Look out for division by 0
if (denominator ~= 0)
   result = numerator / denominator;
end
```

Here is another way of handling this. In the following example, the `if` statement either performs the `sprintf`, or else it performs the division, but it will not do both.

Notice how a single equal sign would indicate assignment, e.g., `a=1`, but when comparing two values, two equal signs are used, e.g., `if (a==1)`. This way it is clear to the computer which operation to perform (assignment or comparison). Unfortunately, this is a common source of error in a program.

```
% Look out for division by 0
if (denominator == 0)
   sprintf('Cannot perform division since denominator is 0')
else
   result = numerator / denominator;
end
```

As might be expected, `if` statements can be nested. For example, the following code checks a pair of indices ($m$ and $n$) before allowing them to be used to access a matrix's value. This kind of check is useful to make sure that the program does not crash. Before running this code, however, values for $m$, $n$, and $A$ (the matrix) need to be defined.

```
A = [1 2; 3 4; 5 6; 7 8];
m=0;
n=1;
```

Now try the following code. Note that `if and(m>0, n>0)` would be equivalent to `if ((m>0) && (n>0))` in C/C++. Experiment with different values for $m$ and $n$. An easy way to try this repeatedly is to use MATLAB's built-in editor, and save these commands in a file (for example, "checkindices.m"). When you want to run it, simply type the filename without the extension (e.g., `checkindices`) at the command prompt. The code does have a precondition, in that it expects $A$, $m$, and $n$ to be defined before this code is executed.

```
% Check that the indices m and n are valid
[rows, cols] = size(A);
if and(m>0, n>0)
   if and(m<=rows, n<=cols)
      sprintf(' A ( %d, %d ) = %d', m, n, A(m,n))
   else
      sprintf('One of the indices m or n is too big.')
   end
```

```
else
   sprintf('The indices m and n must be greater than 0.')
end
```

Also, version 7 of MATLAB was enhanced to allow the `&&` operator to be used just like in C/C++. (This does not work with version 6.) Here is a short example.

```
>> array = [37    85    59];
>> index = 3;
>> if (index >=1 ) && (index <= length(array))
    array(index) = array(index)+2;
end
>> array

array =

    37    85    61
```

The `else` statement goes with the nearest `if`. Note that `else if` is treated differently than `elseif`. The command `else if` is treated as two separate statements; an `end` is expected for both the second `if` as well as the first. The command `elseif` is a continuation of the first `if`, much like a `switch/case` statement. The two examples below show how the `if` statement can be structured.

```
% One way to structure the if statement
if (a == 1)
    sprintf('a = 1')
else if (a == 2)
        sprintf('a = 2')
    end
end

% Another way to structure the if statement
if (a == 1)
    sprintf('a = 1')
elseif (a == 2)
    sprintf('a = 2')
end
```

Any `if` should have a corresponding `end`, unless it is an `elseif`. For clarity, it is better to use `elseif`, or to put `else` and `if` on different lines.

### 2.3.5 Loops

Repetition is common in programs, and MATLAB provides the `for` and `while` loops for this purpose. The `end` statement, naturally enough, marks the end of the loop.

```
count = 0;
while (count < 10)
   B(count+1) = count*2;
   count = count+1;
end
```

Alternatively, the following code does the same thing, as an example of a `for` loop.

```
for count=0:9
   B(count+1) = count*2;
end
```

One final point is that MATLAB is able to process array operations efficiently. Often, loops such as these can be eliminated, as the example below shows.

```
count=0:9;
B(count+1) = count*2;
```

### 2.3.6 Continuing a Line

In the MATLAB examples in this book, you may observe ellipses at the end of a line (that is, three periods in a row, "..."). This simply tells MATLAB that the command is continued on the next line, which is a necessity when observing maximum column widths in a print out.

## 2.4 Arithmetic Examples

The term *vector* has two meanings in this book, depending on the context. First, the word vector is used to mean the same thing as an array. Second, it can also mean a magnitude and angle forming a set of polar coordinates (also called a phasor,

especially if the angle changes, i.e., it spins like the hand on a watch, even if it's counterclockwise). The respective meaning should be apparent in the text. With matrices, order is important. For example, suppose you have matrices $A$, $B$, and $C$. Multiplying them together, $(AB)C$ is not the same as $A(BC)$. Fortunately, MATLAB takes care of the details of complex vector and matrix calculations.

Here are some examples of common arithmetic operations in MATLAB. First, we will define some variables to work with.

```
  To get started, select "MATLAB Help" from the Help menu.

>> A = [20 1 9 16]

A =

    20     1     9    16

>> A2 = [20, 1, 9, 16]   % Commas can separate entries

A2 =

    20     1     9    16
```

As we see above, adding commas after each element also works, and is a bit clearer.

```
>> B = [ 5 4 3 2]

B =

     5     4     3     2

>> C = [ 0 1 2 3; 4 5 6 7; 8 9 10 11]

C =

     0     1     2     3
     4     5     6     7
     8     9    10    11

>> D = [ 8; 12; 0; 6]    % Semicolons separate rows
```

```
D =

     8
    12
     0
     6

>> F = [ 31 30 29 28; 27 26 25 24; 23 22 21 20]

F =

    31    30    29    28
    27    26    25    24
    23    22    21    20
```

Semicolons can be used to separate rows of a matrix, as seen above, where all of the operations are assignment statements. Most of the examples below are not assignment statements; that is, the results are just printed to the screen and do not affect the above matrices.

The `size` command returns the number of rows and number of columns for a given matrix. We see here that $A$ and $B$ are both arrays, matrices with only one row. Meanwhile, $C$ is a matrix of three rows and four columns. $D$ is also an array, since it has only one column.

```
size(A)

ans =

     1     4

>> size(C)

ans =

     3     4

>> size(D)

ans =
```

```
     4     1
```

A variable followed by a period then the single-quote character indicates the transpose of that variable. The notation:

```
A.'
```

simply represents the transpose of the array $A$. We can obtain the same effect by using `transpose(A)` instead. The example below shows the transpose of $A$, that is, views it as a 4 by 1 matrix instead of a 1 by 4 matrix. We can verify this with the `size` function. On a matrix like $C$, the result is to swap the rows with the columns. Similarly, we see that $C$ is a 3 by 4 matrix, while the transpose of $C$ is 4 by 3.

```
>> A.'    % transpose A

ans =

    20
     1
     9
    16

>> size(A.')

ans =

     4     1

>> C.'    % transpose C

ans =

     0     4     8
     1     5     9
     2     6    10
     3     7    11

>> size(C.')
```

```
ans =

     4     3
```

Addition and subtraction can be carried out as expected. Below, we see that a constant can be added to every element of a matrix, e.g., `A+7`. We can add every element of two matrices together, as in `A+B`. Subtraction is easily done with `A-B` and `B-A`.

```
>> A+7

ans =

    27     8    16    23

>> A+B         % Note that A and B must be the same size.

ans =

    25     5    12    18

>> A-B

ans =

    15    -3     6    14

>> B-A

ans =

   -15     3    -6   -14
```

To find the summation of an array, use the `sum` function. It also works with two-dimensional matrices, though it will add all of the numbers along the columns and return an array. Notice below that there are two ways of expressing the same thing. The command `sum(A+B)` says to add matrix $A$ to matrix $B$, then find the sum of the result. The second way to do this, `sum(A) + sum(B)` says to find the sum of $A$ and the sum of $B$, then add the two sums together. As in any programming language,

there are often multiple ways to do the same thing. This becomes important when performance is poor; it may be possible to restate the operations in a more efficient way.

```
>> sum(A)

ans =

    46

>> sum(B)

ans =

    14

>> % Add A to B, then find their sum
>> sum(A+B)

ans =

    60

>> % find sum of A and sum of B, then add
>> sum(A) + sum(B)

ans =

    60
```

The next examples are of multiplication operations. Multiplication can be a bit tricky, since there are several similar operations. First, let us start by trying to multiply $A$ with $B$.

```
>> A*B
??? Error using ==> *
Inner matrix dimensions must agree.
```

We cannot multiply a $4 \times 1$ matrix with a $4 \times 1$ matrix! If we want to multiply two matrices together, we must make sure that the first matrix has as many columns as

the second matrix has rows. In this case, we must transpose either $A$ or $B$ in order to multiply it by the other. Note that when we multiply $A$ by the transpose of $B$, we are multiplying the $n^{\text{th}}$ elements together, and adding the results.

```
>> A.'*B   % Multiply transpose(A) by B

ans =

   100    80    60    40
     5     4     3     2
    45    36    27    18
    80    64    48    32

>> A*B.'   % Multiply A by transpose(B).

ans =

   163

>> A(1)*B(1) + A(2)*B(2) + A(3)*B(3) + A(4)*B(4)

ans =

   163
```

What if we want to multiply the $n^{\text{th}}$ elements together, but not add the results? This can be accomplished by putting a period before the multiplication sign, to say that we want to multiply the individual elements together. Notice that now we do not have to transpose either array.

```
>> A.*B    % ans = [A(1)*B(1)  A(2)*B(2)  A(3)*B(3)  A(4)*B(4)]

ans =

   100     4    27    32
```

We can multiply an array by a scalar value, as seen below. Also, we can take the results of such an operation and add them together with the `sum` function.

```
>> A*4         % multiply all values of A by 4
```

```
ans =

    80     4    36    64

>> sum(A*4)

ans =

   184
```

When it comes to multiplying matrices with arrays, the results are as one would expect from linear algebra.

```
>> [1 2 3]*C      % Multiply row vector [1 2 3] by C
% result(1) =  0*1 + 4*2 + 8*3
% result(2) =  1*1 + 5*2 + 9*3
% result(3) =  2*1 + 6*2 + 10*3
% result(4) =  3*1 + 7*2 + 11*3
% Results are all in 1 row.

ans =

    32    38    44    50

>> C*[1 2 3 4].'  % Multiply C by [1 2 3 4] as a column vector
%  result(1) =   0*1  +  1*2  +  2*3  +  3*4
%  result(2) =   4*1  +  5*2  +  6*3  +  7*4
%  result(3) =   8*1  +  9*2  + 10*3  + 11*4
% Results are all in 1 column.

ans =

    20
    60
   100
```

There are several different ways to multiply $F$ and $C$. Trying `F*C` does not work, because the dimensions are not correct. If $F$ or $C$ is transposed, then there is no

problem. The numbers from these two matrices are shown multiplied and added in the variable *result*, showing that the multiplication result is what we should expect. We use the three dots (ellipsis) after *result* below to continue on the next line; this makes the lines fit.

```
>> F.'*C   % transpose(F) * C

ans =

   292   373   454   535
   280   358   436   514
   268   343   418   493
   256   328   400   472

>> result = ...
[31*0+27*4+23*8, 31*1+27*5+23*9, 31*2+27*6+23*10, 31*3+27*7+23*11;
 30*0+26*4+22*8, 30*1+26*5+22*9, 30*2+26*6+22*10, 30*3+26*7+22*11;
 29*0+25*4+21*8, 29*1+25*5+21*9, 29*2+25*6+21*10, 29*3+25*7+21*11;
 28*0+24*4+20*8, 28*1+24*5+20*9, 28*2+24*6+20*10, 28*3+24*7+20*11]

result =

   292   373   454   535
   280   358   436   514
   268   343   418   493
   256   328   400   472

>> F*C.'

ans =

         172         644        1116
         148         556         964
         124         468         812

>>    result = ...
[31*0+30*1+29*2+28*3, 31*4+30*5+29*6+28*7, 31*8+30*9+29*10+28*11;
 27*0+26*1+25*2+24*3, 27*4+26*5+25*6+24*7, 27*8+26*9+25*10+24*11;
 23*0+22*1+21*2+20*3, 23*4+22*5+21*6+20*7, 23*8+22*9+21*10+20*11]
```

```
result =

         172          644         1116
         148          556          964
         124          468          812
```

We can change the order of the matrices, and find that we get the same results as above, except transposed. An easy way to verify this is to subtract one (transposed) result from the other, to see that all subtraction results are zero.

```
>> C.'*F       % gives same answer as F.'*C, except transposed.

ans =

   292   280   268   256
   373   358   343   328
   454   436   418   400
   535   514   493   472

>> (F.'*C).' - C.'*F

ans =

     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0

>> C*F.'       % gives same answer as F*C.', except transposed.

ans =

         172          148          124
         644          556          468
        1116          964          812

>> (F*C.').' - C*F.'

ans =
```

```
     0     0     0
     0     0     0
     0     0     0
```

Suppose we want to multiply every element of $C$ with the corresponding element of $F$. We can do this with the `.*` operator, as we did before. Incidentally, reversing the order of operands produces the same result.

```
>> C.*F

ans =

     0    30    58    84
   108   130   150   168
   184   198   210   220

>> result = [0*31, 1*30, 2*29, 3*28;
             4*27, 5*26, 6*25, 7*24;
             8*23, 9*22, 10*21, 11*20]

result =

     0    30    58    84
   108   130   150   168
   184   198   210   220
```

Like multiplication, division has a couple of different interpretations, depending on whether `./` or `/` is specified. For example, `A./B` specifies that elements of $A$ should be divided by corresponding elements of $B$.

```
>> A = [20 1 9 16];
>> B = [5 4 3 2];
>> A./B

ans =

    4.0000    0.2500    3.0000    8.0000
```

```
>> result = [ 20/5, 1/4, 9/3, 16/2 ]

result =

    4.0000    0.2500    3.0000    8.0000
```

We can specify that elements of $B$ should be divided by corresponding elements of $A$, in two different ways: using the backslash `A.\B` or the forward slash `A./B`. Referring to the values of `result` from above, we can divide the value 1 by each element. Also shown below is that we can divide each element of matrix $C$ by the value 5.

```
>> A.\B

ans =

    0.2500    4.0000    0.3333    0.1250

>> B./A

ans =

    0.2500    4.0000    0.3333    0.1250

>> 1./result

ans =

    0.2500    4.0000    0.3333    0.1250

>> C/5

ans =

         0    0.2000    0.4000    0.6000
    0.8000    1.0000    1.2000    1.4000
    1.6000    1.8000    2.0000    2.2000
```

Next, we divide the elements of matrix $C$ by the corresponding elements of $F$. The

very first result is 0/31, or 0. If we switched the two matrices, i.e., using $F$'s elements for nominators (the top part of the fractions) while $C$'s elements are used for denominators (the bottom part of the fractions), it would still work, but the division by zero would cause the first result to be `Inf` (infinity).

```
>> C./F

ans =

         0    0.0333    0.0690    0.1071
    0.1481    0.1923    0.2400    0.2917
    0.3478    0.4091    0.4762    0.5500

>> result = [ 0/31, 1/30, 2/29, 3/28;
              4/27, 5/26, 6/25, 7/24;
              8/23, 9/22, 10/21, 11/20]

result =

         0    0.0333    0.0690    0.1071
    0.1481    0.1923    0.2400    0.2917
    0.3478    0.4091    0.4762    0.5500
```

Finally, we can divide one matrix by another. Here is an example, that uses "nice" values for the second matrix. The `inv` function returns the matrix's inverse. We expect the result of matrix division to be the same as multiplication of the first matrix by the inverse of the second. In other words, $M/N = M * N^{-1}$, just as it works with scalar values. The code below verifies that our results are as we expect them.

```
M = [1 2 3; 4 5 6; 7 8 9]

M =

     1     2     3
     4     5     6
     7     8     9

>> N = [1 2 0; 3 0 4; 0 5 6]
```

```
N =

     1     2     0
     3     0     4
     0     5     6

>> inv(N)

ans =

    0.3571    0.2143   -0.1429
    0.3214   -0.1071    0.0714
   -0.2679    0.0893    0.1071

>> M/N         % This is the same as M*inv(N)

ans =

    0.1964    0.2679    0.3214
    1.4286    0.8571    0.4286
    2.6607    1.4464    0.5357

>> M*inv(N)

ans =

    0.1964    0.2679    0.3214
    1.4286    0.8571    0.4286
    2.6607    1.4464    0.5357
```

There are many other operators in MATLAB. For a list, type `help .`, that is, the word `help` followed by a space then a period.

## 2.5 Functions

MATLAB supports functions. For greater flexibility, store the functions separately, with the same filename as the name of the function. In MATLAB, a function's local variables are removed after the function executes. This can be frustrating if the

function needs to be debugged. Once the program works well, it is a good idea to encapsulate it into a function.

For example, the following function takes two scalar values as arguments ($x$ and $y$), and returns two values ($r$ and *theta*). Whatever values $r$ and *theta* have at the end of the function will be returned. This function should be saved as "convert_to_polar.m," in the local workspace (the default).

```
%
% Comments
%
function [r, theta] = convert_to_polar(x, y)
```

It would be invoked by the following command. The names of the parameters do not need to be the same as those in the function's definition.

```
>> [vector_length, vector_angle] = convert_to_polar(a, b);
```

## 2.6 How NOT to Plot a Sinusoid

Sinusoidal functions are important to signal processing. Suppose we want to make a graph of a 200 Hz sinusoid. This sounds simple enough, but can present several hurdles, as this section demonstrates.

We had an expression for sinusoids already, $a \times \cos(2\pi f t + \phi)$. If we want to, we could use the sine function instead, but this expression will do. We know the frequency, and can use $a = 1$ and $\phi = 0$ to keep things simple.

The plan is to generate an array of the cosine values, then plot the array. A first try would probably look like:

```
for t=1:100
    x = cos(2*pi*200*t);
end
plot(x);
```

But this code results in disappointment, since the graph appears blank. Actually, the graph does contain a single point at (1,1), though this is far from our desired sinusoid. A quick check of variable $x$ reveals that it is a scalar value. We need to specify that MATLAB store $x$ as an array. We try again, using $t$ as an index.

```
for t=1:100
    x(t) = cos(2*pi*200*t);
end
plot(x);
```

This brings us a step closer, since we now see a straight line plotted on the graph. But why do we see a straight line? Upon inspection, we notice that all $x$ values are 1. A few examples on a calculator reveal that the values for $x$ are actually correct. The problem here is the argument to the cosine function; $2\pi$ corresponds to a rotation around the circle. So we can always add $2\pi$ to the argument of a sinusoid without changing the result, i.e.,

$$\cos(0) = \cos(2\pi) = \cos(2\pi \times 2).$$

Since $t$ always has integer values in the above code, the cosine function's argument always is just a $2\pi$ multiple of the one before it. We can fix this by introducing a fractional step to the `for` loop. Notice how we change the upper limit on $t$ too, to keep the number of points the same.

```
for t=1:0.01:2
    x(t) = cos(2*pi*200*t);
end
plot(x);
```

This code gives us an error:

```
??? Subscript indices must either be real positive integers
or logicals.

>> t

t =

    1.0100
```

We see that $t$ has a value of 1.01, and using 1.01 to index an array will give us a problem in just about any computer language. In other words, we cannot use $x(t)$ unless $t$ has an integer value. To fix this, we can introduce a new variable $n$ just to act as an index for array $x$.

```
n = 1;
for t=1:0.01:2
    x(n) = cos(2*pi*200*t);
```

```
    n = n + 1;
end
plot(x);
```

This code executes well, but still plots a straight line. The problem is again in the argument to the cos function. We had the right idea of using fractional values in the argument, but we did not carry it far enough. When we multiply 200 by $t$, we generate values from 200 to 400, with a step of 2. The problem remains that we use integer multiples of $2\pi$ for our cosine arguments. To try to fix this, we can make the step value smaller, as below. We will reduce the upper limit, to keep the size of our array the same.

```
n = 1;
for t=1:0.001:1.1
    x(n) = cos(2*pi*200*t);
    n = n + 1;
end
plot(x);
```

Finally, we have working code that produces something that resembles a sinusoid. To make it smoother, we can alter the step and upper limit again.

```
n = 1;
for t=1:0.0001:1.01
    x(n) = cos(2*pi*200*t);
    n = n + 1;
end
plot(x);
```

Now we have a smooth-looking sinusoid. But the x-axis has the indices, not the actual $t$ values. To improve this, we modify the `plot` function to specify the x-axis values:

```
plot(1:0.0001:1.01, x);
```

We are not finished yet. The code we have works, but it could be more efficient and compact. We really do not even need the `for` loop with MATLAB. Instead we can express $t$ as a range, then use it as part of the expression for $x$, then again in the `plot` command.

```
t = 1:0.0001:1.01;
x = cos(2*pi*200*t);
plot(t, x);
```

Notice how MATLAB stores $x$ as an array. The critical difference here is the argument to the cosine function contains a range. When we multiply a range by a constant (or three constants, in this case), we still end up with a range. MATLAB allows the cosine function to work with arrays as well as scalar values, and returns an array. In fact, we could even combine the statements above:

```
plot((1:0.0001:1.01), cos(2*pi*200*(1:0.0001:1.01)));
```

Compaction like this detracts from the code's readability. Thus, the three-line example above makes a better solution. But there is one important thing it lacks: comments. Other improvements that we might desire are a title, and labels for the x-axis and y-axis. We end this discussion with an improved solution (below). We also include an `axis` command, that specifies what part of the graph we want to see, i.e., we want to see the x-axis from 1 to 1.01, and we want to view the y-axis from $-1$ to 1. The final result appears in Figure 2.1.

```
% let t be a small amount of time
t = 1:0.0001:1.01;
% find a 200 Hz sinusoid based on t
x = cos(2*pi*200*t);
plot(t, x);
title('A 200 Hz sinusoid');
xlabel('Time');
ylabel('Amplitude');
axis([1 1.01 -1 1]);
```

## 2.7 Plotting a Sinusoid

The following program plots a sinusoid, given the frequency, magnitude and phase angle. It also works for more than one sinusoid at a time.

```
%
%  Given frequencies, magnitudes and phases,
%  plot a set of sinusoids (up to 4) (based on plotharmonic.m)
%
```

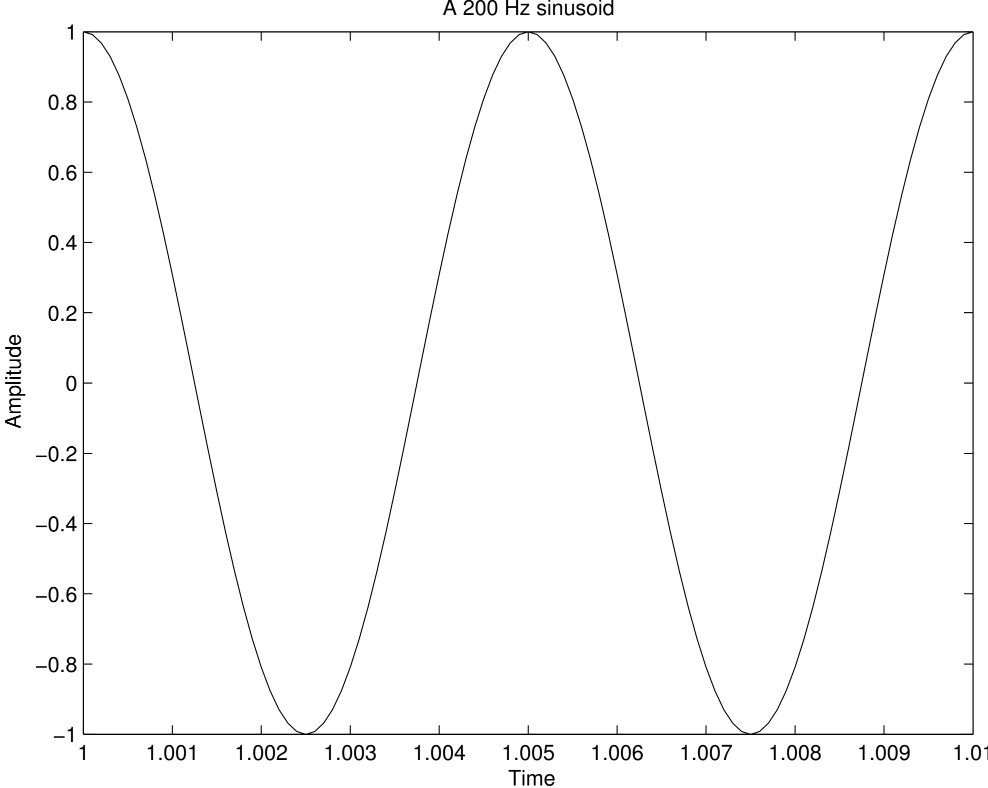



**Figure 2.1:** A 200 Hz sinusoid produced by example MATLAB code.

```
%  usage:  plotsinusoids(freq, mag, phase)
%
function plotsinusoids(freq, mag, phase)

max_plots = 4;     % Maximum number of sinusoids to plot at once
num_points = 200;  % Number of points per repetition

% Check parameters
if (or((length(mag)~=length(freq)), ...
       (length(phase)~=length(freq))))
    error('Error - freq, mag, and phase must be the same size')
end

% We want this for 2 repetitions, and num_points per rep.
freq_min = min(freq);
step = 2/(freq_min*num_points);
t = 0:step:2*(1/freq_min);

i=1;
my_limit = min(length(freq), max_plots);
while (i <= my_limit)
    x =  mag(i)*cos(2*pi*freq(i)*t + phase(i));
    my_title = ...
        sprintf('Sinusoid with f=%d, mag=%4.2f, phase=%4.2f', ...
        freq(i), mag(i), phase(i));
    subplot(my_limit,1,i);
    plot(t,x);
    title(my_title);
    ylabel('Amplitude');
    i = i + 1;
end
xlabel('Time (sec)');
```

The following command uses the above function. Notice how it does not return a value, so we do not precede the call with a `variable =` assignment.

```
plotsinusoids(100, 1.5, pi/6);
```

This produces the graph shown in Figure 2.2.

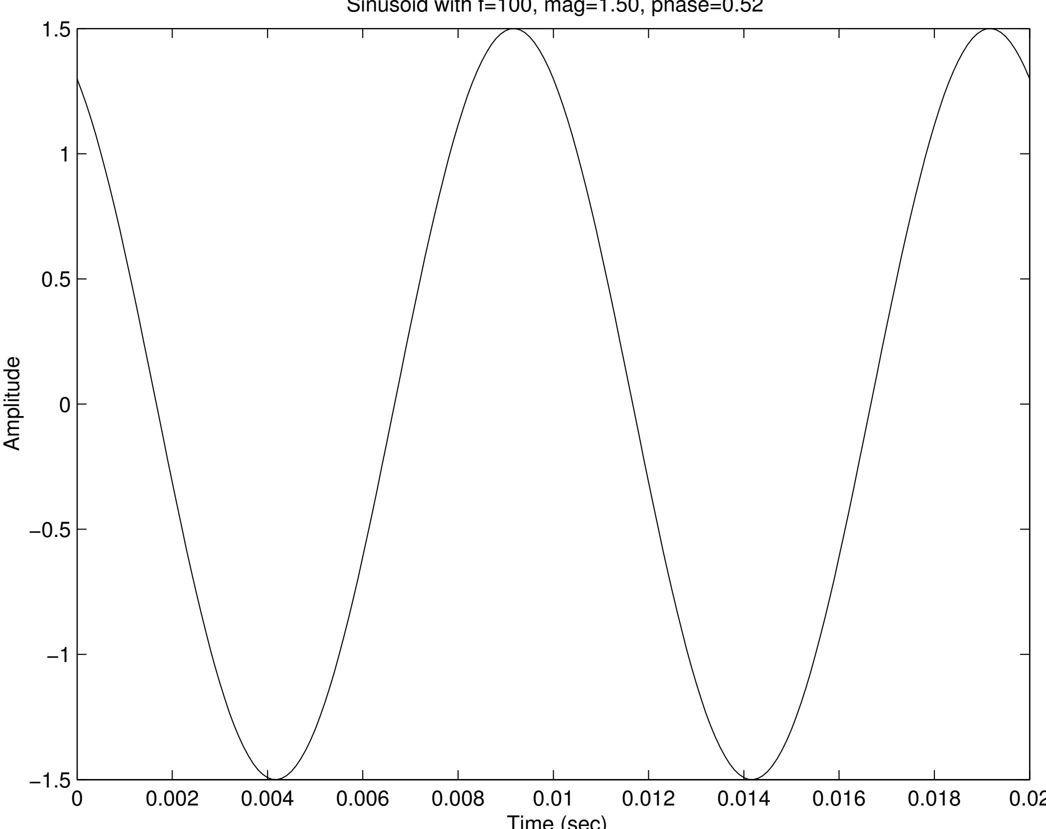



**Figure 2.2:** Using the "plotsinusoids" function.

## 2.8 Plotting Sinusoids a Little at a Time

The next program came about because of a need to see when two sinusoids repeat, also known as the *period*. Take two sinusoids with frequencies of 25 Hz and 33 Hz, for example. Both repeat themselves over and over, but what if they were added together? The resulting sinusoid would repeat itself, but when? To find out, we need the greatest common divisor. Fortunately, MATLAB provides such a function:

```
>> gcd(25, 33)

ans =

     1
```

This means that the signal $\cos(2\pi 25t) + \cos(2\pi 33t)$ repeats every second, or $1/gcd(25, 33)$. Figure 2.3 demonstrates this. The signal is plotted for time 0 to 1 second in the top half, and from 1 to 2 seconds in the bottom half. While the signal has sections that look similar, careful examination reveals that the signal does not actually repeat until 1 second has elapsed.

What if we want to see a close-up view of the two sinusoids? The following code does just that. It plots two sinusoids, and shows one section at a time. The `pause` function gives the user a chance to see the graph before a new graph is plotted. With it, we can see that the two sinusoids start at the same value, but do not match up at this value until the very end. Figure 2.4 shows the output of this program for the end of the sinusoids.

```
%
%  show_sins.m
%  Show 2 sinusoids, 100 samples at a time.
%

% Feel free to change any of the default values below
show_num = 100;       % How many samples to show at a time
count = 100;          % How many samples to advance at a time
t = 0:0.001:1;        % Our simulated time variable
freq1 = 25;           % frequency1
freq2 = 33;           % frequency2
```

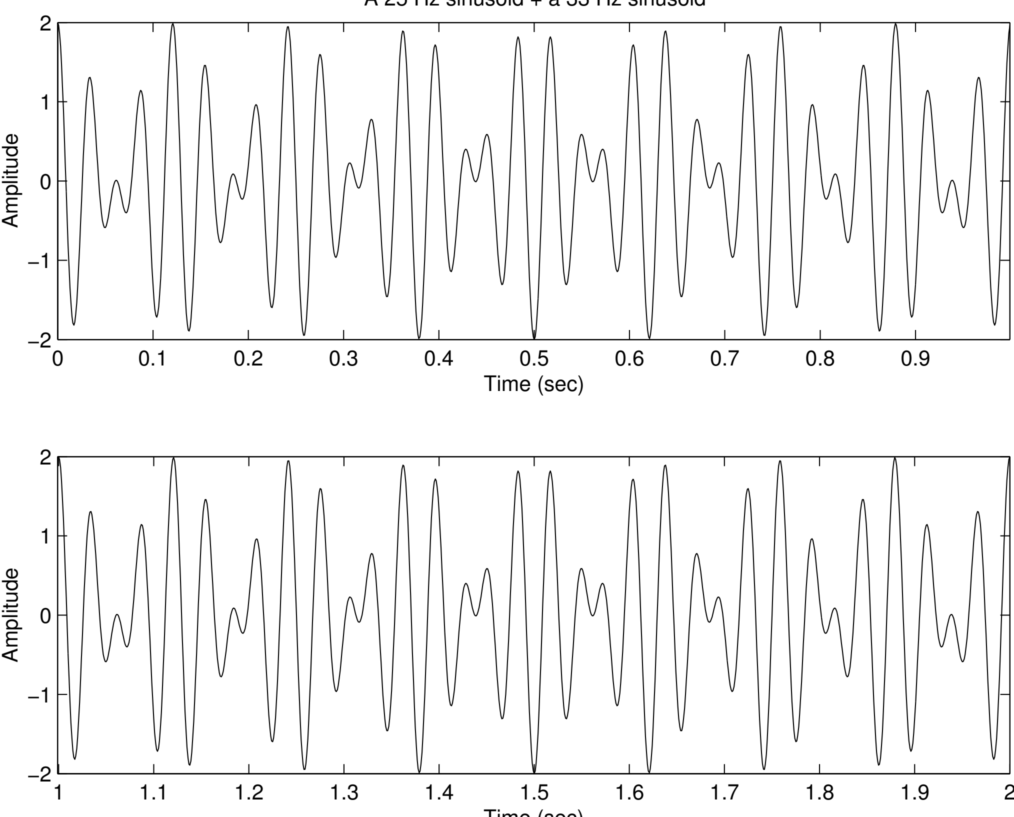



**Figure 2.3:** This signal repeats itself every second.

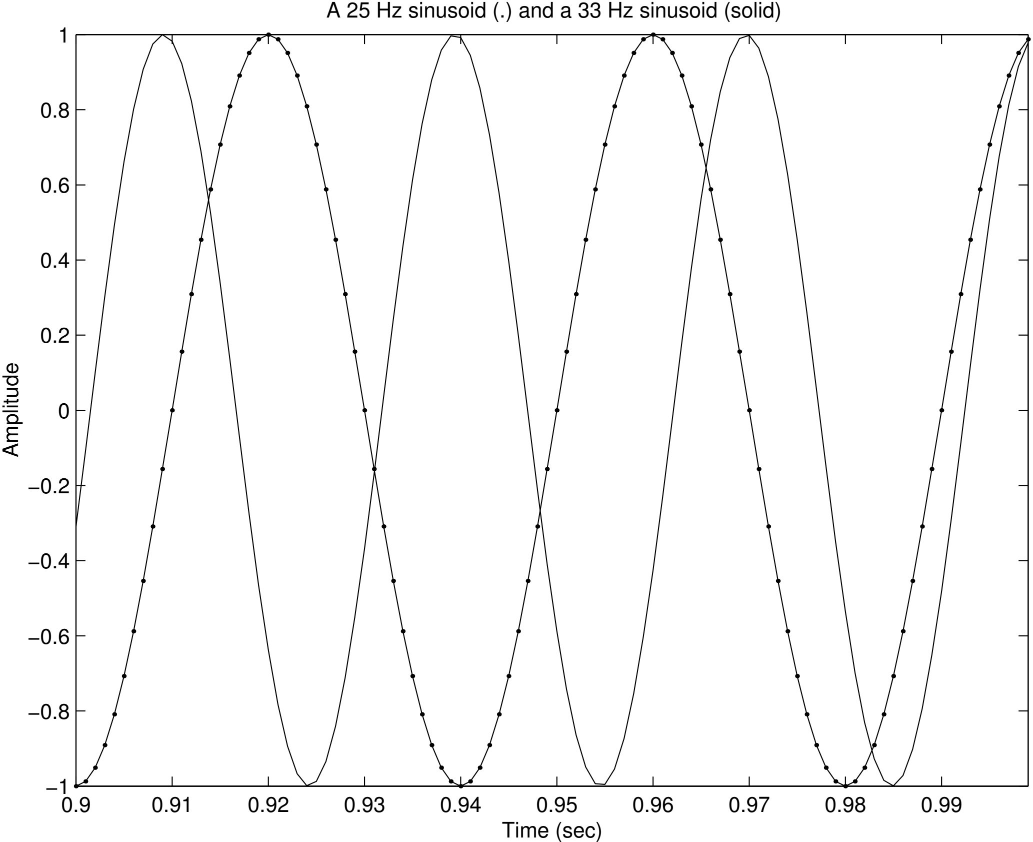



**Figure 2.4:** A close-up view of two sinusoids from 0.9 to 1 second.

```
x1 = cos(2*pi*freq1*t);
x2 = cos(2*pi*freq2*t);
mytitle = ...
   sprintf('A %d Hz sinusoid (.) and a %d Hz one (solid)', ...
   freq1, freq2);
start = 1;
last = show_num;
done = 0;
while (done ~=1)
    plot(t(start:last), x1(start:last), 'k.-', t(start:last), ...
       x2(start:last), 'g')
    axis tight;  % Make the axis look good
    xlabel('Time (sec)');
    ylabel('Amplitude');
    title(mytitle);
    pause(1);

    % Update array offsets
    last = last + count;
    start = start + count;
    if (start >= length(x1))
        done = 1;
    end
    if (last > length(x1))
        last = length(x1);
    end
end
```

## 2.9 Calculating Error

A common operation is to find the difference between two signals. For example, a signal can be compressed, then uncompressed with a "lossy" compression algorithm. (This is where some error in the signal is acceptable.) The amount of error between the original and the reconstructed versions of the signal can be found by summing the absolute values of the differences. Is the absolute value function, `abs`, really necessary? Consider the following two signals, $x$ and $y$ :

```
>> x = [1, 2, 5, 0, -2];
>> y = [-1, 4, 0, 5, -2];
```

```
>> sum(x-y)

ans =

     0
```

Clearly $x$ and $y$ are not equal, but since the positive differences and the negative differences cancel each other out, this simple method makes them *appear* to be equal. The error between two signals, $x$ and $y$, can be found with the following command. Note that $x$ and $y$ must have the same length.

```
>>   error = sum(abs(x - y))

error =

    14
```

Another way to measure error is the root mean square error (RMSE). The code below calculates this. First, *diff* finds the difference between signals $x$ and $y$. Next, we find *sigma_squared*, also known as $\sigma^2$ or variance, calculated by squaring each element in *diff*, and adding the results. Finally, we can compute the root mean square error.

```
diff = x - y;
sigma_squared = sum(diff.*diff);
rmse = sqrt(sigma_squared/length(x))
```

We could have used the command:

```
sigma_squared = diff*diff.';
```

instead of the second line above, since it returns the same answer (and the matrix multiplication takes care of the summation).

## 2.10 Sometimes 0 Is Not Exactly 0

In programming, a common condition in an `if` or `while` statement is a comparison of some variable with zero. When dealing with floating-point numbers, though, the

computer will do exactly what we tell it, and conclude that 0.000000000000000000000000000000000000000001 is not equal to 0.

```
>> % mynumber should be 0.0 after the next statement
mynumber = sqrt(7)^2 - 7;

% Here we confirm that mynumber is zero, right?
if (mynumber == 0.0)
   sprintf('mynumber is zero')
else
   sprintf('mynumber is not zero')
end

ans =

mynumber is not zero

>>
```

What is this? If *mynumber* is not zero, then what is it?

```
>> mynumber

mynumber =

  8.8818e-016
```

The number is very, very small, but not quite zero. It *should* be zero, but due to realities of precision, there is a little error.

### 2.10.1 Comparing Numbers with a Tolerance

When a value is very close to zero, but technically not zero, we can compare it with a certain tolerance, as the following example demonstrates.

```
>> % mynumber should be 0.0 after the next statement
mynumber = sqrt(7)^2 - 7;

% Set a tolerance limit
tolerance = 0.000001;

% Since we know the value of mynumber is very small,
% but not exactly 0, we will use the following comparison.
if and((mynumber < tolerance), (mynumber > -tolerance))
   sprintf('mynumber is zero')
else
   sprintf('mynumber is not zero')
end

ans =

mynumber is zero

>>
```

With the tolerance factored in, the comparison gives us the results that we expect. This is only one way of doing the comparison, and the following example shows the same idea, only with the absolute value function `abs`, to make the code a little more compact and readable.

```
>> % mynumber should be 0.0 after the next statement
mynumber = sqrt(7)^2 - 7;

% Set a tolerance limit
tolerance = 0.000001;

% Since mynumber is very small, but not exactly 0,
% we will use the following comparison.
%
% This example shows that we could use the abs() function.
if (abs(mynumber) < tolerance)
   sprintf('mynumber is zero')
else
```

```
   sprintf('mynumber is not zero')
end

ans =

mynumber is zero

>>
```

As we saw before, this code gives us the answer we expect.

This type of problem can also occur when comparing two numbers to one another. We may desire that they be declared equal when they are sufficiently close enough. Of course, the amount of tolerance is up to us.

```
>> % compare 2 very close numbers
mynumber1 = sqrt(7)^2
mynumber2 = 7
tolerance = 0.000001;

% This comparison gives us misleading results.
if (mynumber1 == mynumber2)
   sprintf('mynumber1 is equal to mynumber2')
else
   sprintf('mynumber1 and mynumber2 are not equal')
end

mynumber1 =

    7.0000


mynumber2 =

     7


ans =
```

```
mynumber1 and mynumber2 are not equal

>>
```

So, according to the example above, 7 does not equal 7.0000. Of course they should be equal, but what is shown on screen is not entirely what is stored internally. Again, this problem can be fixed by altering the comparison statement.

```
>> % compare 2 very close numbers
mynumber1 = sqrt(7)^2
mynumber2 = 7
tolerance = 0.000001;

% This comparison tells us that the two numbers are equal,
% up to our tolerance.
if (abs(mynumber1 - mynumber2)< tolerance)
   sprintf('mynumber1 is equal to mynumber2')
else
   sprintf('mynumber1 and mynumber2 are not equal')
end

mynumber1 =

    7.0000


mynumber2 =

     7


ans =

mynumber1 is equal to mynumber2

>>
```

The above comparison, with the tolerance included, gives us the answer we expect. Notice how the `abs` function is used. This keeps us from having to make two

comparisons. If the absolute value function were not included in the above code, then it would only work when $mynumber1$ was greater than or equal to $mynumber2$.

Of course, rounding or truncating would also work, and the following subsection gives examples of these functions.

### 2.10.2 Rounding and Truncating

There are four different rounding and truncating functions that we will examine: `round`, `ceil`, `floor`, and `fix`, each of which converts a floating-point number to an integer. Since these functions can work with arrays as well as single values, we show them working on an array of numbers so that their precise function becomes apparent. First, we see the `round` function returns the integer closest to the floating-point number, with a cutoff of 3.5.

```
>> round([3.01, 3.49, 3.5, 3.99])

ans =

     3     3     4     4

>> round([-3.01, -3.49, -3.5, -3.99])

ans =

    -3    -3    -4    -4
```

The `floor` function returns the integer to the left of the floating-point number on the number line.

```
>> floor([3.01, 3.49, 3.5, 3.99])

ans =

     3     3     3     3

>> floor([-3.01, -3.49, -3.5, -3.99])

ans =
```

```
    -4    -4    -4    -4
```

The `ceil` function, whose name is short for "ceiling," works in a manner opposite to that of the `floor` function, giving the integer to the right of the floating-point number on the number line.

```
>> ceil([3.01, 3.49, 3.5, 3.99])

ans =

     4     4     4     4

>> ceil([-3.01, -3.49, -3.5, -3.99])

ans =

    -3    -3    -3    -3
```

Finally, the `fix` function performs truncation, eliminating the fractional part.

```
>> fix([3.01, 3.49, 3.5, 3.99])

ans =

     3     3     3     3

>> fix([-3.01, -3.49, -3.5, -3.99])

ans =

    -3    -3    -3    -3
```

## 2.11 MATLAB Programming Tips

The following bits of advice should help to get started, and if you get stuck.

- If something does not work, check the dimensions of the data. It may work with the transpose.

- If a plot does not show up, the data could be an empty matrix.
- Sound (such as a *.wav* file) may have 2 channels, so do not forget to manipulate both.
- If the `wavplay` command does not work, try `sound`.
- If something calls for the discrete Fourier transform (DFT), try using the fast Fourier transform (FFT).
- Most programs should work with the student edition.
- Your programs may be slow, so be patient. If you are working with a lot of data, you may want to let your program run overnight.
- It is easy to write a script to run the same program(s) over and over again, with different parameters.
- Use the graphical user interface to highlight a section of code, and comment it out (or uncomment it later).
- When a command works well, copy and paste it to a file. It is easy to construct a program this way, by trying different things on the command line and saving the best ones.
- If a function has a problem, comment out the `function` line, and be sure to define variables in place of the parameters.
- When debugging, use the `break` command to stop a program, and check values. That is, when trying to find a problem, put this keyword in the program to stop it, with the understanding that it will be removed later. You can also set breakpoints within the code, to achieve the same effect.
- The keywords `try` and `catch` can be used to have your program handle errors. For example, Chapter 9, "The Wavelet Transform," includes a program that uses the `wfilters` command, and uses `try` and `catch` to print a short message indicating that it needs the wavelets toolkit.

## 2.12 MATLAB Programming Exercises

Several programming exercises designed to help you familiarize yourself with MATLAB follow. The exercises build upon each other, so it is best to go through them in order. Note that the subjective term "smooth" means that the changes between successive points are small.

1. Create a signal $x_1$, where $x_1 = 2\cos(2\pi 15t)$.
   Make $t$ a range large enough to show 2 repetitions of $x_1$.
   Plot $x_1$.
   What are the minimum and maximum values of $x_1$ from the graph?
   Make the increment of $t$ small enough that the plot of $x_1$ looks smooth (if it already looks smooth, make it even smoother), then replot $x_1$.

   **Answer:**
   One common problem with this is that a straight line appears instead of a sinusoid. The problem stems from the argument to the `cos` function, namely, making $t$ go from 0 to some limit with a command like `t=0:100`. Since the count is by 1, $t$ is always an integer, and the argument to the `cos` function is always an integer multiple of $2\pi$. Since the `cos` function repeats every $2\pi$, the answer is always the same. To fix this, let the range of $t$ increment by a fraction, such as `t=0:0.001:1`.

   A 15 Hz sinusoid repeats 15 times per second. So one full repetition is 1/15. Here is a nonsmooth plot:

   ```
   t = 0:0.01:(2*1/15); x1 = 2*cos(2*pi*15*t); plot(x1)
   ```

   Here is a smooth plot:

   ```
   t = 0:0.001:(2*1/15); x1 = 2*cos(2*pi*15*t); plot(x1)
   ```

   The minimum and maximum values are -2 and +2, respectively.

2. Let $x_2 = x_1 + 2.5\cos(2\pi 30t)$, and adjust $t$ as needed.
   Plot $x_2$.
   Again, from the plot, what are the minimum and maximum values?

   **Answer:**

   ```
   t = 0:0.001:(2*1/15);
   x2 = 2*cos(2*pi*15*t) + 2.5 * cos(2*pi*30*t);
   plot(x2)
   ```

   The minimum and maximum values are approximately -2.7 and +4.5, respectively.

Note: MATLAB has functions `min`, `max`, `mean` (average), `median`, and `sort` built-in. The questions below ask you to write your own code for these functions, as if they were not built-in. The idea here is for you to get some programming experience with MATLAB, with some easy-to-verify problems.

---

3. Write MATLAB code to give minimum and maximum values of $x_1$ and $x_2$. When you are happy with your code, make functions from it called "mymin" and "mymax," respectively.

**Answer:**

```
%
% Mymax : return the maximum value of an array.
%

function max_val = mymax(x)

% Get an initial guess for the max
max_val = x(1);

% Now go through the rest of the array,
% 1 element at a time
for k=2:length(x)
    % if the value at the current offset
    % is bigger than our current max,
    if (x(k) > max_val)
        % ..then update our current max.
        max_val = x(k);
    end
end

% "max_val" will be returned automatically.
```

Now for the minimum-finding program.

```
%
% Mymin : return the minimum value of an array.
```

```
%

function min_val = mymin(x)

% Get an initial guess for the min
min_val = x(1);

% Now go through the rest of the array,
% 1 element at a time
for k=2:length(x)
    % if the value at the current offset
    % is smaller than our current min,
    if (x(k) < min_val)
        % ..then update our current min.
        min_val = x(k);
    end
end

% "min_val" will be returned automatically.
```

Finally, we can test these solutions.

```
>> mymin(x1)

ans =

    -2

>> mymin(x2)

ans =

   -2.6992

>> mymax(x1)

ans =

     2
```

```
>> mymax(x2)

ans =

    4.5000
```

4. Find $y_1[n] = x_1[n] - .9x_1[n-1]$, using $x_1$ from above.
   Plot $y_1$ (adjusting $t$ as needed).

   **Answer:**
   Array $t$ does not need to be adjusted, but we have to be careful about $n$. That is, $x_1[n-1]$ does not exist when $n = 1$ (MATLAB will not let us index an array at location 0). Therefore, we have to start $n$ at 2, so that $n-1$ is a valid array value. Also, notice that we use parentheses for MATLAB code, in place of the square brackets.

   ```
   for n=2:length(x1)
      y1(n) = x1(n) - .9 * x1(n-1);
   end
   plot(y1)
   ```

5. Create $y_2[n]$ to be the average of the current and previous two values of $x_2[n]$.
   Plot $y_2$.

   **Answer:**
   We have to be careful with our starting value for $n$, as in the problem above.

   ```
   for n=3:length(x2)
      y2(n) = (x2(n) + x2(n-1) + x2(n-2))/3;
   end
   plot(y2)
   ```

6. Create $y_3[n] = x_2[n] - y_3[n-1]$.
   Plot $y_3$.
   What will happen as $n$ approaches infinity?

**Answer:**
You have to give $y_3$ an initial value to use.

```
y3(1) = 0;
for n=2:length(x2)
   y3(n) = x2(n) - y3(n-1);
end
plot(y3)
```

To see what happens when $n$ approaches infinity, we will calculate more values of $y_3$, past the end of $x_2$'s input.

```
for n=length(x2)+1:200
   y3(n) = 0 - y3(n-1);
end
plot(y3)
```

We see from the plot that it repeats as a sinusoid forever.

7. Make a random function that returns an array of $m$ pseudorandom integers between a low and high value specified. Use the `rand` function in your function, and call your solution "myrand." Notice that "myrand" should have three parameters: the size of the array to return ($m$), the lowest integer value it should return, and the highest integer value it should return.

**Answer:**
From the `help rand` function, we can see how to use the `rand` function. For example, `rand(1,8)` returns 8 random values between 0 and 1 as an array. The numbers should range from low to high, so we multiply the values by $(high - low)$. Next, we add $low$ to the results.

```
low = 10; hi = 90; n = 8;
round(rand(1,n)*(hi-low) + low)
```

Now we will test it out:

```
>> x = round(rand(1,100)*(hi-low) + low);
>> plot(x)
>> min(x)
```

```
ans =

    10

>> max(x)

ans =

    90
```

Now we will put it in a function:

```
%
% Return a vector of m random values
% between low and high.
%
% Usage: result = myrand(low, high, m)
%
function result = myrand(low, hi, m)

result = round(rand(1,m)*(hi-low) + low);
```

8. With an array of 8 values between 0 and 99 (use "myrand"), write a function to return the values in sorted order, from lowest to highest.

**Answer:**

```
%
%  mysort.m
%
%  Given an array of numbers, sort them.
%  (Use a bubble-sort since it is simple).
%
function sorted = mysort(unsorted)

sorted = unsorted;
done = false;
```

```
while (~done)
    done = true;
    % Go through the whole array,
    % until no switches are made.
    for k=2:length(sorted)
        % check the current value
        % to its left neighbor
        if (sorted(k-1) > sorted(k))
            % Switch the two
            temp = sorted(k);
            sorted(k) = sorted(k-1);
            sorted(k-1) = temp;
            done  = false;
        end
    end
end


>> unsorted = myrand(0, 99, 8)

unsorted =

    93    91    41    88     6    35    81     1

>> mysort(unsorted)

ans =

     1     6    35    41    81    88    91    93
```

9. Calculate the average from an array, and call your function “average.”

**Answer:**
Yes, this could be done easily with the following one line command:

```
>> sum(unsorted)/length(unsorted)

ans =
```

```
54.5000
```

But in the spirit of doing things for ourselves, here is an alternate solution.

```
%
% average.m
%
% Find the average of an array of numbers.
%
function avg = average(x)

mysum = 0;
for k=1:length(x)
    mysum = mysum + x(k);
end
avg = mysum/k;
```

10. Write a function to determine if the number passed to it is even or odd. Your function "evenodd" should print the result to the screen.

**Answer:**
The `ceil` and `floor` functions both return an integer value. If a value divided by 2 is the same as the `ceil` or `floor` of that number divided by 2, then it must be even.

```
%
% evenodd.m
%  Print whether a number is even or odd.
%
% Usage:  evenodd(unsorted_array)
%
function evenodd(value)

if ((value/2) == floor(value/2))
    % The number is even.
    disp('even');
else
```

```
    % The number is odd.
    disp('odd');
end
```

A couple of tests for this function follow.

```
>> evenodd(7)
odd
>> evenodd(8)
even
```

One interesting variation on this problem is to try to have it return a boolean value (`true` or `false`) instead of printing to the screen.

11. Find the median value from an array, and call this function "mymedian." (A median is the value in the middle of a sorted array. If there are two values in the middle, which happens when the number of elements is even, the median should then be the average of those two numbers.)

**Answer:**

```
%
% mymedian.m
%  Return the median value of an array
%
% Usage:  m = mymedian(unsorted_array)
%
function m = mymedian(unsorted)

% Sort the array
sorted = mysort(unsorted);
L = length(sorted);
k = ceil(L/2);

% Check to see if there are an even
% or odd number of array elements
if (k == L/2)
    % There are an even number.
    % Average the middle two numbers
```

```
        m = (sorted(k) + sorted(k+1))/2;
    else
        % There are an odd number.
        % Just get the middle number
        m = sorted(k);
    end
```

12. Use the `disp` and `sprintf` functions as needed to print the time in a nice format.

    **Answer:**

    ```
    time = clock;
    disp(sprintf('Current time %2d:%2d:%2d', ...
       time(1,4), time(1,5), round(time(1,6))))
    ```

## 2.13 Other Useful MATLAB Commands

Below are some other MATLAB commands that are worth experimenting with.

```
sprintf('text. integer %d  float %f', 3, 2.7)
```

(Notice that there is no semicolon after `sprintf` since we want it to appear on the screen.)

```
plot(1:length(x), x, 'r')
break
return
close all
clear all
who
whos
save
load
zplane
fir1
vpa
```

Command for a 3D surface plot:

```
surf
```

Commands that work with files:

```
fopen
fread
fwrite
fprintf
fprintf(1,'text for a file');
fscanf
fclose
```

To find the magnitude and angle of a complex number (convert to polar):
`abs` (gives magnitude)
`angle` (gives phase angle)

Commands for working with images:

```
x = imread('lena.tif', 'tif');
figure(1)
colormap(gray(256))
image(x)
```

`rgb2hsv()` converts Red-Green-Blue to Hue-Saturation-Intensity

Commands to work with sound:

```
audiorecorder
wavrecord
wavread
wavplay
sound
wavwrite
```

Time-related functions:

```
tic / toc
clock
cputime
```

Wavelet-related commands:

```
wfilters
wavedec
dwt / idwt
dwt2 / idwt2
```

## 2.14 Summary

This chapter presents an overview of MATLAB, and demonstrates some useful commands. Matrix operations are important to DSP. In fact, we can implement transforms with matrix multiplications.

## 2.15 Review Questions

All questions should be answered with MATLAB, unless otherwise noted.

1. Write a MATLAB function to sort an array of numbers in descending order (highest to lowest).

2. Write a MATLAB function that removes duplicate numbers. That is, given an array [5, 2, 2, 1, 4, 4, 4], it should return [5, 2, 1, 4]. Assume that the first occurrence of a number is kept, such as in the case of the array [7, 3, 8, 3, 7], where your function should return [7, 3, 8].

3. If $x = \{121, 145, 167, 192, 206\}$, what is $y = 3x - 4$? Is $sum(y)$ the same value as $3sum(x) - 4$?

4. Use the `plot` command to graph $x$ and $y$ from above. Both $x$ and $y$ should appear on the same graph.

5. Using MATLAB, plot the sinusoid $2\cos(2\pi 1500t + \pi/2)$, starting at $t = 0$ seconds. Be sure to use enough points to make a smooth graph, and only show a few repetitions.

6. Write a MATLAB function called "myfun" to compute

$$2N^2 - 4N + 6$$

for any value $N$. What will be the result if $N$ is an array? If your function does not work when $N$ is an array, modify it so that it will. It should return an array of the same length.

7. Floating-point (real) numbers are stored in binary with finite precision on computers.

   a. Using MATLAB, what is the smallest number that we can store? In other words, we can store $1/(2^n)$ in a variable when $n$ is small, but how large can $n$ be before the variable is considered zero? (Use the default precision.)

   b. What is the largest number that we can store? In other words, we can store $2^n$ in a variable, but how big can $n$ be? What happens when $n$ is too large?

   c. What about the number $1 + 1/(2^n)$, how large can $n$ be? Is your answer the same as in part `a`? Why or why not?

8. The following MATLAB code should return the magnitudes and phases for an array of complex numbers. The line marked "problem" actually has *two* problems with it.

```
%  This function should return the magnitude and phase.
%
function [mag, phi] = my_function(y)
% input Y is an array of complex numbers

yr = real(y);
yi = imag(y);
for i=1:length(yr)
    phi(i) = atan(yi(i)/yr(i)); % problem
    mag(i) = sqrt(yr(i)*yr(i)+yi(i)*yi(i));
end
```

   a. The first problem is the ambiguity discussed in Chapter 1, "Introduction." Explain what this problem is, and demonstrate it via examples.

   b. For the second problem, consider what happens when it processes the following code, which makes it crash! What is the problem?

```
y = [ 1+2j, 3j, 4+5j ];
[xm, xp] = my_function(y);
```

# Chapter 3

# Filters

We are often interested in manipulating a signal. For example, you might turn up the volume on a stereo system, boost the bass (low-frequency sounds), or adjust sounds in other frequency ranges with the equalizer. These examples are not necessarily digital signal processing, since they can also be done with analog parts, but they do serve as examples of ways we might want to change a signal. Filters allow us to change signals in these ways. How does a filter function? Architecturally, how do we make a filter? We will examine these questions and more in this chapter.

Figure 3.1 (top) shows an example signal, the low temperatures from Chapter 1, "Introduction." In the middle, we see the output from a *lowpass* filter, i.e., a filter that keeps the low-frequency information. Notice how much it looks like the original. Also, you may see that the filtered version has one more value than the original, owing to the effect of the filter (the number of filter outputs equals the number of inputs plus the number of filter coefficients minus one). The bottom graph on this figure shows the output from a *highpass* filter, which allows the rapidly changing part of the signal through. We see that most of the points are around zero, indicating little change. However, a visible dip around output 5 shows how the big change between input samples 4 and 5 becomes encoded here.

Shown in Figure 3.2 is the original frequency content of the example input signal (top). The distinctive pattern seen here owes its shape to the sharp drop between the end of the input signal and zeros used to pad the signal. The *sinc* function can be used to model this phenomenon, and often *windows* are used to minimize this effect by allowing the signal to gradually rise at the beginning, then slowly taper off at the end. A triangle function is one example of such a window.

In the middle of Figure 3.2, the lowpass impulse response appears, which gives us an idea of how the filter will affect our signal. The low frequencies will get through it well, but the higher frequencies will be attenuated (zeroed out). For example, an

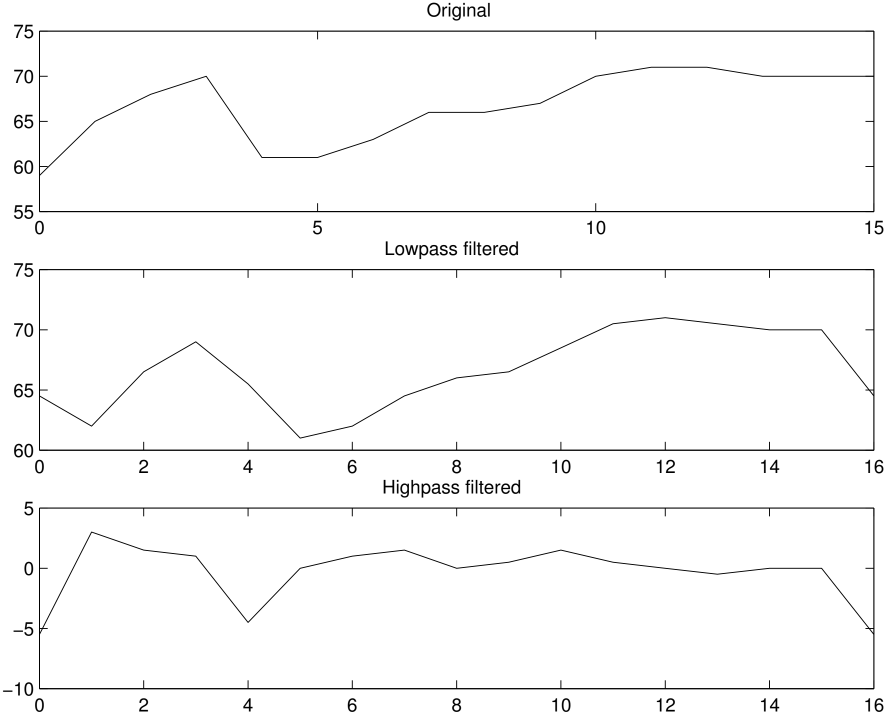



**Figure 3.1:** An example signal, filtered.

input frequency of 60 Hz will still be present in the output, but only at about 10% of its original strength. We can see this by looking at the spot marked 60 along the x-axis, and looking up to see the lowpass impulse response is right around the 10% mark on the y-axis.

The bottom plot on Figure 3.2 has the highpass impulse response, showing the opposite effect of the plot above it. This one will allow the high-frequency content through, but attenuate the low-frequency content. Note that both filters here are not ideal, since they have a long, slow slope. We would expect that a good filter would have a sharper cutoff, and appear more like a square wave. These filters were made using only 2 filter coefficients, [0.5, 0.5] for the lowpass filter, and [0.5, -0.5] for the highpass filter. These particular values will be used again, especially in Chapter 9, "The Wavelet Transform." For the moment, suffice it to say that having few filter coefficients leads to impulse responses like those seen in Figure 3.2.

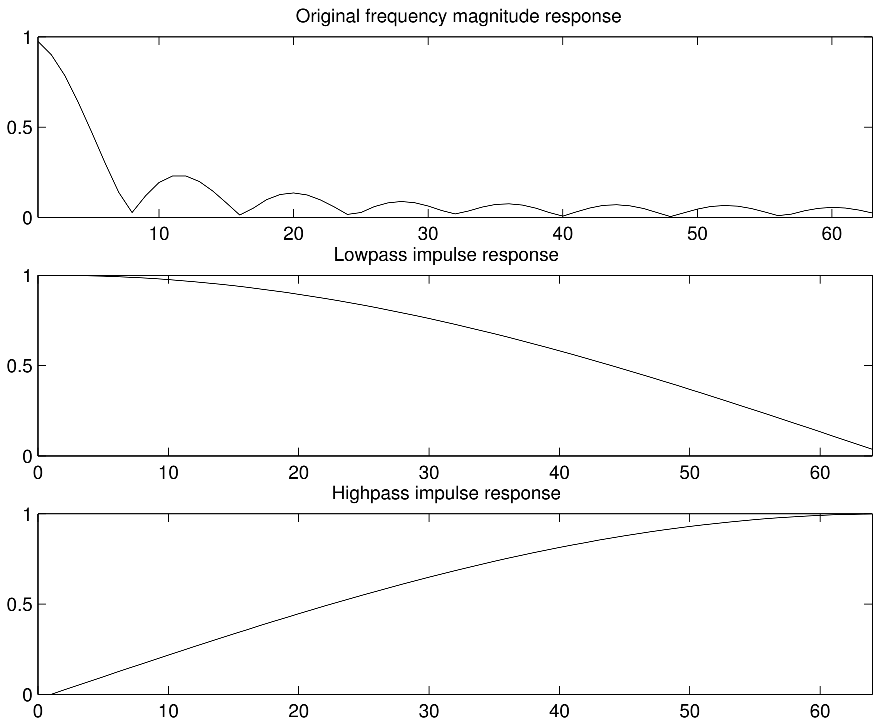



**Figure 3.2:** The frequency content of the example signal, and low/highpass filters.

*Finite Impulse Response (FIR) filters* are a simple class of filters that perform much of the work of digital signal processing. They are composed of multipliers,

adders, and delay elements. In diagrams showing filters, we show a plus sign within a circle. Strictly speaking, this is a summation whenever the number of inputs is greater than two. Architecturally, the summation can be implemented with an adder tree.

Multipliers are rather complex parts, so sometimes they may be replaced by simpler parts. For example, to multiply by 8, it is easier to shift the number three places to the left (since $2^3 = 8$). Of course, the drawback here is that a shift register is not as flexible as a multiplier, and this would only work if the value-to-multiply-by is always a power of 2.

The delay elements can be thought of as registers, in fact, since registers are used to store a value for a short period of time, a register actually implements a delay element. Having two registers in a row has the effect of delaying the signal by two time units. In Figure 3.3, we show this for the general case. The output from the delay unit(s) appears as a time-shifted version of the input. When the second input value enters a single delay unit, the first input value would exit, leading to $x[n]$ entering on the left and $x[n-1]$ exiting on the right. If we consider a group of $k$ delay units, $x[n-k]$ would exit as $x[n]$ enters.

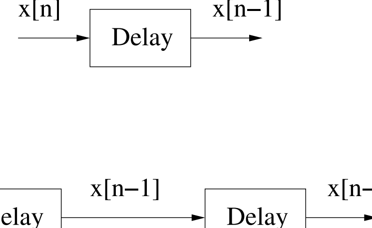



**Figure 3.3:** A digital signal, delayed, appears as a time-shifted version of itself.

The name "Finite Impulse Response" comes from the way the filter affects a signal. An *impulse function* is an interesting input, where the signal is 0 everywhere, except for one place where it has the value of 1 (a single unit). An FIR filter's response to this input is finite in duration, and thus it bears the name Finite Impulse Response filter.

```
unit impulse function x[n] = 1, when n = 0
                      x[n] = 0, when n ≠ 0
```

## 3.1 Parts of a Filter

As mentioned above, there are only a few types of different parts that go to make filters. These are adders (including summations), multipliers, and delay units. In fact, the Infinite Impulse Response (IIR) filter is also made of these basic parts.

To see how filters work, we will start with an adder, Figure 3.4, then work with a multiplier, Figure 3.5. Think of the adder or multiplier as working on one value per signal at a time. The index is very important, and being off by one position affects the outputs. We will come back to this idea, since a delay unit has this effect on the index.

---

**Example:**

If $a$ = [1, 2, 3, 4] and $b$ = [2, 1, 2, 1], what is $c$? See Figure 3.4.

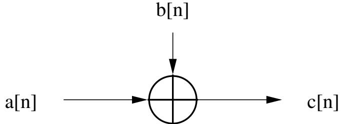



**Figure 3.4:** An adder with two signals as inputs.

**Answer:**

$c[0] = a[0] + b[0]$, $c[1] = a[1] + b[1]$, $c[2] = a[2] + b[2]$, $c[3] = a[3] + b[3]$

$c[0] = 1 + 2$, $c[1] = 2 + 1$, $c[2] = 3 + 2$, $c[3] = 4 + 1$

$c$ = [3, 3, 5, 5]

Thus, we conclude that $c[n] = a[n] + b[n]$.

---

**Example:**

If $a$ = [1, 2, 3, 4] and $b$ = [2, 1, 2, 1], what is $c$? See Figure 3.5.

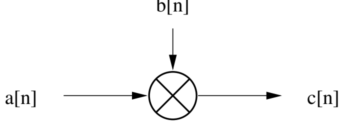



**Figure 3.5:** A multiplier with two signals as inputs.

**Answer:**

$c[0] = a[0] \times b[0]$, $c[1] = a[1] \times b[1]$, $c[2] = a[2] \times b[2]$, $c[3] = a[3] \times b[3]$

$c[0] = 1 \times 2$, $c[1] = 2 \times 1$, $c[2] = 3 \times 2$, $c[3] = 4 \times 1$
$c$ = [2, 2, 6, 4]
Thus, we conclude that $c[n] = a[n] \times b[n]$.

---

**Example:**
If $a$ = [1,2,3,4] and $b$ = [2,1,2,1], what is $c$? See Figure 3.6.

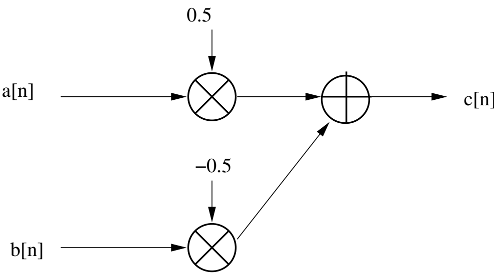



**Figure 3.6:** An example FIR filter with coefficients 0.5 and -0.5.

**Answer:**
$c = [1 \times 0.5 - 2 \times 0.5, 2 \times 0.5 - 1 \times 0.5, 3 \times 0.5 - 2 \times 0.5, 4 \times 0.5 - 1 \times 0.5]$
$c$ = [-0.5, 0.5, 0.5, 1.5]
We see that $c[n] = 0.5 \times a[n] - 0.5 \times b[n]$.

---

The delay unit shifts a signal. We assume that any signal value outside of our index's range is 0, so if a signal is delayed by one unit, we can insert a 0 at the beginning of it. For example, if we delay signal $x$ by one unit to form signal $y$, Figure 3.7, and signal $x$ = [1, 3, 5], then signal $y$ = [0, 1, 3, 5]. How do these two signals relate mathematically?

x[-1] = 0, y[0] = 0,
x[0] = 1, y[1] = 1,
x[1] = 3, y[2] = 3,
x[2] = 5 y[3] = 5

Clearly, $y[n] = x[n-1]$.

Delay units are sometimes shown with "$z^{-1}$" in place of the "D." The reason for this will have to wait until Chapter 8, "The $z$-Transform," but for now, just treat the two as equivalent.

**Figure 3.7:** Signal $y$ is a delayed version of $x$.

## 3.2 FIR Filter Structures

Now that we have seen how the parts make a filter, we will demonstrate some FIR filters and discuss some important characteristics: describing FIR filters by equations, and how the unit impulse function works with them. The FIR filter structure is quite regular, and once the numbers corresponding to the multipliers are known, the filter can be completely specified. We call these numbers the *filter coefficients*. One unfortunate use of the terminology is to call the filter's *outputs* coefficients as well. In this book, "coefficients" will be used to denote the filter coefficients only.

In Figure 3.8, we see an example FIR filter. Notice that the arrows show direction as left to right only, typical of FIR filters. Sure, an arrow points from the input down to the delay unit, but significantly no path exists from the output side back to the input side. It earns the name *feed-forward* for this reason.

Suppose Figure 3.8 has an input $x[n] = [1, 0]$. The outputs would be $[0.5, 0.5]$, and zero thereafter. We find these outputs by applying them, in order, to the filter structure. At time step 0, $x[0]$ appears at the input, and anything before this is zero, which is what the delay element will output. At the next time step, everything shifts down, meaning that $x[0]$ will now be output from the delay element, meanwhile $x[1]$ appears as the current input.

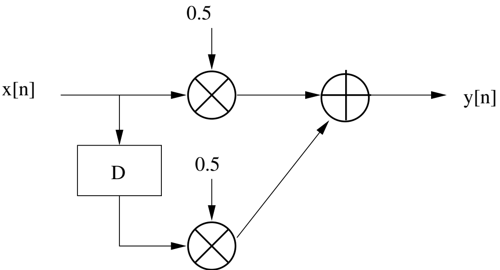



**Figure 3.8:** FIR filter with coefficients {0.5, 0.5}.

How would you express $y[n]$ as an equation? As we observe from calculating $y[0]$ and $y[1]$, we really have the same pattern:

$$y[0] = 0.5x[0] + 0.5x[-1]$$

$$y[1] = 0.5x[1] + 0.5x[0].$$

This generalizes to:

$$y[n] = 0.5x[n] + 0.5x[n-1].$$

A finite and discrete signal's *support* is simply its index range. Think of a signal's support as the amount of samples where the signal has a defined value (which could also include zero). Suppose that you measure the low temperature over a week in winter, in degrees Celsius. If the temperatures measured are $\{2, 1, 0, -1, 0, 2, 0\}$, the support would be still be 7, even though some of the values were 0. For analysis, we might assume that all values before the first sample and after the last sample are 0.

---

**Example:**
For Figure 3.9, if $x = \{1\}$, what is $y$? Express $y$ as an equation.

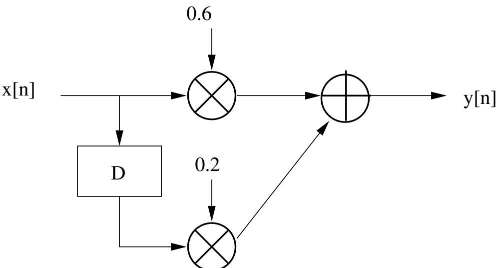



**Figure 3.9:** An example FIR filter with coefficients 0.6 and 0.2.

**Answer:**
It is easy to see that for an input of 1, the output is 0.6. But we must not stop there, since this 1 will be the delay unit's output on the next time step. We can (and should) assume that all future inputs outside of the ones given are zero. On the next time step, we compute $y[1] = 0.2$. For the following time step, notice that the last nonzero input (i.e., 1) no longer appears, and we can stop here.

$$y = [0.6, 0.2]$$

To find the equation, we observe that for input $n$ we have output of $0.6 \times (\textit{current input}) + 0.2 \times (\textit{previous input})$, leading to the generalization:

$$y[n] = 0.6x[n] + 0.2x[n-1].$$

---

Figure 3.10 shows the general form of the FIR filter, for $K + 1$ filter coefficients. (There are $K + 1$ of them because we start at 0 and count to $K$.) The number of filter coefficients is also called the number of *taps*. By convention, the number of taps equals the number of filter coefficients. So a filter with coefficients $\{b_0, b_1, ...$ $, b_K\}$ has $K + 1$ taps, since there are $K + 1$ total filter coefficients. However, it is said to be of order $K$. In other words, the *order* of the filter and the taps express the same idea, but with a difference of 1.

With the structure of Figure 3.10 [11], it is possible to determine the output. It is also possible to determine an equation for the output, which is

$$y[n] = b[0]x[n-0] + b[1]x[n-1] + b[2]x[n-2] + ... + b[K]x[n-K].$$

Notice that whatever index is used for $b[\cdot]$ is also used in $x[n - \cdot]$. This means we can represent everything on the righthand side of the equation as a summation.

$$y[n] = \sum_{k=0}^{K} b[k]x[n-k]$$

---

**Example:**
Given $b = [0.1, 0.2, 0.3]$ and $x = [1, 0, 0, 2, 0, 1, 4, 3]$, what is $y$?
**Answer:**
$y[n] = x[n-2]b[2] + x[n-1]b[1] + x[n]b[0]$

This can be found by noting the above equation, then making a chart.

```
y[0] =    0 b[2]  +    0 b[1]  + x[0]b[0]
y[1] =    0 b[2]  + x[0]b[1]  + x[1]b[0]
y[2] = x[0]b[2]  + x[1]b[1]  + x[2]b[0]
y[3] = x[1]b[2]  + x[2]b[1]  + x[3]b[0]
etc.
```

Plugging in the values of $x[n]$ to form columns multiplied by a scalar:

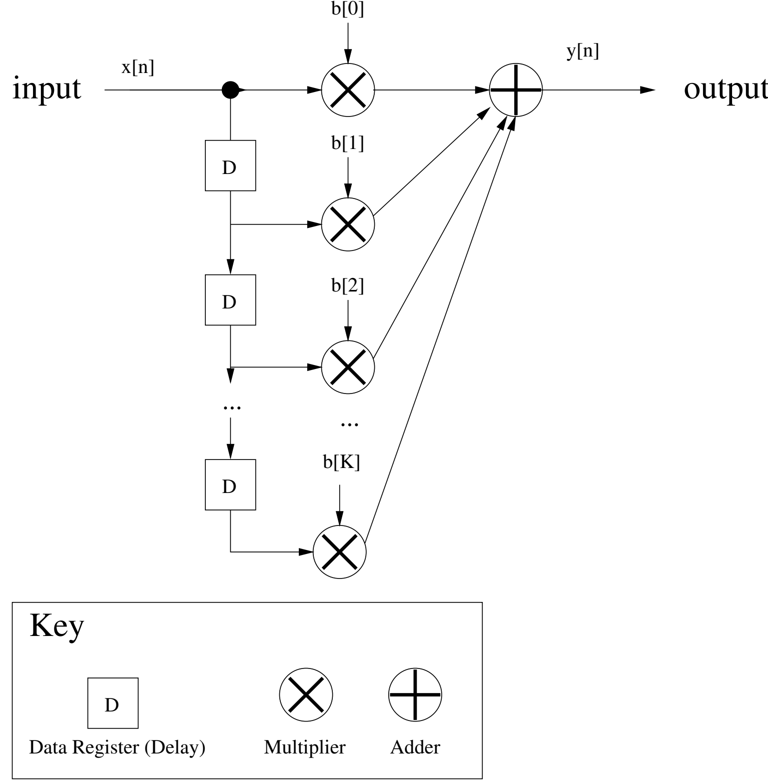



**Figure 3.10:** General form of the FIR filter.

$$y[] = \begin{matrix}0\\0\\1\\0\\0\\2\\0\\1\\4\\3\end{matrix} \times b[2] + \begin{matrix}0\\1\\0\\0\\2\\0\\1\\4\\3\\0\end{matrix} \times b[1] + \begin{matrix}1\\0\\0\\2\\0\\1\\4\\3\\0\\0\end{matrix} \times b[0] = \begin{matrix}0.1\\0.2\\0.3\\0.2\\0.4\\0.7\\0.6\\1.4\\1.8\\0.9\end{matrix}$$

Therefore, $y[n] = [0.1, 0.2, 0.3, 0.2, 0.4, 0.7, 0.6, 1.4, 1.8, 0.9]$.

---

This is nothing more than a matrix multiplied by a vector. In other words,

$$y[] = \begin{bmatrix}0 & 0 & 1\\0 & 1 & 0\\1 & 0 & 0\\0 & 0 & 2\\0 & 2 & 0\\2 & 0 & 1\\0 & 1 & 4\\1 & 4 & 3\\4 & 3 & 0\\3 & 0 & 0\end{bmatrix} \times \begin{bmatrix}0.3\\0.2\\0.1\end{bmatrix}.$$

Generalizing this for any $x$ with eight elements and $b$ with three elements, we have:

$$\begin{bmatrix}y[0]\\y[1]\\y[2]\\y[3]\\y[4]\\y[5]\\y[6]\\y[7]\\y[8]\\y[9]\end{bmatrix} = \begin{bmatrix}0 & 0 & x[0]\\0 & x[0] & x[1]\\x[0] & x[1] & x[2]\\x[1] & x[2] & x[3]\\x[2] & x[3] & x[4]\\x[3] & x[4] & x[5]\\x[4] & x[5] & x[6]\\x[5] & x[6] & x[7]\\x[6] & x[7] & 0\\x[7] & 0 & 0\end{bmatrix} \times \begin{bmatrix}b[2]\\b[1]\\b[0]\end{bmatrix}.$$

We can write one of the values of $y$, say at index 5, as:

$$y[5] = x[3] \times b[2] + x[4] \times b[1] + x[5] \times b[0].$$

With the solution to the above question in mind, notice that this equation

$$y[n] = x[n-2]b[2] + x[n-1]b[1] + x[n-0]b[0]$$

is really the same as

$$y[n] = \sum_{k=0}^{K} b_k x[n-k]$$

with $b_k$ meaning the same thing as $b[k]$.

We call this process *convolution*, and use the symbol "*" (an asterisk) for it. We could say that $y[n] = b * x[n]$, where the argument $[n-k]$ is assumed for $x$.

---

**Example:**
Given $b = [0.3, 0.9, 0.4, 0.7]$, and

$$y[n] = \sum_{k=0}^{K} b_k x[n-k]$$

sketch the filter that implements this equation. Be sure to label it correctly.
**Answer:**
The answer is shown in Figure 3.11.

---

An FIR filter may be represented as a rectangle with numbers inside, as in Figure 3.12. The reason for this is to make it a little easier to work (on an abstract level) with FIR filters.

The input $x[n] = [1]$ is very special; we call it the *unit impulse*, since it has a single nonzero value of one unit. An interesting feature is that it gives us the filter coefficients of an FIR filter. We call $h[n]$ the unit impulse response. In fact, $h$ equals $b$, meaning that the impulse response is the same as the filter coefficients, at least for an FIR filter.

Following is a way to do convolution by hand, in a way analogous to multiplication (see for reference McClellan [6]). First, line up to two signals on the left,

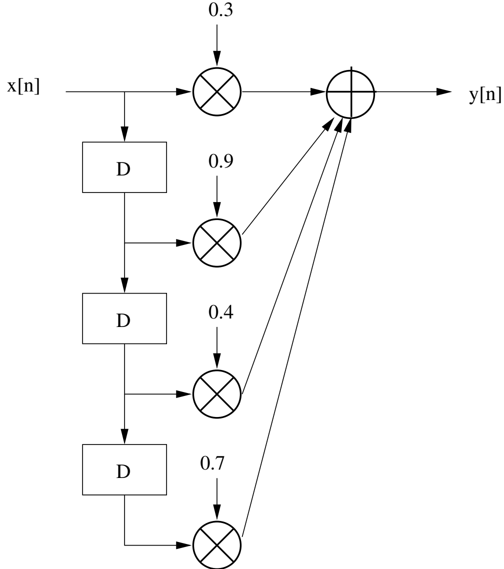



**Figure 3.11:** An example FIR filter.

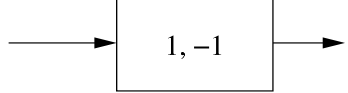



**Figure 3.12:** A representation of an FIR filter.

one above the other. This also works with filter coefficients. Next, multiply the first value of the second signal by each value of the first, and write the result below, starting from the left. Repeat this for each of the second signal's values, indenting the results. Finally, add the values along each column.

For example, suppose signal $x = [1, 2, 3, 4, 5]$ were the input to an FIR filter with coefficients $b = [6, 7, 8]$. What is the output $y$? Following the algorithm given above, we generate this:

```
1  2  3  4  5
6  7  8
-------------
6 12 18 24 30
   7 14 21 28 35
      8 16 24 32 40
-------------------
6 19 40 61 82 67 40.
```

Therefore, we find $y = [6, 19, 40, 61, 82, 67, 40]$. Note that we get the same answer no matter which of the two signals we decide to put in the first line. In other words, we arrive at the same values for $y$ if we convolve $[6, 7, 8]$ with $[1, 2, 3, 4, 5]$ instead of convolving $[1, 2, 3, 4, 5]$ with $[6, 7, 8]$. This can be confirmed below.

```
6  7  8
1  2  3  4  5
-------------
6  7  8
  12 14 16
     18 21 24
        24 28 32
           30 35 40
-------------------
6 19 40 61 82 67 40
```

## 3.3 Causality, Linearity, and Time-Invariance

There are three notable properties of a system:

1. causality
2. linearity

3. time-invariance

A system is *causal* when it uses only current and/or previous inputs to calculate the current output. That is, if the equation describing the inputs to the outputs is given with indexes of $n$ on the left side of the equation, and $n$, $n-1$, $n-2$, etc., on the righthand side. Similarly, we say that a signal is causal when it has only zeros for index values less than zero. In other words, it starts at or after index 0.

---

**Example:**
Which of the following filters are causal? ($K$ is positive)

$y[n] = \sum_{k=0}^{K} b_k x[n-k]$
$y[n] = \sum_{k=0}^{K} b_k x[n+k]$
$y[n] = \sum_{k=-K}^{0} b_k x[n-k]$
$y[n-1] = x[n]$
$y[n] = 0.1x[n]$
$y[n] = 0.1x[n] + 0.4x[n-1] + 0.8x[n-2]$
$y[n] = 0.1x[n] - 0.4x[n-1] - 0.8x[n-2]$
$y[n] = 0.1x[n] + 0.4x[n+1] + 0.8x[n-1]$
$y[n] = 0.1x[n] + 0.4x[n+1] + 0.8x[n+2]$
**Answer:**
*yes, no, no, no, yes, yes, yes, no, no*

---

When a system has an output expressed as a sum of inputs multiplied by constants, as with FIR filters, then this system is *linear*. A linear system has scaling and additivity properties. Scaling means that we get the same result whether we multiply the input by a constant or the output by that constant, as in Figure 3.13. For a linear system, $y_1 = y_2$. For example, suppose we have a system $y[n] = 2x[n] + x[n-1]$. If we replaced each value of $x$ with $ax$, then we would calculate $2ax[n] + ax[n-1]$ to find our output. This is the same as $a(2x[n] + x[n-1])$, the same as if our output were multiplied by $a$, which is what scaling means.

Additivity means that we can add two signals together before applying a linear filter, or we can apply the linear filter to each signal and add their results, as in Figure 3.14, i.e., $y_3 = System(x_1 + x_2) = y_1 + y_2$. Either way, we end up with the same answer.

These properties hold true as long as the inputs are multiplied by constants (i.e., not multiplied by each other). That is, the system described by $y[n] = c_0 x[n] + c_1 x[n-1] + c_2 x[n-2] + ... + c_K x[n-K]$ is linear ($y[n]$ is the current output, $x[n]$

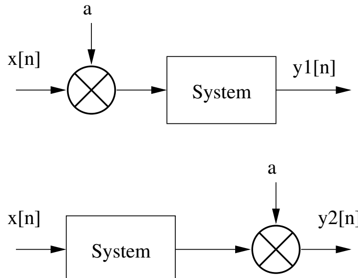



**Figure 3.13:** Linear condition 1: scaling property.

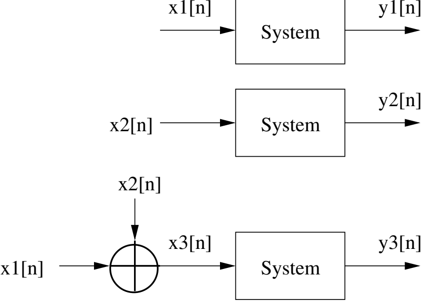



**Figure 3.14:** Linear condition 2: additive property.

is the current input, $x[n-1]$ is the previous input, and all $c$ values are constants). When two such systems are placed in series, the order does not matter. For example, consider the system $y_1[n] = 2x[n]$ and the system $y_2[n] = x[n] + x[n-1]$, both of which are linear. If $x[n]$ is operated on by system 1, and the result ($y_1[n]$) is operated on by system 2, the result will be $z_1[n] = (2x[n]) + (2x[n-1])$. Changing the order of systems would give us $z_2[n] = 2(x[n] + x[n-1])$. As can be seen, $z_1$ and $z_2$ are equal.

As a counter example of linearity, consider the system where $y[n] = x[n]^2$. Let $y_1[n] = x_1[n] \times x_1[n]$, and $y_2[n] = x_2[n] \times x_2[n]$. Defining $x_3[n] = x_1[n] + x_2[n]$, we find $y_3[n] = (x_1[n]+x_2[n]) \times (x_1[n]+x_2[n]) = x_1[n]^2 + x_2[n]^2 + 2x_1[n]x_2[n]$. Clearly, this is not the same as $y_1[n] + y_2[n]$. Therefore, this system is not linear.

*Time-invariant* means that a system's output reflects the same shift as the input signal. That is, if the input $x[n]$ is delayed by $k$ units, then the input will appear as $x[n-k]$, and the output $y[n]$ will appear as $y[n-k]$. Another way of saying this is that the results are the same, regardless of whether the input is delayed or the output is delayed. FIR filters are time-invariant, as long as its coefficients do not change with time. (Adaptive filters are an exception; they change their coefficients according to the input, for example, to try to predict the next input.) One system that is NOT time-invariant is the down-sampler. A down-sampler will get rid of values, such as down-sampling by 2 gets rid of every other value. An input delay of one time unit means that the down-sampler would output the odd-indexed values instead of the even ones.

---

**Example:**
Suppose $system_1$ is an FIR filter with $h_1[n] = \{1, 1, 1, 1\}$, and $system_2$ is an FIR filter with $h_2[n] = \{2, 0, 2\}$. Let $x[n] = \{2, 4, 6, 8, 10\}$. If $output_1 = h_2 * (h_1 * x)$ and $output_2 = h_1 * (h_2 * x)$ How do $output_1$ and $output_2$ relate to each other? Why?
**Answer:**
They are equal. Both $system_1$ and $system_2$ are Linear, and Time-Invariant (LTI), so the effect of the convolution operations is equivalent to $(h_2 * h_1) * x$.

---

**Test for Causality:**

- Express the system in terms of an input/output equation. For example, $y[n] = system(x) = 2x[n] - 3x[n-1]$.
- Examine the input indices versus the output indices. If one or more input index is greater than the output index, then the system is noncausal. Otherwise, it is causal. For example, the index for output $y$ is $n$, while the indices for input

$x$ are $n$ and $n-1$. Since $n \geq n$ and $n \geq n-1$, we know that this system is causal.

**Test for Linearity:**

- Express the system in terms of an input/output equation, for example, $system(x) = 2x[n] - 3x[n-1]$.
- Test $a \times system(x)$ for equality with $system(a \times x)$. For example,

$$\begin{aligned} system(a \times x) &= 2(ax[n]) - 3(ax[n-1]) \\ &= a(2x[n]) + a(-3x[n-1]) \\ &= a(2x[n] - 3x[n-1]) \\ &= a \times system(x). \end{aligned}$$

- Test $system(x_1 + x_2)$ for equality with $system(x_1) + system(x_2)$. For example,

$$\begin{aligned} system(x_1 + x_2) &= 2(x_1[n] + x_2[n]) - 3(x_1[n-1] + x_2[n-1]) \\ &= 2x_1[n] + 2x_2[n] - 3x_1[n-1] - 3x_2[n-1] \\ &= (2x_1[n] - 3x_1[n-1]) + (2x_2[n] - 3x_2[n-1]) \\ &= system(x_1) + system(x_2). \end{aligned}$$

  Both tests are true, therefore this system is linear.

**Test for Time-Invariance:**

- Express the system in terms of an input/output equation. For example, $system(x) = 2x[n] - 3x[n-1]$.
- Examine the system to see if an arbitrary number of delays has the same impact regardless of whether it is applied to the input or the output. In other words, test $delay(system(input))$ for equality with $system(delay(input))$. Note that we should consider all $delay()$ functions; if the test fails for a delay of one time unit but not for a delay of two, we should conclude that the system is NOT time-invariant.

  For example, delaying $x$ by $d$ time units means that we replace the index $n$ with $n - d$. Thus, $delay_d(x) = x[(n-d)]$, meaning that

$$system(delay_d(x)) = 2x[(n-d)] - 3x[(n-d)-1].$$

We know from above that

$$system(x) = 2x[n] - 3x[n-1],$$

so delaying this produces

$$delay_d(system(x)) = 2x[(n-d)] - 3x[(n-d)-1].$$

LTI systems are implemented with convolution, and the convolution operator is *commutative*, meaning that the order of the operands does not matter, as in $h_1 * h_2 = h_2 * h_1$. It is also *associative*, as we saw above, meaning that we get the same answer over three operands, no matter which two we convolve first, $h_2 * (h_1 * x) = (h_2 * h_1) * x$. Convolution also has the *distributive* property [12]. For example, with multiplication and addition, we know that $a(b+c) = ab+ac$. This also works with convolution and addition, as in $x * (h_1 + h_2) = x * h_1 + x * h_2$. Note that addition is NOT distributive over convolution. Just as $a + (bc) \neq (a+b)(a+c)$ (except under Boolean algebra), $x + (h_1 * h_2) \neq (x * h_1) + (x * h_2)$. To show this, try it with a very simple set of signals, such as $x = \{x_0, x_1, x_2\}$, $h_1 = \{a\}$, and $h_2 = \{b\}$.

## 3.4 Multiply Accumulate Cells

A *Multiply Accumulate Cell* (MAC) is a regular structure that performs a multiplication and an addition. In hardware design, regularity is important. The following figure, Figure 3.15, shows a multiply accumulate cell. As its name implies, it performs multiply and add operations.

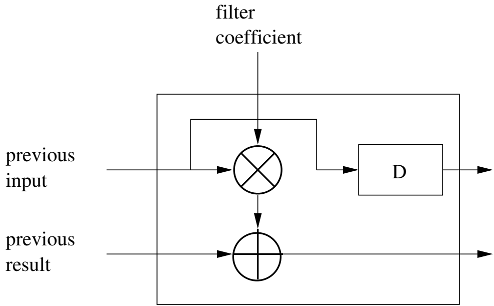



**Figure 3.15:** Multiply accumulate cell.

MACs can be arranged in a line, as shown in Figure 3.16. To alter this filter for more taps, simply add more MACs to the right of it. From this figure, it is easy to verify that this matches the general FIR filter form—just rearrange it to have the adders along the top and the delay units down the left side.

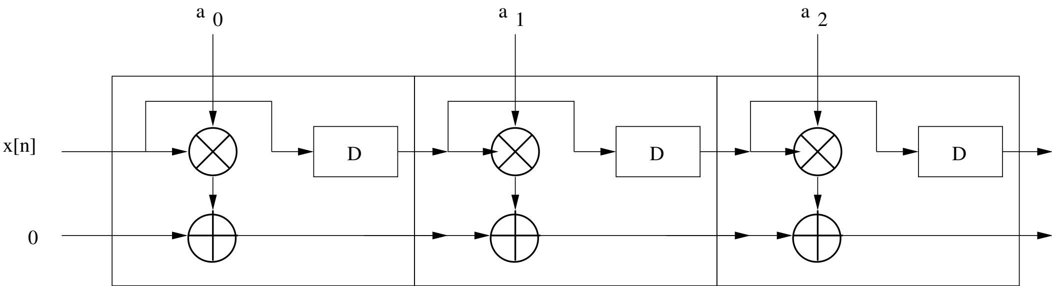



**Figure 3.16:** Multiply accumulate cells as a filter.

Similar to MACs are *Processing Elements* (PEs), which are less standardized. PEs are modular like MACs, though their operations are not fixed.

## 3.5 Frequency Response of Filters

FIR filters can implement several different functions just by changing the filter coefficients. The functions of a filter depend upon how it treats an input signal in the frequency-domain. That is, an input signal will have different frequencies, and the filter will leave some frequencies alone while getting rid of others. Thus, a filter can be described as *lowpass*, *highpass*, *bandpass*, *bandstop*, or *notch*, depending on which frequencies pass through the filter relatively unchanged. Some people prefer the term “bandreject” to “bandstop,” since these filters reject a band of frequencies. Also, the word “notch” is often used as just another name for a bandstop filter, though notch filters usually reject a very narrow frequency range. There are other types of filters, such as differentiators, allpass, and adaptive filters, but they are outside the scope of this chapter. Also, a system is a superset of a filter. In other words, a filter is a type of system, but a system is not necessarily a filter.

The terms lowpass and highpass are fairly self-explanatory. A good example of a lowpass filter’s frequency magnitude response appears in Figure 3.17, while the frequency magnitude response for an example highpass filter can be seen in Figure 3.18. Along the x-axis we have a percentage of the frequencies from 0 to 1, though the actual frequencies will depend upon the sampling rate $f_s$ and range from 0 Hz to $f_s/2$ Hz, which is explained in Chapters 4 and 5, “Sinusoids” and “Sampling.”

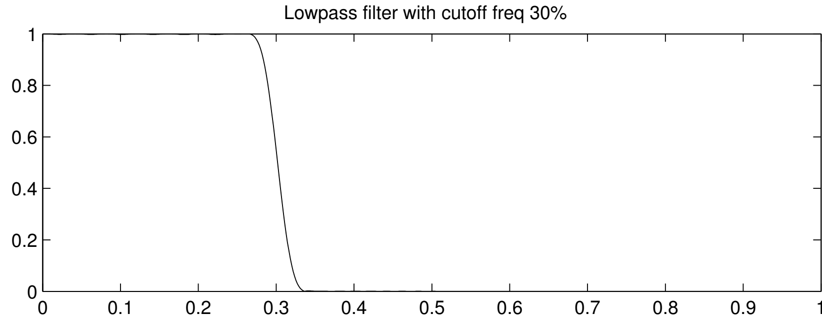



**Figure 3.17:** Frequency magnitude response for a lowpass filter.

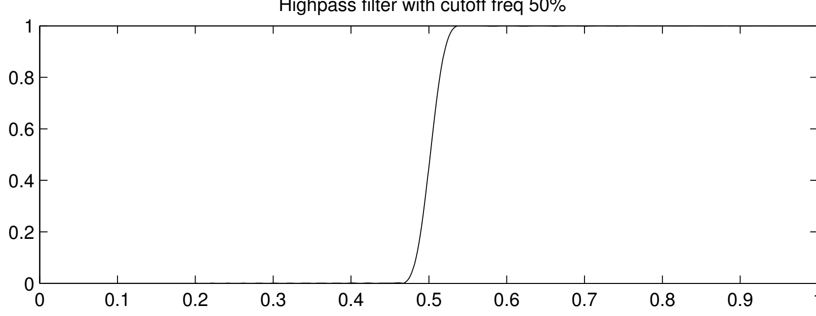



**Figure 3.18:** Frequency magnitude response for a highpass filter.

So what makes Figures 3.17 and 3.18 good representatives of low- and highpass filters? Both examples clearly show the sharp cutoff between the passband (the frequencies that pass through the filter) and the stopband (the frequencies that the filter attenuates). Refer to Figure 3.19 for a graphical representation of these terms. Filters have a transition band between the passband and the stopband, where the frequency magnitude response falls from about 100% to about 0%. But we have to be careful to say "about," since filters have passband ripple and stopband ripple. Instead of remaining a constant magnitude response in the passband, for example, the response actually undulates up and down much like a sine wave, called ripple. When specifying the transition band, we factor in these ripples. Figure 3.19 shows a close-up view of the passband ripple (top) followed by the transition band (middle) and the stopband ripple (bottom). This figure differs from Figure 3.17 (aside from the fact that it has a different cutoff frequency) in that the y-axis has very small increments for the top and bottom subplots. Passband and stopband ripple occurs in both figures, but simply is not visible in Figure 3.17. Why is the stopband ripple shown as a series of small peaks? The plot we see is the frequency magnitude response, which comes from the discrete Fourier transform. The DFT returns complex data, and we use the function $\sqrt{real^2 + imaginary^2}$ to find the frequency magnitude response. So instead of seeing a sinusoid-like pattern around the x-axis, we see only positive values. Are the peaks in the stopband getting smaller from left to right? Yes, and this is common.

A lowpass filter may be called an *averaging* filter, while a highpass filter could be referred to as a *differencing* filter, since these terms describe the filter's functions. For example, if an FIR filter has two coefficients of 0.5 and 0.5, we see that the output, $y[n] = 0.5x[n] + 0.5x[n-1]$, or equivalently, $y[n] = (x[n] + x[n-1])/2$, finds the average of the current and previous inputs. If we examine the frequency response of this averaging filter, we see that low-frequency (slowly changing) parts appear in the output, while the filter decreases high-frequency (quickly changing) parts. For the differencing filter, we could have coefficients of {0.5, -0.5}, where $y[n] = (x[n] - x[n-1])/2$. This gives the change between the two samples, scaled by 1/2 to be consistent with the averaging filter. As one may expect, the frequency response of such a filter would de-emphasize the slowly changing part of a signal, and emphasize the quickly changing part, leading us to conclude that it is a highpass filter.

*Bandpass* means that frequencies within a certain band (or range) pass through the filter, while frequencies outside that range are *attenuated* (reduced in amplitude). For example, a bandpass filter may allow frequencies between 10 kHz and 20 kHz to exit the filter intact, while greatly reducing any frequencies under 10 kHz or above 20 kHz. Figure 3.20 shows the frequency magnitude response of an example

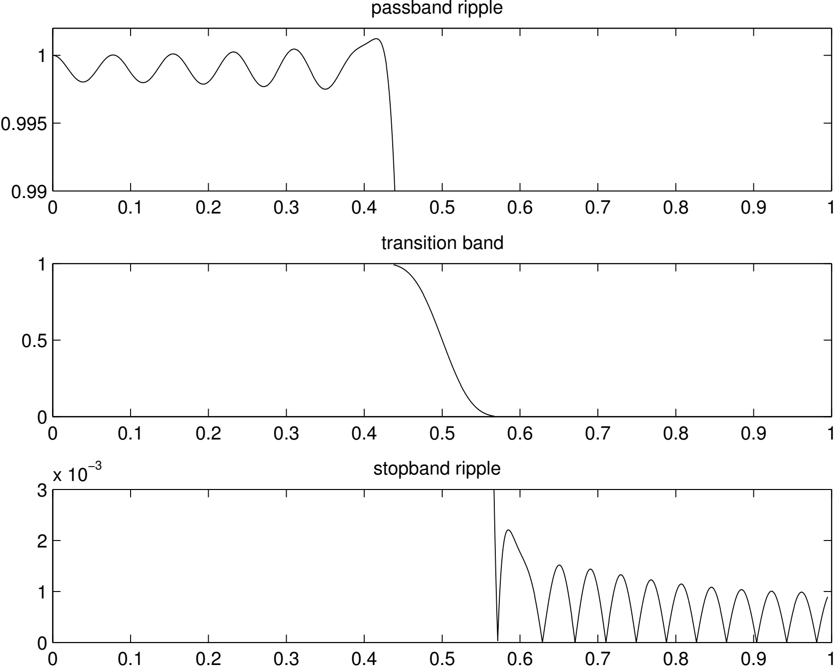



**Figure 3.19:** Passband, transition band, and stopband, shown with ripples.

bandpass filter. A *bandstop* filter is the opposite of the bandpass filter; it attenuates frequencies within a certain range, while passing through any frequencies above or below that range. In Figure 3.21 we see an example bandstop filter, made with the same parameters as the bandpass filter, but obviously these two have an opposite effect on a signal.

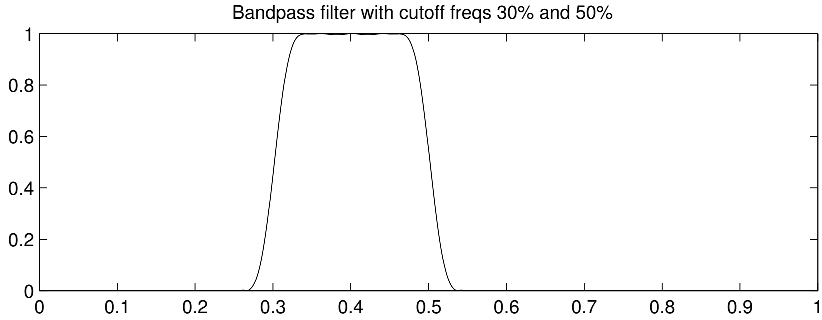



**Figure 3.20:** Frequency magnitude response for a bandpass filter.

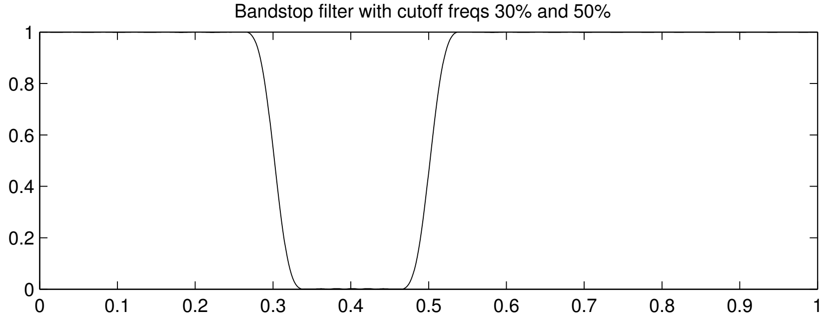



**Figure 3.21:** Frequency magnitude response for a bandstop filter.

Two other example filters' responses are shown, as special cases of the above. Figure 3.22 shows a bandstop filter with a very narrow frequency range being attenuated. Notch filters like this one eliminate noise at a specific frequency. Finally, Figure 3.23 shows the frequency magnitude response for a bandpass filter, except that it has two passbands. Such a filter would be great for getting rid of parts of the signal with undesired frequencies, allowing frequencies in one band or the other

to pass through.

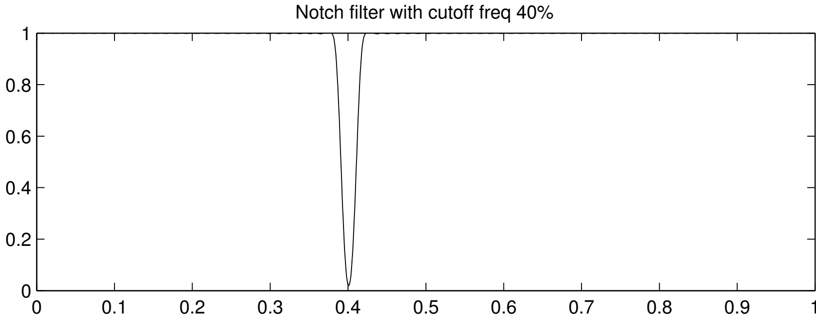



**Figure 3.22:** A notch filter.

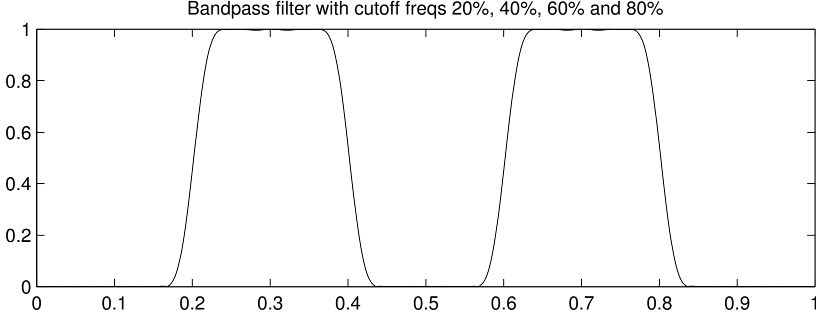



**Figure 3.23:** Frequency magnitude response for a bandpass filter with two passbands.

To see how a filter behaves, we can view a plot of its frequency magnitude response, as we did above. To get the frequency magnitude response, first perform the Discrete Fourier Transform (DFT, Chapter 6, "The Fourier Transform") on the filter coefficients. These coefficients may be zero padded to make the curves smooth. An assumption is that the number of coefficients passed to this function is less than 128, a convenient power of 2. The example below looks at two filters, a lowpass filter and a highpass filter. We show only the first half of the spectrum. The second half will be only a mirror image, since the input signal is real.

```
% Show freq magnitude response for an FIR filter's coefficients
% We assume the number of coefficients is less than 128.
%

function freq_mag_resp(h)

% Zero pad so that the results will look nice
x = h;
x(length(h)+1:128) = 0;

% Get the frequency magnitude response of x
X = fft(x);
X_mag  = abs(X);

% Display the results
half = ceil(length(X_mag)/2);
plot(1:half, X_mag(1:half), 'b');
title('frequency magnitude response');
```

Filter design is discussed in Chapter 10, "Applications." MATLAB provides some functions to do this in the signal processing toolbox, including `fir1` and `fir2`. For example, the graph in Figure 3.21 can be made with the following code. Notice that this code does much the same thing as the above *freq_mag_resp* function, but with more compact code.

```
% Get bandstop filter coefficients
B2 = fir1(100, [0.3, 0.5], 'stop');
x = zeros(1, 1000);
x(50) = 1;
Y2 = fft(conv(x,B2));
half = 1:ceil(length(Y2)/2);
plot(half/max(half), abs(Y2(half)), 'b');
```

To make a highpass filter like Figure 3.18, modify the above code to use different filter coefficients:

```
B4 = fir1(100, 0.5, 'high');
```

Also, Figure 3.23's frequency magnitude response can be found with the following filter.

```
B5 = fir1(100, [0.2, 0.4, 0.6, 0.8], 'bandpass');
```

## 3.6 IIR Filters

The Finite Impulse Response (FIR) filter uses feed-forward calculations only. If we allowed feed-back, then the filter's response to an impulse would not necessarily be finite. Therefore, filters that feed-back are called *Infinite Impulse Response (IIR)* filters.

Consider the filter pictured in Figure 3.24. The equation describing its output is:

$$y[n] = 0.4y[n-1] + 0.6x[n] + 0.2x[n-1].$$

Often, the delayed output terms appear first, as above. The output $y[\,]$ appears on both sides of the equation. This means that the current output depends partly on the *previous* output.

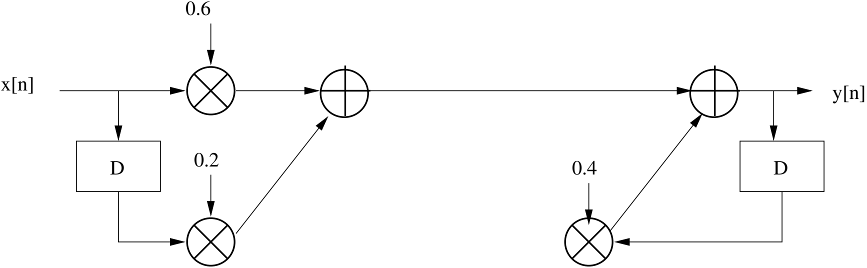



**Figure 3.24:** A filter with feed-back.

If an impulse function passes through this filter, we have the following outputs:

| input | output |
|---|---|
| 0 | 0 |
| 1 | 0.6 |
| 0 | 0.44 |
| 0 | 0.176 |
| 0 | 0.0704 |
| 0 | 0.02816 |
| 0 | 0.011264. |

The output gets smaller each time, and does approach zero. Recall limits of functions; this is important in analyzing these filters. For this filter, since all future impulse-function inputs will be 0, the future outputs will be 0.4 of the previous output, due to this specific feed-back coefficient. Theoretically, the output will

never reach zero, but practically speaking, a digital filter has finite precision, so there will come a time when it outputs zero as the answer.

However, suppose we have the filter of Figure 3.25. At first glance, this filter looks the same as the one in Figure 3.24. The difference is small, but significant: the coefficients are different. Specifically, the feed-back coefficient is 1.

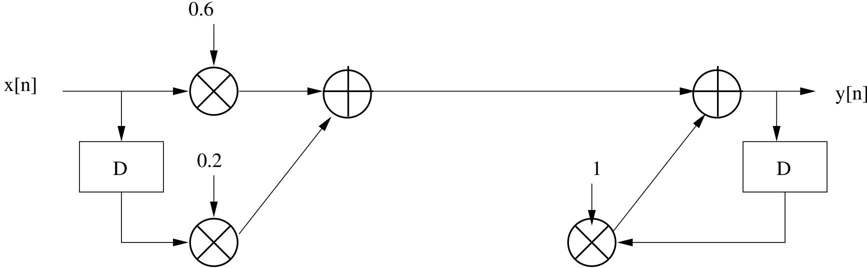



**Figure 3.25:** Another filter with feed-back.

The output for this filter is:

| input | output |
|---|---|
| 0 | 0 |
| 1 | 0.6 |
| 0 | 0.8 |
| 0 | 0.8 |
| 0 | 0.8. |

The filter will produce the same output, 0.8, as long as the input remains zero. This is an *infinite impulse response* (IIR) filter.

Infinite impulse response filters, like the one shown in Figure 3.26, may have outputs that are not so well behaved. Once again, the feed-back coefficient has been changed, this time to 1.1.

The output for the filter in Figure 3.26 is:

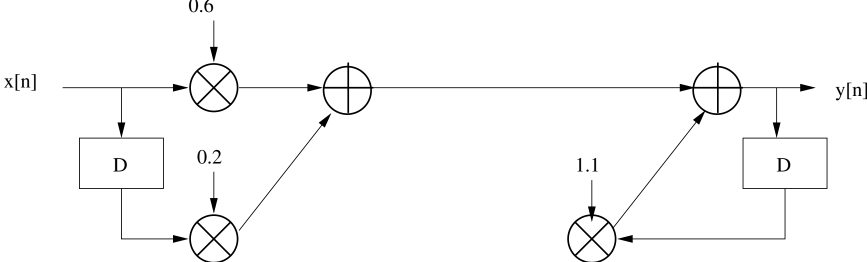



**Figure 3.26:** A third filter with feed-back.

| input | output |
|---|---|
| 0 | 0 |
| 1 | 0.6 |
| 0 | 0.86 |
| 0 | 0.946 |
| 0 | 1.0406 |
| 0 | 1.14466 |
| 0 | 1.259126. |

The output will continue to get larger in magnitude. This demonstrates a problem with IIR filters, that the output may approach infinity (positive or negative infinity). It is also possible for an IIR filter's output to oscillate toward infinity; that the output becomes larger in magnitude, even though the sign may change. A filter is said to be *stable* when the output approaches zero once the input falls to zero. If the output oscillates at a constant rate, we call it *conditionally stable.* If the output approaches positive or negative infinity (such as the example above), or if it oscillates toward infinity, we say that the IIR filter is *unstable.*

The general form of the IIR filter is shown in Figure 3.27 [11]. Other texts refer to this as *Direct Form I*, due to the direct correspondence between the figure and the equation that describes it.

## 3.7 Trends of a Simple IIR Filter

Suppose we have a simple IIR filter, as in Figure 3.28.

By examining the figure, we can find the input/output relation as:

$$y[n] = bx[n] + ay[n-1].$$

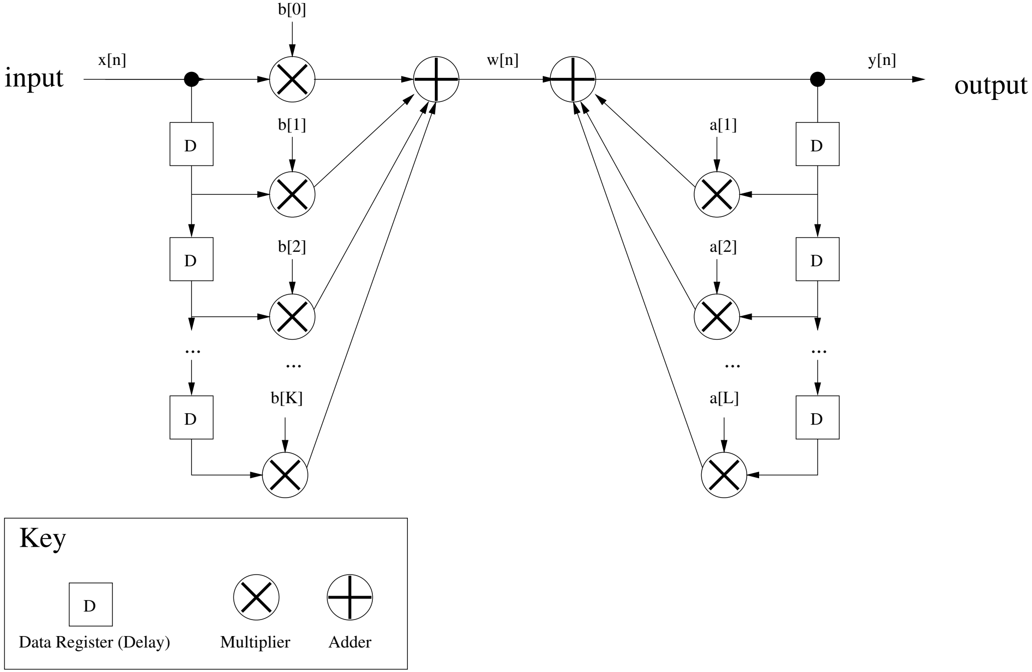



**Figure 3.27:** General form of the IIR filter.

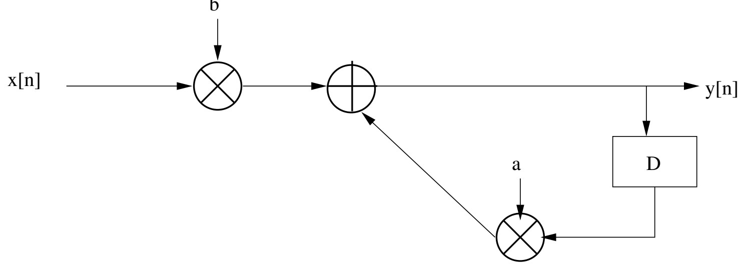



**Figure 3.28:** A simple IIR filter.

For an impulse function as input, only the initial value of $y$ would be directly impacted by $x$. That is, assuming that the system is initially at rest ($y[-1] = 0$), we compute the first output:

$$y[0] = bx[0] + 0.$$

For the impulse function, only $x[0]$ has a nonzero value, and $x[0] = 1$. So the output becomes:

$$y[0] = b$$

and

$$y[n] = ay[n-1], \qquad n > 0.$$

How the output looks over time depends directly on the value of $a$. The following code gives us the output, shown in Figure 3.29.

```
N = 20;
y(1) = 10;
for n=2:N+1
    y(n) = a * y(n-1);
end
```

From the graphs, we see the output either approaches zero (for values $a = -0.5$ and $a = 0.5$), oscillates ($a = -1$), stays constant ($a = 1$), or gets larger in magnitude for successive outputs ($a = -2$ and $a = 2$). Later, in Chapter 8, we will see why this is the case.

## 3.8 Correlation

Correlation determines how much alike two signals are, using convolution. Correlation by itself generates a single number, called a *correlation coefficient* [13]. A large, positive correlation coefficient indicates a relation between the signals, while a correlation coefficient close to zero means that the two signals are not (or that they are not lined up properly). A negative number indicates a negative correlation, meaning that one signal tends to decrease as the other increases [14].

The problem occurs when the input signals are not lined up with one another, since a single correlation calculation may give the appearance that the two signals are not related. To fix this, one of the signals should be shifted, but how much to shift it is not known. Of course, a human observer could say how one signal should be shifted to line it up with the other. This is an example of a problem that is easy for a human to solve, but difficult for a computer. Therefore, one signal is shifted up to $N$ times, and the maximum value from the *cross-correlation* is chosen.

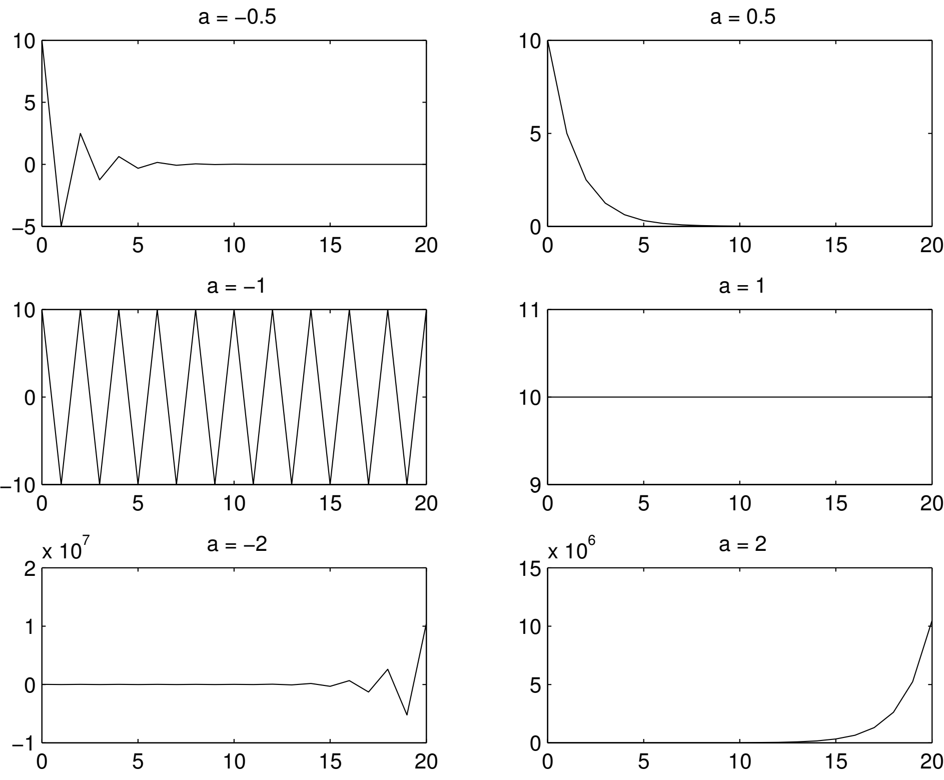



**Figure 3.29:** Output from a simple IIR filter.

Correlation can be performed with convolution, by reversing one signal. Instead of considering signal $y[k]$ being indexed from 0 to some positive limit $K$, we will flip it around the zeroth sample, and consider that it starts at $-K$. For example, if $y[n] = \{5, 3, 1, 6\}$, we interpret this as $y[0] = 5, y[1] = 3, y[2] = 1$, and $y[3] = 6$. If we flip this signal around, we would have $y[-3] = 6$, $y[-2] = 1$, $y[-1] = 3$, and $y[0] = 5$. When we convolve $y$ with $x$, we would sum $x[n] \times y[n-k]$. Using the flipped version of $y$, the only thing to change is the index, e.g., $x[n] \times y[n-(-k)]$ or simply $x[n] \times y[n+k]$.

The output signal from correlation is known as *cross-correlation*, except in the case where the same signal is used for both inputs, which is called *autocorrelation* [12]. However, the number of data samples influences the cross-correlation, so the cross-correlation should be divided by $N$, the number of data samples [14].

$$s_{x,y}[k] \approx \frac{1}{N} \sum_{n=0}^{N-1} x[n]y[n+k]$$

The above equation gives an approximation for $s_{x,y}[k]$, since it does not take the signal's average into account, and assumes that it is 0 for both $x$ and $y$. If the signal averages are nonzero, we need to use the following equation (*cross-covariance*) instead:

$$s_{x,y}[k] = \sum_{n=0}^{N-1} x[n]y[n+k] - \frac{(\sum_{n=0}^{N-1} x[n])(\sum_{n=0}^{N-1} y[n])}{N}.$$

Obtaining a good estimate for the cross-correlation means that we must find the autocovariance as well:

$$s_{x,x} = \sum_{n=0}^{N-1} x[n]^2 - \frac{(\sum_{n=0}^{N-1} x[n])^2}{N}.$$

To find $s_{y,y}$, use the above equation for $s_{x,x}$, and replace $x$ with $y$.

The cross-correlation is estimated to be:

$$\rho_{x,y}[k] = \frac{s_{x,y}[k]}{\sqrt{s_{x,x}s_{y,y}}}.$$

For the autocorrelation, replace $y$ with $x$. $s_{x,x}[k]$ can be found by substituting $x$ for both parameters in the equation for $s_{x,y}[k]$.

$$\rho_{x,x}[k] = \frac{s_{x,x}[k]}{s_{x,x}}.$$

Note that $\rho_{x,y}[k]$ is an *estimate* of the correlation, since we are dealing with sampled data, not the underlying processes that create this data. As you may have guessed, these formulas come from probability and statistics. The cross-correlation is related to variance, $\sigma^2$. For more information, see *Probability and Statistics for Engineers and Scientists, Third Edition* by Walpole and Myers [13].

A few examples about correlation follow. Here we have $x$ and $y$, two signals that are identical, but shifted. To compute the correlation, we use a simple approximation. First, we find the point-for-point multiplication of $x$ and $y$, with the codes that follows.

```
x*y.'
```

Notice that we use the transpose of $y$ (otherwise, we would get an error). Also, after the first computation, we use a different range for $y$ by specifying `[y(10), y(1:9)]`, which concatenates the last value for $y$ with the first nine values. There is no need to find the sum of the point-for-point multiplication with the `sum( )` command, since this is done automatically. (Remember that $[a\ b\ c] \times \text{transpose}[d\ e\ f] = ad + be + cf$, a single value.) Finally, we divide by the number of samples. This code only works because we have been careful to keep $x$ and $y$ the exact same length, and because $x$ and $y$ happen to have an average of zero.

```
>> x = [ 0 0 1 5 1 -2 -3 -2 0 0 ];
>> y = [ 1 5 1 -2 -3 -2 0 0 0 0 ];
>> x*y.'/length(x)

ans =

   -0.8000

>> x*[y(10), y(1:9)].'/length(x)

ans =
```

```
     2

>> x*[y(9:10), y(1:8)].'/length(x)

ans =

    4.4000

>> x*[y(8:10), y(1:7)].'/length(x)

ans =

     2

>> x*[y(7:10), y(1:6)].'/length(x)

ans =

   -0.8000

>> x*[y(6:10), y(1:5)].'/length(x)

ans =

   -1.9000

>> x*[y(5:10), y(1:4)].'/length(x)

ans =

   -1.3000

>> x*[y(4:10), y(1:3)].'/length(x)

ans =

   -0.4000

>> x*[y(3:10), y(1:2)].'/length(x)
```

```
ans =

   -1.3000

>> x*[y(2:10), y(1)].'/length(x)

ans =

   -1.9000
```

Notice how the answer varied from $-1.9$ to 4.4, depending on the alignment of the two signals. At the largest magnitude's value (furthest away from 0), 4.4, the signal given for $y$ was actually `[y(9:10), y(1:8)]`.

```
>> [y(9:10), y(1:8)]

ans =

     0     0     1     5     1    -2    -3    -2     0     0

>> x

x =

     0     0     1     5     1    -2    -3    -2     0     0
```

As the above output shows, the maximum value occurred when the two signals were exactly aligned. A second example is given below. Here, we keep $x$ the same, but we use a negated version of $y$. Notice that the values generated by our simple correlation function are the same values as above, except negated. Again, the value largest in magnitude (-4.4) occurs when the signals are aligned.

```
>> clear all
>> x = [ 0 0 1 5 1 -2 -3 -2 0 0 ];
>> y = [ 1 5 1 -2 -3 -2 0 0 0 0 ];
>> yn = -y
```

```
yn =

    -1    -5    -1     2     3     2     0     0     0     0

>> x*yn.'/length(x)

ans =

    0.8000

>> x*[yn(10), yn(1:9)].'/length(x)

ans =

    -2

>> x*[yn(9:10), yn(1:8)].'/length(x)

ans =

   -4.4000

>> x*[yn(8:10), yn(1:7)].'/length(x)

ans =

    -2

>> x*[yn(7:10), yn(1:6)].'/length(x)

ans =

    0.8000

>> x*[yn(6:10), yn(1:5)].'/length(x)

ans =
```

```
    1.9000

>> x*[yn(5:10), yn(1:4)].'/length(x)

ans =

    1.3000

>> x*[yn(4:10), yn(1:3)].'/length(x)

ans =

    0.4000

>> x*[yn(3:10), yn(1:2)].'/length(x)

ans =

    1.3000

>> x*[yn(2:10), yn(1)].'/length(x)

ans =

    1.9000
```

Technically, the value returned above is not the correlation, but the cross-covariance. One problem is that the shortcut used above only works when the signal has an average of 0. Another problem with this value is that it does not tell us if the two signals are an exact match. That is, in the first example above, the cross-covariance between $x$ and $y$ was found to be 4.4, but if we were to compare signal $x$ with some other signal, could the cross-covariance be higher? Actually, the answer is yes! The following two commands show why this is the case. Taking the signal $y$ and multiplying each value by 10, then finding the cross-covariance, leads to an answer 10 times the original.

```
>> clear all
>> x = [ 0 0 1 5 1 -2 -3 -2 0 0 ];
>> y = [ 1 5 1 -2 -3 -2 0 0 0 0 ];
```

```
>> y10 = y*10

y10 =

    10    50    10   -20   -30   -20     0     0     0     0

>> x*[y10(9:10), y10(1:8)].'/length(x)

ans =

    44
```

To fix this, we find the cross-correlation, which uses two more pieces of information: the autocovariance for each signal. In the code, these are variables $sxx$ and $syy$. The variable $sxy$ is very similar to the above examples, except that no assumption is made about the signal averages.

The code below finds the cross-correlation between two signals.

```
%
%  correlate.m - How much are 2 signals alike?
%
%  Usage:  [rho] = correlate(k, x, y);
%     inputs:  k = shift (integer between 0 and length(y)-1)
%              x, y = the signals to correlate
%     output:  rho = the correlation coefficient -1..0..1
%
%  Note: This function assumes length(x) = length(y).
%
function [rho] = correlate(k, x, orig_y)

% Shift y by k units
y = [orig_y(k+1:length(orig_y)), orig_y(1:k)];

N = length(x);

sxx = x*x.' - sum(x)*sum(x)/N;
syy = y*y.' - sum(y)*sum(y)/N;
sxy = x*y.' - sum(x)*sum(y)/N;
```

```
rho = sxy / sqrt(sxx*syy);
```

When run, we see that the output rho ($\rho$), the cross-correlation, is a much better value to use since it gives us an indication (up to 1) of how well the two signals match.

```
>> clear all
>> x = [ 0 0 1 5 1 -2 -3 -2 0 0 ];
>> y = [ 1 5 1 -2 -3 -2 0 0 0 0 ];
>> yn = -y;
>> y10 = y*10;
>> rho = correlate(8, x, y)

rho =

     1

>> rho = correlate(8, x, y10)

rho =

     1

>> rho = correlate(8, x, yn)

rho =

    -1
```

The last example shows that a negative correlation exists between $x$ and $yn$ (the negated version of $y$). The cross-correlation function takes care of the fact that the signals may be scaled versions of each other. That is, we found that $x$ and $y10$ above were a match, even though $y10$'s values were 10 times that of $x$'s.

Let us look at another example. What if we try to correlate $x$ to a signal that is not like it? We will call the second signal *noise*.

```
>> clear all
>> x = [ 0 0 1 5 1 -2 -3 -2 0 0 ];
```

```
>> noise = [ 0  -1 0 1 0 -1 1 0 1 -1];
>> for i=0:length(x)
    my_rho(i+1) = correlate(i, x, noise);
end
>> my_rho

my_rho =

  Columns 1 through 6

    0.2462   -0.2462   -0.3077    0.3077    0.3693    0.4308

  Columns 7 through 11

   -0.2462   -0.3077   -0.3077    0.0615    0.2462

>> min_rho=min(my_rho); max_rho=max(my_rho);
>> if (abs(min_rho) > abs(max_rho))
    min_rho
else
    max_rho
end

max_rho =

    0.4308
```

The array *my_rho* stores all correlation coefficients for these two signals. To see how well $x$ and *noise* match up, we need to examine both the minimum as well as the maximum values. With cross-correlation, we are able to say that two signals match, or that they do not match well.

---

**Example:**
The following example comes out of an image processing project. In Figure 3.30, we see two rectangles. Our goal is to automatically match these two objects, something people can do easily, but is a difficult pattern-matching process for a computer.

The rectangles are similar, though one is smaller than the other, as well as being

**Figure 3.30:** Two rectangles of different size (scale) and rotation.

rotated (i.e., one is longer than it is wide, the other is wider than it is long). To have the computer determine that they are a match, we will trace the boundaries of these objects, and calculate the center point. From this point, we will measure the distance to each point that makes up the boundary, from the upper-left corners clockwise around the object. When we finish, we have a one-dimensional view of these objects, as seen in Figure 3.31. The solid line represents the object on the left in Figure 3.30, while the dotted points represent the object on the right.

Notice that the two have the same length. This is necessary for correlation, so the smaller of the two was "stretched" to make it longer. The original signals are $d1$ and $d2$, and the code below shows a simple way to stretch the shorter signal $d2$ to be the same length as $d1$. The first line creates an array of points at fractional increments. The second line makes a new data signal called $d3$ that is a copy of $d2$, with redundant points. Better ways to interpolate a signal exist (see the `spline` command), but this method suffices for this example.

```
pts = 1:length(d2)/length(d1):length(d2)+1;
d3 = d2(round(pts(1:length(d1))));
```

Once we have two signals to compare that are the same length, we can use the correlation function, and take the largest value as an indication of how well the objects match.

```
>> for k=1:length(d1)
    rhos(k) = correlate(k, d1, d3);
end
>> disp(sprintf('max correlation coeff: %5.4f', max(rhos)));
max correlation coeff: 0.9994
```

Our code gives us a 0.9994 correlation coefficient, indicating a very good match between the two objects, as expected.

---

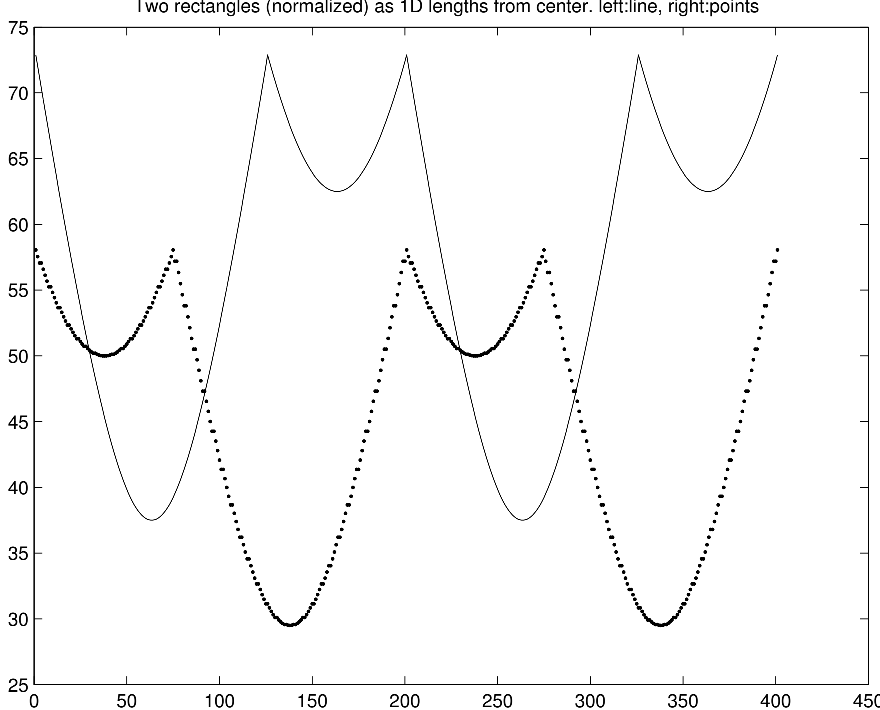



**Figure 3.31:** Two rectangles represented as the distance from their centers to their edges.

**Example:**
To show that the previous example was not just a coincidence, we will repeat the above experiment with a rectangle and a triangle, Figure 3.32.

**Figure 3.32:** A rectangle and a triangle.

This time, we see that the resulting one-dimensional signals are different, as we might expect (Figure 3.33). With the rectangle, we see the 1D signal gradually get smaller, then get larger again until it reaches the same height. Next, it gets a bit smaller, then back to its original height. This pattern repeats. From the center, the corners are going to be the furthest points away, and we see this as the top height. As we trace the boundary of the rectangle, the distance to the center gets a little smaller, then larger until we reach the next corner. As we trace the next side, the distance gets smaller and then back to the original distance at the corner. Since the second side is not as long as the first, the gradual dip in height is not as prominent. With the triangle, the corners are not equidistant from the center (due to the use of a simple averaging function to find the center), thus we see three peaks in the dotted one-dimensional object representation, but two of them are not as far away from the center as the third one. We started from the upper-right corner, though the starting position does not matter since the correlation calculations consider the shift of the signal.

```
max correlation coeff: 0.6272
```

When we calculate the correlation coefficients, we find the maximum as only 0.6272. We can conclude that that objects do share some similarity, but they are far from a match.

---

## 3.9 Summary

Finite Impulse Response (FIR) filters are presented in this chapter, including their structure and behavior. Three important properties that FIR filters possess are

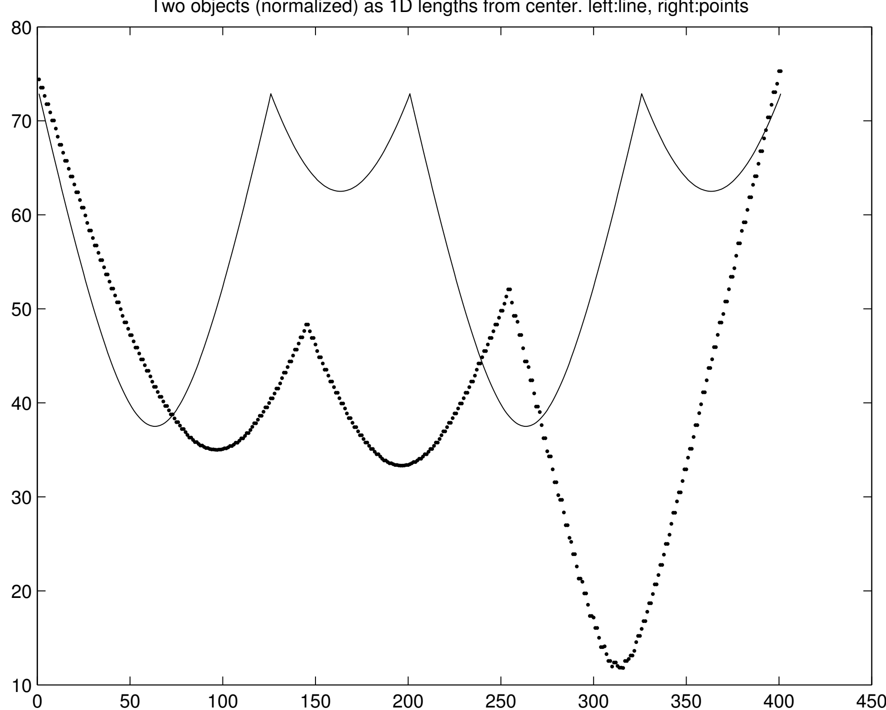



**Figure 3.33:** A rectangle and triangle represented as the distance from their centers to their edges.

causality, linearity, and time-invariance. Infinite Impulse Response (IIR) filters are also a topic of this chapter. Correlation is covered here as well, which is one application of an FIR filter.

## 3.10 Review Questions

1. What does FIR stand for? What does it mean?

2. What does causal mean?

3. What does linear mean?

4. What does time-invariant mean?

5. Write MATLAB code to compute output ($y[n]$) for convolution, given any FIR filter coefficients ($b[k]$) and input ($x[n]$).

6. Filter input $x = [5, 1, 8, 0, 2, 4, 6, 7]$ with a 2-tap FIR filter of coefficients $\{1, -1\}$ by hand (do not use a computer).

7. What word describes the operation below? ________ of $x[n]$ and $b[n]$ produces $y[n]$.
$$y[n] = \sum_{k=0}^{K} b[k]x[n-k]$$

8. Make up a test signal of at least 10 values, and filter it with the following coefficients. Use MATLAB for this question.

   a. $b[k] = \{.4, .4\}$
   b. $b[k] = \{.2, .2, .2, .2\}$
   c. $b[k] = \{.1, .1, .1, .1, .1, .1, .1, .1\}$

   What do you observe? Which set of filter values do the best job of "smoothing out" the signal?

9. What do we call $h[n]$? Why is it significant?

10. What does MAC stand for? What does it mean?

11. For $system_1$ and $system_2$ below, are the systems linear? Are they time-invariant? Are they causal? Explain. Let $x[n]$ be the input and $y[n]$ be the output, where $system_1$ is $y_1[n] = x[n] - x[n-1]$, and $system_2$ is $y_2[n] = 3x[n] + 1$.

12. What is the difference between an FIR filter and an IIR filter? Draw one of each.

13. Suppose we have a system where $y[n] = 2x[n] + x[n-1]$.

    a. Is this system causal?
    b. Is this system linear?
    c. Is this system time-invariant?

14. For the input signal $x[n] = [6, 1, 3, 5, 1, 4]$, and a filter $h[k] = [-1, 3, -1]$, find $y[n]$, where $y[n] = h[k] * x[n]$. Use the algorithm given in section 3.2.

# Chapter 4

# Sinusoids

Most analog signals are either sinusoids, or a combination of sinusoids (or can be approximated as a combination of sinusoids). This makes combinations of sinusoids especially interesting. It is easy to add sinusoids together; pressing keys on a piano or strumming a guitar adds several sinusoids together (though they do decay, unlike the sinusoids we usually study). In this chapter, we will investigate sinusoids and see how they can be added together to model signals. The goal is to gain a better understanding of sinusoids and get ready for the Fourier transform.

## 4.1 Review of Geometry and Trigonometry

Greek letters are nothing to be afraid of. Mathematics uses variables, much like we do in computer programming. These variables are one letter, which allows things to be described concisely. Greek letters are often used simply because we tend to run out of our alphabetic letters. Also, there are letters that are good to avoid, like l, and o, since they look too much like the numbers 1 and 0.

Consider a right triangle (see Figure 4.1). We know, thanks to Pythagoras (or at least his school [15]) that $a^2 + b^2 = c^2$, so if we happen to know only two sides, we can calculate the third[1]. Also, with this information, we can figure out the angles $\theta_1$ and $\theta_2$. Since it is a right triangle, the third angle is 90 degrees by definition.

The cosine function gives us the ratio of the length of the adjacent side ($a$) to the length of the hypotenuse ($c$). In other words,

$$\cos(\theta_1) = \frac{a}{c}.$$

[1] According to [16] the Babylonians knew this, but the Greeks were first to prove it.

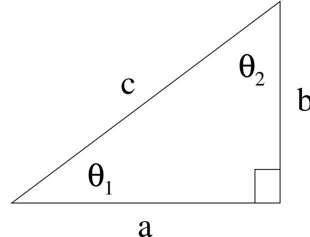



**Figure 4.1:** A right triangle.

The sine function gives us the ratio of the length of the opposite side ($b$) to the length of the hypotenuse ($c$). In other words,

$$\sin(\theta_1) = \frac{b}{c}.$$

The $\cos^{-1}$ and $\sin^{-1}$ functions (also called *arccos* and *arcsin*) give us the angle $\theta_1$ from the ratio of the sides.

## 4.2 The Number $\pi$

It is impossible to talk about sine and cosine functions without talking about the constant $\pi$, the ratio of a circle's diameter to its circumference (see Figure 4.2). You probably remember the equation:

*circumference* $= 2\pi r$, where $r$ is the circle's radius.

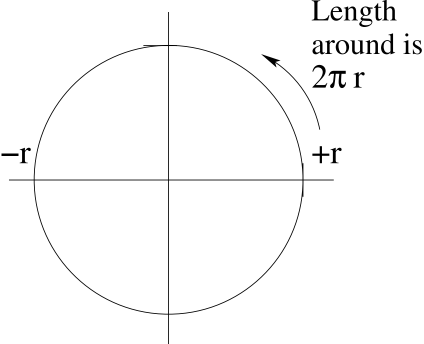



**Figure 4.2:** An example circle.

Pi ($\pi$) is an irrational number. No matter how many digits you calculate it out to be, someone else could go a digit further. To date, no repeating pattern has been found, so unlike rational numbers, you cannot represent $\pi$ in a compact form. For example, with a rational number such as $\frac{1}{3}$, you could write it as $0.3\overline{3}$ to indicate that it repeats forever when represented as a decimal number, or at least it can be represented as one number (1) divided by another (3).

Ancient peoples had various approximations for $\pi$. The Egyptians used 256/81 for $\pi$, while the Babylonians calculated with $3\frac{1}{8}$ [17]. We will use the approximation of 3.1415927 for $\pi$.

We will specify sinusoid components with the format

$$a \times \cos(2\pi ft + \phi)$$

where $a$ is the amplitude, $f$ is the frequency, and $\phi$ is the phase angle. The variable $t$ represents time. Sine could be used in place of cosine, if we add $\pi/2$ to the phase angle, since $\cos(\theta) = \sin(\theta + \pi/2)$.

Amplitude ($a$), phase shift ($\phi$) (also called the phase angle), and frequency completely specify a sinusoid. *Amplitude* describes a quantity without assigning it units. *Magnitude* is also a unitless description of a quantity, with one difference: amplitude can be positive or negative, while magnitude is always positive [11]. $|x(t)|$ refers to magnitude, and as you may expect, the `abs` function in MATLAB returns the magnitude. Many people in DSP use magnitude and amplitude interchangeably, for good reason. Viewing a sinusoid component as a phasor, where we draw it in its initial position without concern about rotation with time, we would draw the phasor along the positive x-axis, then rotate it counterclockwise according to the phase angle. A negative length means we would draw the phasor along the negative x-axis, then rotate (again counterclockwise) according to the angle. However, this is exactly the same as drawing the length along the positive x-axis, and rotating it an additional 180 degrees. In other words, $-a \times \cos(2\pi ft + \phi)$ is exactly the same as $a \times \cos(2\pi ft + \phi + \pi)$. This is why we typically use a positive value for the amplitude. The main exception is for the DC component, where frequency equals zero. The phase angle also is assumed to be zero, and thus the amplitude is the only value left to determine the sign of the DC component.

We use $f$ to represent the cyclic frequency, also called simply *frequency*, though sometimes the radian frequency ($\omega$) is used for convenience. It relates to $f$ with the equation $\omega = 2\pi f$. The units of $f$ are in Hz, cycles per second, named for Heinrich Hertz (1857–1894), a German physicist whose experiments confirmed Maxwell's electromagnetic wave theories [18]. Since the unit Hz comes from a person's name, the first letter should always be capitalized.

## 4.3 Unit Circles

A *unit circle* is of particular interest. For a unit circle, the radius $r$ equals 1. Therefore, the circumference, or length of the circle if it were cut and unrolled, is $2\pi$. As a result, we can talk about an angle in terms of the length of the arc between two line segments joined at a point. We call this angle *radians* (rad for short). In Figure 4.3, angle $\theta$ is only a fraction of the circle's circumference, therefore, it is a fraction of $2\pi$. Figure 4.4 shows the arc lengths on a unit circle for several common angles.

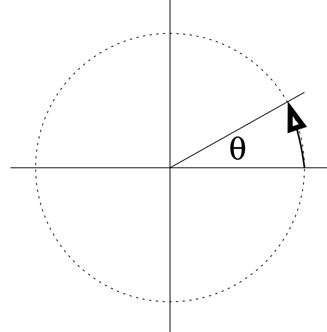



**Figure 4.3:** An angle specified in radians.

No doubt you are also familiar with an angle measured in degrees. When dealing with sinusoidal functions (such as sine, cosine, etc. ), it is important to know whether the function expects the argument to be in degrees or radians. On a scientific calculator, you are likely to have a button labeled DRG, to allow you to specify degrees or radians, though the default is probably degrees. For MATLAB, C/C++ and Java, the arguments to the sin() and cos() functions are assumed to be radians.

To convert between degrees and radians, we recall that one complete rotation around the unit circle's edge is $2\pi$ radians, or $360°$. To convert from degrees to radians, therefore, we multiply by $2\pi/360$. To convert from radians to degrees, we multiply by $360/(2\pi)$. For example, suppose the angle shown in Figure 4.3 is $30°$,

$$\begin{aligned}(30)(2\pi/360) &= 60\pi/360 \\ &= \pi/6 \; radians.\end{aligned}$$

Converting $\pi/6$ back to degrees can be done in a similar fashion:

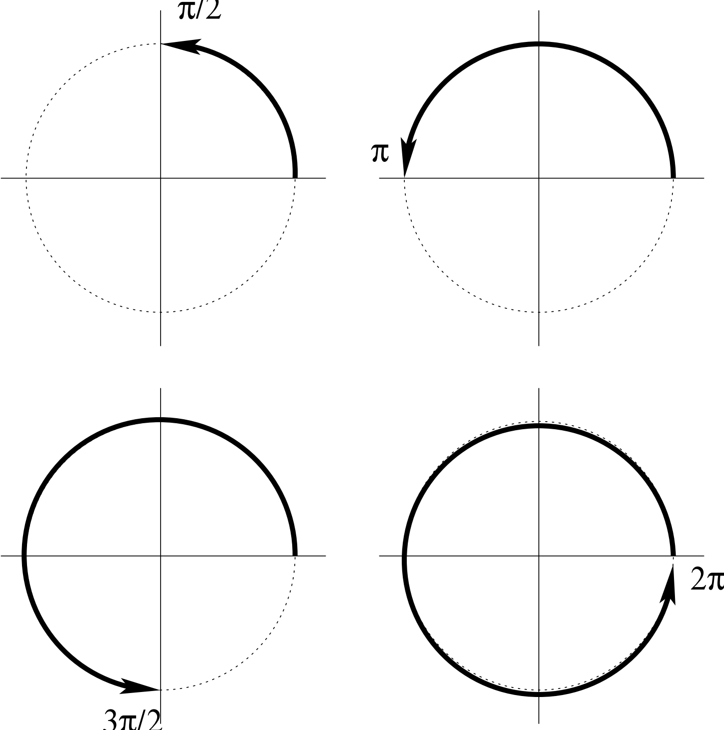



**Figure 4.4:** An angle specified in radians.

$$\begin{aligned}(\pi/6)(360/(2\pi)) &= 360\pi/(12\pi) \\ &= 360/12 \\ &= 30^\circ.\end{aligned}$$

Quiz: What is larger, $\cos(0)$ or $\cos(1,000,000)$? At first glance, one unfamiliar with the cosine function might assume that the latter is larger, simply because the argument is larger. But since sinusoids repeat, there is no reason for this to be the case. In fact, one might remember that $\cos(0)$ is 1, and that cos(*any number*) always generates a value between $+1$ and $-1$. Therefore, $\cos(0)$ is going to be greater than or equal to cos(*any number*). In fact, $\cos(1,000,000) \approx 0.9368$, correct? If not, it is likely due to your calculator being in degrees mode! Since it was not specified that the angle is in degrees, one should assume that it is in radians.

Another thing to consider when dealing with sinusoid functions is that they are repetitive. The angle specified by 0 radians is the same as that specified by $2\pi$ radians. Thus, you can always add or subtract $2\pi$ to/from an angle. In fact, you can add or subtract any integer multiple of $2\pi$ to or from an angle. Sometimes this makes things easier.

## 4.4 Principal Value of the Phase Shift

A sinusoid stretches out infinitely in both directions. If we want to view it in terms of a time axis, we can use the positive peaks of the sinusoid to relate to our time axis. Figure 4.5 shows an example sinusoid, $\cos(2\pi 60t)$, with a frequency of 60 Hz. Therefore, the period of time before this sinusoid repeats is $\frac{1}{60}$, or 0.0167 seconds. Vertical lines show where one repetition starts and ends, at $t = -0.0167/2$ and $t = +0.0167/2$. This corresponds to arguments of $-\pi$ and $+\pi$ for the cos function, that is, let $t = (\frac{1}{60})(\frac{-1}{2})$ to make $\cos(2\pi 60t)$ equal $\cos(2\pi 60(1/60)(-1/2)) = \cos(-\pi)$.

But WHICH peak do we normally use? In Figure 4.5 it is very easy, but a sinusoid function often has a nonzero phase angle. Since it repeats every $2\pi$, we could pick any peak that we want to classify the signal. This means that there will always be a positive peak between -$\pi$ and $+\pi$, in other words, $-\pi < \phi \leq +\pi$ where $\phi$ is the phase shift [6].

So why not use the positive peak closest to zero? This makes a lot of sense, and we give this peak a special name: we say it is the *principal value* of the phase shift. To find this peak, we can always add/subtract $2\pi$ from the phase shift, until it is in the range we desire.

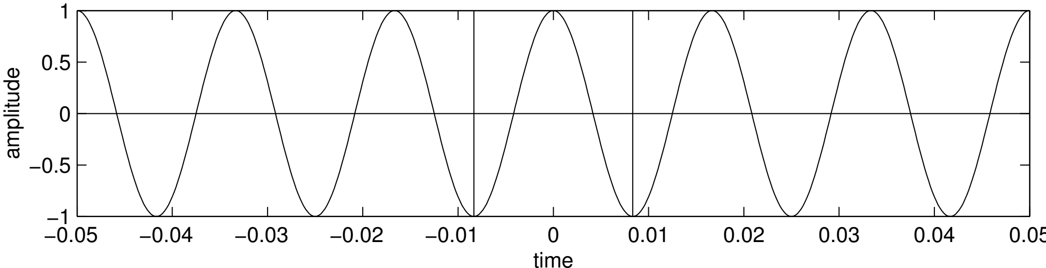



**Figure 4.5:** A 60 Hz sinusoid.

## 4.5 Amplitudes

Usually, amplitude values are positive, and for good reason. Consider two sinusoids of the same frequency, $x_1 = a \times \cos(2\pi f t + \phi_1)$ and $x_2 = -a \times \cos(2\pi f t + \phi_2)$. We know that the minimum and maximum values for the cosine function are -1 and +1. Therefore, $x_1$ varies between $+a$ and $-a$, and $x_2$ varies between $-a$ and $+a$. Given that the two sinusoids have the same frequency, they must look a lot alike. In fact, the only difference between them is their phase angles, a difference of $\pi$. That is, $\phi_2 = \phi_1 + \pi$. This makes a lot of sense considering the conversion of a complex value of $x + jy$ to polar form $r\angle\theta$: the equation $(\sqrt{x^2 + y^2})$ leads to a positive value for $r$. A vector with a negative length would be the same as a vector with a positive length, rotated a half-revolution, as demonstrated in Figure 4.6.

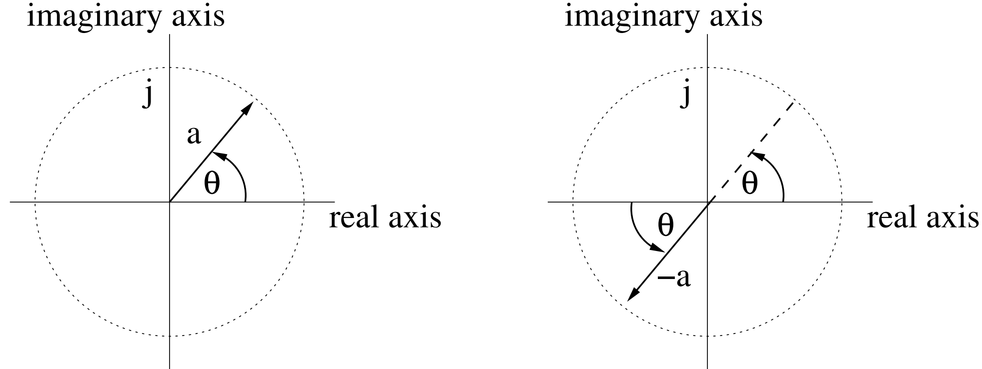



**Figure 4.6:** A vector of $-a$ at angle $\theta = a$ at angle $(\theta + \pi)$.

Signals are often represented as a sum of sinusoids. When these sinusoids are related, we call the signal a harmonic.

## 4.6 Harmonic Signals

Real-world signals can be decomposed into a set of sinusoids added together. A *harmonic* signal is composed of several sinusoids added together, each with an integer multiple of some base frequency, called the *fundamental frequency*. For example, the following signal is a harmonic:

$$x(t) = 0.4\cos(2\pi 15t + \pi/5) + 0.7\cos(2\pi 30t) + \cos(2\pi 45t - \pi/4).$$

Signal $x(t)$ is of the form

$$a_1 \times \cos(2\pi(1)f_0 t + \phi_1) + a_2 \times \cos(2\pi(2)f_0 t + \phi_2) + a_3 \times \cos(2\pi(3)f_0 t + \phi_3)$$

where, for $x(t)$, $a_1 = 0.4$, $a_2 = 0.7$, $a_3 = 1$, $\phi_1 = \pi/5$, $\phi_2 = 0$, $\phi_3 = -\pi/4$, and $f_0 = 15$ Hz. Any signal of this form is a harmonic.

If an amplitude value is 0, then the corresponding sinusoid will not contribute to the function. In other words, $0 \times \cos(anything)$ evaluates to 0. If we have a function like

$$x_2(t) = 0.1\cos(2\pi 100t - \pi/6) + 1.3\cos(2\pi 300t + \pi) + 0.5\cos(2\pi 400t + 2\pi/3)$$

then $x_2(t)$ is a harmonic with $a_1 = 0.1$, $a_2 = 0$, $a_3 = 1.3$, $a_4 = 0.5$, $\phi_1 = -\pi/6$, $\phi_2 = 0$, $\phi_3 = \pi$, $\phi_4 = 2\pi/3$, and $f_0 = 100$ Hz.

The following MATLAB code is a function to plot harmonic signals.

```
%
%  Given magnitudes, phases, and the fundamental frequency,
%    plot a harmonic signal.
%
%  Usage: plotharmonic(freq, mag, phase)
%
%  where  freq = the fundamental frequency (a single value),
%          mag = the amplitudes (a list),
%        phase = the phases (a list).
%
function plotharmonic(freq, mag, phase)

num_points = 200;   % Number of points per repetition

% Check parameters
if (size(mag) ~= size(phase))
```

```
    sprintf('Error - magnitude and phase must be same length')
end

% We want this for 2 repetitions, and num_points per rep.
step = 2/(freq*num_points);
t = 0:step:2*(1/freq);

clear x;
x = 0;
for i=1:length(mag)
    x = x + mag(i)*cos(2*pi*i*freq*t + phase(i));
end

my_title = sprintf('Harmonic signal: %d sinusoids',length(mag));
plot(t,x);
title(my_title);
xlabel('time (sec)');
ylabel('Amplitude');
```

For example, the following commands plot $x_2(t)$ from above. It is assumed that the code from above is saved with "plotharmonic.m" as the filename.

```
Mag = [ 0.1  0  1.3  0.5];
Phase = [ -pi/6  0  pi  2*pi/3];
plotharmonic(100, Mag, Phase);   % Fundamental freq. is 100 Hz
```

The resulting plot can be seen in Figure 4.7.

One additional piece of information is if there is a vertical shift of the signal. That is, what if the graph shown in Figure 4.7 was not centered around the x-axis? This can be accomplished by adding a constant to the signal. For example, if we add 100 units to the signal, it would look the same, but it would be centered around the line $y = 100$. This piece of information can be added to the above harmonic signal format, simply by including the term $a_0 \cos(2\pi(0)f_0 t + \phi_0)$. Starting the multiples of $f_0$ at zero means that the first component is simply a constant. There is no need to specify a $\phi_0$ term, since $\cos(\phi_0)$ would just be a constant, and $a_0$ already specifies this. Therefore, we will assume $\phi_0$ is zero. This term is often called the *DC component*, for Direct Current.

---

**Example:**

For the signals below, say which ones are harmonic and which are not. If they

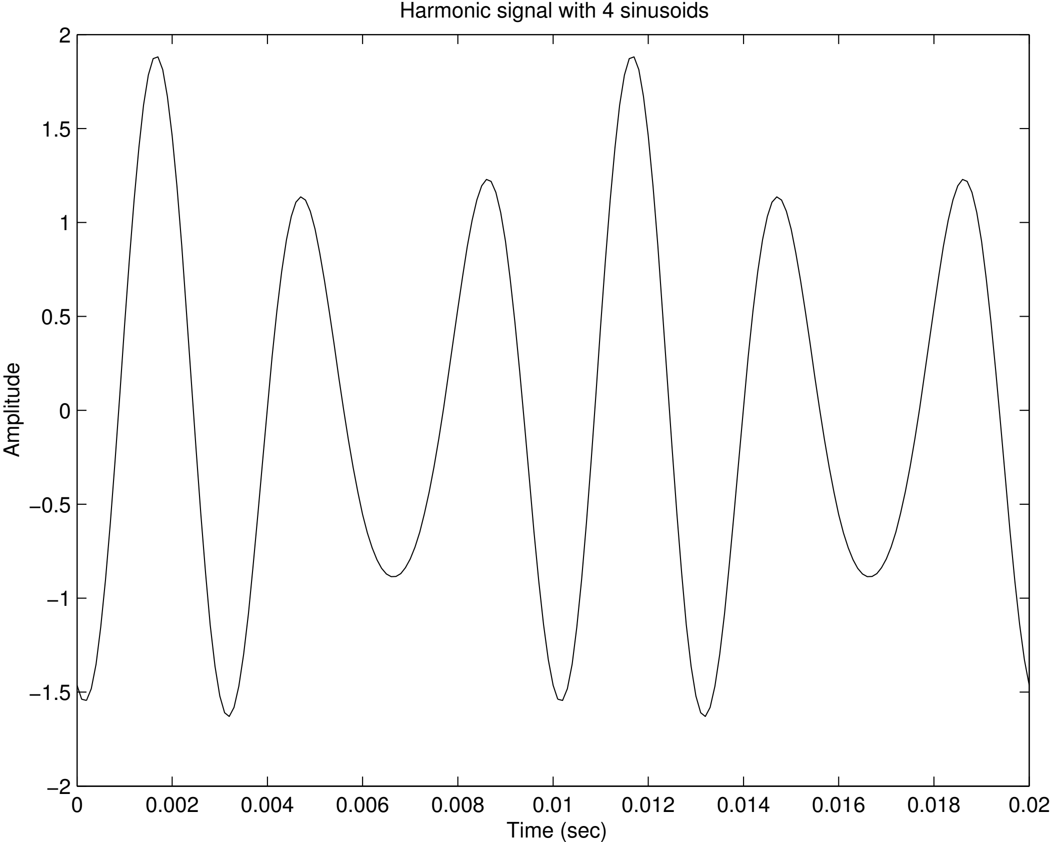



**Figure 4.7:** A harmonic signal.

are harmonic, say what the fundamental frequency is.

$x_1(t) = 2\cos(2\pi 7t) + 3\cos(2\pi 35t)$
$x_2(t) = 2\cos(2\pi^2 t) + 3\cos(2\pi t)$
$x_3(t) = 2\cos(2\pi 7t + \pi) + 3\cos(2\pi t - \pi/4)$

**Answer:**

To be harmonically related, each sinusoid must have a frequency that is an integer multiple of some fundamental frequency. That is, they should be in the following form. Remember that $k$ is an integer.

$$x(t) = \sum_{k=1}^{N} a_k \cos(2\pi k f_0 t + \phi_k)$$

So we put the above signals in this form, and see if the $k$ values are integers, which they must be for the sinusoids to be harmonically related. Looking at the arguments of the cosine function for $x_1$, let $k_1$ and $k_2$ be values of $k$.

$2\pi 7t = 2\pi k_1 f_0 t + \phi_k$, Let $\phi_k = 0$
$f_0 = 7/k_1$

$2\pi 35t = 2\pi k_2 f_0 t + \phi_k$, Let $\phi_k = 0$
$f_0 = 35/k_2$

Substituting in $f_0$, we get

$35/k_2 = 7/k_1$
$k_1/k_2 = 7/35$
$k_1/k_2 = 1/5$

So if $k_1 = 1$, then $k_2 = 5$

$f_0 = 35/k_2 = 35/5 = 7$ Hz.

Therefore, $x_1(t)$ has integer $k$ values, and is harmonic. $x_3(t)$ is, too. $x_2(t)$ does not have integer $k$ values (due to the frequency of $\pi$) and is, therefore, not harmonic. The third signal, $x_3$, is a bit tricky because the frequency components are not lined up as we may expect. $f_0 = 1$ Hz, $a_0 = 1$, $a_k = \{3, 0, 0, 0, 0, 0, 2\}$, and $\phi_k = \{-\pi/4, 0, 0, 0, 0, 0, \pi\}$.

---

We could consider harmonic a signal with a DC component, too. Without loss of generality, we can have index $k$ start at 0, implying that we have a frequency component of $0 * f_0$, or 0 Hz. Since designating a value for the phase angle would only result in the amplitude $a_0$ being multiplied by a constant, we should set the phase angle to 0. Thus, $a_0$ completely specifies the DC component.

Representing a signal that has discontinuities, such as a square wave, with a sum of harmonically related sinusoids is difficult. Many terms are needed [19]. Called *Gibbs' phenomenon*, we cannot perfectly represent a discontinuous signal as a sum of sinusoids, but our approximation will have overshoot, undershoot, and ripples.

Fourier analysis works best with a *periodic* signal. We implicitly expect that the first and last sample are linked, that if we were to take one more reading of the signal, that this reading would be the same as the very first sample, or at least a value that we have seen before. Whether or not this $N + 1$ value matches value 0 depends upon how many samples we have taken. Also, we assume that a sufficient number of samples has been recorded, given by the sampling period. If the signal truly is periodic, and we have a good sampling rate, and we take exactly one period's worth of samples, then the sample at $N + 1$ would be equal to sample 0.

Sometimes we do not have a periodic signal. In this case, we may have a discontinuity between the last sample and the first. For example, consider the signal below, Figure 4.8:

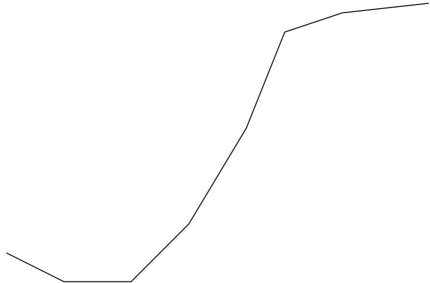


**Figure 4.8:** A short signal.

If we treat this signal as being periodic, then we expect that it repeats forever, and that we can copy and paste this signal to itself, such as in Figure 4.9.

Here we see that there is a discontinuity in the middle, and this is difficult for a Fourier series to handle (remember Gibb's phenomenon). Thus, we should consider the Fourier analysis to give us an *approximation* of the original signal. We would expect that the approximation is not as good around a discontinuity as it is for the rest of the signal. We call this difficulty *edge effects*.

Even with edge effects, Fourier analysis will return a good approximation to the

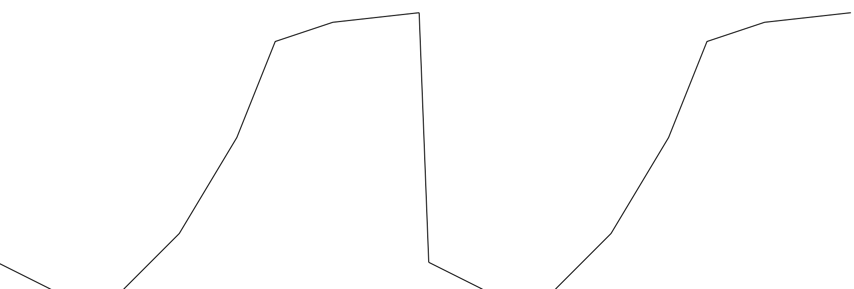


**Figure 4.9:** The short signal, repeated.

original signal. In fact, *any nonrandom signal can be approximated with Fourier analysis.* If we take a random signal and find the Fourier series of it, we will find that it approximates the signal for the values that we have seen. However, we may be tempted to use this to predict future values of the signal. If it is truly random, though, any matching between the signal and a prediction would be coincidence.

## 4.7 Representing a Digital Signal as a Sum of Sinusoids

A digital signal can be represented as a sum of sinusoids. The following three figures show how this works. We can take a digital signal and find the Fourier transform of it (the Fourier transform is covered a bit later in this book). The important thing to understand is that the Fourier transform gives us a frequency-domain version of the signal: a list of amplitudes and phase angles corresponding to a harmonically related set of sinusoids.

We explore the fundamental frequency of these sinusoids later; for the present discussion, we will ignore it. The timing of these simulated samples determines the real fundamental frequency, so we will just stick to sample numbers instead of time for the x-axis of the plots.

Refer to Figure 4.10. Here we see the original signal, a list of discrete sample points. Under it, we have a graph of a signal made by adding several sinusoids (8, to be precise). Notice how this composite signal passes through the same points as the original digital signal. If we were to read values from this composite signal with the same timing as the sampling of the digital signal, we would have the same data.

In Figure 4.10, we see that the sum of sinusoids representation is continuous. But we should be careful *not* to conclude that the underlying analog signal, which we are trying to represent as a set of discrete points, actually looks like the composite

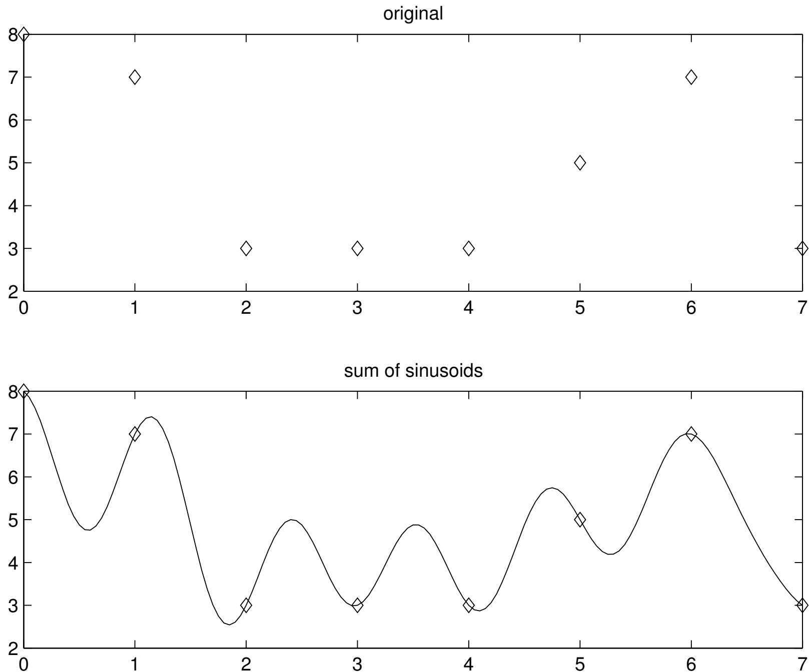



**Figure 4.10:** A digital signal (top) and its sum of sinusoids representation (bottom).

signal. We may not know what it looks like, and the hill we have between sample 2 and sample 3 (as well as between sample 3 and sample 4) could mislead us.

The next two graphs, Figures 4.11 and 4.12, show the sinusoids that we added together to get the composite one. Yes, the first one shown is a sinusoid of zero frequency! The asterisks show the values that would be found if the sinusoids were sampled with the same timing as the original signal. Notice how the number of sinusoids (8) returned from the Fourier transform matches the number of original points (8).

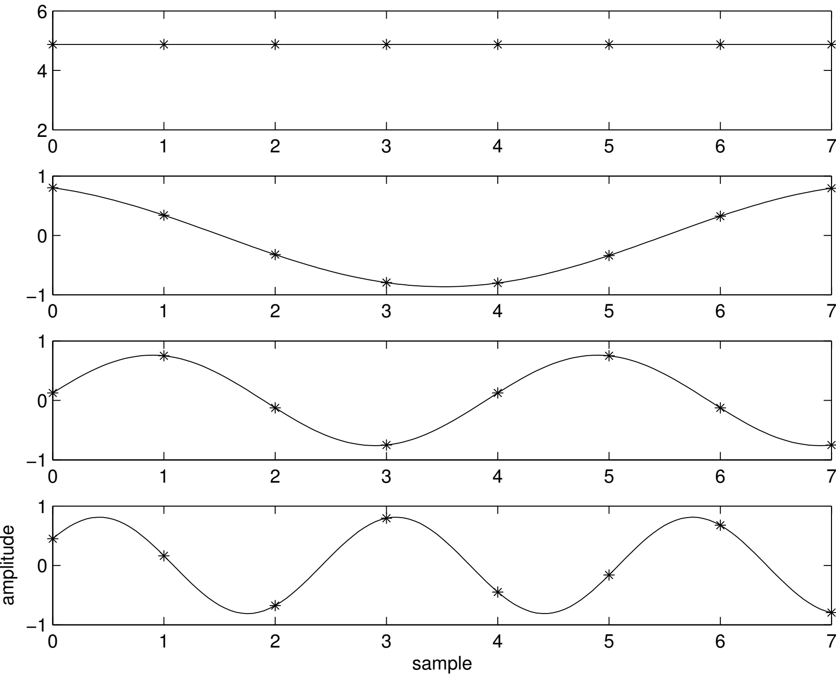



**Figure 4.11:** The first four sinusoids in the composite signal.

As an exercise, let's approximate the value of the second-to-last sample. In Figure 4.11, we see that sample #6 of each sinusoid has the following approximate values: 4.9, 0.3, -0.1, and 0.7. Turning our attention to Figure 4.12, we see this next-to-last sample's values are roughly: 0.4, 0.7, -0.1, and 0.3. If we add these up together, we get 7.1. Looking back at the original Figure 4.10, we see that the value for sample #6 is right around 7. In fact, the only reason we did not get exactly the same number as in the first figure was due to the error induced by judging the

graphs by eye.

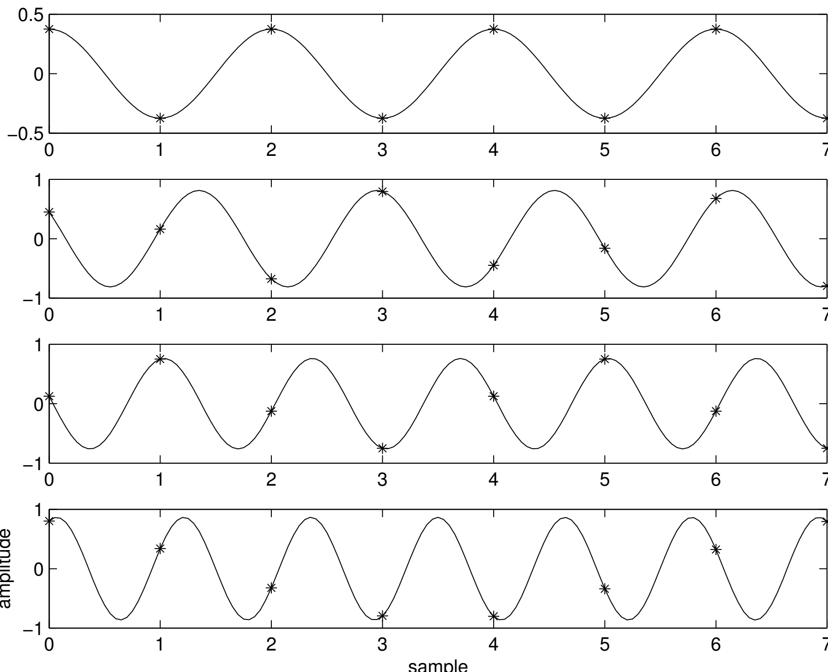



**Figure 4.12:** The last four sinusoids in the composite signal.

In a similar exercise, we can note the maximum values for each of the sinusoids in Figures 4.11 and 4.12. These are: 4.88, 0.86, 0.76, 0.81, 0.38, 0.81, 0.76, and 0.86. They correspond to the amplitudes of each of the sinusoids, the $a_k$ values that we would need for a harmonic representation. The other pieces of information (besides the fundamental frequency) would be the phase angles. These are a bit more difficult to read off of a graph, but we can make approximations for them. Remembering that $\cos(0) = 1$, we notice that the value returned from the cosine function decreases as the angle gets larger, until it reaches $-1$ at $\pi$, then it increases until it gets to 1 again at $2\pi$. We can use this observation to get a rough idea about the phases. Looking at the second graph of Figure 4.11, we see that the sinusoid has its maximum point right at sample 0. The maximum point would correspond to the amplitude, or more exactly $a_1 \times \cos(2\pi 1 f_0 t + \phi_1)$ when the argument to the cosine function is 0, as it would be at time 0. So we would have $2\pi 1 f_0 0 + \phi_1 = 0$, or simply $\phi_1 = 0$. Now, to find the next phase angle, $\phi_2$. At sample 0 (third

graph on Figure 4.11), we notice that our value would be about 0.1. Solving for $\phi_2$, $a_2 \times \cos(2\pi 1 f_0 t + \phi_2)$ at time $0 \approx 0.1$. This means that $a_2 \times \cos(\phi_2) \approx 0.1$, and we already have an approximation of $a_2$ as 0.8. So we can solve to find:

$$0.8\cos(\phi_2) \approx 0.1$$
$$\cos(\phi_2) \approx \frac{0.1}{0.8}$$
$$\phi_2 \approx arccos\left(\frac{0.1}{0.8}\right)$$
$$\phi_2 \approx 83^\circ$$
$$\phi_2 \approx \frac{46\pi}{100}.$$

But there could be a problem: how do we know that the sinusoid will increase at the next time step? To put this another way, we know that the cosine function returns the same value for $\theta$ as it would for $2\pi - \theta$, so which value should we use for our phase?

To solve this problem, we borrow from calculus (in a very painless way!). We have a graphical representation for the sinusoid that we are after, and we can tell just by looking at the graph if it increases or decreases. In the case of the third graph on Figure 4.11, we see that it increases. If we add a little bit to the phase angle, simulating a very small time increment, the value returned by $a_2 \times \cos((a\ little\ bit) + \phi_2)$ should be a little more than $a_2 \times \cos(\phi_2)$. Let's use this to test our value for $\phi_2$.

$$0.8cos\left(\frac{46\pi}{100}\right) = 0.1003$$
$$0.8cos\left(0.01 + \frac{46\pi}{100}\right) = 0.0923$$

But this shows that the sinusoid would *decrease*, contrary to our graph. Therefore, we know that the phase angle must be our previous estimate negated, or $-46\pi/100$. We can (and should) double-check our work by applying the same trick as above.

$$0.8cos\left(\frac{-46\pi}{100}\right) = 0.1003$$
$$0.8cos\left(0.01 - \frac{46\pi}{100}\right) = 0.1082$$

We see that the next values increase, just as we observe from the graph. We can

continue in like fashion for the rest of the graphs.

The program below demonstrates the sum-of-sinusoids concept. It does not require any parameters, since it makes the "original" signal randomly. This program finds the frequency-domain representation $X$ of the random signal $x$, then uses the frequency-domain information to get the original signal back again (as the array $my_sum_of_sins$). We see that, when run, this program calculates the error between the original signal and the reconstructed one, an error so small that it is due only to the precision used.

```
% sum_of_sins.m
% Show a random signal as a sum of sins
%

% Make our x signal
x = round(rand(1,8)*10);
Xsize = length(x);

% Get FFT of x
X = fft(x);
Xmag = abs(X);
Xphase = angle(X);

% Show the freq-domain info as sum of sinusoids
% Find the IFFT (the hard way) part 1
n=0:Xsize-1;  %m=0:Xsize-1;
% Do the sinusoids as discrete points only
s0 = Xmag(1)*cos(2*pi*0*n/Xsize + Xphase(1))/Xsize;
s1 = Xmag(2)*cos(2*pi*1*n/Xsize + Xphase(2))/Xsize;
s2 = Xmag(3)*cos(2*pi*2*n/Xsize + Xphase(3))/Xsize;
s3 = Xmag(4)*cos(2*pi*3*n/Xsize + Xphase(4))/Xsize;
s4 = Xmag(5)*cos(2*pi*4*n/Xsize + Xphase(5))/Xsize;
s5 = Xmag(6)*cos(2*pi*5*n/Xsize + Xphase(6))/Xsize;
s6 = Xmag(7)*cos(2*pi*6*n/Xsize + Xphase(7))/Xsize;
s7 = Xmag(8)*cos(2*pi*7*n/Xsize + Xphase(8))/Xsize;
% Redo the sinusoids as smooth curves
t = 0:0.05:Xsize-1;
smooth0 = Xmag(1)*cos(2*pi*0*t/Xsize + Xphase(1))/Xsize;
smooth1 = Xmag(2)*cos(2*pi*1*t/Xsize + Xphase(2))/Xsize;
smooth2 = Xmag(3)*cos(2*pi*2*t/Xsize + Xphase(3))/Xsize;
smooth3 = Xmag(4)*cos(2*pi*3*t/Xsize + Xphase(4))/Xsize;
```

```
smooth4 = Xmag(5)*cos(2*pi*4*t/Xsize + Xphase(5))/Xsize;
smooth5 = Xmag(6)*cos(2*pi*5*t/Xsize + Xphase(6))/Xsize;
smooth6 = Xmag(7)*cos(2*pi*6*t/Xsize + Xphase(7))/Xsize;
smooth7 = Xmag(8)*cos(2*pi*7*t/Xsize + Xphase(8))/Xsize;
% Find the IFFT (the hard way) part 2
my_sum_of_sins = (s0+s1+s2+s3+s4+s5+s6+s7);
smooth_sum = smooth0 + smooth1 + smooth2 + smooth3;
smooth_sum = smooth_sum + smooth4 + smooth5 + smooth6 + smooth7;


% Show both discrete points and smooth curves together
xaxis1 = (0:length(smooth0)-1)/20;  % for 8 points
xaxis2 = (0:length(s0)-1);
figure(1);
subplot(4,1,1);  plot(xaxis1, smooth0, 'g', xaxis2, s0, 'r*');
subplot(4,1,2);  plot(xaxis1, smooth1, 'g', xaxis2, s1, 'r*');
subplot(4,1,3);  plot(xaxis1, smooth2, 'g', xaxis2, s2, 'r*');
subplot(4,1,4);  plot(xaxis1, smooth3, 'g', xaxis2, s3, 'r*');
ylabel('amplitude');
xlabel('sample');
figure(2);
subplot(4,1,1);  plot(xaxis1, smooth4, 'g', xaxis2, s4, 'r*');
subplot(4,1,2);  plot(xaxis1, smooth5, 'g', xaxis2, s5, 'r*');
subplot(4,1,3);  plot(xaxis1, smooth6, 'g', xaxis2, s6, 'r*');
subplot(4,1,4);  plot(xaxis1, smooth7, 'g', xaxis2, s7, 'r*');
ylabel('amplitude');
xlabel('sample');

% Show the original versus the reconstruction from freq-domain
figure(3);

subplot(2,1,1); plot(0:Xsize-1, x, 'bd');
title('original');
min_y = floor(min(smooth_sum));
max_y = ceil(max(smooth_sum));
axis([0, Xsize-1, min_y, max_y]); % Make the axes look nice
xaxis1 = (0:length(smooth_sum)-1)/20;
xaxis2 = (0:length(s0)-1);
subplot(2,1,2);
```

```
plot(xaxis1, smooth_sum, 'b', xaxis2, my_sum_of_sins, 'rd');
title('sum of sinusoids');
Error = my_sum_of_sins - x
```

We assume that the signal contains only real information. In general, this is not the case. However, since we started with a signal of all real values, we will end up with a signal of all real values, since we do not change the information, we only transform it from the time-domain to the frequency-domain, then transform it back. The Fourier transform results in a complex signal, though the original signal was real. To reconcile this, we simply treat the original real signal as if it were complex, but the imaginary part is zero. The signal returned from the inverse-transform must also have an imaginary part of zero for all values. The program performs the inverse-transform in a sense, but it does not take the complex part into consideration, as it would have to do if the original signal were complex. In other words, the program treats each sinusoid as if it were $X_{mag}[k]\cos(2\pi k f_0 t + X_{angle}[k])$, when in fact each sinusoid is (from Euler's formula) $X_{mag}[k]\cos(2\pi k f_0 t + X_{angle}[k]) + jX_{mag}[k]\sin(2\pi k f_0 t + X_{angle}[k])$. We are able to get away with this simplification since we know in advance that the complex parts, $j\sin(\circ)$, must cancel each other out.

## 4.8 Spectrum

A *spectrum* is the name given to the frequency plot of a signal. The spectrum is a graph of the frequencies that make up a signal, with their amplitudes shown in relation to each other. A separate graph shows the phases for each sinusoid.

Previously, in Figure 4.10, we saw that a digital signal can be represented by a sum of sinusoids. These sinusoids can then be graphed individually, as shown in Figures 4.11 and 4.12. Sometimes this sinusoid information provides insight into the signal. For example, if the original signal is music, then the sinusoids tell us about the instruments that are being played. If the original signal was of a person talking, with a high-pitched extraneous sound such as a train whistle, we might be able to remove the higher-frequency whistle.

Plotting the information given to us by the sinusoids results in a spectrum. Continuing with the example from the last section, we see that one piece of information we are missing is the fundamental frequency. As stated before, we need to know how much time there is between samples before we can know the frequencies. But we will do our best without this. We see that the sinusoids repeat an integral amount in each graph, from 0 repetitions (0 frequency) up to 7 repetitions. That is, the sinusoids repeat with such a frequency that sample #8, if plotted, would be the same as sample

#0. As the number of repetitions gets larger, so does the corresponding frequency. Suppose, for the sake of argument, that the original signal was sampled at a rate of 125 milliseconds per sample, so we read each sample 125 milliseconds after the last. The second graph, on Figure 4.11, shows that the sinusoid repeats once every 1000 milliseconds, or once per second (it takes another 125 milliseconds before it gets to sample #8, which is identical to sample #0). Thus, it has a frequency of 1 Hz. Similarly, we see that the next graph has a frequency of 2 Hz, then 3 Hz for the next graph, etc., up to 7 Hz for the last graph of Figure 4.12. For the remainder of this chapter, we assume that we have this fundamental frequency of 1 Hz. Recalling our amplitudes {4.88, 0.86, 0.76, 0.81, 0.38, 0.81, 0.76, 0.86} from the previous section, and getting the corresponding phases in degrees $\{0, 22, -81, -56, 0, 56, 81, -22\}$, we can plot this information as follows, Figure 4.13. In the top part of the figure, we see a frequency magnitude plot, since it shows the positive amplitudes versus the frequencies. The bottom part of Figure 4.13 shows a frequency phase plot, since it shows the phase angles for each corresponding sinusoid frequency.

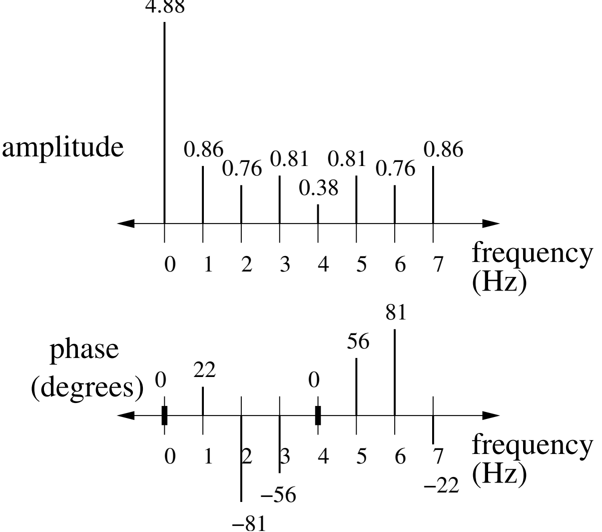



**Figure 4.13:** Frequency magnitude spectrum and phase angles.

One should observe the pattern of the spectrum: for the frequency magnitude plot, we see a mirror image centered around 4 Hz. On the phase plot, again the pattern centers around 4 Hz, but we have a flipped as well as mirrored image.

So far we have seen how to get the spectrum from the amplitudes and phases returned by the Fourier transform. To find the spectrum of an abstract signal, such as a mathematical model, use the inverse Euler's formula. Here we have only an

introduction; this topic is covered extensively in Chapter 7, "The Number $e$." For example, to find and plot the magnitude spectrum of $x(t) = 2 + 2\cos(2\pi(200)t)$, we put it in terms of cosine functions.

$$\begin{aligned} x(t) &= 2 + 2\cos(2\pi 200t) \\ &= 2\cos(0) + 2\cos(2\pi 200t) \end{aligned}$$

Next, use inverse Euler's formula:

$$\cos(\phi) = (e^{j\phi} + e^{-j\phi})/2.$$

Notice how this results in two frequency components: one at the frequency given, and one at the negative frequency. For the spectrum, this implies that the range of frequencies will be different from what we saw with the previous example. Actually, the information stays the same, only our spectral plot has different, but equivalent, frequency values. This is explained in Chapter 6, "The Fourier Transform."

$$\begin{aligned} x(t) &= 2(e^{j0} + e^{-j0})/2 + 2(e^{j2\pi(200)t} + e^{-j2\pi(200)t})/2 \\ &= 2(1+1)/2 + (2/2)(e^{j2\pi(200)t} + e^{-j2\pi(200)t}) \\ &= 2 + e^{j2\pi(200)t} + e^{-j2\pi(200)t} \end{aligned}$$

This results in a magnitude of 2 at frequency 0 Hz, and a magnitude of 1 at frequencies 200 Hz and -200 Hz. This is shown in the magnitude plot, Figure 4.14.

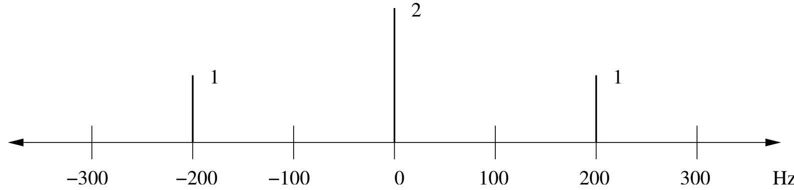



**Figure 4.14:** Spectrum plot: magnitude of $x(t) = 2 + 2\cos(2\pi(200)t)$.

---

**Example:**
For the signal $x(t)$ below,

$$x(t) = 4\ \cos(2\pi 100t + 3\pi/4) + 2\ \cos(2\pi 200t) + 5\ \cos(2\pi 300t - 3\pi/5)$$

draw the magnitude spectrum and the phase spectrum of this signal.

**Answer:**

$$\begin{aligned} x(t) &= \frac{4}{2}e^{j(2\pi 100t+3\pi/4)} + \frac{4}{2}e^{-j(2\pi 100t+3\pi/4)} \\ &+ \frac{2}{2}e^{j(2\pi 200t)} + \frac{2}{2}e^{-j(2\pi 200t)} \\ &+ \frac{5}{2}e^{j(2\pi 300t-3\pi/5)} + \frac{5}{2}e^{-j(2\pi 300t-3\pi/5)} \end{aligned}$$

$$\begin{aligned} x(t) &= 2\ e^{j2\pi 100t}e^{j3\pi/4} + 2\ e^{-j2\pi 100t}e^{-j3\pi/4} \\ &+ e^{j2\pi 200t} + e^{-j2\pi 200t} \\ &+ 2.5\ e^{j2\pi 300t}e^{-j3\pi/5} + 2.5\ e^{-j2\pi 300t}e^{j3\pi/5} \end{aligned}$$

Pulling the frequencies out $\{\ 100, -100, 200, -200, 300, -300\ \}$, and marking the corresponding magnitudes $\{2, 2, 1, 1, 2.5, 2.5\}$, gives us the graph of Figure 4.15. For each frequency, we also note the phases, $\{\ 3\pi/4, -3\pi/4, 0, 0, -3\pi/5, 3\pi/5\}$. The phase angles are shown in Figure 4.16.

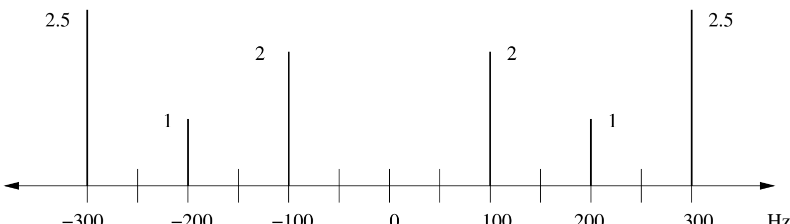



**Figure 4.15:** Spectrum plot: magnitudes.

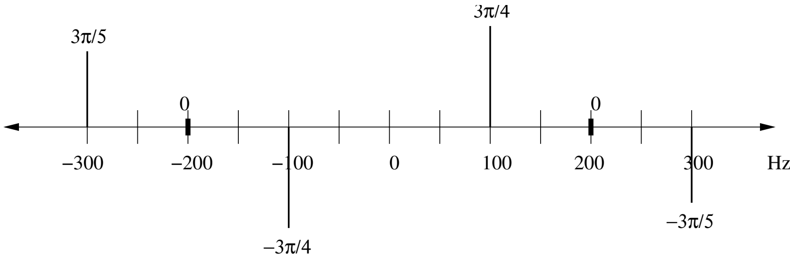



**Figure 4.16:** Spectrum plot: phase angles.

## 4.9 Summary

Real-world signals have sinusoid components, unless they are truly random, in which case they can still be approximated with a sum of sinusoids. Since all analog signals can be represented (or at least approximated) with a sum of sinusoids, we can view a signal in terms of the frequencies that comprise it. Given a sum-of-sinusoids equation for a signal, we use the inverse Euler's formula to convert to a sum of complex exponentials, such as $(a_1/2)e^{j2\pi f_1 t}e^{j\phi_1} + (a_1/2)e^{j2\pi(-f_1)t}e^{j(-\phi_1)}$. From this form, we can make a plot of the amplitudes ($a_1/2$, $a_1/2$) versus the frequencies ($f_1$, $-f_1$), as well as a plot of the phase angles versus the frequencies. Together, we call this information the spectrum, a view of the signal in the frequency-domain.

We will see how to get the frequency-domain information from a signal of only time-domain points using the Fourier transform, in Chapter 6.

## 4.10 Review Questions

1. What are the minimum and maximum values for $x(t)$?

$$x(t) = 3\cos(2\pi 100t + \pi/6)$$

2. Given $\sin(\theta) = \cos(\theta - \pi/2)$, if $x(t) = 3\sin(2\pi 100t + \pi/6)$, what is this function in terms of cos?

3. The analog signal, $x(t)$, is:

$$x(t) = 3\cos(2\pi 2000t + \pi/4) + 2\cos(2\pi 5000t) + \cos(2\pi 11000t - \pi/7)$$

   a. plot each frequency component (in the time-domain) separately
   b. graph this function (in the time-domain)
   c. represent $x(t)$ in terms of a fundamental frequency, amplitudes and phases.

4. The analog signal, $x(t)$, is:

$$x(t) = 3\cos(2\pi 3000t) + \cos(2\pi 1000t + \pi/3) + 4\cos(2\pi 5000t - \pi/3)$$

   Represent $x(t)$ in terms of a fundamental frequency, amplitudes and phases.

5. What is the difference between amplitude and magnitude? What does $|x(t)|$ mean?

6. DC component specifies the "shift of the signal" ________. ($f = 0$)

Phase-shift specifies shift of signal ____________. ($\phi$)

7. What is another name for a plot of the frequency information of a signal?
8. Which signal, a square wave or a triangular wave, is easier to approximate with a series of harmonically related sinusoids? Why?
9. Why are amplitudes always positive?
10. Since $x(t)$ below is in the form $a_0 + \sum_{k=1}^{N} a_k \cos(2\pi(k)f_0 t + \phi_k)$, what are the values for $N$, $a_k, \phi_k$, and $f_0$ ? What special name do we give this signal $x(t)$?

$$x(t) = 4 + 3\cos(2\pi(1)20t + \pi/2) + 7\cos(2\pi(2)20t - 2\pi/3)$$

# Chapter 5

# Sampling

We live in an analog world, in that the signals that interest us are typically continuous. Our computers are digital devices, meaning that they can store only finite, discrete data, limited in both precision and magnitude. Thus, we need to approximate signals before we can store and manipulate them. How do we bridge the gap between the real world and the synthetic one of our computers? This chapter addresses this central question.

*Sampling* is the process of getting a digital signal from an analog one. When we *sample* a signal, we record, every so often, a value. Sampling can lead to many data values. For example, a 3-minute song, sampled at 16 bits at 44,100 Hz (for two channels), produces:

$$(3\ minutes) \times \left(16\ \frac{bits}{sample}\right) \times \left(44100\ \frac{samples}{second}\right) \times 2.$$

Changing the units to make them uniform makes this:

$$\left(3\ minutes \frac{60\ seconds}{1\ minute}\right) \times \left(16 \frac{bits}{sample}\frac{1\ byte}{8\ bits}\right) \times \left(44100\ \frac{samples}{second}\right) \times 2$$

$$(180\ seconds) \times \left(2\ \frac{bytes}{sample}\right) \times \left(44100\ \frac{samples}{second}\right) \times 2$$

$$180 \times (2\text{ bytes}) \times 44100 \times 2 = 31{,}752{,}000\text{ bytes, approximately 30 MB.}$$

But do we really need to store 30 MB of data for a 3-minute song? Actually, there are other ways of storing this data. The MP3 format is one such solution. It is popular due to the fact that a song stored as an MP3 sounds very close to the original, while only requiring about 10% of the space.

## 5.1 Sampling

Converting an analog signal to a digital one is a necessary step for a computer to analyze a signal: modern computers are digital machines, and can store only digital values.

In the continuous function $x(t)$, we replace $t$, the continuous variable, with $nT_s$, a discrete value. We use $n$ to index the discrete array, and $T_s$ is the *sampling period*, the amount of time between two samples. The sampling time is the inverse of the *sampling frequency* $f_s$, that is, $T_s = \frac{1}{f_s}$.

Before we can talk about sampling, let us call the frequency range of the signal that we are interested in the *bandwidth*. Sometimes, we simply use the highest frequency as the bandwidth, implying that we are interested in all frequencies between 0 Hz and the maximum. Other times, it may be limited, e.g., the visible light spectrum.

The term *critical sampling* applies to the case when the sampling rate is exactly twice the bandwidth, $B$. This means that we will record the signal just fast enough to properly reconstruct it later, i.e., $f_s = 2B$. We get this value based on Nyquist's criterion, $f_s \geq 2B$, and choosing the lowest possible sampling frequency.

*Oversampling* is taking samples more frequently than needed (or more than 2 times the bandwidth). This results in more samples than are needed. You can use oversampling, but you may not want to due to hardware limitations, like not being able to process it fast enough, or not being able to store it in memory. Oversampling *may* be something you would want to do if you are working with cheap hardware, such as an inexpensive digital-to-analog converter, which appears to be the case with CD players [6]. An inexpensive digital-to-analog converter may use a simple step function to recreate the signal. Therefore, the more samples it has to work with, the smaller the time between the samples, and the better it approximates the original. As a counter example, reading (and storing) the temperature every microsecond would produce a massive volume of data with no gain for predicting tomorrow's weather.

*Undersampling* occurs when we do not take samples often enough, less than twice the bandwidth. You would not want to use this, since you cannot reconstruct the signal. Also, any analysis done on an undersampled signal will likely be erroneous (as we say in computer programming, "garbage in, garbage out").

$x[n] = x(nT_s)$ describes the sampling process. $T_s$ is the sampling time, while $n$ is an index. This means that the continuous signal $x$ is converted into a digital representation. $x[n]$ is *not exactly the same* as $x(t)$. It cannot be, since $x(t)$ is continuous. At best, $x[n]$ is a good approximation of $x(t)$.

The *period* of a signal is the length of time before it repeats itself. For a sinusoid,

we only need to know the frequency $f$, since we define the period as $T = \frac{1}{f}$.

---

**Example:**
If we used $x(t)$ below and sampled it at 20 kHz, how many samples would we have after 60 ms?

$$x(t) = 3\cos(2\pi 404t + \pi/4) + 2\cos(2\pi 6510) + \cos(2\pi 660t - \pi/5)$$

**Answer:**
The first thing to notice is that the number of samples really does not depend on the signal itself. That is, all we need to answer this question is the 20 kHz sampling rate ($f_s$) and the total time 60 ms.

The sampling period is 1/(20 kHz) or 1/(20,000 Hz). Since Hz is in cycles/sec, 1/Hz would be sec/cycles, or just seconds. We take the first sample at time = 0, then wait 1/(20,000) sec, and take another, and continue until 60 ms = $60 \times 10^{-3}$ seconds = 0.060 seconds is up.

To find the numerical answer, multiply 0.060 by 20,000, then add 1 (since the first sample was taken at time 0). For answering this (or any) question on a test, it is important to show your thought process. A good answer would include the analysis (and wording) as above.

---

**Example:**
if $x(t) = 4\cos(2\pi 250t + 2\pi/7)$, what is the period of this sinusoid?

**Answer:**
All that we really need is the frequency, 250 Hz. $T = \frac{1}{f} = \frac{1}{250\ Hz} = 0.004$ seconds.

---

Suppose that we have a complex signal, one with many harmonically related sinusoids. To find the period, we must find the largest frequency, $f_0$, such that all other frequencies are integer multiples of it. With two or more sinusoids, this base frequency can be found with the greatest common divisor (*gcd*) function. For three frequencies $f_1$, $f_2$, and $f_3$, we want to find $f_0$ such that $f_1 = k_1 f_0$, $f_2 = k_2 f_0$, and $f_3 = k_3 f_0$. The variables $k_1$, $k_2$, and $k_3$ must be integers.

## 5.2 Reconstruction

*Reconstruction* is the process of recreating an analog signal from digital samples. This conversion from digital to analog involves interpolation, or “filling in” the information missing between samples. There are several ways of doing this, the most obvious being a linear method of “connecting the dots” by drawing line segments between the samples.

An analog-to-digital converter, written for short as ADC, *samples* a continuous signal and produces a discrete one. The opposite of the ADC is the DAC, or digital-to-analog converter. When a D-to-A converter produces an analog signal between discrete data points, we call this *interpolation*, just as one may interpolate or “fill in” the values between data points. One simple way to do this is by using a zero-order hold for a pulse function[20]; this creates a step function so that time between two samples is filled by the first sample of the two. Linear interpolation is another choice for interpolation, where the pulse function is based upon the variable $t$. An example of this is a triangle function, where the pulse ramps up until the midpoint, then ramps down again. The following equation describes the process of digital-to-analog conversion [6]:

$$x(t) = \sum_{n=-\infty}^{\infty} x[n]p(t - nT_s)$$

where $p(t)$ is the pulse function. It could be simply a sustained version of the unit impulse function. In fact, we can use a very similar equation with the impulse function to describe sampling, as done in [14]. The index does not, of course, actually run from $-\infty$ to $\infty$, but this is the most general way to define this without knowing the support.

## 5.3 Sampling and High-Frequency Noise

When sampling a signal with high-frequency noise, we will see the effects of the noise even when the sampling frequency is of lower frequency than the noise. This is because the noise will make each sample a little larger or smaller than it would otherwise be. Figure 5.1 demonstrates this idea. Samples of a signal with some high-frequency noise are shown, with the original signal (without noise) shown as a solid line. An alias of the noise will show up in the sampled signal as a frequency component, explained in the following section.

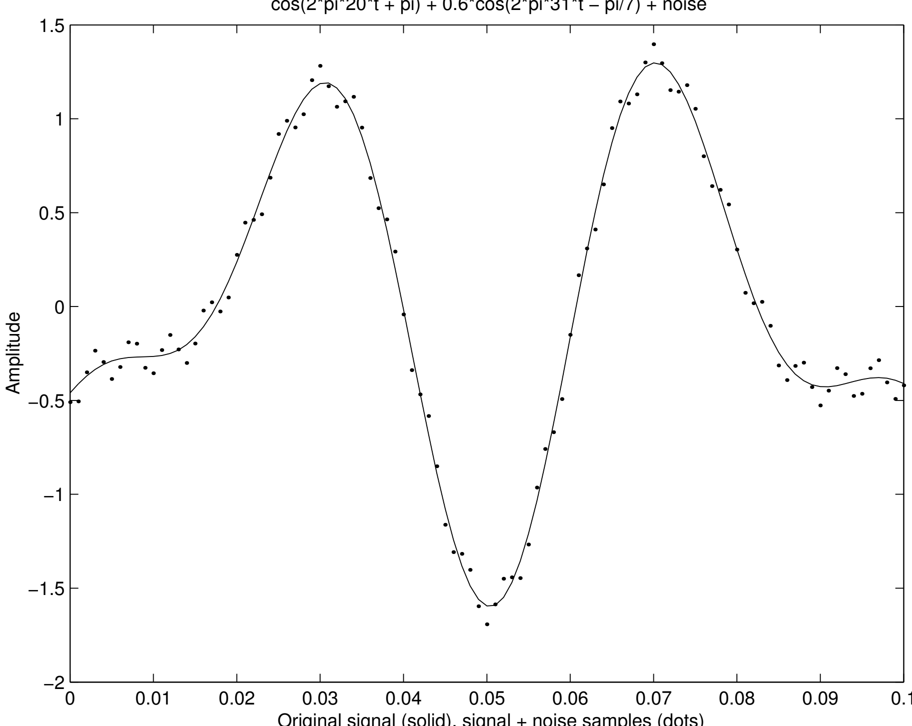



**Figure 5.1:** Sampling a noisy signal.

## 5.4 Aliasing

A common problem that arises when sampling a continuous signal is *aliasing*, where a sampled signal has replications of its sinusoidal components which can interfere with other components. Aliasing occurs when sampling a periodic signal, namely a sinusoid (or a signal made up of sinusoids) such as $a \times \cos(2\pi f t + \phi)$. When we sample this signal, we do so at the rate $f_s$. Any sinusoid component above $f_s$ will appear in the signal as if it were less than $f_s$.

This can best be explained with an example. Suppose we have a sinusoidal signal $x(t) = \cos(2\pi 800t)$. When we sample it, we replace $t$ in the continuous signal $x(t)$ with regularly spaced time points $nT_s$, i.e.,

$$x[n] = x(nT_s).$$

Suppose that we sample this 800 Hz sinusoid at $f_s = 600$ samples/second. When we examine the frequency content of the sampled signal, we see that it shows up as a 200 Hz sinusoid. What happened? Instead of seeing the 800 Hz sinusoid, we see one at 200 Hz instead. Let's look at what the sampled signal is:

$$x(t) = \cos(2\pi 800t)$$

$$x[n] = \cos(2\pi 800nT_s)$$

$$x[n] = \cos(2\pi(200 + 600)nT_s)$$

$$x[n] = \cos(2\pi 200nT_s + 2\pi 600nT_s)$$

$T_s = 1/f_s$, so $T_s = 1/600$. We will replace this in the right-most term.

$$x[n] = \cos(2\pi 200nT_s + 2\pi 600n(1/600))$$

$$x[n] = \cos(2\pi 200nT_s + 2\pi n)$$

The cosine function is periodic with period $2\pi$, which means that $\cos(\theta + 2\pi) = \cos(\theta)$. Just as we can add $2\pi$ to the cosine argument and get the same result, we can add $2\pi$ times any integer and get the same result (even if the integer is negative). We defined index $n$ as an integer, so the $2\pi n$ term can be removed from the above cosine argument, leaving us with $\cos(2\pi 200nT_s)$. When we sample a 800 Hz sinusoid at 600 samples/second, we cannot distinguish it from a 200 Hz sinusoid. If we had a 200 Hz component in the signal too, then the 800 Hz component would interfere with it.

Aliasing occurs on television whenever we see a car whose tires appear to be

spinning in the wrong direction. A television broadcast can be thought of as a series of images, sampled at a regular rate, appearing on screen. If the wheels happen to rotate less than a full circle between frames (images), then they appear to be turning slowly in the opposite direction.

This can be generalized as follows. Imagine that the frequency of a sinusoidal component is replaced by the frequency plus an integer multiple of the sampling frequency, i.e., we substitute $f_0 + kf_s$ in place of $f$.

$$x(t) = a \times \cos(2\pi(f_0 + kf_s)t + \phi)$$

$$x(t) = a \times \cos(2\pi f_0 t + 2\pi k f_s t + \phi)$$

Now we sample this signal at $T_s$ intervals. Note that there is no problem with the sinusoid until we sample it.

$$x[n] = x(nT_s) = a \times \cos(2\pi f_0 n T_s + 2\pi k f_s n T_s + \phi)$$

Since $T_s = \frac{1}{f_s}$, we can eliminate the $f_s T_s$ term.

$$x[n] = x(nT_s) = a \times \cos(2\pi f_0 n T_s + 2\pi k n + \phi)$$

Both $k$ and $n$ are integers, so their product is also an integer. This means that:

$$a \times \cos(2\pi f_0 n T_s + 2\pi k n + \phi) = a \times \cos(2\pi f_0 n T_s + \phi).$$

In other words, once we sample a signal, we cannot tell a sinusoid in it (such as 800 Hz in the example above) from another sinusoid (such as 200 Hz) that contains an integer multiple of the sampling frequency, in this case, 600 Hz. Clearly, this means that the sampling frequency $f_s$ must be at least greater than the largest frequency component, assuming that we have spectral content starting at 0 Hz. We will see other restrictions on $f_s$ in the next few sections.

Aliasing means that if we look at the spectrum of a sinusoid of $f$ Hz, there will be a component at $f$, but there will also be components at $(f + 1f_s)$, then at $(f + 2f_s)$, then at $(f + 3f_s)$, etc.

### 5.4.1 Aliasing Example

The program below demonstrates this problem graphically. We simulate sampling two sinusoids, $2\cos(2\pi 100t + \pi/3)$ and $2\cos(2\pi 600t + \pi/3)$, at 500 samples/second. They are given the same magnitude and phase for a reason: the program shows that the sampled versions are identical.

```
%
% Example of aliasing with
%       2 cos(2 pi 100 t + pi/3)
% and   2 cos(2 pi 600 t + pi/3)
%
% if these are sampled at 500 samples/sec,
% then the sampled versions are identical!
%

freq = 100;    % example frequency
phase = pi/3;  % example phase
mag = 2;       % example magnitude
fs = 500;      % sampling frequency
Ts = 1/fs;     % sampling period
k = 1;         % number of repetitions

num_points = 200;   % How many points to use
                    %  200 makes it look smooth
num_samples = 11;   % How many samples to simulate reading
                    %  11 puts "sampled" under "analog"

step = 2/(freq*num_points);  % get a nice step size
t = 0:step:2*(1/freq);       % "time"
n = 0:num_samples-1;         % sample index

% x and y are simulated analog functions
x = mag*cos(2*pi*freq*t + phase);
y = mag*cos(2*pi*(freq+k*fs)*t + phase);

% x2 and y2 are simulated sampled version of x and y
x2(n+1) = mag*cos(2*pi*freq*n*Ts + phase);
y2(n+1) = mag*cos(2*pi*(freq+k*fs)*n*Ts + phase);
```

Both $x2$ and $y2$ have the same values. To show the simulated sampled versions, the following code plots both, and generates Figure 5.2.

```
% Plot the "analog" signals
subplot(2,1,1);
plot(t, x, 'r.-', t, y,'b-');
my_title = sprintf('Simulated analog signals, x=dots  y=solid');
```

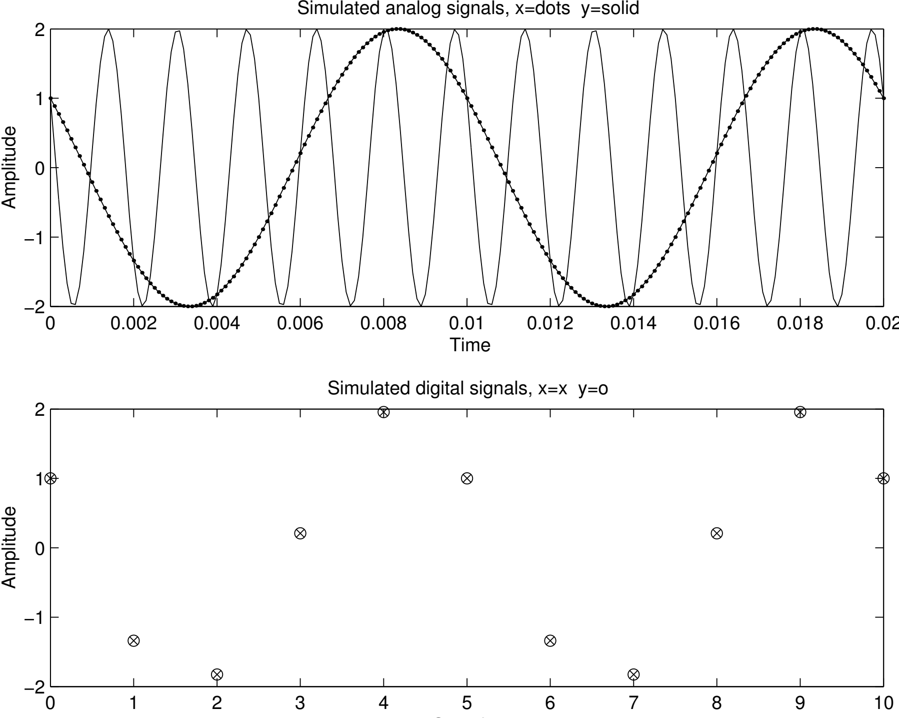



**Figure 5.2:** Aliasing demonstrated.

```
title(my_title);
xlabel(' time ');
ylabel('Amplitude');

% Plot the "sampled" signals
subplot(2,1,2);
plot(n,x2,'rx', n,y2,'bo');
my_title = sprintf('Simulated digital signals, x=x  y=o');
title(my_title);
xlabel(' samples ');
ylabel('Amplitude');
```

It is easy to figure out when to take samples. Since the sampling frequency is 500 samples/sec, the sampling period must be $1/500 = 0.002$ seconds. Taking our first sample at time $t = 0$, we will take samples again at 0.002 seconds, 0.004 seconds, 0.006 seconds, and so forth.

### 5.4.2 Folding

We saw that any frequency greater than the sampling frequency will show up as a lower frequency. *Folding* is another way of looking at the aliasing phenomenon, where the observed frequency of a sampled signal differs from the actual frequency, even though it is less than the sampling frequency but greater than half the sampling frequency. Some texts use the term "folding" [11] [2] [6], while others mention this only as "aliasing" [9] [20] [21]. It is a type of aliasing, but here we prefer the term "folding" to distinguish it from aliasing, since the phase angle is affected. You may also see $f_s/2$ referred to as the *folding frequency* if, for example, a 600 Hz sinusoid is sampled at $f_s = 1000$ samples/sec, and the observed frequency is $1000 - 600 = 400$ Hz. This comes from the fact that:

$$\begin{aligned}
\cos(2\pi 600 nT_s + \phi) &= \cos(2\pi(1000-400)nT_s + \phi) \\
&= \cos(2\pi 1000 nT_s - 2\pi 400 nT_s + \phi) \\
&= \cos(2\pi 1000 n/1000 - 2\pi 400 nT_s + \phi) \\
&= \cos(2\pi n - 2\pi 400 nT_s + \phi).
\end{aligned}$$

As before, we can remove an integer multiple of $2\pi$ from the cos function. Since $n$ is defined as an integer, we can remove the $2\pi n$ term:

$$\cos(2\pi n - 2\pi 400 nT_s + \phi) = \cos(-2\pi 400 nT_s + \phi).$$

It may seem strange to have a negative argument to the cosine function, but there is a trigonometric identity that can help: $\cos(-\theta) = \cos(\theta)$. Applying this to the equation above, we have:

$$\cos(-2\pi 400 n T_s + \phi) = \cos(2\pi 400 n T_s - \phi).$$

The frequency now appears to be 400 Hz, but the phase angle's value $\phi$ is negated. Therefore, if we were to plot the frequency magnitude response for this signal $x(t) = \cos(2\pi 600n/1000 + \phi)$, we would see a frequency of 400 Hz. If we plotted the phase angles, it would show up as $-\phi$.

The code below shows that the cosine function gives the same results whether the argument to it is positive or negative. Figure 5.3 shows the result of this code, with $\cos(-2\pi 10t - \pi/3)$ on the left half of the figure and $\cos(2\pi 10t + \pi/3)$ on the right.

```
%
% Demonstrate that cos(-theta) = cos(theta)
%

t=-0.1:0.001:0.1;  % time
phi = pi/3;        % example phase
freq = 10;         % example frequency

x = cos(2 * pi * -freq * t - phi);
y = cos(2 * pi * freq * t + phi);
t1 = -0.1:0.001:0;
t2 = 0:0.001:0.1;

%plot(t,x,'b.', t, y, 'g*')  % This plot shows both overlapping

plot(t1,x(1:ceil(length(x)/2)),'b*', t2, ...
        y(ceil(length(y)/2):length(y)), 'gd')
axis([-0.1 0.1 -1 1])
title('cos(-theta) and cos(theta) produce the same results');
xlabel('cos(-2\pi f t-\phi) (.) and cos(2\pi f t+\phi) (*)')
```

An alternate `plot` command appears as a comment within the code. It shows that the two cosine functions overlap each other. If you choose to experiment with it, do not forget to comment out the other `plot` command.

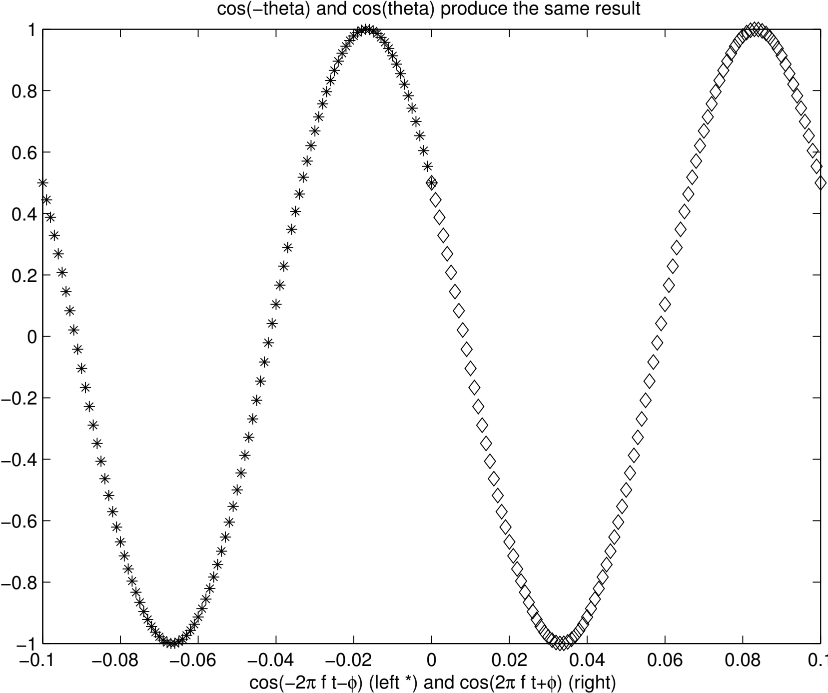



**Figure 5.3:** $\cos(-2\pi 10t - \pi/3)$ and $\cos(2\pi 10t + \pi/3)$ produce the same result.

Folding occurs when the sampled signal has a sinusoidal component that is greater than $f_s/2$, but less than $f_s$. In general, we can replace some frequency $f$ with $(f_s - f_0)$ to give:

$$\begin{aligned}
\cos(2\pi f n T_s + \phi) &= \cos(2\pi(f_s - f_0)nT_s + \phi) \\
&= \cos(2\pi f_s n T_s - 2\pi f_0 n T_s + \phi) \\
&= \cos(2\pi n - 2\pi f_0 n T_s + \phi) \\
&= \cos(-2\pi f_0 n T_s + \phi) \\
&= \cos(2\pi f_0 n T_s - \phi).
\end{aligned}$$

The value of $f_0$ must be less than the original $f$, otherwise the folded frequency will be outside the spectrum. For example, if we start with $\cos(2\pi 400 n T_s + \phi)$, with $T_s = 1000Hz$, and replace 400 with $1000-600$, we would end up with $\cos(2\pi 600 n T_s - \phi)$. This would not show up in the spectrum (from $-f_s/2$ to $f_s/2$), except as $\cos(2\pi 400 n T_s + \phi)$, which is what we started with.

### 5.4.3 Locations of Replications After Sampling

Once a signal is sampled, an infinite number of replications of frequency components will appear at regular intervals. This is based on the periodicity of the sinusoids, that is, $\cos(2\pi f t) = \cos(2\pi f t + 2\pi n)$, where $n$ is an integer, just as $\cos(\pi/6) = \cos(\pi/6 + 2\pi) = \cos(\pi/6 + 2\pi 2)$. This means that sampling a signal will put *all* frequency content in the range 0 .. $f_s$.

Since $n$ and $k$ are both integers by definition, their product $nk$ is also an integer.

$$\cos(2\pi f_1 n T_s) = \cos(2\pi f_1 n T_s + 2\pi n k)$$

$$= \cos\left(2\pi n \left(\frac{f_1}{f_s} + \frac{f_s}{f_s}k\right)\right)$$

Thus, $\cos(2\pi f_1 n T_s)$ is indistinguishable from $\cos(2\pi(f_1 + k f_s) n T_s)$, for all integers $k$. So when sampling at $f_s$ samples per second, $\cos(2\pi f_1 t)$ is indistinguishable from $\cos(2\pi(f_1 + k f_s)t)$, meaning that sampling creates a replica of a frequency at every integer multiple of the sampling frequency plus that frequency. We will look at negative values of $k$ next.

The cosine function is an even one, meaning that $\cos(-\theta) = \cos(\theta)$, or that

$$\cos(-2\pi f_1 t) = \cos(2\pi f_1 t).$$

This means that all frequency content on the spectrum between 0 and $+\infty$ will be mirrored between $-\infty$ and 0.

Therefore, a frequency of $f_1$ Hz will also appear at $-f_1$ Hz. When combined with aliasing, we see that it will also appear between $\frac{f_s}{2}$ and $f_s$. Suppose that we let $k = -1$:

$$\cos(2\pi f_1 n T_s) = \cos\left(2\pi n\left(\frac{f_1}{f_s} - \frac{f_s}{f_s}\right)\right).$$

So $\cos(2\pi f_1 n T_s)$ also appears as $\cos(2\pi(f_1 - f_s)nT_s)$.

---

**Example 1:**
Let $f_1 = 1$ Hz and $f_s = 4$ samples/second:

$$\cos(2\pi(1)nT_s) = \cos(2\pi n(1-4)T_s)$$

$$= \cos(2\pi n(-3)T_s)$$

$$= \cos(2\pi n(3)T_s).$$

See Figure 5.4. Note that we would also have a replica at $-1$ Hz. Since there are an infinite number of replications, we only show a few, though there are replications at $\pm 5$ Hz and $\pm 7$ Hz.

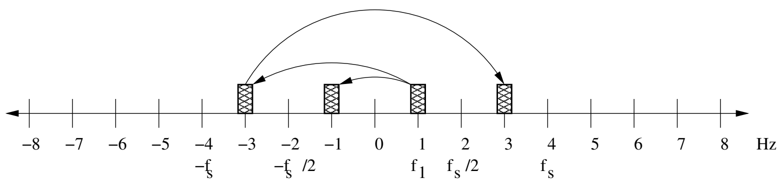



**Figure 5.4:** Replications for $f_1 = 1$ Hz and $f_s = 4$ samples/second.

---

**Example 2:**
Let $f_1 = 2.5$ Hz and $f_s = 8$ samples/second:

$$\cos(2\pi(2.5)nT_s) = \cos(2\pi n(2.5-8)T_s)$$

$$= \cos(2\pi n(-5.5)T_s)$$

$$= \cos(2\pi n(5.5)T_s).$$

See Figure 5.5. Note that we would also have a replica at −2.5 Hz.

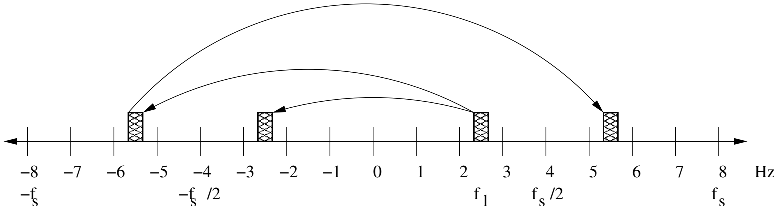



**Figure 5.5:** Replications for $f_1 = 2.5$ Hz and $f_s = 8$ samples/second.

From these examples, we see that $\cos(2\pi f_1 n T_s)$ appears the same as $\cos(\pm 2\pi(f_1 - f_s)nT_s)$, at least when $f_1 < \frac{f_s}{2}$.

---

**Example 3:**
What if $f_1 > \frac{f_s}{2}$? With $k = -1$, we will let $f_1 = 3$ Hz and $f_s = 4$ samples/second:

$$\cos(2\pi(3-4)nT_s) = \cos(2\pi(-1)nT_s)$$

$$= \cos(2\pi n T_s).$$

See Figure 5.6. Note that we would also have a replica at −3 Hz.

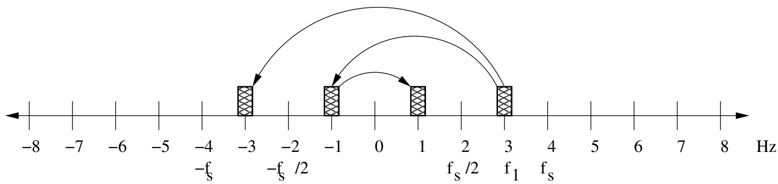



**Figure 5.6:** Replications for $f_1 = 3$ Hz and $f_s = 4$ samples/second.

---

**Example 4:**
What if $f_1 > f_s$? With $k = -1$, we will let $f_1 = 5$ Hz and $f_s = 4$ samples/second:

$$\cos(2\pi(5-4)nT_s) = \cos(2\pi(1)nT_s)$$
$$= \cos(2\pi(-1)nT_s).$$

That is, this will also appear as a negative frequency. See Figure 5.7. Note that we would also have a replica at $-5$ Hz.

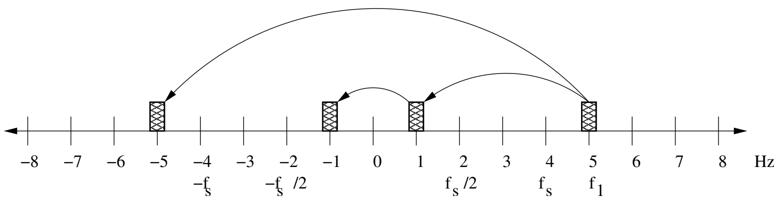



**Figure 5.7:** Replications for $f_1 = 5$ Hz and $f_s = 4$ samples/second.

We see that $\cos(2\pi f_1 nT_s)$ appears the same as $\cos(\pm 2\pi(f_1 - f_s)nT_s)$, regardless of where $f_1$ is in relation to $f_s$.

---

The previous two graphs, Figure 5.6 and Figure 5.7, are very similar. Wouldn't the last example also have a replica at 3 Hz? Repeating the fourth example:

$$\cos(2\pi(5-4)nT_s) = \cos(2\pi(1)nT_s) = \cos(2\pi(-1)nT_s),$$

but $\cos(2\pi(5-4)nT_s)$ also equals $\cos(2\pi(5-8)nT_s)$. If we let $k = -2$.

$$\cos(2\pi(5-8)nT_s) = \cos(2\pi(-3)nT_s) = \cos(2\pi(3)nT_s),$$

so we *do* see replicas at $\pm 3$ Hz, also. In fact, we have a replica of the 5 Hz signal at *all* odd-numbered frequencies on this spectrum, Figure 5.8. This can be a problem—we cannot tell the 3 Hz signal in Example 3 from the 5 Hz signal in Example 4. If we had both a 3 Hz and 5 Hz signal in these examples, they would interfere with each other.

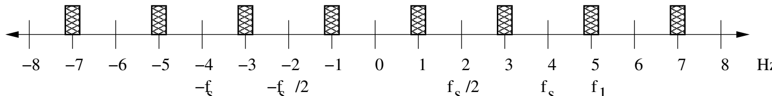



**Figure 5.8:** Replications for $f_1 = 3$ or 5 Hz and $f_s = 4$ samples/second.

## 5.5 Nyquist Rate

In the previous sections, we saw that the sampling rate $f_s$ must be greater than the largest frequency component to avoid aliasing. We also saw how this largest frequency $f$ should be less than half of the sampling rate $f_s$, to avoid folding. In the next section, we will see that the *bandwidth* is the important term; up to this point, we have assumed that we are interested in all frequencies from 0 Hz to the largest frequency component. This implies that the bandwidth equals the largest frequency component. In fact, it is more precise to say that the minimum sampling frequency is based on the bandwidth. In other words,

$$f_s \geq 2(bandwidth).$$

We call this minimum sampling rate for a signal the *Nyquist rate*. Other texts refer to Shannon's sampling theory, but these are the same thing. To find it, take the bandwidth (often the maximum frequency of the signal), and multiply by a factor of 2. The reason this rate is critical is that it avoids aliasing, a problem brought about with sampling.

---

**Example:**

$$x(t) = 2\cos(2\pi 700t - 5\pi/2) + 3\cos(2\pi 450t) + \cos(2\pi 630 + 2\pi/5)$$

What is the minimum sampling rate for this signal? Assume that we are interested in all frequencies starting with 0 Hz.

**Answer:**
First, put the sinusoids in the form $a_k \times \cos(2\pi f_k t + \phi_k)$. The frequencies ($f_k$) in $x(t)$ are: 700, 450, and 630 Hz. We see that the largest one is 700 Hz. From the question's assumption that we want all frequencies starting with 0 Hz, our bandwidth is therefore 700 Hz − 0 Hz = 700 Hz.

$$Nyquistrate \quad = \quad 2\ (bandwidth)$$

$$\begin{aligned} Nyquistrate &= 2\,(max(700\ Hz,\ 450\ Hz,\ 630\ Hz) - 0Hz) \\ &= 2(700\ Hz) \\ &= 1400\ Hz \end{aligned}$$

Therefore, our sampling rate should be at least 1400 Hz. This process is called *lowpass sampling*, since it accurately records all frequencies lower than the highest frequency component. However, if we do not need all frequencies between 0 and the maximum frequency (in this case, 700 Hz), section 5.6 shows another way to perform sampling.

---

## 5.6 Bandpass Sampling

It is possible to sample a signal at a rate lower than twice its highest frequency, if the signal has a limited bandwidth. This is called *bandpass sampling.*

Let's start with an example. Suppose there is a signal composed of four sinusoids, with frequencies 1010 Hz, 1020 Hz, 1030 Hz, and 1040 Hz.

$$x(t) = 4\cos(2\pi 1010t) + 6\cos(2\pi 1020t) + 8\cos(2\pi 1030t) + 10\cos(2\pi 1040t)$$

Figure 5.9 shows the spectrum of the sampled signal. The number of simulated samples is chosen to be a multiple of $f_s$, which in this case means that the DFT's analysis frequencies exactly match up to the frequencies present in the signal. In other words, there is no DFT leakage (see section 6.8), so the figures look nice. Also in this figure, only part of the spectrum is shown. The amplitudes below 1000 Hz in Figure 5.9 are all zero. One final note about this figure is that a simulated sampling frequency of 2081 samples/second was chosen because it gives a nice figure. If 2080 samples/second were used, then the frequency component at 1040 Hz would appear to have double the amplitude.

It would be possible to sample signal $x(t)$ at or above 2080 Hz, as Figure 5.9 shows, but what happens when we sample it at 100 Hz? We would have spectral content, as shown in Figure 5.10. As we can see from these graphs, the same information is present. There are advantages to sampling at a lower rate; namely, that the equipment to do so may be cheaper. But why does this work?

The answer lies in the same analysis that we used with aliasing. When a signal is sampled (digitized), values are periodically read from the analog signal. Effectively, this means that we replace $t$ with $n/f_s$ in the analog signal's mathematical representation. We are not likely to *have* a mathematical representation of our analog

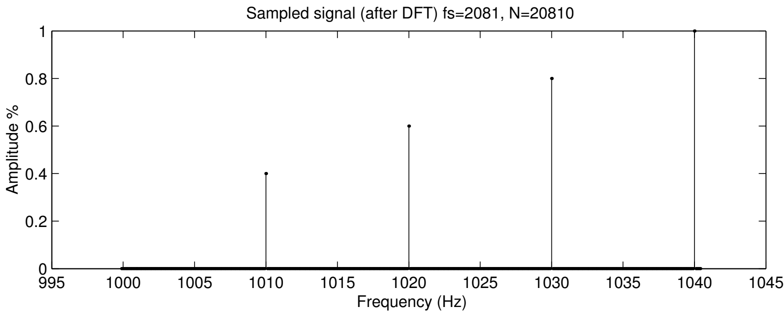



**Figure 5.9:** Example signal sampled at 2081 Hz.

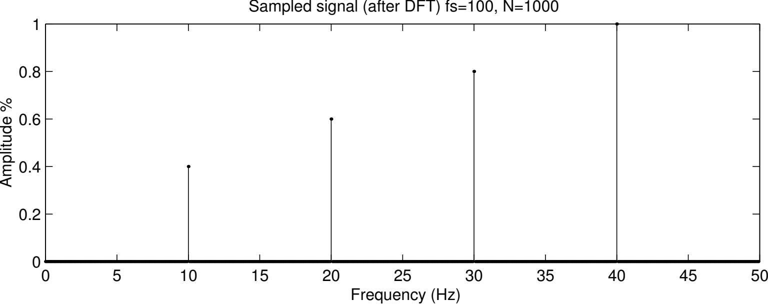



**Figure 5.10:** Example signal sampled at 100 Hz.

signal. After all, if we knew what the analog signal was, why would we bother to sample it? However, any analog signal can be represented as a sum of sinusoids, so it is reasonable to assume that any analog signal has a mathematical representation in a form like this. The form for each sinusoid component is $a \times \cos(2\pi f t + \phi)$. When we sample the signal, these components become $a \times \cos(2\pi f n/f_s + \phi)$. We know that $\cos(2\pi k + \phi) = \cos(\phi)$, as long as $k$ is an integer. This is not surprising; since the cosine function is periodic, it repeats itself over and over again. In fact, it repeats itself infinitely in both positive and negative directions (that is, $k$ can be a positive or negative integer). So a component of an analog signal will repeat itself, given enough time. When $f_s = 100$ samples/second, the first component becomes:

$$\begin{aligned}
4\cos(2\pi f n/f_s) &= 4\cos(2\pi 1010 n/100) \\
&= 4\cos(2\pi(1000 + 10)n/100) \\
&= 4\cos(2\pi 1000 n/100 + 2\pi 10 n/100) \\
&= 4\cos(2\pi 10 n + 2\pi n/10).
\end{aligned}$$

Since $n$ is an integer (i.e., let $k = 10n$), this reduces to:

$$4\cos(2\pi n/10).$$

In other words, the 1010 Hz component *appears* as a 10 Hz component. It is left to the reader to verify that the other components will appear as 20 Hz, 30 Hz, and 40 Hz when sampled at this rate. Also, the reader is encouraged to show that this works when there is a nonzero phase angle, e.g., for $4\cos(2\pi 1010 n/f_s + \pi/4)$. This example works because there is no conflict between frequencies. But what if there already was a 10 Hz component? In that case, the information would be lost, since both the 10 Hz and the 1010 Hz components would appear to be a single component at 10 Hz. Avoiding overlap is one thing to look for when bandpass sampling.

As we saw above, bandpass sampling depends upon the sampling frequency. If $f_s$ were, say, 103 samples per second, then the first component becomes:

$$\begin{aligned}
4\cos(2\pi f n/f_s) &= 4\cos(2\pi 1010 n/103) \\
&= 4\cos(2\pi(927 + 83)n/103) \\
&= 4\cos(2\pi 927 n/103 + 2\pi 83 n/103) \\
&= 4\cos(2\pi 9 n + 2\pi 83 n/103).
\end{aligned}$$

Since $n$ is an integer, this reduces to:

$$4\cos(2\pi 83n/103).$$

We do not stop here. Since $83n/103$ is equivalent to $(103 - 20)n/103$, we can eliminate another $2\pi n$ term, which leaves us with a frequency component that appears to be $-20$ Hz. We do this because it will show up as $-20$ Hz anyway; the signal's spectrum contains information from $-f_s/2$ to $f_s/2$, with the same information on both sides, reflected around 0. This frequency component 83/103 is greater than 103/2 (i.e., $f_s/2$), so it appears in the spectrum as $-20/103$. Since the left half of the spectrum for a real signal is a mirror of the right half, this shows up as $+20$ Hz, too. In other words, $\cos(-\phi) = \cos(\phi)$, for any real angle $\phi$. Likewise, when we remove multiples of 103 from the other components, we get:

$$x[n] = 4\cos(2\pi - 20n/103) + 6\cos(2\pi - 10n/103) + 8\cos(0) + 10\cos(2\pi 10n/103).$$

The information in both the second and the fourth components appears at 10 Hz, meaning that we have lost information. Figure 5.11 shows what the spectrum would look like if we did sample at 103 Hz. An interesting point is that the component at 0 Hz appears to be twice as large, as will any 0 Hz component. The amplitudes are relative to each other, and since we expect, say, 10 Hz to have an equal amplitude at $-10$ Hz, we should expect this for 0 Hz and $-0$ Hz, too. Using 2081 samples/second as we did in Figure 5.9, we avoided doubling the amplitude shown at 1040 Hz.

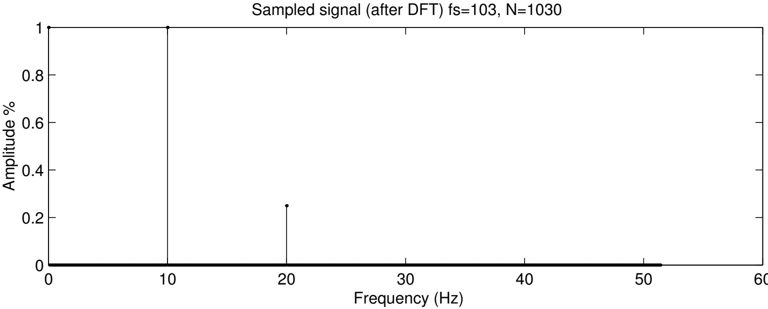



**Figure 5.11:** Example signal sampled at 103 Hz.

All real signal information will show up in the spectrum between 0 and $f_s/2$, even if it has a frequency component above $f_s/2$. This is the idea behind the Nyquist sampling rate; $f_s/2$ must be at least as large as the bandwidth. Since we are likely to know the bandwidth and need to solve for $f_s$, this becomes the relation $f_s \geq 2B$. We can think of the signal information mapping to the spectrum, from 0 to $f_s/2$, in

one of two ways. One is that this spectral information will be repeated, infinitely, for every multiple of $f_s$. Another way is to imagine that any spectral information that exists in the analog signal will map to a frequency between 0 and $f_s/2$, just like a curved mirror reflects everything in front of it, even if the mirror is smaller than the things reflected. If we are careful to avoid overlap (aliasing), we can use this to our advantage.

How do we choose a value for $f_s$? As pointed out previously, one bound is double the bandwidth, i.e., $f_s \geq 2B$. Another thing to consider is that each frequency component should map to a unique frequency between 0 and $f_s/2$. This can be accomplished with the equation below [11].

$$\frac{2f_c - B}{m} \geq f_s \geq \frac{2f_c + B}{m + 1}$$

The variable $f_c$ is the *center frequency*, defined as the midpoint between the lowest frequency of interest and the highest frequency of interest. Another way of viewing it is as the highest frequency of interest minus half the bandwidth. The variable $m$ is an integer, and it gives the number of replications, or how many times the information appears to repeat between 0 and the lowest frequency of interest. It is similar to the number of multiples of $f_s$ that we can remove from our original frequencies $f$.

Recall our earlier example, where we were interested in information between 1010 Hz and 1040 Hz. Here, $B = 1040 - 1010 = 30$ Hz, and $f_c = 1010 + B/2 = 1025$ Hz. We can find ranges for $f_s$ by computing the above equation for increasing values of $m$. If we use $m = 20$, we see that we get the range 2020/20 Hz $\geq f_s \geq$ 2080/21 Hz, or 101 Hz $\geq f_s \geq$ 99.0476 Hz. We see that $f_s = 100$ samples/second falls within this range. We must also make sure that the Nyquist criterion is satisfied, and since 100 Hz $\geq 2B$, or 100 Hz $\geq$ 60 Hz, we see that it is. If we calculate again using $m = 19$, we get the range of 106.3158 Hz $\geq f_s \geq$ 104 Hz. There are two things to notice. First, a sampling rate of 103 samples/second is outside these ranges, which explains why we were not able to use it in the previous example. Second, if we were to use a rate of, say, 105 samples/second (which is in the previous range), we get the graph of Figure 5.12. What is interesting here is that we get the same information, only it is *reversed* from what we expected.

Let's examine the first signal component, for $f_s = 105$ samples/second:

$$\begin{aligned} 4\cos(2\pi f n/f_s) &= 4\cos(2\pi 1010n/105) \\ &= 4\cos(2\pi(1050 - 40)n/105) \\ &= 4\cos(2\pi 1050n/105 - 2\pi 40n/105) \end{aligned}$$

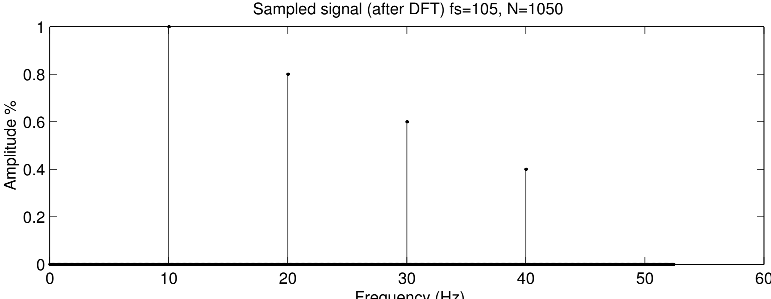



**Figure 5.12:** Example signal sampled at 105 Hz.

$$= \quad 4\cos(2\pi 10n - 2\pi 40n/105).$$

This reduces to:

$$4\cos(-2\pi 40n/105)$$

and since $\cos(-\phi) = \cos(\phi)$, this 1010 Hz component appears at 40 Hz. In a similar fashion, the other frequency components appear reversed from the original.

The information is there, but reversed, when $m$ is an odd number. Whether this is acceptable depends upon the application. Incidentally, if we were to choose $f_s = 110$ samples/second, which is from the range of $f_s$ when $m = 18$, we would have the information in the "correct" order, as shown in Figure 5.13. For a good overview on how the equation that gives the range of $f_s$ was developed, see R. Lyons, *Understanding Digital Signal Processing* [11].

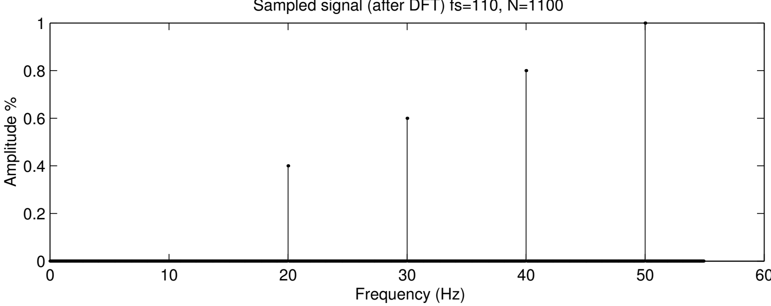



**Figure 5.13:** Example signal sampled at 110 Hz.

Table 5.1 summarizes these analyses. It shows that using 103 samples per second

**Table 5.1:** Frequencies detected with bandpass sampling.

| **Sampling** | **Actual Frequency** | | | |
|---|---|---|---|---|
| **Frequency** | **1010** | **1020** | **1030** | **1040** |
| 100 | 10 | 20 | 30 | 40 |
| 103 | 20 | 10 | 0 | 10 |
| 105 | 40 | 30 | 20 | 10 |
| 110 | 20 | 30 | 40 | 50 |

results in two frequencies mapping to the same observed value. The other sampling frequencies are fine to use, since each actual frequency maps to a unique observed frequency.

## 5.7 Summary

This chapter covers the topic of sampling. Sampling is the process of reading signals via a computer, also called Analog to Digital Conversion (ADC). We can go the other way and convert from digital back to analog, with a Digital to Analog converter (called a DAC). When dealing with sampling, however, there are some pitfalls to be wary of such as aliasing and folding.

Aliasing means that a frequency $f_1$, sampled at $f_s$ samples per second as in $\cos(2\pi f_1 n T_s)$, appears as $\cos(2\pi(f_1 + k f_s) n T_s)$, for all positive $k$ values.

Folding means that frequency $f_1$, sampled at $f_s$ samples per second as in $\cos(2\pi f_1 n T_s)$, appears as $\cos(2\pi(f_1 - k f_s) n T_s)$, for all positive $k$ values. Many textbooks combine aliasing and folding, since it is simply a matter of letting $k$ be negative. Note that a negative frequency between $-f_s/2$ and 0 will appear as a positive frequency between $f_s/2$ and $f_s$. That is, if $-f_s/2 < f_1 < 0$, it will have a replication at $f_1 + f_s$. Let $f_1 = -f_s/3$, and it will have a replication at $f_s - f_s/3 = 2f_s/3$, which is clearly between $f_s/2$ and $f_s$.

Since the cosine function is an even function, $\cos(+\theta) = \cos(-\theta)$, so that a frequency component $\cos(2\pi f_1 n T_s)$ also appears as $\cos(2\pi(-f_1) n T_s)$, this should come as no surprise. Euler's inverse formula $\cos(\theta) = (e^{j\theta} + e^{j(-\theta)})/2$ means that for every sinusoid term, there will be two components in the spectrum; at plus and minus the frequency. Every frequency component $\cos(2\pi f_1 n T_s)$ will appear in the digital spectrum as $\cos(\pm 2\pi(f_1 \pm k f_s) n T_s)$.

The Nyquist criteria must be observed, but bandpass sampling makes it possible to use replications to our advantage by sampling at a lower-than-expected rate.

## 5.8 Review Questions

1. If the sampling period is 8 ms, what is the sampling frequency?
2. What is oversampling? Why would you use it (or why would you not use it)?
3. What is undersampling? Why would you use it (or why would you not use it)?
4. Modify the aliasing example code (from section 5.4.1) to demonstrate folding.
5. What does "reconstruction" mean?
6. What do "ADC" and "DAC" mean?
7. If you attached the output of an ADC to the input of a DAC, would the DAC's output be exactly the same as the ADC's input? Why or why not?
8. What does $x[n] = x(nT_s)$ imply?
9. Show that aliasing occurs for more than one sinusoid, such as when we sample $x(t) = 3\cos(2\pi 300t + \pi/3) + 8\cos(2\pi 800t - \pi/5)$ at $f_s = 500Hz$.
10. Suppose we sampled the following signal $x(t)$ at $f_s = 10,000$ samples/second.

$$x(t) = 3\cos(2\pi 500t - \pi/5) + 6\cos(2\pi 700t + \pi/5) + 4\cos(2\pi 600t + 2\pi/7)$$

    a. How many samples would we have after 16 ms?

    b. What is the equation for $x[n]$?

    c. Write out what $x[0]$ and $x[1]$ are, and compute the value for $x[0]$.

    d. Using bandpass sampling, is the sampling rate OK? What about low-pass sampling? Explain.

11. Suppose we have a signal with frequencies between 18 kHz and 24 kHz. If we sampled the signal such that there would be 2 replicas of the signal (between −18 kHz and +18 kHz), what would our sampling frequency(ies) be?

12. Suppose you want to sample a signal with a frequency range from 15 kHz to 20 kHz. List ALL possible sampling frequencies that you can use (assuming you use bandpass sampling). How many possible sampling frequencies are there if you were to use lowpass sampling instead?

13. Suppose we have a signal with frequencies between 10 MHz and 15 MHz. If we sampled the signal, what possible sampling frequency(ies) could we use? Note how many replicas of the signal there would be (i.e., between $-10$ MHz and $+10$ MHz).

14. The analog signal, $x(t)$, is:

$$x(t) = 3\cos(2\pi 2000t + \pi/4) + 2\cos(2\pi 5000t) + \cos(2\pi 11000t - \pi/7).$$

a. What is the bandwidth of $x(t)$?
b. What is the equation for $x[n]$?

15. Assume the signal:

$$x(t) = 3\cos(2\pi 2000t + \pi/4) + 2\cos(2\pi 5000t) + \cos(2\pi 11000t - \pi/7)$$

is sampled at 10 kHz.

a. What does the $x[n]$ equation become?
b. Plot $x[n]$.
c. If $x(t)$ is sampled at 10 kHz, what aliases does the 5 kHz signal have?
d. What is the critical Nyquist frequency?
e. Could you use bandpass sampling with this signal? Why (or why not)?

16. Suppose we have a signal of interest between 6 kHz and 8 kHz, and we plan to use bandpass sampling.

a. What is the carrier (center) frequency?
b. What is the bandwidth?
c. If we sample this signal at 3500Hz, would this cause aliasing?
d. What sampling frequencies can we use for 1, 2, and 3 spectral replications?

17. Suppose that we have the following signals:

$x_1(t) = 6\cos(2\pi 7000t + \pi/2)$
$x_2(t) = 4\cos(2\pi 8000t)$

$x_3(t) = 2\cos(2\pi 6000t)$.

a. Plot each signal. On the same page, also show the plot of the signal with the frequency argument negated (i.e., $\cos(-2\pi ft - \phi)$).
b. What can you conclude about these plots?
c. Would your conclusion be the same if the sine function were used instead? Why or why not? (Do not show a graph, but do explain your answer.)

18. If we used $x(t)$ below and sampled it at 50 kHz, how many samples would we have after 25 ms?

$$x(t) = 3\cos(2\pi 2000t + \pi/4) + 2\cos(2\pi 5000t) + \cos(2\pi 11000t - \pi/7)$$

19. Given $x[n] = x(nT_s) = \cos(2\pi 1000nT_s)/2 + \cos(2\pi 2000nT_s + \pi/4)$ and $f_s = 8$ kHz, use MATLAB to find the first 10 $x[n]$ samples ($n = 0..9$).

20. Suppose you want to sample a signal with a frequency range from 14 kHz to 20 kHz. What is the lowest sampling frequency that you can use?

21. Suppose you have a signal of frequencies between 1 kHz and 22 kHz, and there is noise between 100 kHz and 102 kHz.

    a. If you sample at 44,000 samples/second, does the noise interfere with the signal?

    b. If you sample at 30,000 samples/second, there is a problem. What is the problem?

22. Suppose we have a signal with frequencies between 24 kHz and 28 kHz. If we sampled the signal such that there would be 2 replicas of the signal (between $-24$ kHz and $+24$ kHz), what would our sampling frequency be?

# Chapter 6

# The Fourier Transform

Many different fields, including medicine, optics, physics, and electrical engineering, use the Fourier Transform (FT) as a common analysis tool. In practice, the standards by compression groups JPEG (Joint Photographic Experts Group) and MPEG (Motion Picture Experts Group) use a modified form of the Fourier transform. Essentially, it allows us to look at frequency information instead of time information, which people find more natural for some data. For example, many stereo systems have rows of little lights that glow according to the strength of frequency bands. The stronger the treble, for example, the more lights along the row are lit, creating a light bar that rises and falls according to the music. This is the type of information produced by the Fourier transform.

The Discrete Fourier Transform (DFT) is the version of this transform that we will concentrate on, since it works on discrete data. Our data are discrete in time, and we can assume that they are periodic. That is, if we took another $N$ samples they would just be a repeat of the data we already have, in terms of the frequencies present. In this case, we use the DFT, which produces discrete frequency information that we also assume is periodic. Below, in Figure 6.1, we see the frequency magnitude response graph for the "ee" sound. We got this by applying the discrete Fourier transform to the sound file. The figure shows the whole range along the top, and a close-up view on the bottom.

The following graph, Figure 6.2, shows the frequency magnitude response for another sound file: a brief recording of *Adagio from Toccata and Fuge in C*, written by J.S. Bach. The top of this figure shows the entire frequency range, from 0 to 22,050 Hz, while the bottom part shows a close-up view of the first 4500 frequencies. An interesting thing to notice is that the spikes in magnitude are regularly spaced. This happens often with real signals, especially music. For example, right before 1000 Hz, we see three spikes increasing in magnitude, corresponding to three different

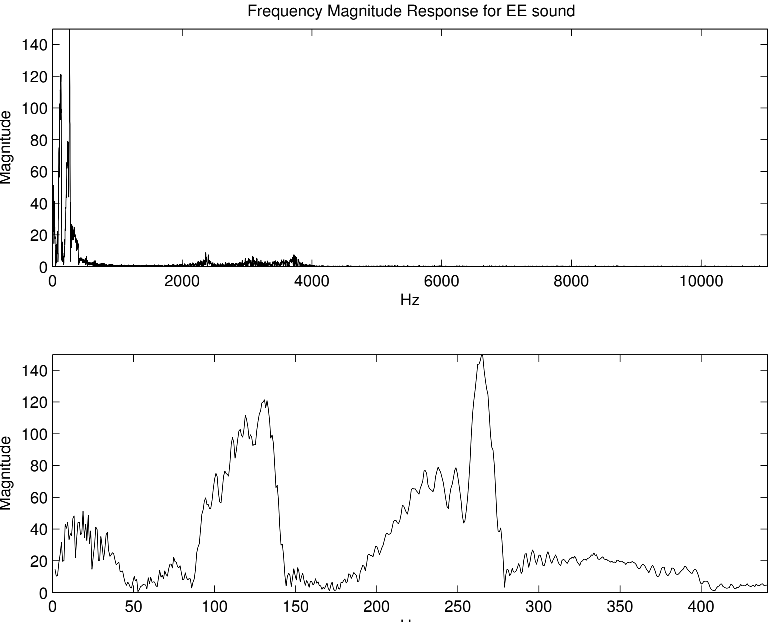



**Figure 6.1:** A person vocalizing the "ee" sound.

(but related) frequencies. We call this harmonics. This can be seen even more clearly in Figure 6.3, where a sustained note from a flute is played. Four frequencies are very pronounced, while most of the other frequencies are 0.

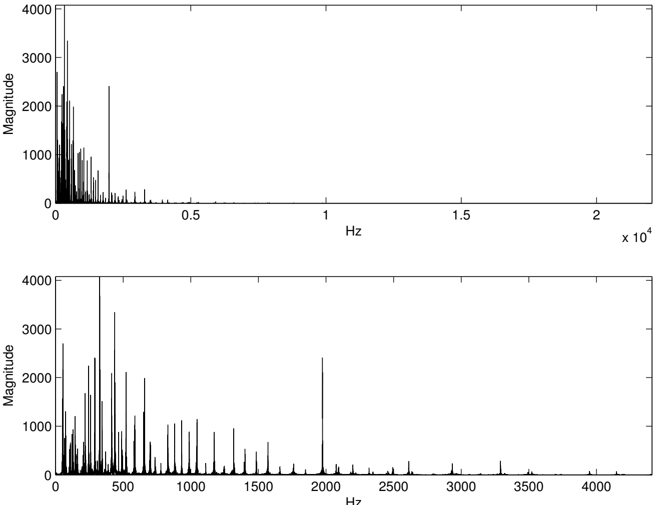



**Figure 6.2:** J.S. Bach's *Adagio from Toccata and Fuge in C*—frequency magnitude response.

The frequency range appearing above (0 to 22,050 Hz) was not arbitrarily chosen. Compact Disks (CDs) store music recorded at 44,100 samples per second, allowing for sounds in the range of 0 to 22,050 Hz, which is slightly greater than the maximum frequency that we can hear. It should not be surprising that almost all of the frequency content in Bach's music shown in Figure 6.2 is below 4000 Hz. The organ, for which this music was written, can produce a very wide range of sound. Some organs can produce infrasound notes (below 20 Hz, which most humans cannot hear), and also go well beyond 10 kHz. But these high notes are not necessary for pleasing music. To put this in perspective, consider that most instruments (guitar, violin, harp, drums, horns, etc.) cannot produce notes with fundamental frequencies above 4000 Hz, though instruments do produce harmonics that appear above this frequency [22]. A piano has a range of 27.5 Hz to just over 4186 Hz.

The Fourier transform is a way to map continuous time, nonperiodic data to continuous frequency, nonperiodic data in the frequency-domain. A variation called Fourier Series works with periodic data that is continuous in time, and turns it into a nonperiodic discrete frequency representation. When the data are discrete in time, and nonperiodic, we can use the discrete time Fourier transform (DTFT)

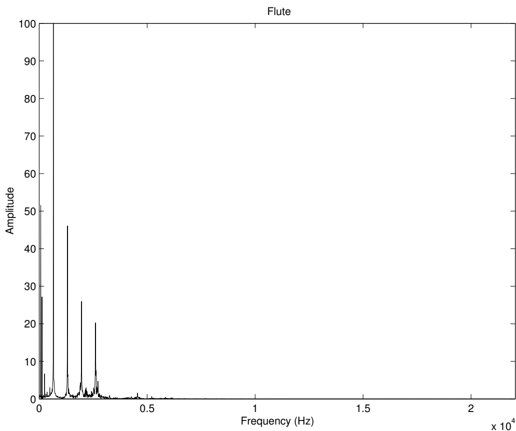



**Figure 6.3:** A sustained note from a flute.

to get a periodic, continuous frequency representation. Finally, when the data are discrete in time (and we can assume that they are periodic), we use the DFT to get a discrete frequency, assumed periodic representation.

## 6.1 Fast Fourier Transform Versus the Discrete Fourier Transform

The Fast Fourier Transform (FFT) is a clever algorithm to implement the DFT. As expected, it gives the same results as the DFT, but it arrives at them much faster due to the efficiency of the algorithm. The FFT was a major breakthrough, since it allowed researchers to calculate the Fourier transform in a reasonable amount of time. Figure 6.4 demonstrates this difference. The bottom curve is a plot of $Nlog_2(N)$, while the other is $N^2$. For example, a late-model Pentium-4 processor at 2 GHz does on the order of 2 billion calculations per second. An algorithm (such as the DFT) that performs $N^2$ operations would take about 5 seconds to complete for 100,000 data samples. For 100,000,000 data samples, this would take *about 2 months* to compute! In contrast, an algorithm (such as the FFT) that performs $Nlog_2(N)$ operations for 100,000,000 data samples would need only 1.33 seconds.

MATLAB provides both `fft` and `ifft` functions. For optimum speed, the FFT needs the data size to be a power of 2, though software (such as `fft` command in MATLAB) does not require this. For most applications, zeros can be appended to the data without negatively affecting the results. In fact, zeros are often appended to get better looking results. It does not add any information, but it changes the

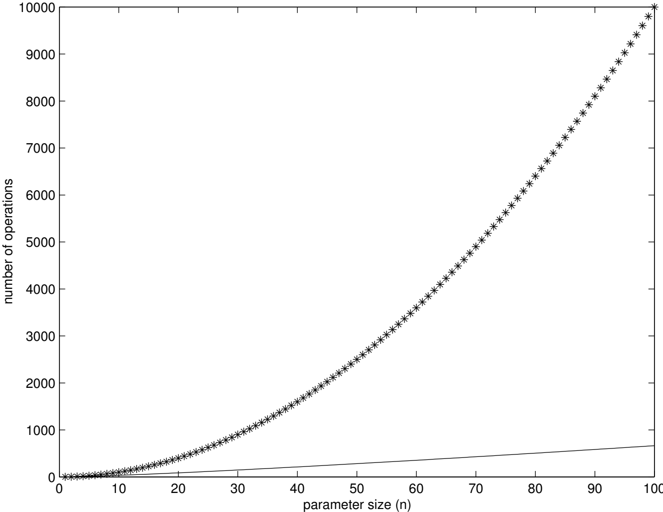



**Figure 6.4:** Comparing $Nlog_2(N)$ (line) versus $N^2$ (asterisks).

analysis frequencies used by the FFT/DFT. This technique is called *zero-padding*.

We saw in Chapter 3, "Filters," that filters perform convolution. One can use the FFT to compute convolution efficiently. Other uses include analyzing the effect that a filter has on a signal, and designing FIR filters (with the Inverse FFT). Any time that a problem requires the DFT, the FFT can be used instead.

## 6.2 The Discrete Fourier Transform

This is the Discrete Fourier Transform (DFT)

$$X[m] = \sum_{n=0}^{N-1} x[n](\cos(2\pi nm/N) - j\sin(2\pi nm/N))$$

where $m = 0..N-1$. We call this the "rectangular form" of the DFT, and there are many variations of this transform. For example, one common way to express it uses the $e^{-j2\pi nm/N}$ term instead of the sinusoids, which can make analysis by a human easier.

$$X[m] = \sum_{n=0}^{N-1} x[n]e^{-j2\pi nm/N}$$

Here is a MATLAB function that calculates the DFT. Its purpose is to demonstrate how the DFT can be computed, but it does not have the efficiency of the `fft` function built in to MATLAB. It returns a complex array, which stores two pieces of information per complex number, corresponding to an $x$ and $y$ coordinate. Normally, we would need the magnitudes and phase angles. Just as we can convert from Cartesian coordinates to polar coordinates, we can convert from the complex array information to the magnitude and phase information.

```
function [X] = dft(x)
%
% Demonstrate DFT
%

Xsize = length(x);

% do DFT (the hard way)
for m=0:Xsize-1
   mysumm = 0;
   for n=0:Xsize-1
      mysumm = mysumm + x(n+1) * (cos(2*pi*n*m/Xsize) ...
         - j*sin(2*pi*n*m/Xsize));
   end
   X(m+1) = mysumm;
end
```

All that needs to be returned is the vector of complex numbers, or variable $X$. The magnitudes and phase angles can be calculated, though this information makes $X$ redundant. To convert, we can use the `abs` and `angle` functions, such as:

```
Xmag = abs(X);
Xphase = angle(X);
```

Also notice that the comments are honest. This is not a very efficient way to obtain the Fourier transform! But it is meant to explain how the DFT can be calculated.

For the DFT, notice how the arguments of the cosine and sine functions are the same, yet the arguments are functions of both $n$ and $m$. In effect, this transform spreads the original one-dimensional data over a two-dimensional matrix. Let's illustrate this with an example. The following code finds the DFT of an example

signal (it uses `fft`, but `dft` would also work assuming that the above program is present).

```
>> fft([6, 4, 9, 0, 1, 5, 2, 7])

ans =

  Columns 1 through 4

  34.0000   9.2426 - 1.3431i  -4.0000 - 2.0000i   0.7574 +12.6569i


  Columns 5 through 8

  2.0000    0.7574 -12.6569i  -4.0000 + 2.0000i   9.2426 + 1.3431i
```

Table 6.1 shows a matrix where we calculate the Fourier transform of the example signal. The rows correspond to $n$ and we see the values from the example signal running down each of the columns ($m$). Observe how each row has a corresponding signal sample (time-domain data), while the sum of each column results in the DFT of the signal (frequency domain data). If we were to find the sum of magnitudes along each row (i.e., `sum(abs(Matrix(r,:)))`), we would get the time-domain sample multiplied by the number of points ($N$). Every $e^{-j\theta}$ value has a magnitude of 1; these values are complex vectors of unit magnitude. This 2D matrix comes from the expression for the DFT, $e^{-j\theta}$, where $\theta = 2\pi nm/N$, and $N = 8$ (our sample size). The values of $2\pi nm/N$ for a single column run from 0 to $2\pi m(N-1)/N$. Variable $m$ also runs from 0 to $(N-1)$. The inverse DFT, which we will cover soon, essentially multiplies table entries by an $e^{+j\theta}$ vector, cancelling out the original complex vectors and giving us back the time-domain data.

How to add the values of complex exponentials together may not be obvious. We can always convert these to Cartesian coordinates (of the form $a + jb$) first. Table 6.2 shows another way of looking at this data, with the rectangular form of the DFT equation. We can easily verify that the sum of a column equals the DFT results given by MATLAB, since we add the real parts and imaginary parts separately.

Each column results in a single frequency-domain point. That is, for each output $X[m]$, the frequencies used to find $X[m]$ are $2\pi m(0)/N$, $2\pi m(1)/N$, ..., $2\pi m(N-1)/N$. The number of frequencies used depends entirely upon how many points we have, which is determined by our sampling frequency. Therefore, the analysis frequencies are given by the following relation:

**Table 6.1:** Example DFT calculations.

| | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
|---|---|---|---|---|---|---|---|---|
| 0 | $6e^{-j2\pi 0}$ | $6e^{-j2\pi 0}$ | $6e^{-j2\pi 0}$ | $6e^{-j2\pi 0}$ | $6e^{-j2\pi 0}$ | $6e^{-j2\pi 0}$ | $6e^{-j2\pi 0}$ | $6e^{-j2\pi 0}$ |
| 1 | $4e^{-j2\pi 0}$ | $4e^{-j2\pi 0.125}$ | $4e^{-j2\pi 0.25}$ | $4e^{-j2\pi 0.375}$ | $4e^{-j2\pi 0.5}$ | $4e^{-j2\pi 0.625}$ | $4e^{-j2\pi 0.75}$ | $4e^{-j2\pi 0.875}$ |
| 2 | $9e^{-j2\pi 0}$ | $9e^{-j2\pi 0.25}$ | $9e^{-j2\pi 0.5}$ | $9e^{-j2\pi 0.75}$ | $9e^{-j2\pi}$ | $9e^{-j2\pi 1.25}$ | $9e^{-j2\pi 1.5}$ | $9e^{-j2\pi 1.75}$ |
| 3 | $0e^{-j2\pi 0}$ | $0e^{-j2\pi 0.375}$ | $0e^{-j2\pi 0.75}$ | $0e^{-j2\pi 1.125}$ | $0e^{-j2\pi 1.5}$ | $0e^{-j2\pi 1.875}$ | $0e^{-j2\pi 2.25}$ | $0e^{-j2\pi 2.625}$ |
| 4 | $1e^{-j2\pi 0}$ | $1e^{-j2\pi 0.5}$ | $1e^{-j2\pi 1}$ | $1e^{-j2\pi 1.5}$ | $1e^{-j2\pi 2}$ | $1e^{-j2\pi 2.5}$ | $1e^{-j2\pi 3}$ | $1e^{-j2\pi 3.5}$ |
| 5 | $5e^{-j2\pi 0}$ | $5e^{-j2\pi 0.625}$ | $5e^{-j2\pi 1.25}$ | $5e^{-j2\pi 1.875}$ | $5e^{-j2\pi 2.5}$ | $5e^{-j2\pi 3.125}$ | $5e^{-j2\pi 3.75}$ | $5e^{-j2\pi 4.375}$ |
| 6 | $2e^{-j2\pi 0}$ | $2e^{-j2\pi 0.75}$ | $2e^{-j2\pi 1.5}$ | $2e^{-j2\pi 2.25}$ | $2e^{-j2\pi 3}$ | $2e^{-j2\pi 3.75}$ | $2e^{-j2\pi 4.5}$ | $2e^{-j2\pi 5.25}$ |
| 7 | $7e^{-j2\pi 0}$ | $7e^{-j2\pi 0.875}$ | $7e^{-j2\pi 1.75}$ | $7e^{-j2\pi 2.625}$ | $7e^{-j2\pi 3.5}$ | $7e^{-j2\pi 4.375}$ | $7e^{-j2\pi 5.25}$ | $7e^{-j2\pi 6.125}$ |
| $\Sigma$ | $34e^{j2\pi 0}$ | $9.3e^{-j2\pi 0.023}$ | $4.5e^{-j2\pi 0.426}$ | $12.7e^{j2\pi 0.24}$ | $2e^{-j2\pi 0}$ | $12.7e^{-j2\pi 0.24}$ | $4.5e^{j2\pi 0.4}$ | $9.3e^{j2\pi 0.023}$ |

**Table 6.2:** Example DFT calculations (rectangular form).

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 6.0 + 0.0j | 6.0 + 0.0j | 6.0 + 0.0j | 6.0 + 0.0j | 6.0 + 0.0j | 6.0 + 0.0j | 6.0 + 0.0j | 6.0 + 0.0j |
| 1 | 4.0 + 0.0j | 2.8 - 2.8j | 0.0 - 4.0j | -2.8 - 2.8j | -4.0 - 0.0j | -2.8 + 2.8j | -0.0 + 4.0j | 2.8 + 2.8j |
| 2 | 9.0 + 0.0j | 0.0 - 9.0j | -9.0 - 0.0j | -0.0 + 9.0j | 9.0 + 0.0j | 0.0 - 9.0j | -9.0 - 0.0j | -0.0 + 9.0j |
| 3 | 0.0 + 0.0j | 0.0 + 0.0j | 0.0 + 0.0j | 0.0 + 0.0j | 0.0 + 0.0j | 0.0 + 0.0j | 0.0 + 0.0j | 0.0 + 0.0j |
| 4 | 1.0 + 0.0j | -1.0 - 0.0j | 1.0 + 0.0j | -1.0 - 0.0j | 1.0 + 0.0j | -1.0 - 0.0j | 1.0 + 0.0j | -1.0 - 0.0j |
| 5 | 5.0 + 0.0j | -3.5 + 3.5j | 0.0 - 5.0j | 3.5 + 3.5j | -5.0 - 0.0j | 3.5 - 3.5j | -0.0 + 5.0j | -3.5 - 3.5j |
| 6 | 2.0 + 0.0j | -0.0 + 2.0j | -2.0 - 0.0j | 0.0 - 2.0j | 2.0 + 0.0j | -0.0 + 2.0j | -2.0 - 0.0j | -0.0 - 2.0j |
| 7 | 7.0 + 0.0j | 4.9 + 4.9j | -0.0 + 7.0j | -4.9 + 4.9j | -7.0 - 0.0j | -4.9 - 4.9j | -0.0 - 7.0j | 4.9 - 4.9j |
| Σ | 34.0 + 0.0j | 9.2 - 1.3j | -4.0 - 2.0j | 0.8 + 12.7j | 2.0 - 0.0j | 0.8 - 12.7j | -4.0 + 2.0j | 9.2 + 1.3j |

$$f_{analysis}[m] = \frac{m f_{sampling}}{N}.$$

## 6.3 Plotting the Spectrum

To get a spectral plot, we will start out with an example signal. The following code sets up a signal, $x$.

```
>> % Set up an example signal
>> n = 0:99; % number of points
>> fs = 200;  % sampling frequency
>> Ts = 1/fs; % sampling period
>> % x is our example signal
>> x = cos(2*pi*20*n*Ts + pi/4) + ...
    3*cos(2*pi*40*n*Ts - 2*pi/5) + ...
    2*cos(2*pi*60*n*Ts + pi/8);
>>
```

Now we need the frequency information from $x$, which we can get from the Fourier transform. Also, we will establish the index $m$.

```
>> X = fft(x);
>> m = 0:length(X)-1;
```

It may be helpful to know the frequency resolution; the following code displays it.

```
>> disp(sprintf('Freq resolution is every %5.2f Hz',...
    fs/length(X)));
Freq resolution is every  2.00 Hz
>>
```

To see the spectrum, we will show both the frequency magnitude response and a plot of the phase angles.

```
>> % Plot magnitudes
>> subplot(2,1,1);
>> stem(m*fs/length(X),abs(X), 'b');
>> ylabel('magnitude');
```

```
>> xlabel('frequency (Hz)');
>> title('Frequency magnitude response');
>> % Plot phase angles
>> subplot(2,1,2);
>> stem(m*fs/length(X),angle(X), 'b');
>> ylabel('phase angle');
>> xlabel('frequency (Hz)');
>> title('Phase angle plot');
```

When we run the previous code, we see the graphs as shown in Figure 6.5, and clearly there is frequency content at 20 Hz, 40 Hz, and 60 Hz, with magnitudes of 50, 150, and 100, corresponding to relative magnitudes of 1, 3, and 2 as in our original signal. We notice a few annoying things about this figure, though. First, since the original signal $x$ is real, the frequency magnitude response has a mirror image, so we only really need the first half of it. For the phases, the pattern is also reflected around the x-axis, so again we only need half of the plot. The phase plot contains a lot of information, actually too much. At frequency 80 Hz, for example, we see that the amplitude approximates zero, while the phase angle is relatively large. (We expect the phase angles to be between $-\pi$ and $+\pi$.)

To solve the first problem, we will simply plot the first half of both the magnitudes and phases. We can set up a variable called *half_m* to give us half the range of $m$, and use it in place of $m$. But when we use it, we must be careful to add 1 to it before using it as an array offset, since it starts at 0.

```
half_m = 0:ceil(length(X)/2);
stem(half_m*fs/length(X),abs(X(half_m+1)), 'b');
```

For the second problem, we need a little more information.

```
>> X(2)

ans =

 -1.6871e-13 - 2.2465e-15i

>> angle(X(2))

ans =

   -3.1283
```

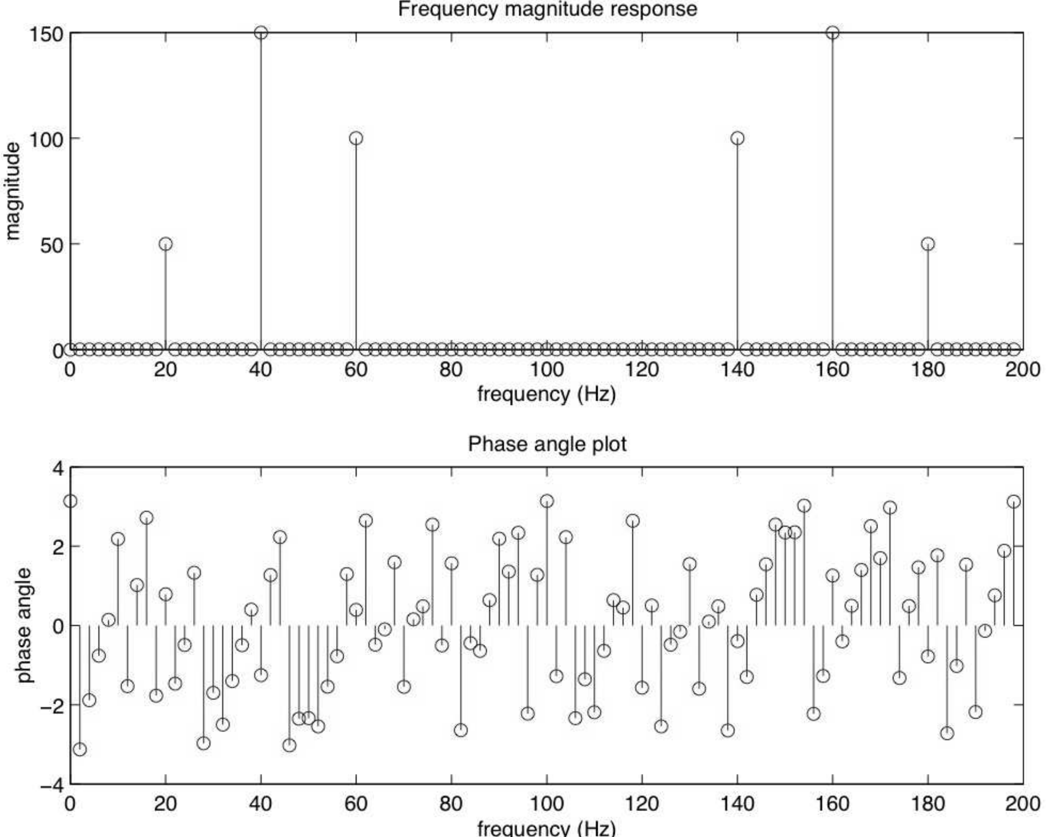



**Figure 6.5:** Spectrum for an example signal.

```
>> abs(X(2))

ans =

   1.6873e-13
```

The above commands confirm our suspicion that the phase angles are calculated and shown for every $X$ value even when they are close to zero. Fixing this requires the use of a tolerance value. If the magnitude is smaller than the tolerance, then we should assume a zero phase. After all, the magnitudes correspond to how much a sinusoid contributes to our signal. If a sinusoid contributes about zero, then it makes no sense to keep a nonzero phase angle for it.

```
>> % The next 3 lines allow us to ignore phases
>> % that go with very small magnitudes.
>> tolerance = 0.00001;
>> X2 = ceil(abs(X) - tolerance);
>> X3 = round(X2 ./ (X2+1));
>> % X3 above is a vector of 0s and 1s
```

The above commands may be a bit confusing, but these lines set up a binary vector. The subtraction operation separates the magnitudes for us; since all magnitudes are positive, any negative values after the subtraction correspond to phases that we should ignore, and the `ceil` function will make them zero. Now the problem is to map the nonzero values in $X2$ to 1. Dividing each value of $X2$ by itself (plus 1) will return values that are either 0 or almost 1, while avoiding a divide-by-zero error. The `round` function simply takes care of making the "almost 1" values equal to 1. The code results in vector $X3$, consisting of zeros for the phases to ignore, and ones for the phases we want to see. We can then use it in the stem plot below, to zero-out any phases with magnitudes close to zero. The following plot does not use *half_m* as above, but we can put all of this together.

```
subplot(2,1,2);
stem(m*fs/length(X),angle(X).*X3, 'b');
```

Now we can put all of this together into a program, to plot the spectrum. When we run this program, we get the graphs shown in Figure 6.6, with the magnitudes and phase angles plotted versus the frequencies.

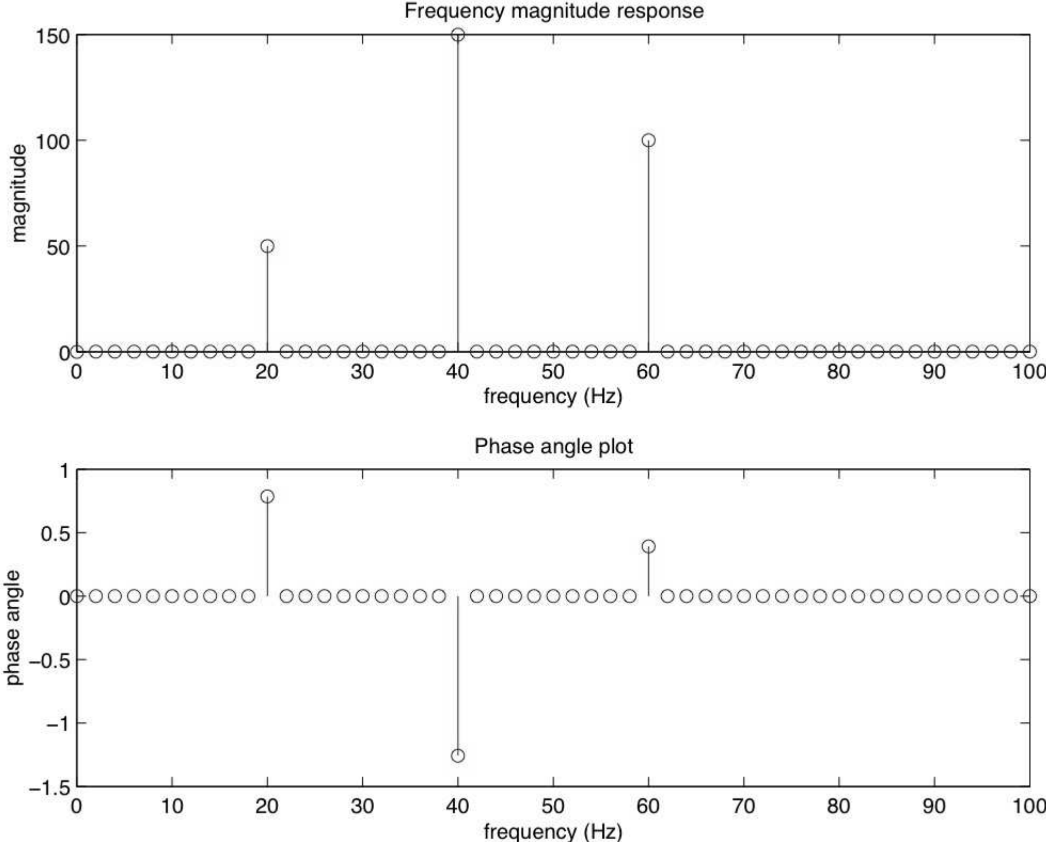



**Figure 6.6:** Improved spectrum for an example signal.

This program is shown in its entirety, since its pieces are shown above. To summarize what it does, first it makes an example signal by adding several sinusoids. Next, it finds the Fourier transform, then prints the frequency resolution. After this, it plots the magnitudes. It also plots the phases, but first it zeros-out the phases that have a tiny corresponding magnitude.

```
% spectrum.m
%
% Show the spectrum of an example signal.
%

% Set up an example signal
n = 0:99; % number of points
fs = 200;  % sampling frequency
Ts = 1/fs; % sampling period
% x is our example signal
x = cos(2*pi*20*n*Ts + pi/4) + ...
    3*cos(2*pi*40*n*Ts - 2*pi/5) + ...
    2*cos(2*pi*60*n*Ts + pi/8);

% Find the spectrum
X = fft(x);
%m = 0:length(X)-1;
half_m = 0:ceil(length(X)/2);

disp(sprintf('Freq resolution is every %5.2f Hz',...
    fs/length(X)));

% Plot magnitudes
subplot(2,1,1);
%stem(m*fs/length(X),abs(X), 'b');
stem(half_m*fs/length(X),abs(X(half_m+1)), 'b');
ylabel('magnitude');
xlabel('frequency (Hz)');
title('Frequency magnitude response');
% Plot phase angles
subplot(2,1,2);
% The next 3 lines allow us to ignore phases
% that go with very small magnitudes.
tolerance = 0.00001;
```

```
X2 = ceil(abs(X) - tolerance);
X3 = round(X2 ./ (X2+1));
% X3 above is a vector of 0s and 1s
%stem(m*fs/length(X),angle(X).*X3, 'b');
stem(half_m*fs/length(X), ...
    angle(X(half_m+1)).*X3(half_m+1), 'b');
ylabel('phase angle');
xlabel('frequency (Hz)');
title('Phase angle plot');
```

## 6.4 Zero Padding

When we zero-pad a signal in the time-domain, we get a smoother-looking frequency resolution. Why does this happen? Actually, the length of the sampled signal determines the frequency resolution, and zero-padding does not add any new information. Though the zero-padded signal results in a more visually appealing frequency-domain representation, it does not add to the resolution of the transform.

Why can we zero-pad our time-domain signal? Part of the explanation has to do with the continuous Fourier transform. For convenience, we use the radian frequency, $\omega = 2\pi f$. Using $\omega$ also allows us to use $f(t)$ for our function's name, without confusing it with frequency.

$$F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-j\omega t}dt$$

Suppose $f(t)$ is a signal that has zero value outside the interval $[a, b]$, and assume $a < b < c$. We can, therefore, replace the previous equation with the following:

$$F(\omega) = \int_{a}^{c} f(t)e^{-j\omega t}dt$$

$$= \int_{a}^{b} f(t)e^{-j\omega t}dt + \int_{b}^{c} f(t)e^{-j\omega t}dt.$$

Since $f(t) = 0$ outside the interval $[a, b]$ by definition, the second integral evaluates to 0. We can add 0 to the righthand side of the equation,

$$F(\omega) = 0 + \int_{a}^{b} f(t)e^{-j\omega t}dt.$$

Therefore, for continuous signals, $F(\omega)$ is the same whether or not we consider

$f(t)$ outside of its support. For a discrete signal, the number of points for the frequency-domain signal equals the number of points in the time-domain signal, so adding zeros means that there will be more frequency-domain points. The analysis frequencies, as a result, become finer since they are closer together.

## 6.5 DFT Shifting Theory

There is a DFT shifting theory, stating that sampling a signal will give the same results (in terms of frequency magnitude response), even when the samples are shifted, such as removing the first $k$ samples and appending them to the end of the sample sequence. This is an important theory, since it tells us that there is no critical time to start recording samples. The program below demonstrates this idea. Notice how the `plot` command puts two signals on the graph at the same time, one in blue ('b') and the other in green ('g'). Try other colors like red ('r') and black ('k').

```
% Demonstrate DFT Shifting theory
%

% parameters to modify
number_of_samples = 16;
shift = 3;      % Must be less than number_of_samples
fs = 2000;      % sampling frequency

% Make our example signal
Ts = 1/fs;
range = 1:number_of_samples;
x(range) = 2*cos(2*pi*400*(range-1)*Ts) ...
         + cos(2*pi*700*(range-1)*Ts);

% Make y a shifted version of x
y = [x(shift:number_of_samples), x(1:shift-1)];

% Get fft of each signal
X = fft(x);
Y = fft(y);

% Find magnitudes
```

```
Xmag = abs(X);
Ymag = abs(Y);
% Find phase angles
Xphase = angle(X);
Yphase = angle(Y);

% Show results in a graph
subplot(3,1,1), plot(range, x,'bd', range, y,'g*');
mystr = strcat('original (blue diamond) and shifted', ...
    '(green *) versions of the sampled signal');
title(mystr);
subplot(3,1,2), plot(range, Xmag,'bd', range, Ymag,'k*');
title('Xmagnitude = diamond, Ymagnitude = *');
subplot(3,1,3), plot(range, Xphase,'bd', range, Yphase,'k*');
title('Xphase = diamond, Yphase = *');
```

Figure 6.7 shows what the output for this program looks like. Though the phase angles are different, the frequency magnitudes are the same for the original as well as the shifted signal. We should expect the phase angles to be different, since these values determine how a composite signal "lines up" with with the samples. Otherwise, we would not be able to get the original signals back via the inverse transform.

## 6.6 The Inverse Discrete Fourier Transform

This is the Inverse Discrete Fourier Transform (IDFT):

$$x[n] = \frac{1}{N}\sum_{m=0}^{N-1} X[m](\cos(2\pi nm/N) + j\sin(2\pi nm/N))$$

where $n = 0..N-1$. Notice how similar it is to the DFT, the main differences being the $1/N$ term, and the plus sign in front of the complex part. The plus sign is no coincidence, the inverse transform works by using what we call the *complex conjugate* of the forward transform. A complex conjugate, stated simply, occurs when the complex part has a negated sign. For example, $4 - j2$ and $4 + j2$ are complex conjugates. A superscripted asterisk denotes complex conjugation. If we say $a = 3 + j7$ and $b = 3 - j7$, then we could also say $a = b^*$ or $b = a^*$.

An alternate form of the IDFT follows, using Euler's formula to replace the sinusoids with exponentials.

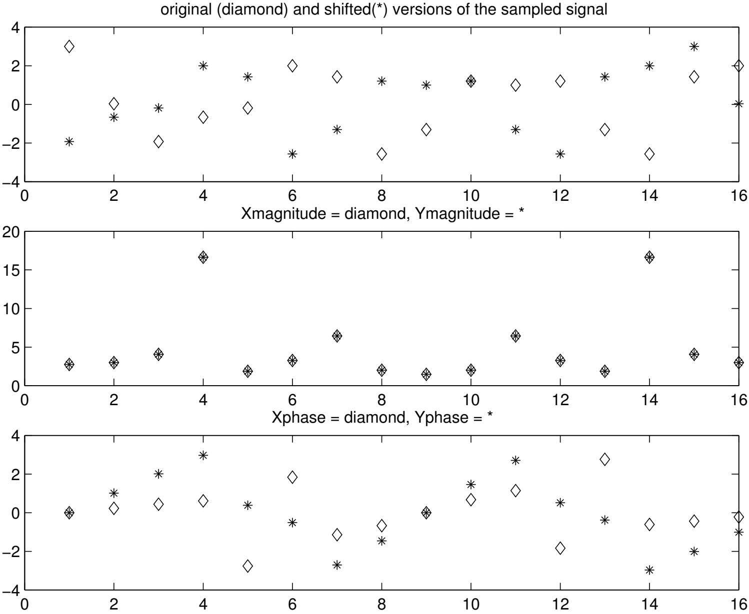



**Figure 6.7:** Example output of DFT-shift program.

$$x[n] = \frac{1}{N} \sum_{m=0}^{N-1} X[m] e^{j2\pi nm/N}$$

Below is a MATLAB function that calculates the Inverse Discrete Fourier Transform (IDFT). Like the DFT program above, this program intends only to demonstrate how the IDFT can be calculated. MATLAB has an `ifft` function that gives the same results, only faster.

```
function [x] = idft(X)
% Find the 1D Inverse DFT of an input signal.
% It returns the IDFT of the signal as a
% vector of complex numbers.
%
% Usage:
%     [x] = idft(X);
%
% This function is NOT very efficient,
```

```
% but it demonstrates how the IDFT is done.

Xsize = length(X);
% Do reconstruction
for n=0:Xsize-1
   for m=0:Xsize-1
      arra(n+1,m+1) = X(m+1) * (cos(2*pi*m*n/Xsize) + ...
         j*sin(2*pi*m*n/Xsize));
   end
end

for n=0:Xsize-1
   mysumm = 0;
   for m=0:Xsize-1
      mysumm = mysumm + arra(n+1,m+1);
   end
   % Keep only the real part
   %x(n+1) = real(mysumm) / Xsize;
   x(n+1) = mysumm / Xsize;
end
```

Below is shown a MATLAB session that demonstrates the DFT and IDFT functions. First, the signal *mysignal* is given some example values. Next, the "dft" function is called. Notice that the "dft" function will return three values, but here we only use the first.

After the DFT values are stored in variable *M*, the inverse discrete Fourier transform is computed, and stored in *mysignal_2*. This signal should have the same information as *mysignal*, and this is shown to be true. The `round` function gets rid of the part beyond the decimal point, and the `real` function ignores the imaginary part (which is zero).

```
  To get started, select "MATLAB Help" from the Help menu.

>> mysignal = [ 7 4 3 9 0 1 5 2 ];
>> M = dft(mysignal)

M =

  Columns 1 through 6
```

```
  31.0000             4.1716 - 5.0711i  -1.0000 + 6.0000i
9.8284 - 9.0711i  -1.0000 - 0.0000i   9.8284 + 9.0711i

  Columns 7 through 8

  -1.0000 - 6.0000i   4.1716 + 5.0711i

>> mysignal_2 = idft(M)

mysignal_2 =

  Columns 1 through 6

   7.0000 + 0.0000i   4.0000 + 0.0000i   3.0000 - 0.0000i
9.0000 - 0.0000i   0.0000 - 0.0000i   1.0000 + 0.0000i

  Columns 7 through 8

   5.0000 + 0.0000i   2.0000 + 0.0000i

>> real(round(mysignal_2))

ans =

     7     4     3     9     0     1     5     2

>> mysignal

mysignal =

     7     4     3     9     0     1     5     2
```

## 6.7 Forward and Inverse DFT

This section shows that when we take a signal, perform the DFT on it, and then perform the IDFT on the result, we will end up with the same values that we started with. We will begin by remembering the formulas.

Discrete Fourier Transform (DFT):

$$X[m] = \sum_{n=0}^{N-1} x[n]e^{-j2\pi nm/N}$$

where $m = 0..N-1$.

Inverse Discrete Fourier Transform (IDFT):

$$x[n] = \frac{1}{N}\sum_{m=0}^{N-1} X[m]e^{j2\pi nm/N}$$

where $n = 0..N-1$.

Let's start with an example signal, $x = \{x_0, x_1, x_2, x_3\}$. We can find the DFT of $x$ using the general formulas above, and since we know that $N = 4$, we can replace it in the formula.

$$X[m] = \sum_{n=0}^{3} x[n]e^{-j2\pi nm/4}$$

$$X[m] = x_0e^{-j2\pi 0m/4} + x_1e^{-j2\pi 1m/4} + x_2e^{-j2\pi 2m/4} + x_3e^{-j2\pi 3m/4}$$

We can then find the inverse transform...

$$x[n] = \frac{1}{4}\sum_{m=0}^{3} X[m]e^{j2\pi nm/4}$$

$$x[n] = \frac{1}{4}(X[0]e^{j2\pi n0/4} + X[1]e^{j2\pi n1/4} + X[2]e^{j2\pi n2/4} + X[3]e^{j2\pi n3/4})$$

...and replace the $X[m]$ terms with their equivalents.

$$x[n] = \frac{1}{4}((x_0e^{-j2\pi 0\times 0/4} + x_1e^{-j2\pi 1\times 0/4} + x_2e^{-j2\pi 2\times 0/4} + x_3e^{-j2\pi 3\times 0/4})e^{j2\pi n0/4} +$$

$$(x_0e^{-j2\pi 0\times 1/4} + x_1e^{-j2\pi 1\times 1/4} + x_2e^{-j2\pi 2\times 1/4} + x_3e^{-j2\pi 3\times 1/4})e^{j2\pi n1/4} +$$

$$(x_0e^{-j2\pi 0\times 2/4} + x_1e^{-j2\pi 1\times 2/4} + x_2e^{-j2\pi 2\times 2/4} + x_3e^{-j2\pi 3\times 2/4})e^{j2\pi n2/4} +$$

$$(x_0e^{-j2\pi 0\times 3/4} + x_1e^{-j2\pi 1\times 3/4} + x_2e^{-j2\pi 2\times 3/4} + x_3e^{-j2\pi 3\times 3/4})e^{j2\pi n3/4})$$

Combining the exponents:

$$x[n] = \frac{1}{4}((x_0 e^{-j2\pi 0\times 0/4 + j2\pi n0/4} + x_1 e^{-j2\pi 1\times 0/4 + j2\pi n0/4} +$$

$$x_2 e^{-j2\pi 2\times 0/4 + j2\pi n0/4} + x_3 e^{-j2\pi 3\times 0/4 + j2\pi n0/4}) +$$
$$(x_0 e^{-j2\pi 0\times 1/4 + j2\pi n1/4} + x_1 e^{-j2\pi 1\times 1/4 + j2\pi n1/4} +$$
$$x_2 e^{-j2\pi 2\times 1/4 + j2\pi n1/4} + x_3 e^{-j2\pi 3\times 1/4 + j2\pi n1/4}) +$$
$$(x_0 e^{-j2\pi 0\times 2/4 + j2\pi n2/4} + x_1 e^{-j2\pi 1\times 2/4 + j2\pi n2/4} +$$
$$x_2 e^{-j2\pi 2\times 2/4 + j2\pi n2/4} + x_3 e^{-j2\pi 3\times 2/4 + j2\pi n2/4}) +$$
$$(x_0 e^{-j2\pi 0\times 3/4 + j2\pi n3/4} + x_1 e^{-j2\pi 1\times 3/4 + j2\pi n3/4} +$$
$$x_2 e^{-j2\pi 2\times 3/4 + j2\pi n3/4} + x_3 e^{-j2\pi 3\times 3/4 + j2\pi n3/4})).$$

Multiplying out the terms:

$$x[n] = \frac{1}{4}((x_0 e^{j2\pi(-0\times 0 + n0)/4} + x_1 e^{j2\pi(-1\times 0 + n0)/4} +$$

$$x_2 e^{j2\pi(-2\times 0 + n0)/4} + x_3 e^{j2\pi(-3\times 0 + n0)/4}) +$$
$$(x_0 e^{j2\pi(-0\times 1 + n1)/4} + x_1 e^{j2\pi(-1\times 1 + n1)/4} +$$
$$x_2 e^{j2\pi(-2\times 1 + n1)/4} + x_3 e^{j2\pi(-3\times 1 + n1)/4}) +$$
$$(x_0 e^{j2\pi(-0\times 2 + n2)/4} + x_1 e^{j2\pi(-1\times 2 + n2)/4} +$$
$$x_2 e^{j2\pi(-2\times 2 + n2)/4} + x_3 e^{j2\pi(-3\times 2 + n2)/4}) +$$
$$(x_0 e^{j2\pi(-0\times 3 + n3)/4} + x_1 e^{j2\pi(-1\times 3 + n3)/4} +$$
$$x_2 e^{j2\pi(-2\times 3 + n3)/4} + x_3 e^{j2\pi(-3\times 3 + n3)/4})).$$

Simplifying:

$$x[n] = \frac{1}{4}((x_0 e^{j2\pi 0/4} + x_1 e^{j2\pi 0/4} + x_2 e^{j2\pi 0/4} + x_3 e^{j2\pi 0/4}) +$$

$$(x_0 e^{j2\pi(n)/4} + x_1 e^{j2\pi(n-1)/4} + x_2 e^{j2\pi(n-2)/4} + x_3 e^{j2\pi(n-3)/4}) +$$
$$(x_0 e^{j2\pi(2n)/4} + x_1 e^{j2\pi(2n-2)/4} + x_2 e^{j2\pi(2n-4)/4} + x_3 e^{j2\pi(2n-6)/4}) +$$
$$(x_0 e^{j2\pi(3n)/4} + x_1 e^{j2\pi(3n-3)/4} + x_2 e^{j2\pi(3n-6)/4} + x_3 e^{j2\pi(3n-9)/4})).$$

Simplifying even more ($e^{j0} = 1$):

$$x[n] = \frac{1}{4}((x_0 + x_1 + x_2 + x_3)+$$

$$(x_0 e^{j2\pi(n)/4} + x_1 e^{j2\pi(n-1)/4} + x_2 e^{j2\pi(n-2)/4} + x_3 e^{j2\pi(n-3)/4})+$$
$$(x_0 e^{j2\pi(2n)/4} + x_1 e^{j2\pi(2n-2)/4} + x_2 e^{j2\pi(2n-4)/4} + x_3 e^{j2\pi(2n-6)/4})+$$
$$(x_0 e^{j2\pi(3n)/4} + x_1 e^{j2\pi(3n-3)/4} + x_2 e^{j2\pi(3n-6)/4} + x_3 e^{j2\pi(3n-9)/4})).$$

Let's see what we get when $n = 1$:

$$x[1] = \frac{1}{4}((x_0 + x_1 + x_2 + x_3)+$$

$$(x_0 e^{j2\pi(1)/4} + x_1 e^{j2\pi(0)/4} + x_2 e^{j2\pi(-1)/4} + x_3 e^{j2\pi(-2)/4})+$$
$$(x_0 e^{j2\pi(2)/4} + x_1 e^{j2\pi(0)/4} + x_2 e^{j2\pi(-2)/4} + x_3 e^{j2\pi(-4)/4})+$$
$$(x_0 e^{j2\pi(3)/4} + x_1 e^{j2\pi(0)/4} + x_2 e^{j2\pi(-3)/4} + x_3 e^{j2\pi(-6)/4})).$$

Remember that these exponential values, such as $2\pi(-6)/4$, correspond to angles, and any angle greater than $2\pi$ can have an integer multiple of $2\pi$ removed from it. For example, $e^{j2\pi(-6)/4} = e^{-j12\pi/4} = e^{-j3\pi} = e^{-j(2\pi+\pi)} = e^{-j\pi}$.

$$x[1] = \frac{1}{4}((x_0 + x_1 + x_2 + x_3)+$$

$$(x_0 e^{j\pi/2} + x_1 + x_2 e^{-j\pi/2} + x_3 e^{-j\pi})+$$
$$(x_0 e^{j\pi} + x_1 + x_2 e^{-j\pi} + x_3 e^{-j2\pi})+$$
$$(x_0 e^{j\pi 3/2} + x_1 + x_2 e^{-j\pi 3/2} + x_3 e^{-j\pi}))$$

Next, we will replace the exponentials with sinusoids, using Euler's formula.

$$x[1] = \frac{1}{4}((x_0 + x_1 + x_2 + x_3)+$$

$$(x_0(\cos(\pi/2) + j\sin(\pi/2)) + x_1 + x_2(\cos(\pi/2) - j\sin(\pi/2)) + x_3(\cos(\pi) - j\sin(\pi)))+$$
$$(x_0(\cos(\pi) + j\sin(\pi)) + x_1 + x_2(\cos(\pi) - j\sin(\pi)) + x_3(\cos(2\pi) - j\sin(2\pi)))+$$
$$(x_0(\cos(3\pi/2) + j\sin(3\pi/2)) + x_1 + x_2(\cos(3\pi/2) - j\sin(3\pi/2)) + x_3(\cos(\pi) - j\sin(\pi))))$$

Now, we can calculate the sinusoids (Table 6.3), and plug in the values.

**Table 6.3:** Sinusoids that simplify things for us.

| | |
|---|---|
| $\cos(\pi/2) = 0,$ | $\sin(\pi/2) = 1$ |
| $\cos(\pi) = -1,$ | $\sin(\pi) = 0$ |
| $\cos(2\pi) = 1,$ | $\sin(2\pi) = 0$ |
| $\cos(3\pi/2) = 0,$ | $\sin(3\pi/2) = -1$ |

$$x[1] = \frac{1}{4}((x_0 + x_1 + x_2 + x_3)+$$

$$(x_0(j) + x_1 + x_2(-j) + x_3(-1 - j0))+$$

$$(x_0(-1 + j0) + x_1 + x_2(-1 - j0) + x_3(1 - j0))+$$

$$(x_0(j(-1)) + x_1 + x_2(-j(-1)) + x_3(-1 - j0)))$$

Simplifying:

$$x[1] = \frac{1}{4}((x_0 + x_1 + x_2 + x_3)+$$

$$(x_0(j) + x_1 + x_2(-j) + x_3(-1))+$$

$$(x_0(-1) + x_1 + x_2(-1) + x_3(1))+$$

$$(x_0(-j) + x_1 + x_2(j) + x_3(-1))).$$

Grouping terms:

$$x[1] = \frac{1}{4}(x_0(1 + j - 1 - j) + x_1(1 + 1 + 1 + 1) + x_2(1 - j - 1 + j) + x_3(1 - 1 + 1 - 1))$$

$$x[1] = \frac{1}{4}(x_0(0) + x_1(4) + x_2(0) + x_3(0))$$

$$x[1] = x_1.$$

Thus, the inverse transform gives us back the data that we had before the forward transform. The $\frac{1}{N}$ term appears in the inverse transform to scale the outputs back to the original magnitude of the inputs. We only show this for one of the original values, but the interested reader should be able to verify that this works for all four of the original values.

## 6.8 Leakage

The DFT does not have infinite resolution. A consequence of this is that sometimes frequencies that are present in a signal are not sharply defined in the DFT, and the frequency's magnitude response appears to be spread out over several analysis frequencies. This is called *DFT leakage*. The following code demonstrates this. In the code below, we simulate two sampled signals: $x1$ and $x2$. If you look closely, you will see that the equations defining these two signals are the same. What we change are the parameters; the sampling frequencies ($fs1$ and $fs2$), which alter the sampling periods ($Ts1$ and $Ts2$), and the number of samples to read (given by the lengths of $n1$ and $n2$).

```
fs1 = 1000;          fs2 = 1013;
Ts1 = 1/fs1;         Ts2 = 1/fs2;
n1 = 0:99;           n2 = 0:98;

x1 = 3*cos(2*pi*200*n1*Ts1 - 7*pi/8) + 2*cos(2*pi*300*n1*Ts1) ...
     + cos(2*pi*400*n1*Ts1 + pi/4);

x2 = 3*cos(2*pi*200*n2*Ts2 - 7*pi/8) + 2*cos(2*pi*300*n2*Ts2) ...
     + cos(2*pi*400*n2*Ts2 + pi/4);

mag1 = abs(fft(x1));
mag2 = abs(fft(x2));
```

The results are shown in the following graphs. When we take 100 samples of signal $x$ at 1000 samples per second, we get the spectral plot shown in Figure 6.8.

$$x(t) = 3\cos(2\pi 200t - 7\pi/8) + \cos(2\pi 300t) + 2\cos(2\pi 400t + \pi/4)$$

It shows what happens when the analysis frequencies happen to match up nicely with the frequencies present in the signal. The second figure represents the same signal, only with the number of samples $N$, and the sampling frequency $f_s$ changed to 99 samples at 1013 samples per second, Figure 6.9. Since

$$f_{analysis}[m] = mf_s/N,$$

changing either (or both) of these parameters affects the analysis frequencies used in the DFT. For the second figure, the analysis frequencies do not match up with the frequencies of the signal, so we see the spectral content spread across ALL frequencies. The signal has not changed, but our view of the frequency information

has. A trained individual would interpret the second figure as if it were the first figure, however.

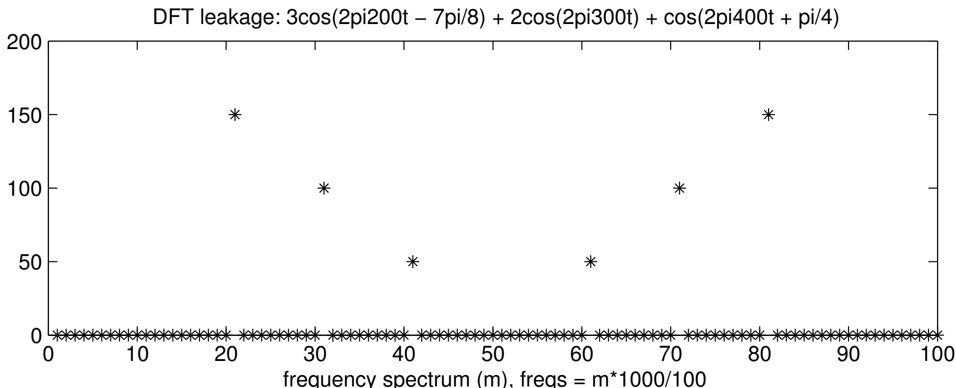



**Figure 6.8:** Frequency content appears at exact analysis frequencies.

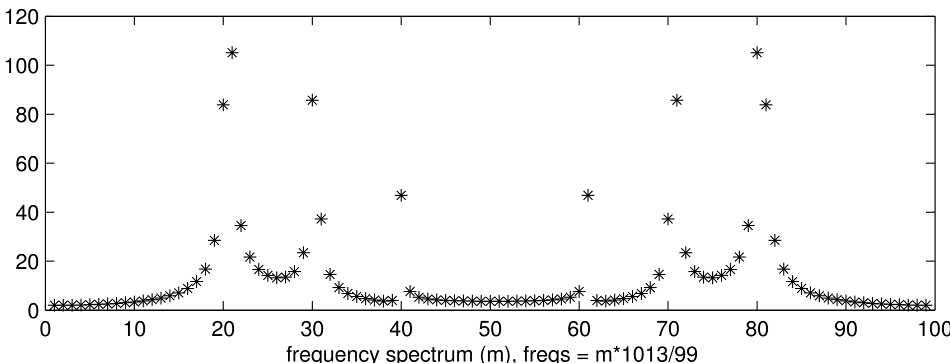



**Figure 6.9:** Frequency content appears spread out over analysis frequencies.

A *window* is a way of modifying the input signal so that there are no sudden jumps (discontinuities) in it. Even if you do not use a window function, when you sample a signal, you impose a rectangular window function on it [11]. In other words, the input is considered 0 for all values before you start sampling and all values after you stop. The sudden jumps (discontinuities) show up in the frequency response as the sinc function. Leakage occurs when the analysis frequencies do not

land on the actual frequencies present. This means that the actual information "leaks" into other DFT output bins; it shows up as other frequencies. Windowing reduces the sidelobes of the sinc function (of the Continuous Fourier Transform, or CFT for short), which in turn decreases the effect of leakage, since the DFT is a sampled version of the continuous Fourier transform [11].

In other words, looking at the CFT of a signal component (i.e., a single sinusoid), we see the sinc function. Looking at the CFT of a windowed version of the same signal, we see a sinc function with lower sidelobe levels (though a wider main lobe). If we sample the CFT, we might be lucky and sample it right at the point between sidelobes, so we have many zeros and a single spike. Realistically, our samples will land where we have one or two spikes, and several nonzero values, due to the sidelobes. The lower these sidelobes, the better our DFT indicates the frequencies that are actually present.

## 6.9 Harmonics and Fourier Transform

Harmonics and the Fourier transform are closely linked. Harmonics refers to the use of sinusoids related by a fundamental frequency $f_0$, that is, adding sinusoids of frequencies $f_0$, $2f_0$, $3f_0$, etc.

The following program demonstrates harmonics. Its objective is to approximate a triangle wave with a sum of sinusoids. When we run it, we see that the approximation gets better and better as more sinusoids are added. We use a fundamental frequency of 1 Hz. After creating signal $x$ as a triangle wave, the program finds the DFT of it using the `fft` command in MATLAB. Next, it finds the magnitudes and phases for all of the sinusoids corresponding to the DFT results. Finally, the program shows the sum of the sinusoids, pausing briefly between iterations of the loop to show the progress. Essentially, this performs the inverse DFT. Figure 6.10 shows what the progress looks like about one-sixth of the way through (the solid line is the original signal, and the sum of sinusoids is shown as a dash-dot line). Running the program to completion shows that the final approximation appears directly over top of the original signal.

```
%
%  Show how the DFT function can represent a triangle wave
%

% first, make a triangle wave
for i=1:20
    x(i) = i;
```

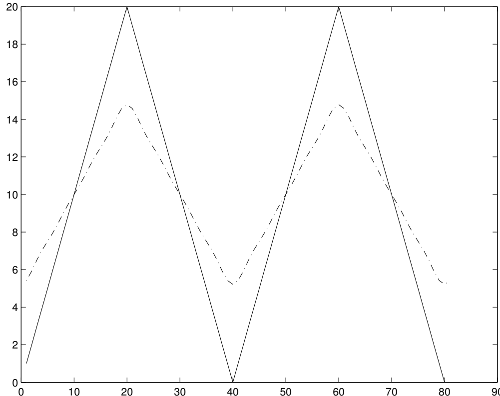



**Figure 6.10:** Approximating a triangle wave with sinusoids.

```
end
k=20;
for i=21:40
    k=k-1;
    x(i) = k;
end
for i=41:60
    x(i) = i-40;
end
k=20;
for i=61:80
    k=k-1;
    x(i) = k;
end
% OK, now we have the triangle wave as signal x

% Find the DFT of it, and scale the result.
% In other words, represent signal x as a sum of sinusoids
[y] = fft(x);
% Scale the result, so that it is the same size as original
y = y / length(x);
% Convert to polar coordinates (magnitudes and phases)
mag = abs(y);
phi = angle(y);
```

```
% Now, reconstruct it.
% This shows what the "sum of sinusoids" version looks like
t=0:(1/length(mag)):1;
f = 1;  % Our fundamental frequency is 1, since time t=n*m/N
a = 0;
% Show it, adding another sinusoid each time
for k=0:length(mag)-1
   a=a+mag(k+1)*(cos(2*pi*f*k*t+phi(k+1)) ...
     + j*sin(2*pi*f*k*t+phi(k+1)));
   plot(1:length(x), x, 'r', 1:length(a), real(a), 'b-.')
   pause(0.1);
end
```

The first thing this program does is create a signal to work with; a triangle wave. Below, we show how the program works with a square wave. Note the final `for` loop, enclosing `plot` and `pause` commands. We could use the "plotharmonic" function instead, but this program shows the reconstruction of the original signal as a step-by-step process.

Try the above program again, replacing the triangle wave above (program lines 5–22) with the following signals. It is a good idea to type `clear all` between them.

```
% Make a saw-tooth wave
for i=1:40
    x(i) = i;
end
for i=41:80
    x(i) = i-40;
end
% Now we have the saw-tooth wave as signal x
```

The progress of the approximation (one-sixth of the way) appears in Figure 6.11. This figure also shows the result when all terms are used. Notice that the biggest difference between the original and the approximation is the final point. The sinusoid approximation anticipates that the pattern will repeat.

```
% Make a square wave
for i=1:20
    x(i) = 0;
end
for i=21:40
```

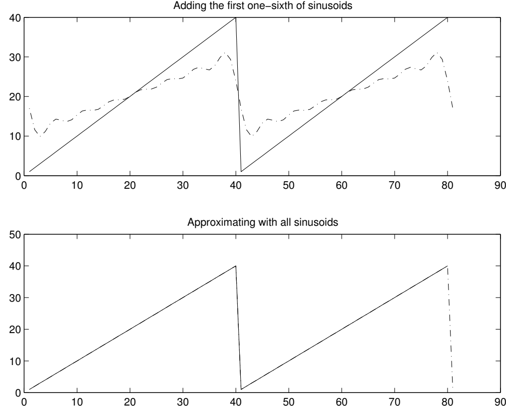



**Figure 6.11:** Approximating a saw-tooth wave with sinusoids.

```
    x(i) = 1;
end
for i=41:60
    x(i) = 0;
end
for i=61:80
    x(i) = 1;
end
% Now we have the square wave as signal x
```

Figure 6.12 shows the approximation of the previous square wave for one-sixth of sinusoids as well as all sinusoids.

```
% Make a combination saw-tooth square wave
for i=1:40
    x(i) = i;
end
for i=41:80
    x(i) = 40; %i-40;
end
% repeat
for i=81:120
    x(i) = i-80;
end
for i=121:160
```

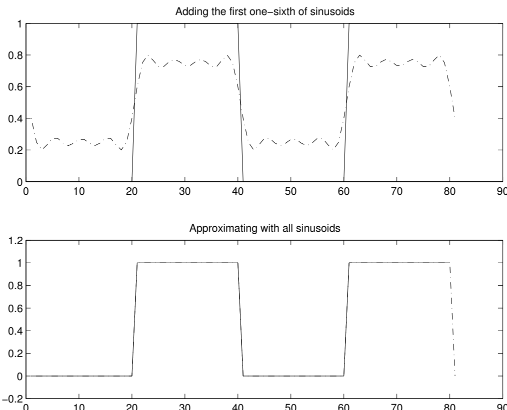



**Figure 6.12:** Approximating a square wave with sinusoids.

```
        x(i) = 40; %40;
    end
```

Figure 6.13 shows the approximation of a combined sawtooth-square wave for use of one-sixth of the terms as well as all terms.

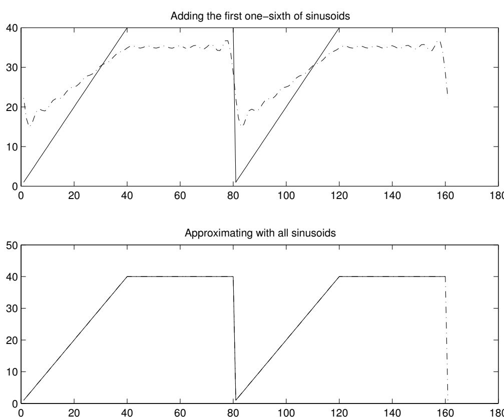



**Figure 6.13:** Approximating a saw-tooth square wave with sinusoids.

The DFT versions of the signals are good, but they are not perfect. Discontinuities in signals, such as the quick rise and quick fall of the square wave, are difficult to represent in the frequency-domain. The simple impulse function demonstrates this. It is very easy to represent in the time-domain:

$$x(t) = 1,\ t = 0$$

$$x(t) = 0,\ otherwise.$$

But to represent this in the frequency-domain, we need an infinite number of sinusoids.

Looking at this problem the other way, consider a simple sinusoid. In the frequency-domain, it is easily represented as a spike of half-amplitude at the positive and negative frequencies (recall the inverse Euler's formula). For all other frequencies, it is zero. If we want to represent this same signal in time, we have $amplitude \times \cos(2\pi ft + \phi)$, which gives us a function value for *any* value of time. Recording this signal's value for every value of time would require writing down an infinite number of terms.

This leads us to the observation that what is well-defined in the time-domain is poorly represented in the frequency-domain. Also, what is easy to represent in the frequency domain is difficult to represent in the time-domain. In effect, this is *Werner Heisenberg's uncertainty principle* [23].

## 6.10 Sampling Frequency and the Spectrum

As we saw with the DFT, when we sample a real signal at a rate of $f_s$ samples/second, the frequency magnitude plot from $f_s/2$ to $f_s$ looks like a mirror image of 0 to $f_s/2$.

A real signal is made up of components such as $\cos(\theta)$, or can be put in this form. If it were a complex signal, there would be a $j\sin(\theta)$ component, or something that can be put in that form (such as a $j\cos(\theta)$ component). Since $\cos(\theta) = \frac{e^{j\theta}}{2} + \frac{e^{-j\theta}}{2}$, a real signal always has a positive and negative frequency component on the spectrum plot.

Given $\cos(\theta) = \frac{e^{j\theta}}{2} + \frac{e^{-j\theta}}{2}$, and given Euler's law:

$$e^{j\theta} = \cos(\theta) + j\sin(\theta)$$

$$e^{j\theta} = \frac{e^{j\theta}}{2} + \frac{e^{-j\theta}}{2} + j\sin(\theta)$$

$$\frac{e^{j\theta}}{2} = \frac{e^{-j\theta}}{2} + j\sin(\theta)$$

$$j\sin(\theta) = \frac{e^{j\theta}}{2} - \frac{e^{-j\theta}}{2}.$$

This means that a complex signal has two frequency components as well. Of course, if we had a signal such as $\cos(\theta) + j\sin(\phi)$, we would have:

$$\cos(\theta) + j\sin(\phi) = \frac{e^{j\theta}}{2} + \frac{e^{-j\theta}}{2} + \frac{e^{j\phi}}{2} - \frac{e^{-j\phi}}{2}.$$

If $\theta$ equals $\phi$, then

$$\cos(\theta) + j\sin(\theta) = e^{j\theta}.$$

For this reason, we concern ourselves with only the first half of the frequency response. When it looks like Figure 6.14, we say it is a lowpass filter. This means that any low-frequency (slowly changing) components will remain after the signal is operated on by the system. High-frequency components will be attenuated, or "filtered out." When it looks like Figure 6.15, we say it is a highpass filter. This means that any high-frequency (quickly changing) components will remain after the signal is operated on by the system.

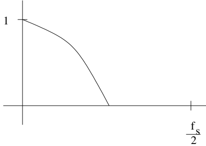



**Figure 6.14:** Frequency response of a lowpass filter.

The frequency response can be found by performing the DFT on a system's output when the impulse function is given as the input. This is also called the "impulse response" of the system. To get a smoother view of the frequency response, the impulse function can be padded with zeros.

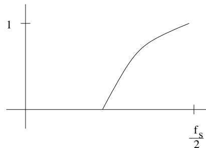



**Figure 6.15:** Frequency response of a highpass filter.

The more samples that are in the input function to the DFT, the better the resolution of the output will be.

---

**Exercise:**

$$w[n] = x[n] + x[n-1]$$
$$y[n] = x[n] - x[n-1]$$

Find and plot the frequency response for $w$ and $y$.

First, assume $x$ is a unit impulse function, and find $w[n]$ and $y[n]$. Pad these signals, say up to 128 values, so that the result will appear smooth. Next, find the DFT (or FFT) of $w$ and $y$, and plot the magnitudes of the result. The plots should go to only half of the magnitudes, since the other half will be a mirror image.

---

## 6.11 Summary

This chapter covers the Fourier transform, inverse Fourier transform, and related topics. The Fourier transform gives frequency (spectral) content of a signal. Given any series of samples, we can create a sum of sinusoids that approximate it, if not represent it exactly.

## 6.12 Review Questions

1. Suppose we were to sample a signal at $f_s = 5000$ samples per second, and we were to take 250 samples. After performing the DFT, we find that the first 10 results are as follows. What does this say about the frequencies that are present in our input signal? (Assume that the other values for $X[m]$ up to $m = 125$ are 0.)

$$X[m] = 10, 0, 0, 2 + j4, 0, 1, 0, 0, 0, 2 - j4$$

2. An input sequence, $x[n]$, has the Fourier transform performed on it. The result is:
   $X[m]$ = {3, 2+j4, 1, 5-j3, 0, 0, 0, 5+j3, 1, 2-j4}.

   a. Find (and plot) the magnitudes and phase angles.

   b. You should notice some symmetry in your answer for the first part. What

kind of symmetry do you expect (for the magnitudes), and why?

3. An input sequence, $x[n]$, has the Fourier transform performed on it. The result is:
   $X[m]$ = {3, 2+j4, 1, 5-j3, 0, 0, 0, 5+j3, 1, 2-j4}.
   Given that $x[n]$ was sampled at $f_s = 100$ samples per second,

   a. What is the DC component for this signal?
   b. What frequencies are present in the input? Rank them in order according to amplitude.
   c. Using MATLAB, find $x[n]$.

4. Given $x_2 = [0.4786, -1.0821, -0.5214, -0.5821, -0.2286, 1.3321, 0.7714, 0.8321]$; find (using MATLAB) the FFT values $X_{magnitude}[m]$ and $X_{phase}[m]$ for $m =$ 0..7. Show all of your work. Also, graph your results.

5. Try a 5-tap FIR filter using the following random data. Note that this command will return different values every time.

   ```
   x = round(rand(1, 20)*100)
   ```

   Use $h_1[k] = \{0.5, 0.5, 0.5, 0.5, 0.5\}$, and compare it to $h_2[k]$ = {0.1, 0.3, 0.5, 0.3, 0.1}, and $h_3[k] = \{0.9, 0.7, 0.5, 0.7, 0.9\}$. Graph the filter's output, and its frequency magnitude response. Be sure to use the same $x$ values. Make one graph with the filters' outputs, and another graph with the frequency responses. Judging each set of filter coefficients as a lowpass filter, which is best? Which is worst? Why?

6. Try an 8-tap FIR filter using the following random data. Note that this command will return different values every time.

   ```
   x = round(rand(1, 20)*100)
   ```

   Use $h_1[k]$ = {0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5}, and compare it to $h_2[k]$ = {0.1, 0.2, 0.3, 0.5, 0.5, 0.3, 0.2, 0.1}, and $h_3[k]$ = {0.9, 0.7, 0.6, 0.5, 0.5, 0.6, 0.7, 0.1}. Graph the filter's output and its frequency magnitude response. Be sure to use the same $x$ values. Make one graph with the filters' outputs, and another graph with the frequency responses. Judging each set of filter coefficients as a lowpass filter, which is best? Which is worst? Why?

7. What is $3e^{-j2\pi 0.2}$ in complex Cartesian coordinates? (That is, in the form $a + jb$.) Hint: use Euler's formula.

8. What is $1.7 - j3.2$ in complex polar coordinates? (That is, in the form $re^{j\phi}$.) Hint: use the magnitude and angle calculations from Chapter 1, "Introduction."

# Chapter 7

# The Number $e$

Euler's formula: so far we have used it like a transform to go from $e^{j()}$ to $\cos() + j\sin()$, or back. But why can we do this? Where does $e$ come from? What does $j$ really mean? These questions and more will be answered in this chapter. Along the way, we have a few important concepts about DSP, including how a complex number can be thought of as a vector on the complex plane (instead of a real number on the real number line), how such a vector can rotate, how we can go between alternate representations of the same information (such as polar coordinates versus Cartesian ones), and how some representations are more suitable for mathematical manipulations than others.

## 7.1 Reviewing Complex Numbers

Instead of using the Cartesian coordinate system to map a variable and a function, we can represent a complex number as a point on this plane. Here we have the x-axis represent the "real" part of a complex number, and the y-axis represent the "imaginary" part of it. Of course, we can also think of this same information in terms of polar coordinates; that is, we can represent it as a point $(x, y)$, or as a vector $r\angle\theta$ (length and angle), Figure 7.1.

The x-position of the vector is given by the function $x = r\ \cos(\theta)$, where $r$ is the radius (or length of the vector), and $\theta$ is the angle between the vector and the x-axis. The y-coordinate of the vector follows as $y = r\ \sin(\theta)$. These functions come from geometry, as Figure 7.2 shows. A 2D vector, with a vertical line segment $y$ units high, makes a right triangle with the x-axis.

Using the notation $j$, we can write the position of the vector as $r\ \cos(\theta) +$

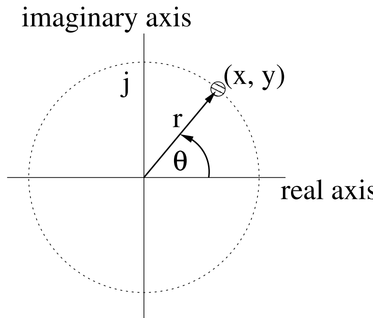



**Figure 7.1:** A complex number can be shown as a point or a 2D vector.

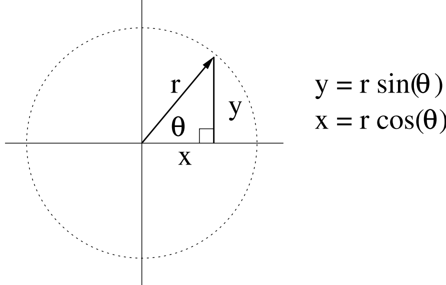



**Figure 7.2:** A vector forms a right triangle with the x-axis.

$j\ r\ \sin(\theta)$. Recall Euler's formula,

$$e^{j\theta} = \cos(\theta) + j \sin(\theta)$$

or, multiplying by a constant,

$$r\ \cos(\theta) + j\ r\ \sin(\theta) = r\ e^{j\theta}$$

so we can substitute $r\ e^{j\theta}$ to represent this vector (also called a phasor).

Imagine that a 2D vector starts on the x-axis, and rotates counterclockwise around the origin (see Figure 7.3). As it rotates, the length of the vector stays the same, but the angle that it makes with the positive x-axis always increases. Eventually, the vector will have traveled completely around and momentarily line up with the positive x-axis again, before repeating its journey around the origin. This is much like the movement of the secondhand around the face of a watch, except in the opposite direction. This rotating vector is also called a rotating phasor. On the Cartesian coordinate system, we can model a rotating vector in terms of sine and cosine functions. As in Figure 7.2, the x-position of the vector is given by the function $x = r\ \cos(\theta)$, with the value of $\theta$ always increasing with time. The y-coordinate of the vector follows as $y = r\ \sin(\theta)$.

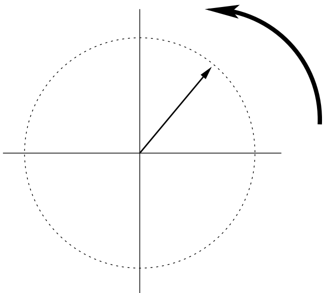


**Figure 7.3:** A rotating vector.

A simple sinusoid has the form $a \times \cos(2\pi ft + \phi)$. Using Euler's formula, this can be represented as $Real\{a \times e^{j(2\pi ft+\phi)}\}$. This form is interesting, because it makes some operations easier. For example, it can be rewritten as $Real\{a \times e^{j\phi} e^{j2\pi ft}\}$. This means that you can think of it as the real part of the result of the phasor $a \times e^{j\phi}$, multiplied by the rotating phasor $e^{j2\pi ft}$ of unit length.

A better way to represent this is with Euler's inverse formula.

$$a \times \cos(2\pi ft + \phi) = \frac{a}{2} e^{j(2\pi ft+\phi)} + \frac{a}{2} e^{-j(2\pi ft+\phi)}$$

Here we have a real cosine breaking down into two complex, rotating vectors. As we will soon see, the complex parts cancel each other out. Again, the vector notation helps us mathematically manipulate these signals.

A few other concepts from math that are good to remember are as follows:

$$e^{a+b} = e^a e^b$$

$$e^{a-b} = e^a e^{-b}.$$

If $e^a = e^b$, then $a = b$.

Next, let's review some properties of $j$.

## 7.2 Some Interesting Properties of $j$

Angular notation represents complex numbers (or polar numbers), such as:

$$\sqrt{-1} = 1\angle 90^\circ.$$

Note that we would get these same numbers if we found the magnitude and angle of the Cartesian vector $0 + 1j$. Angular notation allows us to specify a vector on the complex plane. Multiplying by $j$, or by $-j$, can be interpreted as a rotation of such a vector on the complex plane. This section demonstrates that property.

### 7.2.1 Rotating Counterclockwise

Multiplying by $j$ can be thought of as a counterclockwise rotation of 90 degrees [24]. If we start with a real number, say 1, and multiply it by $j$, we end up with $j$.

$$j = \sqrt{-1}$$

Next, we could multiply this by $j$ again:

$$j^2 = -1.$$

And again:

$$j^3 = -\sqrt{-1}.$$

And again:

$$j^4 = 1.$$

So we end up with the same number that we started with. If we plotted this on the complex plane, we would see that every multiplication by $j$ is equivalent to rotating

the point counterclockwise by 90 degrees.

### 7.2.2 Rotating Clockwise

We can also find an equivalent version of $j^{-1}$ with the following:

$$j^2 = -1.$$

Dividing through by $j$:

$$j = \frac{-1}{j}.$$

Switching sides and multiplying through by $-1$:

$$\frac{1}{j} = -j$$

$$j^{-1} = -j.$$

In fact, multiplying by $j^{-1}$, or $-j$, is like rotation in the clockwise direction. We can show this holds for the general case with the complex number $a + jb$.

$$\begin{aligned}
(a + jb) \times (-j) &= -aj - j^2 b \\
&= b - ja \\
(b - ja) \times (-j) &= j^2 a - jb \\
&= -a - jb \\
(-a - jb) \times (-j) &= j^2 b + ja \\
&= -b + ja \\
(-b + ja) \times (-j) &= bj - j^2 a \\
&= a + jb
\end{aligned}$$

If we plotted the sequence above, assuming that $a$ and $b$ are both positive, we would see the vector made from the origin to the point on the complex plane rotate in the clockwise direction. Figure 7.4 shows this. We can clearly see that the point $a + jb$ rotates around the origin, at $-\pi/2$ increments. The numbers next to the points show the sequence; if we were to multiply the point by $j$, we would see the order reversed.

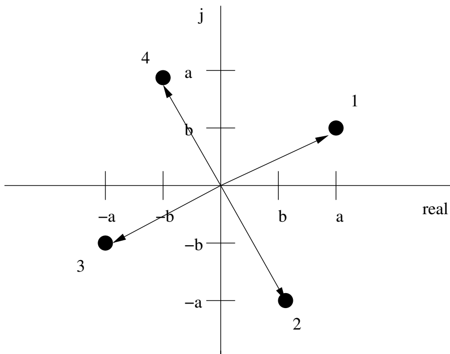



**Figure 7.4:** A vector rotates clockwise by $-\pi/2$ when multiplied by $-j$.

### 7.2.3 Removing $j$ from $\sqrt{-a}$

Just as the quadratic formula uses the $\pm$ signs to indicate two possibilities for roots, we must use care when removing $j$ from under the radical. Assume $a$ is some positive real value. When dealing with $\sqrt{-a}$, we might be tempted to rewrite this as $j\sqrt{a}$. And we might be correct, but there is a second possibility to consider: $-j\sqrt{a}$ would also work. That is, $(\pm j\sqrt{a})^2 = -a$, therefore

$$\sqrt{-a} = \pm j\sqrt{a}.$$

## 7.3 Where Does $e$ Come from?

Where does $e$ come from? First, $e$ is an irregular number, $e = 2.71828....$ It comes from the idea of interest [25]. Suppose you lend two people the same amount of money, at a fixed interest rate. If the first person pays you back the loan and interest within two weeks, while the second person waits one month before repaying, you might think that the second person should pay additional interest. After all, if you had the money back earlier, perhaps you could have lent the money to another person, at the same interest rate. So the next time you lend someone money, you could tell him that you expect the repayment within a month, but that you will compute the interest every day. Every time you calculate the amount to be repaid, the interest applies not only to the amount of the original loan, but also to the interest that the money had "earned" by that point. Thus, compound interest is born.

Simple interest, in contrast, works as follows. Suppose we lend \$1000, with a specified maturity date. When that date comes, the borrower will pay us back \$1000

plus interest. If the borrower decides not to pay back, we wait another time interval with the understanding that the borrower will pay us back $1000 plus 2× interest. If this pattern continues, eventually the borrower will owe us double the original amount, i.e., $2000. But the interest rate never changes. It costs the borrower just as much to delay paying us the $2000 as it does the original $1000. That is, at an interest rate of 5%, the borrower would owe a total of $1000 + $50 in interest after the first time period. But after 20 time periods, the borrower would owe $1000 + $950 in interest up to this point, plus $50 interest for the 20th time period. Shouldn't we charge him more for tying up so much of our money?

A better agreement (for us, the lender) would be to charge interest not only on the original amount borrowed, but on the balance of money owed to that point, leading us to compound interest. With the example of $1000 at 5% interest, the borrower would owe $1000 + $1000×0.05 = $1050 for the first time period, but then $1050 + $1050×0.05 = $1102.50 for the second time period. The amount owed at the end of a time period is (*amount owed at the end of the last time period*)×(1 + *interest rate*). Carrying this to an extreme, where the time between interest calculations gets arbitrarily small, we arrive at the following equation, where $x$ is the interest ($x = 1$ means that we double our money), and $n$ is the number of iterations. For example, with $x = 1$ we expect to get 100% simple interest in the time allotted. For complex interest, we break down the time to repay into $n$ segments, meaning we reevaluate how much the borrower owes $n$ times within the time allotted. With $x = 1$ and $n = 10$, the borrower repays $200 for simple interest and $259.37 for compound interest.

$$e^x = \lim_{n\to\infty}(1 + \frac{x}{n})^n$$

— OR —

$$e^x = (1 + \tfrac{1}{n})^{nx}, \text{ as } n \to \infty$$

The program below demonstrates how $e$ comes from compound interest. Suppose we lend $1000 at an interest rate of 100%, doubling our money with simple interest. With compound interest, we choose how often to calculate the interest, and the more often we do this, the closer to $e \times$ *interest* we get. Table 7.1 was generated using the program below. As we can see from the table, as $n$ gets larger, the amount to repay gets closer to $1000×$e$ (the value of $e \approx 2.71828$). Note that $n$ getting larger does not mean that we lend the money for a longer period, only that there is a smaller time between recalculating interest.

The point is that as $(1/n)$ approaches 0, the value of the compounded money will be $e \times$ *principal*, instead of $2 \times$ *principal*. See Figure 7.5 for a graph of simple

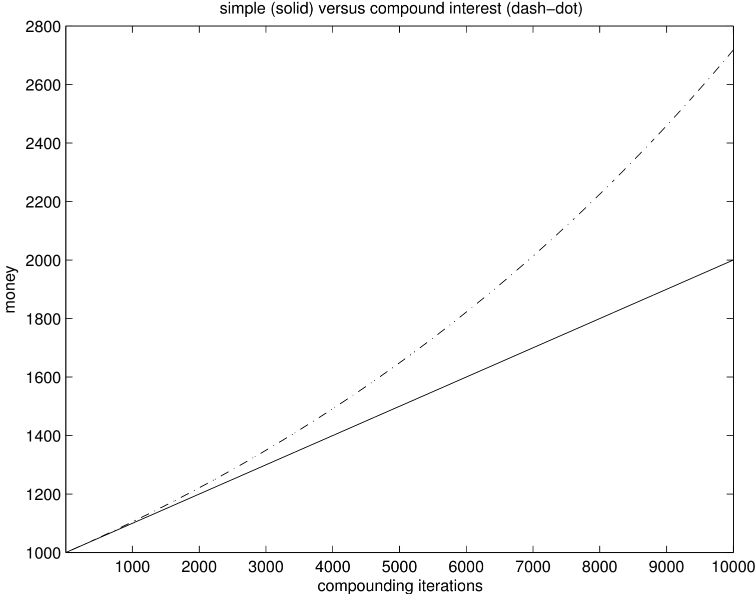



**Figure 7.5:** Simple versus compounded interest.

**Table 7.1:** Compound interest on \$1000 approximates \$1000 $\times e$.

| n | Amount to repay |
|---|---|
| 10 | 2593.74 |
| 100 | 2704.81 |
| 1000 | 2716.92 |
| 10,000 | 2718.15 |
| 100,000 | 2718.27 |
| 250,000 | 2718.28 |
| 500,000 | 2718.28 |

interest (solid line) versus compound interest (the dash-dot line). In other words, $e$ is the limit of the money compounded continuously. Now, we can answer the question as to how Euler's formula works.

```
% interest.m
% Simple interest versus compound interest
% -> e
%

x = 1; % Set x = 1 to find e
n = 100;
principal = 1000;
disp('This program shows how e comes from compound interest.');
disp(sprintf('Lending %6.2f at %d percent interest. ', ...
    principal, round(x*100)));
simple = principal;
compound = principal;
% Calculate the simple interest
simple = simple + principal*x;
% Calculate the compound interest
for i=1:n
    compound = compound*(1+ x/n); % + my_interest;
end
disp('Amount to repay:');
disp(sprintf(' with simple interest = %6.2f', simple));
disp(sprintf(' with compound interest = %6.2f', compound));
disp(sprintf('    compounded %d times.', n));
```

Also included on the CD-ROM is a different version of this program, called `interest2.m`, that produces the graph shown in Figure 7.5.

## 7.4 Euler's Formula

Now we are ready to see where Euler's formula comes from. To find how Euler's formula works, first we need the binomial theorem [25].

$$e^x = 1 + x + x^2/2! + x^3/3! + x^4/4! + \ldots$$

First, we will replace $x$ with $j\theta$,

$$e^{j\theta} = 1 + j\theta + (j\theta)^2/2! + (j\theta)^3/3! + (j\theta)^4/4! + \ldots$$

Now simplify:

$$e^{j\theta} = 1 + j\theta - (\theta)^2/2! - j\theta^3/3! + \theta^4/4! + \ldots$$

Now split the results into the real and imaginary parts:

$$e^{j\theta} = (1 - (\theta)^2/2! + \theta^4/4! + \ldots) + j(\theta - \theta^3/3! + \theta^5/5! + \ldots). \tag{7.1}$$

Recall Maclaurin's formula [26]:

$$f(x) = f(0) + \frac{f'(0)}{1!}x + \frac{f''(0)}{2!}x^2 + \ldots + \frac{f^n(0)}{n!}x^n + \frac{f^{n+1}(0)}{(n+1)!}x^{n+1}. \tag{7.2}$$

Replacing $f(x) = \sin(x)$, we obtain:

$$\sin(x) = \sin(0) + \frac{f'(0)}{1!}x + \frac{f''(0)}{2!}x^2 + \ldots + \frac{f^n(0)}{n!}x^n + \frac{f^{n+1}(y)}{(n+1)!}x^{n+1}$$

where $y > 0$ and $y < x$. The derivatives of the sine function are as follows [26].

$$\begin{aligned}
f(x) &= \sin(x) \\
f'(x) &= \cos(x) \\
f''(x) &= -\sin(x) \\
f'''(x) &= -\cos(x) \\
f''''(x) &= \sin(x), \quad \text{and the above pattern repeats.}
\end{aligned}$$

$$\sin(x) = \sin(0) + \frac{\cos(0)}{1!}x - \frac{\sin(0)}{2!}x^2 + \ldots + \frac{f^n(0)}{n!}x^n + \frac{f^{n+1}(y)}{(n+1)!}x^{n+1}$$

Simplifying and substituting $\theta$ for $x$:

$$\sin(\theta) = \theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} + \ldots \tag{7.3}$$

The cosine function can be broken down in a similar manner.

$$\begin{aligned} f(x) &= \cos(x) \\ f'(x) &= -\sin(x) \\ f''(x) &= -\cos(x) \\ f'''(x) &= \sin(x) \\ f''''(x) &= \cos(x), \quad \text{and the above pattern repeats.} \end{aligned}$$

Replacing $f(x)$ of equation 7.2 with $\cos(x)$:

$$\cos(x) = \cos(0) - \frac{\sin(0)}{1!}x - \frac{\cos(0)}{2!}x^2 + \ldots + \frac{f^n(0)}{n!}x^n + \frac{f^{n+1}(y)}{(n+1)!}x^{n+1}.$$

Simplifying and substituting $\theta$ for $x$:

$$\cos(\theta) = 1 - \frac{\theta^2}{2!} + \frac{\theta^4}{4!} + \ldots \qquad (7.4)$$

Now repeat equation 7.1, but notice how the real parts correspond to the $\cos(\theta)$ function (equation 7.4), and that the imaginary parts correspond to the $\sin(\theta)$ function (equation 7.3).

$$e^{j\theta} = (1 - (\theta)^2/2! + \theta^4/4! + \ldots) + j(\theta - \theta^3/3! + \theta^5/5! + \ldots)$$

This means that this equation can be simplified to:

$$e^{j\theta} = \cos(\theta) + j\sin(\theta)$$

which, of course, is Euler's formula.

## 7.5 Alternate Form of Euler's Equation

As we saw previously, Euler's equation is

$$e^{j\theta} = \cos(\theta) + j\sin(\theta).$$

What if $\theta$ happens to be a negative number? Let $\theta = -\phi$, then replace $\theta$ with $-\phi$ in the above equation. Keep in mind that it does not really matter what symbol we use, as long as we are consistent. On the righthand side, we can replace the sine and cosine functions with a couple of trigonometric identities, that is, $\cos(-\phi) = \cos(\phi)$, and $\sin(-\phi) = -\sin(\phi)$. Therefore, Euler's equation also works for a negative

exponent, i.e.,

$$e^{j(-\phi)} = \cos(-\phi) + j\sin(-\phi)$$
$$e^{-j\phi} = \cos(\phi) - j\sin(\phi).$$

## 7.6 Euler's Inverse Formula

Euler's inverse formula is given as:

$$\cos(\theta) = \frac{1}{2}e^{j\theta} + \frac{1}{2}e^{-j\theta}.$$

That is, a sinusoid can be replaced by a phasor of half amplitude, plus a similar phasor with a negated angle. We call this phasor its *complex conjugate*, and denote the complex conjugate of $z$ with the symbol $z^*$. These phasors are always mirror images of each other, with respect to the x-axis. Figure 7.6 shows this for three different example angles, $\theta$ (in the first quadrant), $\phi$ (in the second quadrant), and $\omega$ (in the third quadrant). The phasor and its complex conjugate are on opposite sides of the "real" x-axis, but the same side of the "imaginary" y-axis, no matter what the angle is. If the phasor rotates, then so does its complex conjugate, but they rotate in opposite directions.

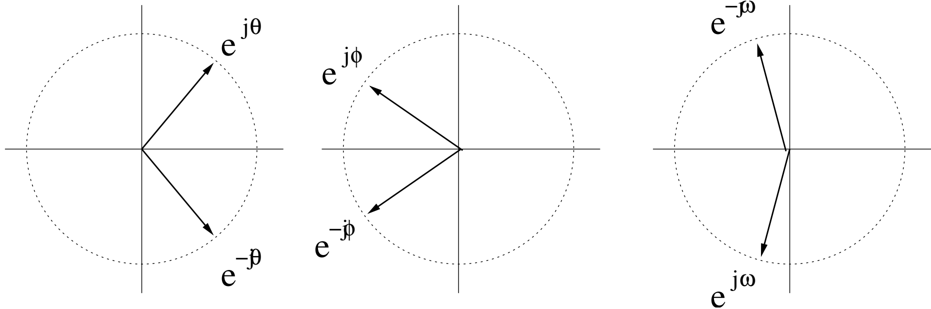



**Figure 7.6:** Phasors and their complex conjugates.

Think of the cosine function as the real part of a complex vector, possibly even one that rotates. The cosine function can also be thought of as the result of adding two vectors that are rotating in opposite directions. This is because adding them together cancels out the imaginary parts. We can verify this formula with Euler's formula, as follows.

$$\frac{1}{2}e^{j\phi} + \frac{1}{2}e^{-j\phi} = \frac{1}{2}(\cos(\phi) + j\sin(\phi)) + \frac{1}{2}(\cos(-\phi) + j\sin(-\phi))$$

$$= \frac{1}{2}\cos(\phi) + \frac{1}{2}j\sin(\phi) + \frac{1}{2}\cos(\phi) - \frac{1}{2}j\sin(\phi)$$

$$= \cos(\phi)$$

From the above analysis, we get the inverse Euler's formula:

$$\cos(\theta) = \frac{1}{2}e^{j\theta} + \frac{1}{2}e^{-j\theta}.$$

---

**Example:**
$x(t) = 3e^{j\pi/6}e^{j2\pi 1000t}$, graph $x(0)$. What significance does this have?

**Answer:**
When $t = 0$, the term $e^{j2\pi ft}$ , becomes 1, so we are left with $3e^{j\pi/6}$, a (nonrotating) vector. The vector's length is given by the amplitude, in this case 3 units. The $\pi/6$ term indicates that it lies in the first quadrant. The significance of showing the vector at time 0 is that it reveals the offset given by the phase shift [6]. Note that the frequency of 1000 Hz does not play a role in this question. See Figure 7.7 for a graph of this example. Notice how easy it is to get the length and phase angle from this representation.

---

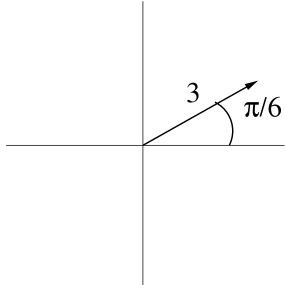



**Figure 7.7:** Graph of $x(0)$ where $x(t) = 3e^{j\pi/6}e^{j2\pi 1000t}$.

The frequency $f$ controls how fast the vector rotates. In fact, if the frequency is negative, the vector will rotate in the *opposite* direction (clockwise). You can verify

this by examining the $x$ and $y$ values over small increments of time for a positive frequency (e.g., 10 Hz) and for a negative frequency (e.g., $-10$ Hz). The code below shows this.

```
f=10;
t=0:0.01:1;
x1 = cos(2*pi*f*t+pi/4);
x2 = cos(2*pi*(-f)*t+pi/4);
plot(1:length(x1),x1,'r',1:length(x2),x2,'b')
```

The term $e^{j\theta}$ usually gives us a complex quantity, but under some conditions, it reduces to a real number only. One example is where $\theta = \pi$. Here we calculate $e^{\pm j\pi}$, starting with the two forms of Euler's formula.

$$\begin{aligned} e^{+j\theta} &= \cos(\theta) + j\sin(\theta) & e^{-j\theta} &= \cos(\theta) - j\sin(\theta) \\ e^{+j\pi} &= \cos(\pi) + j\sin(\pi) & e^{-j\pi} &= \cos(\pi) - j\sin(\pi) \\ \cos(\pi) &= -1 & \sin(\pi) &= 0 \\ e^{+j\pi} &= -1 + j0 & e^{-j\pi} &= -1 - j0 \end{aligned}$$

$$e^{\pm j\pi} = -1$$

As we see above, when the angle $\theta = \pm\pi$, the expression $e^{j\theta}$ reduces to a value on the real axis. This also works for other angle values, namely $\theta = 0$.

## 7.7 Manipulating Vectors

Vectors can be added together. In fact, introductory physics classes usually show this graphically, by moving one 2D vector until its tail connects to the head of the other. The resulting vector can then be drawn from the origin to the head of the vector that moved.

### 7.7.1 Adding Two Vectors

Adding 2D vectors cannot normally be done in vector form, but it is easy in Cartesian form. For example, how would one add $3\angle\pi/6 + 4\angle 5\pi/18$? That is, let $z_1 = 3\angle\pi/6$, $z_2 = 4\angle 5\pi/18$, and $z_3 = z_1 + z_2$.

Converting $z_1$ to Cartesian form:

$$x_1 = 3\cos(\pi/6),\ y_1 = 3\sin(\pi/6)$$

$$x_1 = 2.60,\ y_1 = 1.5.$$

Converting $z_2$ to Cartesian form:

$$x_2 = 4\cos(5\pi/18),\ y_2 = 4\sin(5\pi/18)$$

$$x_2 = 2.57,\ y_2 = 3.06.$$

Adding them:

$$x_3 = x_1 + x_2 = 2.60 + 2.57 = 5.17$$

$$y_3 = y_1 + y_2 = 1.5 + 3.06 = 4.56.$$

Converting from Cartesian form to polar form:

$$r_3 = \sqrt{5.17^2 + 4.56^2} = \sqrt{47.5} = 6.89$$

$$\theta_3 = \arctan(4.56/5.17) = 0.723 \text{ radians}.$$

Therefore, $z_3 = 6.89\angle 0.723$ radians. Of course, rounding will introduce some imprecision.

### 7.7.2 Adding Vectors in General

Just as we can add two 2D vectors of the form $r\angle\phi$, we can add two vectors of the form $re^{j\theta}$.

$$r_1 e^{j\theta_1} + r_2 e^{j\theta_2}$$

Convert to sinusoids with Euler's formula,

$$= r_1\cos(\theta_1) + jr_1\sin(\theta_1)$$

$$+ r_2\cos(\theta_2) + jr_2\sin(\theta_2)$$

$$= (r_1\cos(\theta_1) + r_2\cos(\theta_2)) + j(r_1\sin(\theta_1) + r_2\sin(\theta_2)).$$

To keep this general, we will let $a = (r_1\cos(\theta_1) + r_2\cos(\theta_2))$ and $b = (r_1\sin(\theta_1) + r_2\sin(\theta_2))$, so that the above result becomes $a + jb$. Now we can convert from this complex number to the complex polar form,

$$r_3 = \sqrt{a^2 + b^2}$$

and

$$\theta_3 = \arctan(b/a).$$

Keep in mind the adjustments of $\pm\pi$ if $a$ happens to be negative.

As long as we have $a$ and $b$ computed as above, we can find the resulting $r_3$ and $\theta_3$.

$$r_3 e^{j\theta_3} = r_1 e^{j\theta_1} + r_2 e^{j\theta_2}$$

### 7.7.3 Adding Rotating Phasors

This also works if they are rotating phasors, assuming that they have the same frequency.

$$a_1 e^{j\phi_1} e^{j2\pi ft} + a_2 e^{j\phi_2} e^{j2\pi ft}$$

$$= (a_1 e^{j\phi_1} + a_2 e^{j\phi_2}) e^{j2\pi ft}$$

$$= a_3 e^{j\phi_3} e^{j2\pi ft}$$

where $a_3$ and $\phi_3$ can be found as above (for the nonrotating phasors).

### 7.7.4 Adding Sinusoids of the Same Frequency

In a manner similar to that used for adding rotating phasors, you can also add two sinusoids when the frequencies are the same.

For example,

$$a_1 \cos(2\pi ft + \phi_1) + a_2 \cos(2\pi ft + \phi_2)$$

$$= a_3 \cos(2\pi ft + \phi_3).$$

Section 7.8 shows how to obtain the values for $a_3$ and $\phi_3$.

### 7.7.5 Multiplying Complex Numbers

Figure 7.8 shows two complex numbers, $z_1$ and $z_2$, plotted as vectors.

Adding them together (shown in Figure 7.9):

$$z_3 = z_1 + z_2$$

$$= 5 + 3j + 6 + 2j$$

$$= 11 + 5j.$$

Multiplying them together:

$$z_4 = z_1 z_2 = (5 + 3j)(6 + 2j)$$

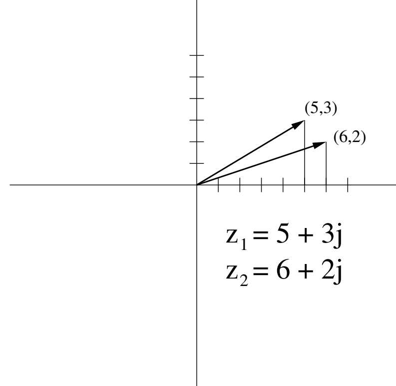



**Figure 7.8:** Two example vectors.

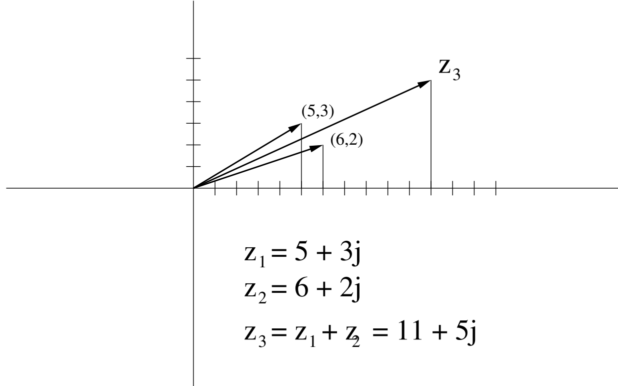



**Figure 7.9:** Adding two example vectors.

$$= 30 + 10j + 18j + 6j^2$$
$$= 30 + 28j - 6$$
$$= 24 + 28j.$$

These can also be multiplied in polar form:

$$z_1 = r_1 \angle \theta_1$$
$$r_1 = \sqrt{5^2 + 3^2} = \sqrt{34}$$
$$\theta_1 = \arctan(3/5)$$
$$z_2 = r_2 \angle \theta_2$$
$$r_2 = \sqrt{6^2 + 2^2} = \sqrt{40}$$
$$\theta_2 = \arctan(2/6)$$
$$z_4 = r_1 r_2 \angle \theta_1 + \theta_2.$$

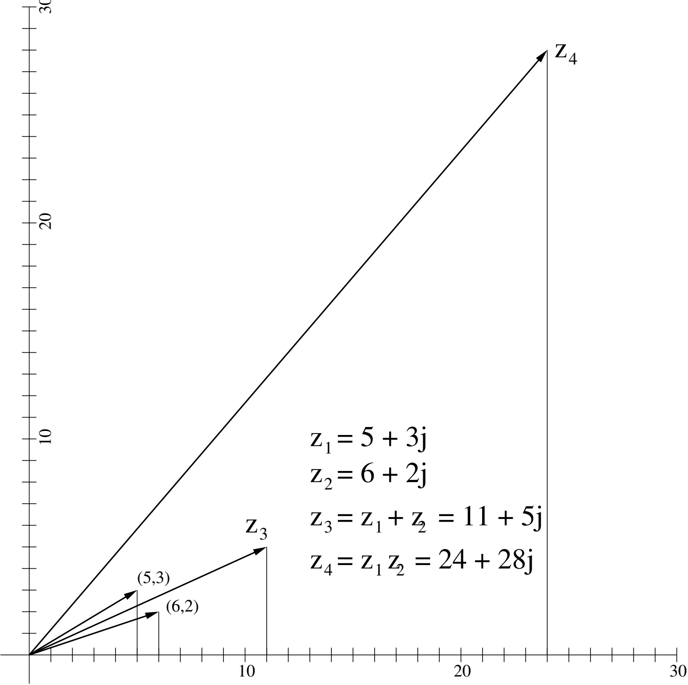



**Figure 7.10:** Adding and multiplying two example vectors.

Figure 7.10 shows the multiplication result for the two example vectors, with the addition result, too.

## 7.8 Adding Rotating Phasors: an Example

You can add two (or more) rotating phasors (or sinusoids), if they have the same frequency. But if you add two together, what do the results look like?

Here is an example:

$$x_1(t) = 2\cos(2\pi 100t + \pi/4)$$

$$x_2(t) = 3\cos(2\pi 100t + \pi/8)$$

$$x_3(t) = x_1(t) + x_2(t).$$

Using Euler's inverse formula,

$$\cos(\theta) = \frac{1}{2}e^{j\theta} + \frac{1}{2}e^{-j\theta}$$

$$x_1(t) = 2\frac{1}{2}e^{j(2\pi 100t+\pi/4)} + 2\frac{1}{2}e^{-j(2\pi 100t+\pi/4)}$$

$$x_2(t) = 3\frac{1}{2}e^{j(2\pi 100t+\pi/8)} + 3\frac{1}{2}e^{-j(2\pi 100t+\pi/8)}$$

$$x_1(t) = e^{j2\pi 100t}e^{j\pi/4} + e^{-j2\pi 100t}e^{-j\pi/4}$$

$$x_2(t) = \frac{3}{2}e^{j2\pi 100t}e^{j\pi/8} + \frac{3}{2}e^{-j2\pi 100t}e^{-j\pi/8}$$

$$x_3(t) = e^{j2\pi 100t}\left(e^{j\pi/4} + \frac{3}{2}e^{j\pi/8}\right) + e^{-j2\pi 100t}\left(e^{-j\pi/4} + \frac{3}{2}e^{-j\pi/8}\right).$$

To progress, we have to convert these vectors to Cartesian coordinates. Rather than convert to Cartesian coordinates $(x, y)$ explicitly, it is easier to use Euler's formula, with the understanding that $x$ equals the real part, and $y$ equals the imaginary part. In effect, we use $j$ just to keep the $x$ and $y$ values straight.

$$e^{j\pi/4} = \cos(\pi/4) + j\ \sin(\pi/4)$$

$$\frac{3}{2}e^{j\pi/8} = \frac{3}{2}\cos(\pi/8) + j\frac{3}{2}\sin(\pi/8)$$

$$e^{j\pi/4} + \frac{3}{2}e^{j\pi/8} = \cos(\pi/4) + \frac{3}{2}\cos(\pi/8) + j(\sin(\pi/4) + \frac{3}{2}\sin(\pi/8))$$

$$\approx 2.093 + j1.296$$

$$\theta \approx \arctan\left(\frac{1.296}{2.093}\right) \approx 32^\circ\left(\frac{2\pi}{360}\right) = \frac{8}{45}\pi$$

$$r \approx \sqrt{2.093^2 + 1.296^2} = \sqrt{6.060265} \approx 2.46$$

$$x_3(t) \approx 2.46e^{j8\pi/45}e^{j2\pi 100t} + 2.46e^{-j8\pi/45}e^{-j2\pi 100t}$$

$$x_3(t) \approx 2.46e^{j(2\pi 100t + 8\pi/45)} + 2.46e^{-j(2\pi 100t + 8\pi/45)}.$$

With Euler's formula,

$$x_3(t) \approx 2.46\cos(2\pi 100t + 8\pi/45) + j2.46\sin(2\pi 100t + 8\pi/45)$$

$$2.46\cos(-(2\pi 100t + 8\pi/45)) + j2.46\sin(-(2\pi 100t + 8\pi/45)).$$

The $j\sin()$ terms cancel each other out, since $\sin(-\phi) = -\sin(\phi)$. Since $\cos(-\phi) = \cos(\phi)$, the cosine terms simply add together.

$$x_3(t) \approx 4.92\ \cos(2\pi 100t + 8\pi/45)$$

We use MATLAB to validate these results. In Figure 7.11, two sinusoids are plotted in the upper part of the graph, corresponding to the previous example, $x_1(t) = 2\cos(2\pi 100t + \pi/4)$ plotted as a solid line with dots on it, and $x_2(t) = 3\cos(2\pi 100t + \pi/8)$, plotted as the solid line. The lower part of the graph also shows two sinusoids, though they happen to be right on top of one another. The

first one, plotted with a solid line, was found with the command `x3 = x1 + x2;`. That is, it is a point-for-point addition of signals $x_1$ and $x_2$. The second sinusoid is $4.92\cos(2\pi 100t + 8\pi/45)$, which is the analytical result obtained above. The program below was used to generate this graph.

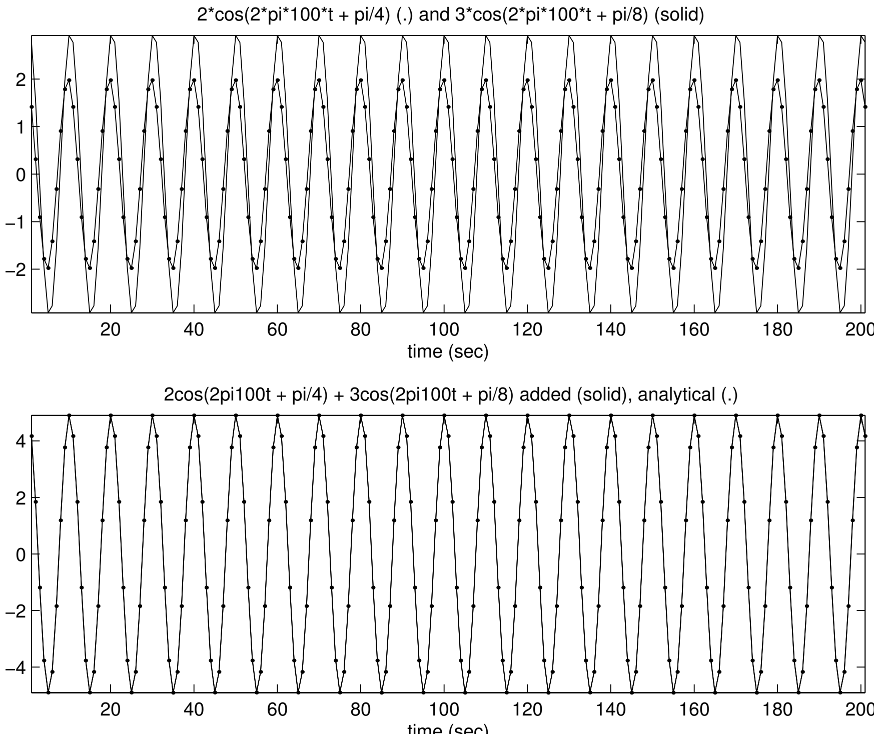



**Figure 7.11:** Two sinusoids of the same frequency added point-for-point and analytically.

```
%
% adding_sins.m
% Show 2 sinusoids of same frequency, and how they look combined.
%

t = 0:0.001:0.2;
x1 = 2*cos(2*pi*100*t + pi/4);
x2 = 3*cos(2*pi*100*t + pi/8);
x3 = x1 + x2;
```

```
x4 = 4.92*cos(2*pi*100*t + 8*pi/45);

subplot(2,1,1);
plot(t, x1, 'b.-', t, x2, 'k');
mystr = '2*cos(2*pi*100*t + pi/4) (.) and';
mystr = strcat(mystr,' 3*cos(2*pi*100*t + pi/8) (solid)');
title(mystr);
xlabel('time (sec)');
axis tight;

subplot(2,1,2);
plot(t, x3, 'b', t, x4, 'r.-');
mystr = '2cos(2pi100t + pi/4) + 3cos(2pi100t + pi/8) added';
title(strcat(mystr,' (solid), analytical (.)'));
xlabel('time (sec)');
axis tight;
```

The addition of two phasors can be seen graphically as in Figure 7.12. Here we take phasor $e^{j2\pi ft}a_1e^{j\phi_1}$ and add it to phasor $e^{j2\pi ft}a_2e^{j\phi_2}$. Since the rotating phasors are the same, we separate them and add the rest, just like we can rewrite $ax+bx$ as $x(a+b)$. This works if the nonrotating vectors have different lengths. For the addition of the nonrotating phasors (how we go from the middle of the figure to the bottom), refer back to Figure 7.9.

$$e^{j2\pi ft}a_3e^{j\phi_3} = e^{j2\pi ft}(a_1e^{j\phi_1} + a_2e^{j\phi_2})$$

Here, two rotating phasors are shown as their rotating part, multiplied by a phasor with the amplitude and phase angle. If they are rotating *at the same frequency*, then the rotating phasor parts are exactly the same, and the amplitude and phase angle parts can be added together.

Now let's take this a step further. Instead of just adding two phasors together, we will add two sinusoids together (Figure 7.13). Remember that Euler's inverse formula gives us two phasors of half amplitude for one cosine term, that is:

$$a \times \cos(2\pi ft + \phi) = \frac{ae^{j(2\pi ft+\phi)}}{2} + \frac{ae^{-j(2\pi ft+\phi)}}{2}$$

$$a \times \cos(2\pi ft + \phi) = \frac{a}{2}e^{j2\pi ft}e^{j\phi} + \frac{a}{2}e^{-j2\pi ft}e^{-j\phi}.$$

In this figure, we take two typical sinusoid components (of the same frequency)

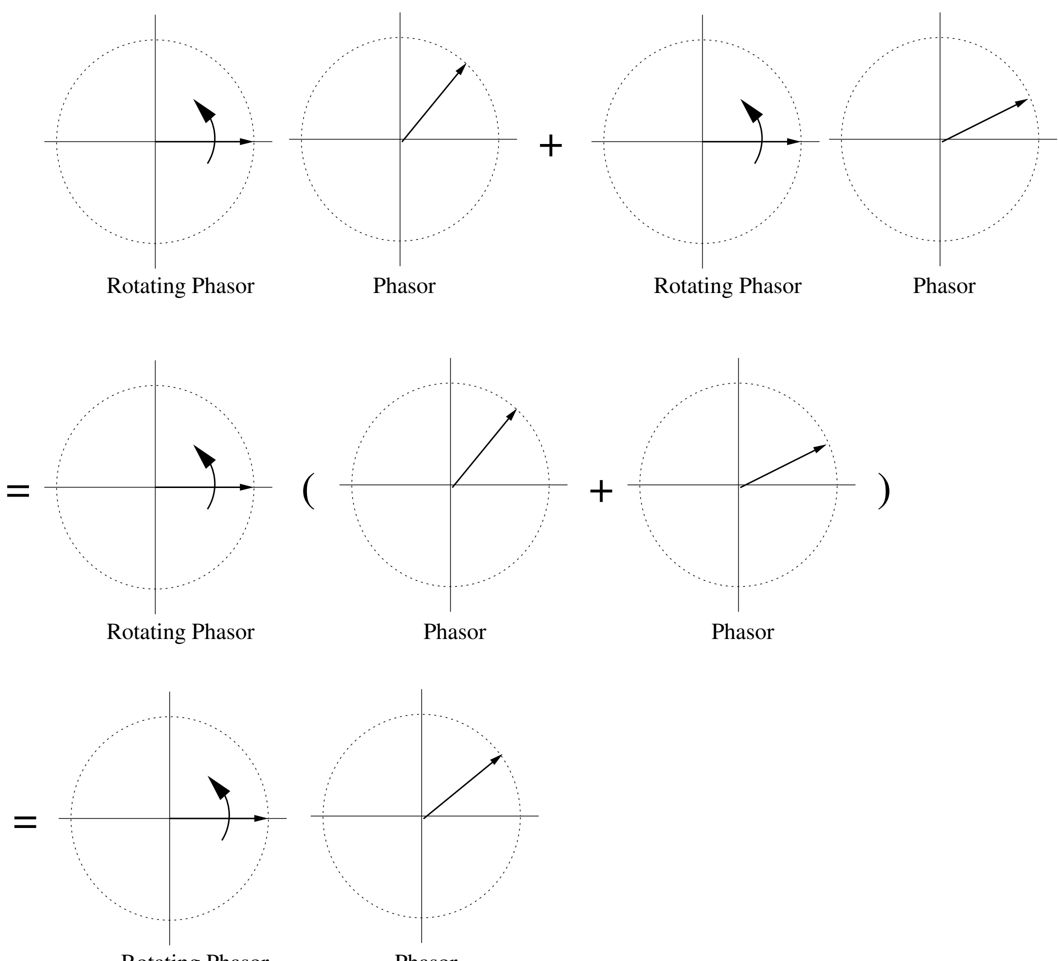



**Figure 7.12:** A graphic representation of adding 2 phasors of the same frequency.

$a_1 \times \cos(2\pi ft + \theta_1)$ and $a_2 \times \cos(2\pi ft + \theta_2)$, visualize them as rotating phasors multiplied by nonrotating phasors, then visualize how they are added together. Note that $r_n = \frac{a_n}{2}$ in this figure.

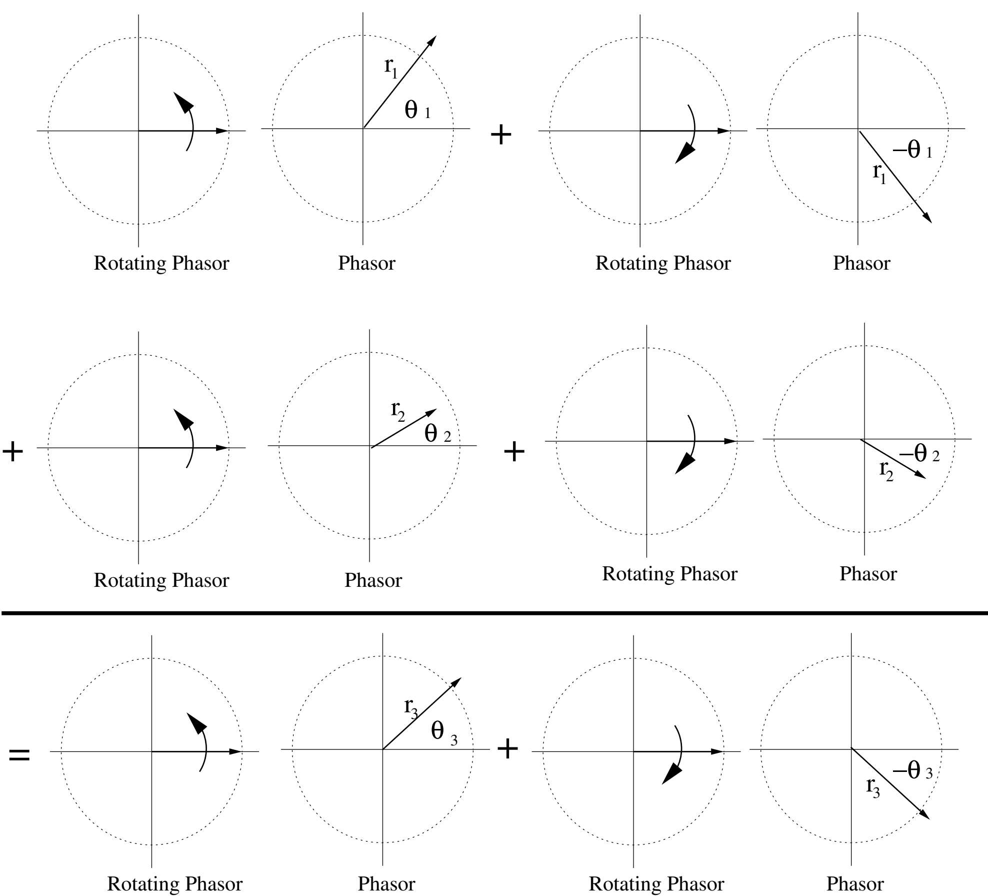



**Figure 7.13:** A graphic representation of adding 2 sinusoids of the same frequency.

$$a_3 \cos(2\pi ft + \theta_3) = a_1 \cos(2\pi ft + \theta_1) + a_2 \cos(2\pi ft + \theta_2)$$

The first row represents $a_1 \cos(2\pi ft + \theta_1)$, shown as a pair of phasors rotating in opposite directions. This comes from Euler's inverse formula, where a cosine function is given as a combination of two phasors (of half the amplitude). Their rotation in opposite directions means that the complex parts cancel each other out, and only the cosine part remains.

The second row represents $a_2 \times \cos(2\pi f t + \theta_2)$, again as a pair of phasors rotating in opposite directions, as given by Euler's inverse formula. On the third row, we have a visualization of the addition result (keeping the left and right sides separate). Notice how similar the left and right sides are; they are complex conjugates of one another.

## 7.9 Multiplying Phasors

We multiply two phasors by putting them in exponential form, multiplying their amplitudes, and adding their angle arguments. To see how adding the angles works, let's look at how a rotating phasor multiplied by another phasor and an amplitude behave when we multiply them using Euler's formula. Here are a couple of equations about adding the arguments of sinusoids [26], which we will need shortly.

$$\cos(\theta_1 + \theta_2) = \cos(\theta_1)\cos(\theta_2) - \sin(\theta_1)\sin(\theta_2)$$

$$\sin(\theta_1 + \theta_2) = \cos(\theta_1)\sin(\theta_2) + \sin(\theta_1)\cos(\theta_2).$$

Rather than write $2\pi f$ over and over again, we will replace it with $\omega$. With these items in mind, we can simplify the following:

$$a \times e^{j\theta} e^{j\omega t} = a \times (\cos(\theta) + j\sin(\theta))(\cos(\omega t) + j\sin(\omega t))$$

$$= a \times \cos(\theta)\cos(\omega t) + ja \times \sin(\theta)\cos(\omega t) + ja \times \cos(\theta)\sin(\omega t) + j^2 a \times \sin(\theta)\sin(\omega t)$$

$$= a \times \cos(\theta)\cos(\omega t) - a \times \sin(\theta)\sin(\omega t) + ja \times \sin(\theta)\cos(\omega t) + ja \times \cos(\theta)\sin(\omega t)$$

$$= a \times (\cos(\theta)\cos(\omega t) - \sin(\theta)\sin(\omega t)) + ja \times (\cos(\theta)\sin(\omega t) + \sin(\theta)\cos(\omega t))$$

$$= a \times \cos(\theta + \omega t) + ja \times \sin(\theta + \omega t).$$

We should see now how multiplying two exponentials simplifies to adding their arguments. Of course, a shortcut would be to replace $e^c e^d$ with $e^{c+d}$, or, more specifically, replace $e^{j\theta} e^{j\omega t}$ with $e^{j(\theta + \omega t)}$. Either way, we arrive at the same result. If we multiply two complex exponents, $z_3 = z_1 z_2$,

$$z_1 = r_1 e^{j\theta_1} = r_1 \cos(\theta_1) + jr_1 \sin(\theta_1)$$

$$z_2 = r_2 e^{j\theta_2} = r_2 \cos(\theta_2) + jr_2 \sin(\theta_2)$$

$$z_3 = (r_1 \cos(\theta_1) + jr_1 \sin(\theta_1))(r_2 \cos(\theta_2) + jr_2 \sin(\theta_2))$$

$$z_3 = r_1 r_2 (\cos(\theta_1 + \theta_2) + j\sin(\theta_1 + \theta_2)).$$

Therefore,

$$z_3 = r_1 r_2 e^{j(\theta_1+\theta_2)}.$$

We simply multiply the amplitudes, and add the angle arguments.

We can also find $z_3 = z_1 z_2$ when $z_1$ and $z_2$ are complex numbers. That is, suppose $z_1 = x_1 + jy_1$, and $z_2 = x_2 + jy_2$. Then $z_3$ can be found by multiplying out the terms.

$$z_3 = x_1 x_2 + jx_1 y_2 + jx_2 y_1 - y_1 y_2$$

Multiplying phasors can be done easily, now that we know the pattern.

## 7.10 Summary

This chapter expands on sinusoids, and covers fundamental ideas about $e$ and $j$. As we saw in earlier chapters, we can decompose a signal into a sum of many sinusoids. Here, we have the underlying theory behind treating a signal as a sum of sinusoids, and the ways in which they can be manipulated mathematically.

Breaking a signal into its component sinusoids is the Fourier transform. Recombining this sinusoidal information (amplitude, frequency, and phase) to recover the original signal is the inverse Fourier transform. We can view the complex magnitudes and phases from the Fourier transform as vectors, or we can view them as complex cosines and sines. Euler's formula (and Euler's inverse formula) allows us to go from one representation to the other. Often, calculations are easier to do in terms of $amplitude \times e^{j\ angle}$, while we want to view the results as a real part ($amplitude \times \cos(angle)$) and an imaginary part ($amplitude \times \sin(angle)$).

## 7.11 Review Questions

1. If $z = (2, 3)$, what is $z$ in terms of $e^j$ notation?

2. What is $5e^{j\pi/4}$ in terms of 2D polar coordinates (radius and angle)?

3. Does $6e^{j2\pi}$ reduce to a real number?

4. If $x(t) = 4e^{j(2\pi 120t+\pi)}$, what is this in terms of sine and cosine?

5. What is $2e^{j3\pi/8}e^{j2\pi 100t}$ in terms of sine and cosine?

6. What is $\Re\{ae^{j(\omega t+\phi)}\}$?

7. Consider the rotating phasor $e^{j\omega t+\phi}$, what does $\omega$ tell us about what it looks like? What does $\phi$ tell us about what it looks like?

8. If $z = a + jb$, what is $z^*$?

9. What is the complex conjugate of $e^{j\phi}$?

10. If $x(t) = a \times \cos(k)$, where $k$ is a constant, what is the frequency?

11. Plot $4 + j5$ on a graph, then multiply it by $j$, and plot the result. Multiply this result by $j$ again, and indicate it on the graph. Repeat this one final time. What do you observe? What angles do the four vectors have?

12. Let $z_1 = 2e^{j2\pi/3}$ and $z_2 = e^{j\pi/7}$. What is $z_3 = z_1 + z_2$? Draw all three vectors on a graph.

13. Let $z_1 = e^{-j\pi/5}$ and $z_2 = 3e^{j\pi/6}$. What is $z_3 = z_1 \times z_2$? Draw all three vectors on a graph.

14. Let $z_1 = 4e^{j(2\pi 100t-5\pi/4)}$ and $z_2 = 5e^{j(2\pi 100t+pi/6)}$. What is $z_3 = z_1 + z_2$? Draw all three vectors on a graph at time = {0 sec, 0.1 sec, and 0.2 sec}.

15. We saw that we can add rotating phasors when they rotate at the same frequency. What happens if we add them when their frequencies do not match? Use MATLAB to find (point by point) and plot $z_3 = z_1 + z_2$, where $z_1 = 4e^{j(2\pi 25t-5\pi/4)}$ and $z_2 = 5e^{j(2\pi 33t+pi/6)}$. Let $t$ range from 0 to 1 with a suitable small increment.

# Chapter 8

# The $z$-Transform

This chapter will address questions about the $z$-transform, an analytical tool for systems. What is $z$? How does this transform work? How can it be used to combine filters? Why do delay units sometimes have a $z^{-1}$ symbol? How does this transform relate to other transforms we have seen? We will answer these questions and more in the following sections.

The $z$-transform is a generalized version of the Fourier transform. Like the Fourier transform, it allows us to represent a time-domain signal in terms of its frequency components. Instead of accessing signal values in term of $n$, a discrete index related to time, we can know the response of the signal for a given frequency (as we did with the Fourier transform). The difference is that we can also specify a magnitude with the $z$-transform.

The $z$-transform serves two purposes. First, it provides a convenient way to notate the effects of filters. So far, we have used the coefficients, $h[n] = \{a, b, c, d\}$, to describe how the output $y[n]$ relates to the input $x[n]$. In $z$-transform notation, we say $Y(z) = H(z)X(z)$, where $H(z)$ is the $z$-transform of $h[n]$. We can think of $H(z)$ as something that operates on $X(z)$ to produce the output $Y(z)$. For this reason, $H(z)$ is also called the *transfer function.* Rather than use $ax[n-k]$ in the equation, we can put the coefficient and the delay ($k$) together, and remove $x$. So the filter with coefficients $h[n] = \{a, b, c, d\}$ can be described instead by the $z$-transform of $h[n]$, $H(z) = az^0 + bz^{-1} + cz^{-2} + dz^{-3}$. A second purpose of the $z$-transform is to tell us about the stability of the filter, but this will be explained a bit later.

## 8.1 The z-Transform

By definition, the *z-transform* of a discrete signal $x[n]$ is

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

where $z$ is a complex number, $z = re^{j\omega}$. Typically, $n$ has a finite range, for example, imagine a signal $x$ with six values. Effectively, in this example, $n$ has a range of only 0 to 5, since any value of $x$ before index 0 or after index 5 is considered to be 0. So the $z$-transform would be:

$$X(z) = x[0]z^0 + x[1]z^{-1} + x[2]z^{-2} + x[3]z^{-3} + x[4]z^{-4} + x[5]z^{-5}.$$

Theoretically however, for a general $x[n]$, $X(z)$ could diverge, since $n$ could be $\infty$. As an example, consider a simple function of a constant value, such as $x[n] = 1$, where $-\infty \leq n \leq \infty$. We can express the $z$-transform compactly with the equation $\sum_{n=-\infty}^{\infty} 1 \times z^{-n}$, but this sum could clearly be infinite. For this reason, we include a *region of convergence* with the $z$-transform, called *RoC* for short. An interesting aspect of the region of convergence is that it depends only on $r$, not $\omega$. Remember that $z = re^{j\omega}$, so $z^{-k} = r^{-k}e^{-jk\omega}$, and no matter what the value of $\omega$, the value for $e^{-jk\omega}$ will be bounded by the maximum and minimum value of the cosine and sine functions. That is, $e^{-jk\omega} = \cos(k\omega) - j\sin(k\omega)$, and the cosine and sine functions return values between $-1$ and 1 (the magnitude of $e^{-jk\omega}$ is 1). Thus, the $e^{-jk\omega}$ will not contribute to the $z$-transform being infinite; either the $r^{-k}$ term makes the $z$-transform infinite, or the $z$-transform converges. Therefore, the region of convergence is circular [20]. If we know that a point is inside (or outside) the region of convergence, all other points of the same magnitude will also be inside (or outside) the region of convergence.

The RoC will be all area on and inside the unit circle, for all practical cases. It is only when we consider all theoretical cases that it gets confusing. In fact, some DSP texts steer clear of the cases where the RoC is not the unit circle.

We can also view the $z$-transform as a geometric series. For example, if $c \neq 0$ and $-1 < d < 1$ [26]:

$$c + cd + cd^2 + \ldots + cd^{n-1} + \ldots = \frac{c}{1-d}.$$

Therefore, if $x[n] = 1$ for all $n \geq 0$,

$$X(z) = \sum_{n=-\infty}^{\infty} 1 \times z^{-n}$$

$$= 1 + 1z^{-1} + 1(z^{-1})^2 + \ldots + 1(z^{-1})^{n-1} + \ldots$$

$$= \frac{1}{1 - z^{-1}} \quad \text{for all } -1 < z^{-1} < 1.$$

Of course, we should check to see if this sum converges or not. In fact, when $|z^{-1}| \geq 1$, the sum does not converge (and we should not use the above fractional expression). This is easy to see; as $n$ gets larger, we add another term $cd^n$ to our sum. If $d$ happens to be a fraction of 1, then the term we add is smaller than the last term we added, and the sum does not grow substantially. For example, suppose $d = \frac{1}{2}$. Adding a millionth term would mean adding $c \times (\frac{1}{2})^{1000000}$, or $\frac{c}{2^{1000000}}$, to the sum, which would not change it dramatically (especially if we compare it to the effect of adding the first term, $c$). However, supposing that $d$ were exactly 1, adding a millionth term would mean adding $c$ to the sum. This would be as significant as adding the first term. If $d$ were greater than 1, then every value for $cd^n$ means adding a larger value than the previous one, meaning that the sum continues to grow significantly.

As a fraction, we know that the expression $X(z) = \frac{1}{1-z^{-1}}$ evaluates to zero when the numerator (top part of the fraction) is zero (though it cannot happen in this case). Also, we know that the fraction will be infinite when the denominator (bottom part) evaluates to zero. This happens when $1 - z^{-1} = 0$, or $z^{-1} = 1$. This is the idea behind the region of convergence, and we will revisit this when we examine the frequency response of a filter.

## 8.2 Replacing Two FIR Filters in Series

When two filters are placed in series, the output will be a convolution of the input with the first filter's coefficients, followed by convolution with the second filter's coefficients. Is it possible then to replace the two filters with a single filter instead? The answer is yes. Figure 8.1 helps to demonstrate this idea.

Two filters can be combined into one filter by convolving the filter coefficients. There are a couple of reasons why this is interesting. First, combining filters in series may lead to a space savings if the design is implemented in hardware. Second, it may be a good idea to break a filter into two filters in series, so that the resulting filters are easier to make. For example, suppose there are two FIR filters in series,

with coefficients {2, 0, 0.5} and {1, −1, 3}, respectively. The effect is the same as having a single FIR filter with coefficients {2, −2, 5.5, −0.5, 1.5}. But which is easier to implement, the first two filters, or the equivalent one? Notice that the first filter can be made without multipliers; all we need is a shifter. For fixed-point numbers, a left-shift is equivalent to multiplying by 2, while a right-shift is equivalent to dividing by 2. For the second filter, no multiplier is needed for multiplying by 1, and multiplying by −1 can be done by flipping the sign bit. The only tricky one is the multiplication by 3, and this can be done with a shift and an add, that is, $x + 2x = 3x$. For a system with just these coefficients, the first two filters can be implemented without multipliers.

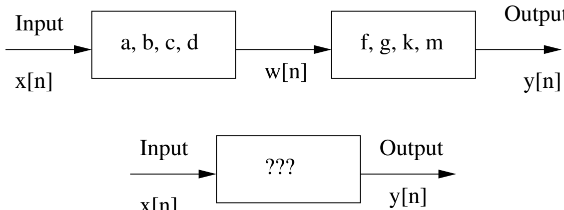



**Figure 8.1:** FIR filters in series can be combined.

Let's examine how we can find the resulting combined coefficients, in the following manner. First, we will write the equations describing the output of the two filters along the top of Figure 8.1, as if they were two separate filters; that is, one filter with coefficients $\{a, b, c, d\}$ and an input labeled $x$ and output labeled $w$, and another filter of coefficients $\{f, g, k, m\}$ with input $w$ and output $y$. All filter coefficients are assumed to be constants.

$$w[n] = ax[n] + bx[n-1] + cx[n-2] + dx[n-3]$$

$$y[n] = fw[n] + gw[n-1] + kw[n-2] + mw[n-3]$$

We use $w$ as a label simply to make this description less confusing, but what we really need is the final output $y$ in terms of the original input $x$. To figure this out, we must have an expression for $w[n-k]$, instead of $w[n]$. In the next step, we use the expression for $w[n]$, but replace $n$ with $n-k$ throughout.

$$w[n-k] = ax[n-k] + bx[n-k-1] + cx[n-k-2] + dx[n-k-3]$$

Putting the output $y[n]$ all in terms of the input, $x[n]$:

$$\begin{aligned} y[n] &= afx[n] + bfx[n-1] + cfx[n-2] + dfx[n-3] \\ &+ agx[n-1] + bgx[n-2] + cgx[n-3] + dgx[n-4] \\ &+ akx[n-2] + bkx[n-3] + ckx[n-4] + dkx[n-5] \\ &+ amx[n-3] + bmx[n-4] + cmx[n-5] + dmx[n-6]. \end{aligned}$$

Simplifying:

$$\begin{aligned} y[n] &= afx[n] + (ag + bf)x[n-1] + (ak + bg + cf)x[n-2] \\ &+ (am + bk + cg + df)x[n-3] + (bm + ck + dg)x[n-4] \\ &+ (cm + dk)x[n-5] + dmx[n-6]. \end{aligned}$$

We see that our expression for $y[n]$ now contains only a linear combination of the input (and delayed versions of it), multiplied by constants. The resulting coefficients are, therefore, $\{af$, $(ag + bf)$, $(ak + bg + cf)$, $(am + bk + cg + df)$, $(bm + ck + dg)$, $(cm + dk)$, $dm\}$. This pattern is the convolution of $\{a, b, c, d\}$ with $\{f, g, k, m\}$.

We have found a resulting filter using convolution. In the next section, we will see how this analysis could be done with the $z$-transform.

## 8.3 Revisiting Sequential Filter Combination with $z$

Let's return to the example in Figure 8.1. We want to replace the two filters $h_1[n] = \{a, b, c, d\}$ and $h_2[n] = \{f, g, k, m\}$ with a single equivalent filter. First, we can write the $z$-transform for these two filters directly, just by examining their coefficients.

$$H_1(z) = az^0 + bz^{-1} + cz^{-2} + dz^{-3}$$

$$H_2(z) = fz^0 + gz^{-1} + kz^{-2} + mz^{-3}$$

To find the equivalent filter, we multiply the $z$-transforms $H_1(z)$ and $H_2(z)$ together. We do this because multiplication in the frequency-domain is the same as convolution in the time-domain, as demonstrated in section 8.8. We will call the $z$-transform for our third filter $H_3(z)$,

$$H_3(z) = H_1(z)H_2(z)$$

$$
\begin{aligned}
H_3(z) &= (az^0 + bz^{-1} + cz^{-2} + dz^{-3})(fz^0 + gz^{-1} + kz^{-2} + mz^{-3}) \\
H_3(z) &= az^0fz^0 + az^0gz^{-1} + az^0kz^{-2} + az^0mz^{-3} \\
&+ bz^{-1}fz^0 + bz^{-1}gz^{-1} + bz^{-1}kz^{-2} + bz^{-1}mz^{-3} \\
&+ cz^{-2}fz^0 + cz^{-2}gz^{-1} + cz^{-2}kz^{-2} + cz^{-2}mz^{-3} \\
&+ dz^{-3}fz^0 + dz^{-3}gz^{-1} + dz^{-3}kz^{-2} + dz^{-3}mz^{-3}.
\end{aligned}
$$

Simplifying:

$$
\begin{aligned}
H_3(z) &= af + agz^{-1} + akz^{-2} + amz^{-3} \\
&+ bfz^{-1} + bgz^{-2} + bkz^{-3} + bmz^{-4} \\
&+ cfz^{-2} + cgz^{-3} + ckz^{-4} + cmz^{-5} \\
&+ dfz^{-3} + dgz^{-4} + dkz^{-5} + dmz^{-6}.
\end{aligned}
$$

Combining terms:

$$
\begin{aligned}
H_3(z) &= af + (ag + bf)z^{-1} + (ak + bg + cf)z^{-2} \\
&+ (am + bk + cg + df)z^{-3} + (bm + ck + dg)z^{-4} \\
&+ (cm + dk)z^{-5} + dmz^{-6}.
\end{aligned}
$$

This result should look familiar; it is the same as when we found $y[n]$ above, except that $x[n]$ is not mentioned. Instead, we have $z^{-k}$ to indicate delays, that is, when $H_3(z)$ is applied to input $X(z)$, we would have the equation:

$$
\begin{aligned}
Y(z) &= H_3(z)X(z) \\
Y(z) &= afX(z) + (ag + bf)X(z)z^{-1} + (ak + bg + cf)X(z)z^{-2} \\
&+ (am + bk + cg + df)X(z)z^{-3} + (bm + ck + dg)X(z)z^{-4} \\
&+ (cm + dk)X(z)z^{-5} + dmX(z)z^{-6}.
\end{aligned}
$$

For example, $X(z)z^{-1}$ in the time-domain means that $x[n]$ is delayed by one unit, or that $afX(z) = afx[n]$, $(ag+bf)X(z)z^{-1} = (ag+bf)x[n-1]$, $(ak+bg+cf)X(z)z^{-2} = (ak + bg + cf)x[n-2]$, and so forth.

## 8.4 Why Is $z^{-1}$ the Same as a Delay by One Unit?

A delay unit is often shown as $z^{-1}$ in a box, since multiplying by $z^{-1}$ in the frequency-domain delays a signal by one sample. To see why this is the case, let's examine two similar FIR filters, one with coefficients $\{1, 0\}$, and the other with coefficients $\{0, 1\}$. These trivial filters are shown in Figure 8.2.

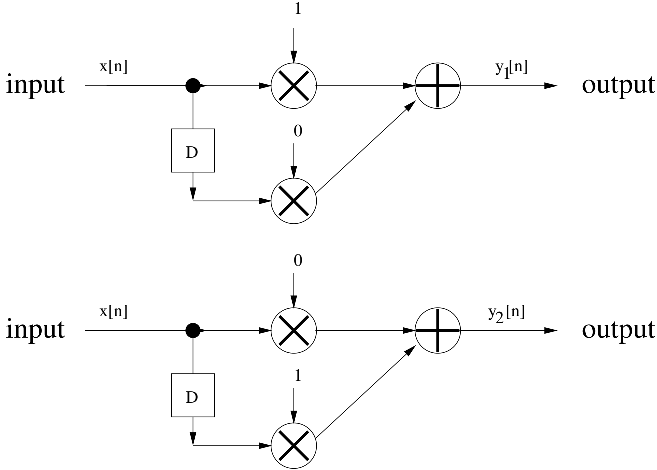



**Figure 8.2:** Two trivial FIR filters.

By inspection, we see that the first filter effectively does nothing, and the output is the same as the input, $y_1[n] = x[n]$. The second filter really does not do anything either, except that the output is a delayed version of the input, i.e., $y_2[n] = x[n-1]$. We can remove the multipliers, since multiplying by zero gives us a zero result, and multiplying a number by one gives the same number back. Since one of the adder's inputs is zero, we can remove the adders from the diagram as well. This leaves us with a very reduced form of these two filters, Figure 8.3.

The first filter has a transfer function $H_1(z) = 1z^0 + 0z^{-1}$, while the second one has $H_2(z) = 0z^0 + 1z^{-1}$. Writing the output in terms of the transfer function and input gives us:

$$
\begin{aligned}
Y_1(z) &= 1X(z)z^0 + 0X(z)z^{-1} \\
&= X(z)z^0 \\
&= X(z) \\
Y_2(z) &= 0X(z)z^0 + 1X(z)z^{-1}
\end{aligned}
$$

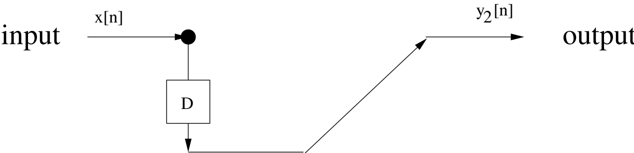



**Figure 8.3:** Two trivial FIR filters, reduced.

$$= \quad X(z)z^{-1}.$$

Here we see that the only difference between the $z$-transform of the first system and the second (delayed by one sample) system is that the power that $z$ is raised to is decreased by 1. In other words, a one unit time delay is equivalent to multiplication in the frequency-domain $z^{-1}$.

## 8.5 What Is $z$?

By definition, $z$ is a complex number.

$$z = re^{j\omega}$$

We say that the $z$-transform is in the frequency-domain because there is a direct link between the $z$-transform and the Fourier transform. Replacing the $z$ in the $z$-transform definition with $re^{j\omega}$ gives us the following equation:

$$X(re^{j\omega}) = \sum_{n=-\infty}^{\infty} x[n]re^{-j\omega n}.$$

This equation should look familiar; all that we have to do to turn this equation into the Fourier transform is to set $r = 1$.

As we saw above, however, we often do not need to use $z$'s definition in order to use the $z$-transform.

## 8.6 How the $z$-Transform Reduces to the Fourier Transform

Earlier, we stated that the discrete Fourier transform is a simplified version of the $z$-transform. Here, we demonstrate this. First, we start with the definition of the $z$-transform.

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

Then, replace $z$ with its definition.

$$X(re^{j\theta}) = \sum_{n=-\infty}^{\infty} x[n](re^{j\theta})^{-n}$$

Simplify:

$$X(re^{j\theta}) = \sum_{n=-\infty}^{\infty} x[n]r^{-n}e^{-j\theta n}.$$

Next, let $\theta = 2\pi m/N$, where $m$ and $n$ are indices, and $N$ is the length of the signal $x$. We assume that signal $x$ is causal, so the first value of $x$ starts at index zero.

$$X(re^{j2\pi m/N}) = \sum_{n=0}^{N-1} x[n]r^{-n}e^{-j2\pi mn/N}$$

If $r = 1$, then $r^{-n} = 1$ for all $n$ values. This leads to:

$$X(e^{j2\pi m/N}) = \sum_{n=0}^{N-1} x[n]e^{-j2\pi mn/N}.$$

Notice that the previous equation matches the definition for the DFT. We usually write the DFT as $X[m]$ on the lefthand side, simply because $m$ is the only variable parameter. Some people use the notation $X(\omega)$ on the lefthand side, for the same reason.

## 8.7 Powers of $-z$

When multiplying powers of $-z$, the chart below can be used to do substitutions. This comes up in filter design, in finding a channel's frequency response. We see that the signs of $-z^{-k}$ alternate for even and odd values of $k$. This comes in handy

when we incorporate up/down-samplers into our architecture.

$$\begin{aligned}
(-z)^{-1} &= \frac{1}{-z} = -z^{-1} \\
(-z)^{-2} &= \frac{1}{(-z)^2} = z^{-2} \\
(-z)^{-3} &= \frac{1}{(-z)^3} = -z^{-3} \\
(-z)^{-4} &= \frac{1}{(-z)^4} = z^{-4}
\end{aligned}$$

## 8.8 Showing that $x[n] * h[n] \leftrightarrow X(z)H(z)$

Here we demonstrate that convolution in the time-domain gives the same results as multiplication in the frequency-domain. In other words,

$$x[n] * h[n] \leftrightarrow X(z)H(z).$$

Let's assume that we are dealing with small signals. Let $h = \{h_0, h_1, h_2\}$ and $x = \{x_0, x_1, x_2, x_3\}$. First, we find the convolution $x * h$ as shown in Table 8.1. As an exercise, calculate $h * x$, and show that you get the same result. It should be apparent how having more (or fewer) values for $x$ or $h$ would affect this example. If not, try this example on your own with $x_0 ... x_4$, then try it again with $h_0 ... h_3$.

Now let's find the values for $X(z)$ and $H(z)$. By definition of the $z$-transform, we have:

$$X(z) = x_0 z^{-0} + x_1 z^{-1} + x_2 z^{-2} + x_3 z^{-3}$$

$$H(z) = h_0 z^{-0} + h_1 z^{-1} + h_2 z^{-2}.$$

Multiplying these together:

$$X(z) \times H(z)$$

$$\begin{aligned}
&= (x_0 z^{-0} + x_1 z^{-1} + x_2 z^{-2} + x_3 z^{-3}) \times (h_0 z^{-0} + h_1 z^{-1} + h_2 z^{-2}) \\
&= x_0 z^{-0} h_0 z^{-0} + x_1 z^{-1} h_0 z^{-0} + x_2 z^{-2} h_0 z^{-0} + x_3 z^{-3} h_0 z^{-0} \\
&\quad + x_0 z^{-0} h_1 z^{-1} + x_1 z^{-1} h_1 z^{-1} + x_2 z^{-2} h_1 z^{-1} + x_3 z^{-3} h_1 z^{-1} \\
&\quad + x_0 z^{-0} h_2 z^{-2} + x_1 z^{-1} h_2 z^{-2} + x_2 z^{-2} h_2 z^{-2} + x_3 z^{-3} h_2 z^{-2}.
\end{aligned}$$

**Table 8.1:** Convolution of $x$ and $h$.

| $x_0$ | $x_1$ | $x_2$ | $x_3$ | | |
|---|---|---|---|---|---|
| $h_0$ | $h_1$ | $h_2$ | | | |
| $x_0h_0$ | $x_1h_0$ | $x_2h_0$ | $x_3h_0$ | | |
| | $x_0h_1$ | $x_1h_1$ | $x_2h_1$ | $x_3h_1$ | |
| | | $x_0h_2$ | $x_1h_2$ | $x_2h_2$ | $x_3h_2$ |
| $x_0h_0$ | $x_1h_0 + x_0h_1$ | $x_2h_0 + x_1h_1$ $+x_0h_2$ | $x_3h_0 + x_2h_1$ $+x_1h_2$ | $x_3h_1 + x_2h_2$ | $x_3h_2$ |

Now we can simplify terms, especially the powers of $z$. Remember that $z^{-a} \times z^{-b} = z^{-a-b}$. Thus, we get:

$$= x_0h_0z^{-0} + x_1h_0z^{-1} + x_2h_0z^{-2} + x_3h_0z^{-3}$$

$$+x_0h_1z^{-1} + x_1h_1z^{-2} + x_2h_1z^{-3} + x_3h_1z^{-4}$$

$$+x_0h_2z^{-2} + x_1h_2z^{-3} + x_2h_2z^{-4} + x_3h_2z^{-5}.$$

The next step is to group terms by the exponent of $z$.

$$= x_0h_0z^{-0} + (x_1h_0 + x_0h_1)z^{-1} + (x_2h_0 + x_1h_1 + x_0h_2)z^{-2}$$

$$+(x_3h_0 + x_2h_1 + x_1h_2)z^{-3} + (x_3h_1 + x_2h_2)z^{-4} + x_3h_2z^{-5}$$

One way to interpret this result is to remember how $z^{-1}$ corresponds to a delay of one time unit. Thus, the first output of a system that computes $X(z) \times H(z)$ would be, in the time-domain, $x_0h_0$, followed by $x_1h_0 + x_0h_1$, $x_2h_0 + x_1h_1 + x_0h_2$, etc.; the same values in the same order as the convolution example.

Another way to see that the results are equivalent is to perform the $z$-transform on the output of the convolution example, which will give us $x_0h_0z^{-0}$, $(x_1h_0 + x_0h_1)z^{-1}$, and so forth.

Thus, we have established that convolution in the time-domain is equivalent to multiplication in the frequency-domain.

## 8.9 Frequency Response of Filters

The $z$-transform can be used with IIR filters, and in fact is a useful analytical tool. The transfer function, $H(z)$ reveals information about the stability of the

filter. With IIR filters, it is possible for the output to approach infinity (or negative infinity), which can be seen with the transfer function, $H(z)$.

The $z$-transform describes the frequency response of filters, simply by finding $z$-transform of the filter's coefficients. For an FIR filter, we find the frequency response with:

$$H(z) = \sum_{k=0}^{N} h[k]z^{-k}$$

where $h[k]$ are the filter's coefficients.

For an IIR filter, the situation is a bit more complicated, since we have both a feed-forward part (coefficients $b[k]$) and a feed-back part (coefficients $a[k]$). We start by looking at the time-domain equation describing the filter's output.

$$y[n] = \sum_{k=0}^{K} b[k]x[n-k] + \sum_{k=1}^{M} a[k]y[n-k]$$

Remember that the feed-back part starts with $a[1]$ instead of $a[0]$, since there is no multiplier at that corresponding location. Next, we find the $z$-transform of this equation,

$$Y(z) = X(z)\sum_{k=0}^{N} b[k]z^{-k} + Y(z)\sum_{k=1}^{M} a[k]z^{-k},$$

where $z = re^{j\omega}$.

The *transfer function* gives us the IIR filter's response. We can specify the effect of the filter without having to give an input:

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

— or —

$$H(z) = \frac{\sum_{k=0}^{N} b[k]z^{-k}}{1 - \sum_{k=1}^{M} a[k]z^{-k}}.$$

Notice that the denominator *could* be zero under the right circumstances (values for $z$). If this happens, our frequency response, $H(z)$, is infinite. We call such a value for $z$ a *pole*, borrowing from the image of a circus tent, where the tent fabric has a pronounced shape from the tent poles, like a three-dimensional plot of $H(z)$'s surface would for certain values of $z$. Also, some values of $z$ could result in a zero value for the function as a whole, from the numerator. These spots we call *zeros*. When we talk about the frequency response for IIR filters, we are primarily concerned with

the locations of the zeros and the poles. If a pole is located outside our region of convergence, then we have a problem: the system is unstable.

A nice attribute of FIR filters is that, since they have no feedback, they do not have poles. This comes from the way the transfer function $H(z)$ is defined. It specifies how the filter maps the input to the output, without specifying the input. Sometimes the filter's output will be zero, such as when the input is zero. For example, if the input $x[n] = 0$ for all $n$, then any FIR filter will output zeros as well, i.e., $y[n] = b[0]x[n] + b[1]x[n-1] + ... + b[k]x[n-k]$. As long as the $x[]$ values are zero, the $y[]$ values will be zero, too. This can be expressed in the frequency-domain, also, with the $z$-transform, as $Y(z) = H(z)X(z)$. If $x[n] = 0$ for all $n$, then $X(z) = 0z^0 + 0z^{-1} ... + 0z^{-n}$, or $X(z) = 0$. $H(z)$ specifies the filter's effect, so $H(z) = b_0 z^0 + b_1 z^{-1} + ... + b_k z^{-k}$. From the input-output equation, $Y(z) = H(z)$ times 0 for this particular $X(z)$, so $Y(z)$ will be zero, too. The fact that the transfer function $H(z)$ has zeros simply indicates that there are some conditions where $H(z)$ can be zero, which means that $Y(z)$ will be zero, too.

For an FIR filter, the transfer function $H(z)$ can be found by the $z$-transform of the filter coefficients, $b[k]$, to which the unit-impulse response $h$ is identical. This is why we use lowercase $h[k]$ for the filter coefficients of an FIR filter, and $H(z)$ for the transfer function. For IIR filters, we have the same idea, but it is a bit more complicated to find the transfer function since we have two sets of filter coefficients.

First, let's revisit the FIR filter, and suppose that we have 3 filter coefficients, $b_0$, $b_1$, and $b_2$, corresponding to the unit-impulse response of $h_0$, $h_1$, and $h_2$. We can write the relation describing the output as:

$$y[n] = b[0]x[n] + b[1]x[n-1] + b[2]x[n-2].$$

If we then find the $z$-transform of the above relation, we have:

$$Y(z) = b[0]X(z)z^0 + b[1]X(z)z^{-1} + b[2]X(z)z^{-2}$$

and simplifying:

$$Y(z) = X(z)(b[0]z^0 + b[1]z^{-1} + b[2]z^{-2}).$$

If we divide out $X(z)$, we get

$$Y(z)/X(z) = b[0]z^0 + b[1]z^{-1} + b[2]z^{-2}$$

which is the same thing as the $z$-transform of the filter coefficients, or $H(z)$ by definition, since $h_k = b_k$ for all integer $k$ values.

Now, for an IIR filter, our approach is the same. We write the input/output relation, find the $z$-transform of it, then rearrange terms until we have $Y(z)/X(z)$

on one side. Thus, we have an expression on the other side for the transfer function $H(z)$. Often, the zeros and poles are plotted on the $z$-plane. The `zplane` function in MATLAB will do this for us.

Suppose that an IIR filter has feed-forward coefficients of $\{b_0, b_1, b_2\}$ and feedback coefficients of $\{a_1, a_2\}$. What is the transfer function? Where are the zeros? Where are the poles?

First, we write the time-domain function describing the input/output:

$$y[n] = b_0x[n] + b_1x[n-1] + b_2x[n-2] + a_1y[n-1] + a_2y[n-2].$$

Finding the $z$-transform:

$$Y(z) = b_0X(z) + b_1X(z)z^{-1} + b_2X(z)z^{-2} + a_1Y(z)z^{-1} + a_2Y(z)z^{-2}.$$

Grouping the terms:

$$Y(z)(1 - a_1z^{-1} - a_2z^{-2}) = X(z)(b_0 + b_1z^{-1} + b_2z^{-2}).$$

Rearranging:

$$\frac{Y(z)}{X(z)} = \frac{b_0 + b_1z^{-1} + b_2z^{-2}}{1 - a_1z^{-1} - a_2z^{-2}}.$$

Since $H(z) = Y(z)/X(z)$ by definition, we have found our transfer function.

Next, to find the zeros, we set the numerator to 0:

$$b_0 + b_1z^{-1} + b_2z^{-2} = 0.$$

Putting this in terms of positive exponents for $z$:

$$b_0z^2 + b_1z + b_2 = 0.$$

Using the quadratic formula, we find the roots are:

$$\frac{-b_1 + \sqrt{(b_1)^2 - 4b_0b_2}}{2b_0}$$

and

$$\frac{-b_1 - \sqrt{(b_1)^2 - 4b_0b_2}}{2b_0}.$$

We find the poles in a similar way. First, set the denominator to 0:

$$1 - a_1 z^{-1} - a_2 z^{-2} = 0.$$

Putting this in terms of positive exponents for $z$:

$$z^2 - a_1 z - a_2 = 0.$$

Using the quadratic formula, we find the roots, noting that $a_0 = 1$, while we negate the $a_1$ and $a_2$ values:

$$\frac{-(-a_1) + \sqrt{(-a_1)^2 - 4(-a_2)}}{2}$$

and

$$\frac{-(-a_1) - \sqrt{(-a_1)^2 - 4(-a_2)}}{2}.$$

---

**Example:**
Suppose we have an IIR filter with feed-forward coefficients {4, 5, 6}, and feed-back coefficients {2, 3}. What is the transfer function? Where are the zeros? Where are the poles?

**Answer:**
We find the transfer function as we did above: First, we write the time-domain function describing the input/output:

$$y[n] = 4x[n] + 5x[n-1] + 6x[n-2] + 2y[n-1] + 3y[n-2].$$

Finding the $z$-transform:

$$Y(z) = 4X(z) + 5X(z)z^{-1} + 6X(z)z^{-2} + 2Y(z)z^{-1} + 3Y(z)z^{-2}.$$

Grouping the terms:

$$Y(z)(1 - 2z^{-1} - 3z^{-2}) = X(z)(4 + 5z^{-1} + 6z^{-2}).$$

Rearranging:

$$\frac{Y(z)}{X(z)} = \frac{4 + 5z^{-1} + 6z^{-2}}{1 - 2z^{-1} - 3z^{-2}}.$$

Since $H(z) = Y(z)/X(z)$ by definition, we have found our transfer function:

$$H(z) = \frac{4 + 5z^{-1} + 6z^{-2}}{1 - 2z^{-1} - 3z^{-2}}.$$

Next, to find the zeros, we set the numerator to 0:

$$4 + 5z^{-1} + 6z^{-2} = 0.$$

Putting this in terms of positive exponents for $z$:

$$4z^2 + 5z + 6 = 0.$$

Using the quadratic formula, we find the roots are:

$$\frac{-5 + \sqrt{(5)^2 - 4 \times 4 \times 6}}{2 \times 4}$$

and

$$\frac{-5 - \sqrt{(5)^2 - 4 \times 4 \times 6}}{2 \times 4}.$$

A MATLAB session showing the calculations follows.

```
>> B = [4, 5, 6];
>> zero1 = (-B(2) + sqrt(B(2)^2 - 4*B(1)*B(3)))/(2*B(1))

zero1 =

  -0.6250 + 1.0533i

>> zero2 = (-B(2) - sqrt(B(2)^2 - 4*B(1)*B(3)))/(2*B(1))

zero2 =

  -0.6250 - 1.0533i
```

We find the poles in a similar way. First, set the denominator to 0:

$$1 - a_1 z^{-1} - a_2 z^{-2} = 0.$$

Putting this in terms of positive exponents for $z$:

$$z^2 - a_1 z - a_2 = 0.$$

Using the quadratic formula, we find the roots, noting that $a_0 = 1$, while we negate the $a_1$ and $a_2$ values:

$$\frac{-(-a_1) + \sqrt{(-a_1)^2 - 4(-a_2)}}{2}$$

and

$$\frac{-(-a_1) - \sqrt{(-a_1)^2 - 4(-a_2)}}{2}.$$

The following MATLAB code demonstrates this.

```
>> A = [1, -2, -3];
>> pole1 = (-A(2) + sqrt(A(2)^2 - 4*A(1)*A(3)))/(2*A(1))

pole1 =

     3

>> pole2 = (-A(2) - sqrt(A(2)^2 - 4*A(1)*A(3)))/(2*A(1))

pole2 =

    -1
```

Finally, we can plot the zeros and poles with the following command:

```
>> zplane([4, 5, 6], [1, -2, -3]);
```

Figure 8.4 shows the plot generated by the above command. Notice that the parameter is a row vector containing the filter's feed-forward coefficients. The second parameter is 1, followed by negated versions of the filter's feed-back coefficients. The reason that they are negated and preceded by a 1 has to do with the transfer function; these vectors are not the filter coefficients, but the constants from the polynomials of the transfer function. Of course, the transfer function depends directly on the filter coefficients, so we see the same numbers (2 and 3) show up in both. Please note that the `zplane` function requires the parameters to be row vec-

tors, otherwise they will not be interpreted as the transfer function's nominator and denominator.

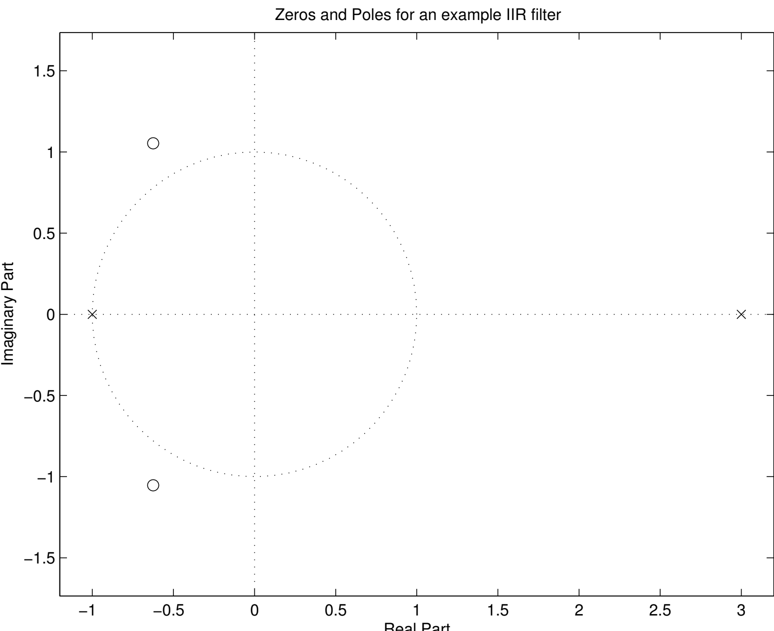



**Figure 8.4:** Example plot of zeros and poles.

---

By definition, a pole is a value for $z$ such that the transfer function becomes infinite. Any $z^{-k}$ quantity can be considered $\frac{1}{z^k}$, and if $z = 0$, this value is infinite. Therefore, for each occurrence of $z^{-k}$ in the numerator, we could consider it to have a corresponding pole at $z = 0$. This is why the `zplane` function will plot poles even when we set the denominator to 1, i.e., there is no feedback. Does this mean that an FIR filter is unstable for zero frequency? No, this is not the case. Zero frequency means that the input never changes, that we have an endless repetition of some value at the input. But a value of $z = 0$ means that the magnitude has a zero value; the angle (frequency) can be any value. In other words, if the frequency is zero and the magnitude is not zero, we have a stability for the filter. How can an FIR filter be made nonstable? This happens only when it has an infinite number of filter coefficients, in which case it will have an infinite sum for its $z$-transform.

## 8.10 Trends of a Simple IIR Filter, Part II

Remember when we examined the output from a simple IIR filter, Figure 3.29? We saw that the feed-back coefficient $a$ played a key role in determining how the output's behavior. Now that we have the $z$-transform, we can predict the behavior based upon $a$. The input/output equation is given as:

$$y[n] = bx[n] + ay[n-1].$$

Taking the $z$-transform, we get:

$$Y(z) = bX(z) + aY(z)z^{-1}$$

$$\frac{Y(z)}{X(z)} = \frac{b}{1 - az^{-1}}.$$

We see that there is a pole at $z = a$, and this pole lies inside the unit circle when $|a| < 1$. This gives the stability that we saw in Figure 3.29 (top). We observe conditional stability in the middle two plots, and unstable cases in the bottom two plots.

## 8.11 Summary

This chapter introduces the $z$-transform as a generalization of the Fourier transform. It is useful in talking about the frequency response of filters, and whether an IIR filter is stable or not. An FIR filter is stable for all practical purposes (barring any theoretically possible but impossible-to-build system, such as one with an infinite number of taps).

The $z$-transform allows us to calculate a filter's effect on an input without having to specify the input, called the transfer function $H(z)$. We have zeros and poles, corresponding to a transfer function's response of zero or $\pm\infty$. A good way to think of $H(z)$ is as a two-dimensional function that gives us a surface, based upon the two-dimensional variable $z$. This chapter demonstrated that multiplication by $z^{-1}$ corresponds to a delay of one unit of time. Also, as we saw in this chapter, multiplication in the frequency-domain is equivalent to convolution in the time-domain.

There is an inverse $z$-transform, but it is outside the scope of this text; see *The Electrical Engineering Handbook* by W. K. Chen (ed), [27] for more information.

## 8.12 Review Questions

1. What is the $z$-transform of $x = \{3, 2, 4\}$?

2. A FIR filter has the following filter coefficients: $\{1, -1\}$. What is the frequency response? Where are the zeros located? Use MATLAB to calculate the output with an input of $\cos(2\pi 100t)$, sampled at $f_s = 200$ samples/second.

3. A FIR filter has the following filter coefficients: $\{1, 1\}$. What is the frequency response? Where are the zeros located? Use MATLAB to calculate the output with an input of $\cos(2\pi 100t)$, sampled at $f_s = 200$ samples/second.

4. A filter has the following transfer function:

$$H(z) = (1 + 3z^{-1} - 4z^{-2})/(1 - z^{-1} - 6z^{-2}).$$

   a. Determine where the zeros and poles are.

   b. Plot these (by hand) on the $z$-plane, with an `x` for a pole and an `o` for a zero. Is this filter stable?

   c. Draw the filter in terms of multipliers, adders, and delay units.

5. What is the $z$-transform of the transfer function, $H(z)$, for the following filters?

   a. FIR filter with coefficients $b[k]$ = {-17, 29, 107, 62}

   b. IIR filter with feed-forward coefficients $b[k] = \{-17, 29, 107, 62\}$ and feed-back coefficients $a[k] = \{4, -7, -26, -15\}$

6. Where are the zeros and poles located for the following transfer functions? Are these filters stable?

   a.

$$H(z) = \frac{1}{\sqrt{2}} - \frac{1}{\sqrt{2}}z^{-1}$$

   b.

$$H(z) = \frac{\frac{1}{\sqrt{2}} - \frac{1}{\sqrt{2}}z^{-1}}{1 - z^{-1}}$$

   c.

$$H(z) = (1 + 4z^{-2})/(1 - 4z^{-1} + 3z^{-2})$$

7. Suppose we have a filter with the following transfer function. What value for $H(z)$ do we have when $z = 2e^{j\pi/6}$? What about when $z = 2e^{-j2\pi 6}$? Find these analytically, and reduce the answer to a single complex value, then write a program to demonstrate this.

$$H(z) = 1 - z^{-1} + z^{-2}$$

# Chapter 9

# The Wavelet Transform

The Discrete Wavelet Transform (DWT) is used in a variety of signal processing applications, such as video compression [28], Internet communications compression [29], object recognition [30], and numerical analysis. It can efficiently represent some signals, especially ones that have localized changes. Consider the example of representing a unit impulse function with the Fourier transform, which needs an infinite amount of terms because we are trying to represent a single quick change with a sum of sinusoids. However, the wavelet transform can represent this short-term signal with only a few terms.

This transform came about from different fields, including mathematics, physics, and image processing. Essentially, people in different areas were doing the same thing, but using different terminology. In the late 1980s, Stéphane Mallat unified the work into one topic [1].

This transform is discrete in time and scale. In other words, the DWT coefficients may have real (floating-point) values, but the time and scale values used to index these coefficients are integers. The wavelet transform is gaining popularity with the recent JPEG-2000 standard, which incorporates multiresolution analysis. This chapter explains how the wavelet transform works, and shows how the wavelet breaks a signal down into detail signals and an approximation.

We call an *octave* a level of resolution, where each octave can be envisioned as a pair of FIR filters, at least for the one-dimensional case. One filter of the analysis (wavelet transform) pair is a lowpass filter (LPF), while the other is a highpass filter (HPF), Figure 9.1. Each filter has a down-sampler after it, to make the transform efficient. Figure 9.2 shows the synthesis (inverse wavelet transform) pair, consisting of an inverse lowpass filter (ILPF) and an inverse highpass filter (IHPF), each preceded by an up-sampler. A lowpass filter produces the average signal, while a highpass filter produces the detail signal. For example, a simple lowpass filter may

have coefficients $\{\frac{1}{2}, \frac{1}{2}\}$, producing outputs $(x[n] + x[n-1])/2$, which is clearly the average of two samples. A corresponding simple highpass filter would have coefficients $\{\frac{1}{2}, -\frac{1}{2}\}$, producing outputs $(x[n] - x[n-1])/2$, half the difference of the samples. While the average signal would look much like the original, we need the details to make the reconstructed signal match the original. Multiresolution analysis feeds the average signal into another set of filters, which produces the average and detail signals at the next octave [1]. The detail signals are kept, but the higher octave averages can be discarded, since they can be recomputed during the synthesis. Each octave's outputs have only half the input's amount of data (plus a few coefficients due to the filter). Thus, the wavelet representation is approximately the same size as the original.

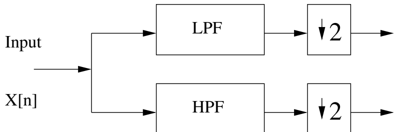



**Figure 9.1:** Analysis filters.

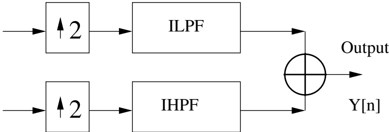



**Figure 9.2:** Synthesis filters.

Using the discrete wavelet transform on images (or other 2D data) can be accomplished with the `dwt2` and `idwt2` functions. Making your own 2D transform can be accomplished in one of two ways; either by applying the low- and highpass filters along the rows of the data, then applying each of these filters along the columns of the previous results, or by applying four matrix convolutions, one for each lowpass/highpass, horizontal/vertical combination (i.e., low-horizontal low-vertical, low-horizontal high-vertical, etc.). *Separability* means that the 2D transform is simply an application of the 1D DWT in the horizontal and vertical directions [31]. The nonseparable 2D transform works a bit differently, since it computes the transform based on a 2D sample of the input convolved with a matrix, but the results are the same. Think of this as two ways of reaching the same answer; just as we can combine two filters into one (section 8.2), we can combine horizontal and vertical operations into a matrix multiplication.

The main difference between the 1D, 2D, and 3D DWT is that a pair of filters

operates on each channel for each dimension. Figures of these should not be taken too literally, since there are more efficient ways to implement all three architectures [32].

This chapter examines how FIR filters can be used to perform the wavelet transform. First, we will look at the Haar transform, which was developed by Alfred Haar before the term “wavelet” was coined [33]. Later, we will examine the wavelet transform with the four-coefficient Daubechies wavelet [2]. The Daubechies wavelets are actually a generalization of the Haar transform, and sometimes the Haar transform is referred to as a 2-coefficient Daubechies wavelet. Daubechies actually has a family of wavelets, using the pattern we see below, except with more coefficients.

## 9.1 The Two-Channel Filter Bank

To demonstrate a general transform, we use Figure 9.3 below. Here, an input signal feeds to two channels, each with a pair of FIR filters. We call this structure a *two-channel filter bank*.

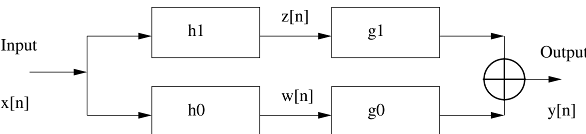



**Figure 9.3:** A two-channel filter bank.

The left half of Figure 9.3 performs the *forward transform*, also called *analysis*, while the right half corresponds to the *inverse transform*, also called *synthesis*. When we talk about a transform such as the Haar transform or the wavelet transform, we refer only to the analysis part, the left half of the figure. On the other hand, the inverse Haar transform takes place on the right half of the figure. Putting the forward and inverse transforms together, we expect that the output of the two-channel filter bank ($y[n]$) is exactly the same as the input ($x[n]$), with the possible exception of a time delay, such as $y[n] = x[n-k]$, where $k = 1$ for two coefficients.

The complementary filters of a filter bank divide the signal into subsignals of low-frequency components and one of high-frequency components. This approach is called *subband coding*.

For the inverse transform, each of the filters' outputs ($w[n]$ and $z[n]$) goes to another FIR filter, after which the two channels are recombined by addition to form $y[n]$. The idea is that $w[n]$ and $z[n]$ are a transformed version of $x[n]$, and that $y[n]$ is the signal after the inverse transform. We expect that $y[n]$ is the same as $x[n]$, assuming that we did not alter the signal along the way. If we wanted to perform

lossy compression, for example, we might alter $z[n]$ so that it can be stored more efficiently. Of course, this would mean that $y[n]$ would not be $x[n]$ exactly, but rather it would be an *approximation* of $x[n]$. The discussion here assumes that we want *perfect reconstruction*, that is, that $y[n]$ will be an *exact* copy of $x[n]$, though it may be a delayed version. To get perfect reconstruction, we have to choose our filter coefficients with care. Let's use the following values:

$$h_0 = \{a, b\}$$

$$h_1 = \{b, -a\}$$

$$g_0 = \{b, a\}$$

$$g_0 = \{-a, b\}.$$

Let's see what the signals $w[n]$ and $z[n]$ would be. This is the forward transform:

$$w[n] = ax[n] + bx[n-1]$$

$$z[n] = bx[n] - ax[n-1].$$

We will also need to know $w[n-1]$ and $z[n-1]$, which can be found by replacing $n$ with $n-1$:

$$w[n-1] = ax[n-1] + bx[n-2]$$

$$z[n-1] = bx[n-1] - ax[n-2].$$

Now we can talk about what $y[n]$ is, after the inverse transform:

$$y[n] = bw[n] + aw[n-1] - az[n] + bz[n-1].$$

However, it is interesting to see how $y[n]$ relates to the original signal, $x[n]$:

$$\begin{array}{lllll} y[n] = & b(ax[n] + bx[n-1]) & + a(ax[n-1] + bx[n-2]) & & \\ & \text{-}\ a(bx[n] - ax[n-1]) & + b(bx[n-1] - ax[n-2]) & & \\ y[n] = & abx[n] & + bbx[n-1] & + aax[n-1] & + abx[n-2] \\ & \text{-}\ abx[n] & + aax[n-1] & + bbx[n-1] & \text{-}\ abx[n-2]. \end{array}$$

Repeating the previous equation, but eliminating the parts that cancel:

$$\begin{array}{lll} y[n] = & bbx[n-1] & + aax[n-1] \\ & + aax[n-1] & + bbx[n-1] \end{array}$$

$$y[n] = (2aa + 2bb)x[n-1].$$

The Haar transform is a very simple wavelet transform. If we carefully choose our coefficients $a$ and $b$ above, we will have the Haar transform. If $2aa + 2bb = 1$, then $y[n] = x[n-1]$, meaning that the output is the same as the input, only delayed by 1. We can handle the coefficients $a$ and $b$ in one of two ways. Either we can make them both 1, which makes the filters easier to implement, and then remember to divide all $y[n]$ values by $(2 \times 1^2 + 2 \times 1^2) = 4$, or we can choose our $a$ and $b$ values to make sure that $(aa + bb) = 1$. If we force $aa = 1/2$, and $bb = 1/2$, then $a = b = 1/\sqrt{2}$. These values correspond to the Haar transform. The reason we want $(aa + bb) = 1$ and not $2(aa + bb) = 1$ has to do with the down/up-samplers, discussed later in this chapter.

## 9.2 Quadrature Mirror Filters and Conjugate Quadrature Filters

Sometimes two-channel filter banks are referred to as *subband coders*. Here, we will use the terms interchangeably, though subband coders may have more than two channels. You may also see a two-channel filter bank called a *quadrature mirror filter* (QMF), or a *conjugate quadrature filter* (CQF), though "two-channel filter bank" is the most general of these three terms. The QMF and CQF both put conditions on the filter coefficients to cancel aliasing terms and get perfect reconstruction. In other words, a quadrature mirror filter and a conjugate quadrature filter are both types of two-channel filter banks, with the same structure. The only differences between a QMF, CQF, or another two-channel filter bank are the filter coefficients.

We specify the coefficients in terms of one of the analysis filters, and we will label these coefficients $h_0$. The other filter coefficients $(h_1, g_0, g_1)$ are typically obtained from $h_0$. After all, we cannot just use any values we want for the filter coefficients or the end result will not match our original signal.

Essentially, a quadrature mirror filter specifies that $h_1$ uses the same filter coefficients as $h_0$, but negates every other value. The reconstruction filter $g_0$ is the same as $h_0$, and the filter coefficients $g_1 = -h_1$, meaning that we copy $h_1$ to $g_1$ and negate all of $g_1$'s values. See Figure 9.4 for an example using two coefficients. An interesting note about the QMF is that this does not work in general. Practically speaking, the QMF has only two coefficients (unless one uses IIR filters).

The conjugate quadrature filter specifies $h_1$ as a reversed version of $h_0$, with

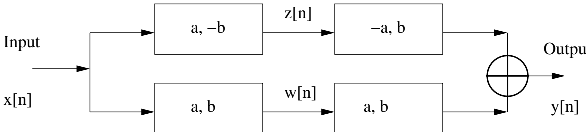



**Figure 9.4:** A quadrature mirror filter for the Haar transform.

every other value negated [34]. If $h_0 = [a, b, c, d]$, then $h_1$ will be $[d, -c, b, -a]$ [1]. For reconstruction, $g_0$ and $g_1$ are reversed versions of $h_0$ and $h_1$, respectively. Using the $h_0$ values from above, we get $g_0 = [d, c, b, a]$, and $g_1 = [-a, b, -c, d]$. Figure 9.5 shows what this looks like for two coefficients.

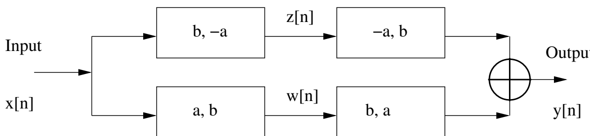



**Figure 9.5:** A conjugate quadrature filter for the Haar transform.

For two coefficients that are the same ($a = b$ in the figures), the QMF and the CQF have identical filter coefficients. The CQF is actually an improvement of the QMF, in the sense that it works for larger filters (assuming an even number of filter coefficients). Notice that both Figures 9.4 and 9.5 specify the Haar transform when $a = b$. We will use this transform often, since it has only two coefficients, but it is an example of a wavelet.

Though there are four filters, the name ("quadrature") actually comes from the fact that the frequency responses of the lowpass and highpass filters are symmetric around the "quadrature" point $\pi/2$. Consider the spectrum of a filter's response: for real signals, we show only the first half, frequencies from 0 to $\pi$. If we divide this into an ideal lowpass and highpass section, we divide it halfway again, i.e., at point $\pi/2$. The "mirror" term comes from the highpass filter's frequency magnitude response being a mirror image of the lowpass filter's frequency magnitude.

## 9.3 How the Haar Transform Is a 45-Degree Rotation

A *basis* is a projection onto a coordinate axis. If we have a point on a graph, we can talk of its distance from the origin, another point used for reference. But this

[1]MATLAB uses $h_1 = [-d, c, -b, a]$.

only narrows it down to a circle, so we need more information: the angle between the vector made by the origin to the point of interest, and another line for reference (i.e., the x-axis). For a point in 2D space, these polar coordinates are sufficient information to unambiguously locate the point with respect to the origin. But we often use the Cartesian basis {1, 0} and {0, 1} to represent this point, which tells about the point in terms of a unit vector along the x-axis and another unit vector along the y-axis (they are normalized). We know that the y-axis intersects the x-axis at the origin, and that there is a 90-degree angle between them (they are orthogonal). Though, generally speaking, we could use a different set of axes (i.e., a different basis).

The Haar basis is $\{\ 1/\sqrt{2},\ 1/\sqrt{2}\ \}$ and $\{1/\sqrt{2},\ -1/\sqrt{2}\ \}$, instead of the normal Cartesian basis. This section answers the question of what the points *look* like in the Haar-domain instead of the Cartesian-domain. What we will see is that the points have the same relationship to one another, but the coordinate system moves by a rotation of 45 degrees.

First, consider the points $(x_1, y_1), (x_2, y_2), etc.$ as the array $\{x_1, y_1, x_2, y_2, etc.\}$. The Cartesian system simply isolates the $x$'s and $y$'s, as in:

$$\begin{bmatrix} [1\ 0] & 0\ 0 \\ [0\ 1] & 0\ 0 \\ 0\ 0 & [1\ 0] \\ 0\ 0 & [0\ 1] \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \end{bmatrix}.$$

Whereas the Haar system does the following:

$$\frac{1}{\sqrt{2}} \begin{bmatrix} [1\ 1] & 0\ 0 \\ [1\ -1] & 0\ 0 \\ 0\ 0 & [1\ 1] \\ 0\ 0 & [1\ -1] \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \end{bmatrix}.$$

For example, the Cartesian coordinates $(1, 10)$ would be mapped to $\left(\frac{1+10}{\sqrt{2}}, \frac{1-10}{\sqrt{2}}\right)$, or $\left(\frac{11}{\sqrt{2}}, \frac{-9}{\sqrt{2}}\right)$ in the Haar-domain.

### 9.3.1 How The Haar Transform Affects a Point's Radius

Consider these numbers to be vectors (i.e., in polar coordinates). The magnitude and angle can be found with a little geometry. In fact, the magnitude can be found using Pythagoras' theorem. The magnitude of the Cartesian coordinates $(1, 10)$ is

$\sqrt{1^2 + 10^2} = \sqrt{101}$. The magnitude of the Haar coordinates $\left(\frac{11}{\sqrt{2}}, \frac{-9}{\sqrt{2}}\right)$ is

$$\sqrt{\left(\frac{11}{\sqrt{2}}\right)^2 + \left(\frac{-9}{\sqrt{2}}\right)^2} = \sqrt{\frac{121}{2} + \frac{81}{2}} = \sqrt{\frac{202}{2}} = \sqrt{101}.$$

This example shows that the magnitudes are the same. In other words, when we consider the point $(1, 10)$ as a distance from the origin and a rotation from the positive x-axis, the Haar transform does not affect the distance. We can show that this is the case for ANY arbitrary point by replacing $(1, 10)$ above with $(x, y)$. In the Haar-domain, we would have points $\left(\frac{x+y}{\sqrt{2}}, \frac{x-y}{\sqrt{2}}\right)$. Now, considering these points as polar coordinates, we see that the magnitude of the Cartesian coordinates is $\sqrt{x^2 + y^2}$. The magnitude of the Haar coordinates is

$$\sqrt{\left(\frac{x+y}{\sqrt{2}}\right)^2 + \left(\frac{x-y}{\sqrt{2}}\right)^2} = \sqrt{\left(\frac{x^2+2xy+y^2}{2}\right) + \left(\frac{x^2-2xy+y^2}{2}\right)}$$

$$= \sqrt{\frac{2x^2+2y^2}{2}} = \sqrt{x^2+y^2}.$$

Therefore, the Haar transform does not affect the distance of a point from the origin. It does, however, affect the angle of the point in relation to the x-axis.

### 9.3.2 How The Haar Transform Affects a Point's Angle

To see how the angle changes for the Haar transform, we will start again with the point $(1, 10)$, which we know becomes $\left(\frac{11}{\sqrt{2}}, \frac{-9}{\sqrt{2}}\right)$ in the Haar-domain. What are the angles that these points make with respect to their x-axes? The formula for finding the angles (labeled *theta*) is given below. Note that `atan` finds the arctangent.

```
if (x > 0)
   theta = atan (y/x)
elseif (x < 0)
   if (y < 0)
      theta = atan (y/x) - pi
   else
      theta = atan (y/x) + pi
   end
else
   % The vector lies on the y-axis.
```

```
    theta = sign(y)*(pi/2)
end
```

The angle made by the Cartesian coordinates $(1, 10)$ is $84.29°$. The angle made by the same coordinates after the Haar transform is $tan^{-1}\left(\frac{-9}{11}\right) = -39.29°$.

Looking at the relationship between the two in general, the $Angle_{Cartesian} = tan^{-1}\left(\frac{y}{x}\right)$, while the $Angle_{Haar} = tan^{-1}\left(\frac{(x-y)/\sqrt{2}}{(x+y)/\sqrt{2}}\right) = tan^{-1}\left(\frac{x-y}{x+y}\right)$.

The formula above is rather cumbersome to deal with, due to the ambiguous nature of the *arctan* function, and because of a potential divide-by-zero problem when $x = 0$. For the first consideration, this becomes a little easier when we replace $x$ and $y$ with their polar coordinate equivalents. That is, we will replace $x$ and $y$ with $r\cos(\theta_1)$ and $r\sin(\theta_1)$, respectively. The Haar-domain values are then $r(\cos(\theta_1) + \sin(\theta_1))/\sqrt{2}$ and $r(\cos(\theta_1) - \sin(\theta_1))/\sqrt{2}$, respectively. Finding the new angle (the Haar-domain angle, $\theta_2$), we plug the Haar-domain values from above into the equation $\theta = tan^{-1}(\frac{y}{x})$ to find:

$$\theta_2 = tan^{-1}\left(\frac{r(\cos(\theta_1) - \sin(\theta_1))/\sqrt{2}}{r(\cos(\theta_1) + \sin(\theta_1))/\sqrt{2}}\right).$$

An interesting thing to notice is that the term $r/\sqrt{2}$ cancels itself out, meaning that the radii of the polar values do not matter. In other words, the value for $\theta_2$ depends solely on $\theta_1$. This should be expected, since we already know from section 9.3.1 that the two radii are equal. So the equation for $\theta_2$ simplifies to:

$$\theta_2 = tan^{-1}\left(\frac{\cos(\theta_1) - \sin(\theta_1)}{\cos(\theta_1) + \sin(\theta_1)}\right).$$

We can divide by $\cos(\theta_1)$, giving us:

$$\theta_2 = tan^{-1}\left(\frac{\frac{\cos(\theta_1)}{\cos(\theta_1)} - \frac{\sin(\theta_1)}{\cos(\theta_1)}}{\frac{\cos(\theta_1)}{\cos(\theta_1)} + \frac{\sin(\theta_1)}{\cos(\theta_1)}}\right)$$

$$\theta_2 = tan^{-1}\left(\frac{1 - \frac{\sin(\theta_1)}{\cos(\theta_1)}}{1 + \frac{\sin(\theta_1)}{\cos(\theta_1)}}\right).$$

We then replace $\frac{\sin(\theta_1)}{\cos(\theta_1)}$ with $tan(\theta_1)$:

$$\theta_2 = tan^{-1}\left(\frac{1 - tan(\theta_1)}{1 + tan(\theta_1)}\right).$$

Since $tan(\pi/4) = 1$, we will replace the 1 in the numerator. Also in the denominator, $tan(\theta_1) = 1 \times tan(\theta_1) = tan(\pi/4)tan(\theta_1)$. This gives us:

$$\theta_2 = tan^{-1}\left(\frac{tan(\pi/4) - tan(\theta_1)}{1 + tan(\pi/4)tan(\theta_1)}\right).$$

Now comes the point of all this manipulation. There is a trigonometric identity[2] that says:

$$tan(\alpha - \beta) = \frac{tan(\alpha) - tan(\beta)}{1 + tan(\alpha)tan(\beta)}.$$

It is no coincidence that the right side of the above identity looks a lot like part of our expression for $\theta_2$. Once we use this identity, it simplifies our equation:

$$\theta_2 = tan^{-1}(tan(\pi/4 - \theta_1))$$

$$\theta_2 = \pi/4 - \theta_1.$$

This says what we expect it to; that the Haar transform is a rotation by $\pi/4$ radians, or 45 degrees.

## 9.4 Daubechies Four-Coefficient Wavelet

Let's also consider the case of four coefficients in a conjugate quadrature filter. Figure 9.6 shows a structure similar to Figure 9.5, except that its FIR filters have four taps (use four coefficients).

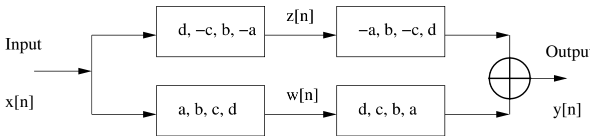



**Figure 9.6:** A two-channel filter bank with four coefficients.

The expressions for $w[n]$ and $z[n]$ can be found in a similar manner as in section 9.1:

$$w[n] = ax[n] + bx[n-1] + cx[n-2] + dx[n-3]$$
$$z[n] = dx[n] - cx[n-1] + bx[n-2] - ax[n-3].$$

[2]Thanks to Mr. Srikanth Tirupathi for pointing this out.

We will also need $w[n-1]$, $w[n-2]$, and so forth, so it is helpful to define these for an integer $k$:

$$\begin{aligned} w[n-k] = &\; ax[n-k] + bx[n-k-1] + cx[n-k-2] + dx[n-k-3] \\ z[n-k] = &\; dx[n-k] - cx[n-k-1] + bx[n-k-2] - ax[n-k-3]. \end{aligned}$$

Now we can put this all together, and express $y[n]$ in terms of $x[n]$ only:

$$\begin{aligned} y[n] = &\; dw[n] + cw[n-1] + bw[n-2] + aw[n-3] \\ &- az[n] + bz[n-1] - cz[n-2] + dz[n-3] \end{aligned}$$

$$\begin{aligned} y[n] = &\; d(ax[n] + bx[n-1] + cx[n-2] + dx[n-3]) \\ &+ c(ax[n-1] + bx[n-2] + cx[n-3] + dx[n-4]) \\ &+ b(ax[n-2] + bx[n-3] + cx[n-4] + dx[n-5]) \\ &+ a(ax[n-3] + bx[n-4] + cx[n-5] + dx[n-6]) \\ &- a(dx[n] - cx[n-1] + bx[n-2] - ax[n-3]) \\ &+ b(dx[n-1] - cx[n-2] + bx[n-3] - ax[n-4]) \\ &- c(dx[n-2] - cx[n-3] + bx[n-4] - ax[n-5]) \\ &+ d(dx[n-3] - cx[n-4] + bx[n-5] - ax[n-6]). \end{aligned}$$

Multiplying through:

$$\begin{aligned} y[n] = &\; adx[n] + bdx[n-1] + cdx[n-2] + ddx[n-3] \\ &+ acx[n-1] + bcx[n-2] + ccx[n-3] + cdx[n-4] \\ &+ abx[n-2] + bbx[n-3] + bcx[n-4] + bdx[n-5] \\ &+ aax[n-3] + abx[n-4] + acx[n-5] + adx[n-6] \\ &- adx[n] + acx[n-1] - abx[n-2] + aax[n-3] \\ &+ bdx[n-1] - bcx[n-2] + bbx[n-3] - abx[n-4] \\ &- cdx[n-2] + ccx[n-3] - bcx[n-4] + acx[n-5] \\ &+ ddx[n-3] - cdx[n-4] + bdx[n-5] - adx[n-6]. \end{aligned}$$

Now we rewrite this equation to line up the $x[n-k]$ values properly:

$$\begin{array}{lllll}
y[n] = & & & & \\
adx[n] & + bdx[n-1] & + cdx[n-2] & + ddx[n-3] & \\
 & + acx[n-1] & + bcx[n-2] & + ccx[n-3] & + cdx[n-4] \\
 & & + abx[n-2] & + bbx[n-3] & + bcx[n-4] \\
+ bdx[n-5] & & & + aax[n-3] & + abx[n-4] \\
+ acx[n-5] & + adx[n-6] & & & \\
-adx[n] & + acx[n-1] & - abx[n-2] & + aax[n-3] & \\
 & + bdx[n-1] & - bcx[n-2] & + bbx[n-3] & - abx[n-4] \\
 & & - cdx[n-2] & + ccx[n-3] & - bcx[n-4] \\
+ acx[n-5] & & & + ddx[n-3] & - cdx[n-4] \\
+ bdx[n-5] & - adx[n-6]. & & &
\end{array}$$

Repeating the above equation, only eliminating the parts that cancel:

$$\begin{array}{llll}
y[n] = & bdx[n-1] & + ddx[n-3] & \\
 & + acx[n-1] & + ccx[n-3] & \\
 & & + bbx[n-3] & + bdx[n-5] \\
 & & + aax[n-3] & + acx[n-5] \\
 & + acx[n-1] & + aax[n-3] & \\
 & + bdx[n-1] & + bbx[n-3] & \\
 & & + ccx[n-3] & + acx[n-5] \\
 & & + ddx[n-3] & + bdx[n-5].
\end{array}$$

The terms $x[n-1]$ and $x[n-5]$ are pesky, since we want each output of $y[n]$ to depend upon only one input of $x[n]$. But if $ac$ happens to equal $-bd$, then these terms will cancel each other out. Therefore, we will make this a requirement. If $ac + bd = 0$, then the $x[n-1]$ and $x[n-5]$ terms are eliminated, and we are left with:

$$y[n] = 2(aa + bb + cc + dd)x[n-3]$$

or that $y[n]$ is a delayed version of $x[n]$, multiplied by a constant. Of course, we could have a second requirement that this constant be 1. The Daubechies coefficients satisfy both of these criteria: (These are the lowpass coefficients from `db2`, obtained from MATLAB)
$a = -0.1294$
$b = 0.2241$
$c = 0.8365$
$d = 0.4830$

$ac = -0.1083$

$bd = 0.1083$.

Errors due to precision come about when using wavelets. The coefficients above are actually an approximation of the following Daubechies coefficients (obtained from [3]):

$$a = \frac{1-\sqrt{3}}{4\sqrt{2}},\ b = \frac{3-\sqrt{3}}{4\sqrt{2}},\ c = \frac{3+\sqrt{3}}{4\sqrt{2}},\ d = \frac{1+\sqrt{3}}{4\sqrt{2}}.$$

Note that the reference gives these as the highpass filter coefficients, which means that the $b$ and $d$ values above were negated. So,

$$ac = \frac{-2\sqrt{3}}{32},\ \ bd = \frac{2\sqrt{3}}{32}.$$

Therefore:

$$ac = -bd.$$

Using these coefficients, we see that:

$$a^2 = \frac{4-2\sqrt{3}}{32},\ b^2 = \frac{12-6\sqrt{3}}{32},\ c^2 = \frac{12+6\sqrt{3}}{32},\ d^2 = \frac{4+2\sqrt{3}}{32}.$$

Finding the sum $aa + bb + cc + dd$ leads to:

$$a^2 + b^2 + c^2 + d^2 = \frac{4 - 2\sqrt{3} + 12 - 6\sqrt{3} + 12 + 6\sqrt{3} + 4 + 2\sqrt{3}}{32}$$

$$a^2 + b^2 + c^2 + d^2 = \frac{4 + 12 + 12 + 4}{32}$$

$$a^2 + b^2 + c^2 + d^2 = 1.$$

This is what we should expect—that the values for $a$, $b$, $c$, and $d$ are carefully chosen so that the output $y[n]$ is just a delayed version of the input $x[n]$. But above we found that $y[n] = 2(aa + bb + cc + dd)x[n-3]$, so these coefficients mean that $y[n] = 2x[n-3]$. Why is the 2 there? The answer for this lies in the down-sampling operation. We have not yet looked at the effect of the down-sampling operation, but it gives us the same outputs, only at 1/2 the scale. That is, if we use the above Daubechies coefficients, but with down-samplers and up-samplers in place, then we get $y[n] = x[n-3]$. This is why $aa+bb+cc+dd = 1$, instead of 1/2. This is shown in section 9.5.

## 9.5 Down-Sampling and Up-Sampling

The *down-sampling* operation is easy enough to envision: only the even indexed values are kept. It squeezes the signal to half of its size. The inverse operation, *up-sampling*, stretches the signal back out, usually by inserting a 0 between each two successive samples. The effect of a down-sampler followed by an up-sampler is that every other value will be 0.

In this text, we show the down-sampler as the arrow followed by the number 2, as in the top left corner of Figure 9.7, and the up-sampler as in the top right corner of the same figure. Alternatively, some texts use the two symbols along the bottom of this figure [35], as in two inputs enter but only one leaves for the down-sampler, and one input enters but two exit the up-sampler.

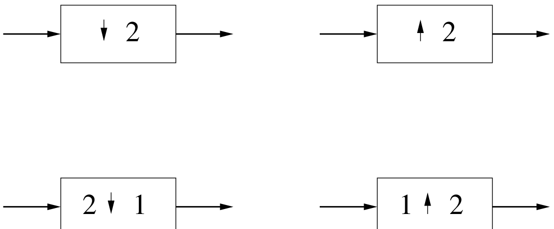



**Figure 9.7:** Different ways to indicate down-samplers and up-samplers.

### 9.5.1 Example Using Down/Up-Samplers

Intuitively, using up- and down-samplers may seem like it cannot work. How can we throw out half of our calculations after filtering without losing information? This section gives a brief example of how this process works.

Figure 9.8 shows an example of a two-channel filter bank with down-samplers followed by up-samplers. The filters are very simple, in fact, these filters would be implemented as in Figure 8.2, which reduces down to Figure 8.3.

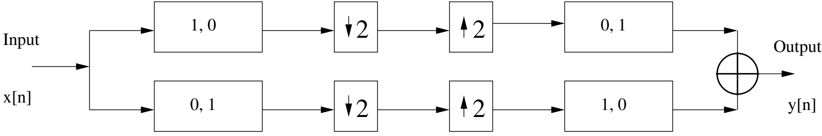



**Figure 9.8:** A simple filter bank demonstrating down/up-sampling.

Since the filter with coefficients {1, 0} allows the input to pass through to the output, we will replace it with a simple line. Likewise, we can replace the filter with

coefficients of {0, 1} with a delay unit. With these modifications in place, we have the revised filter of Figure 9.9.

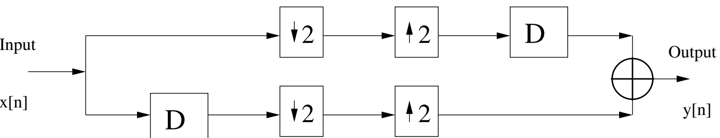



**Figure 9.9:** A simple filter bank demonstrating down/up-sampling, reduced.

Now we are ready to trace an input through to the output, with the values $x[0]$, $x[1]$, $x[2]$, $x[3]$, $x[4]$, $x[5]$, etc. Keep in mind that these inputs are listed in order from left to right, so that $x[0]$ enters the filter bank first, followed by $x[1]$, and so forth.

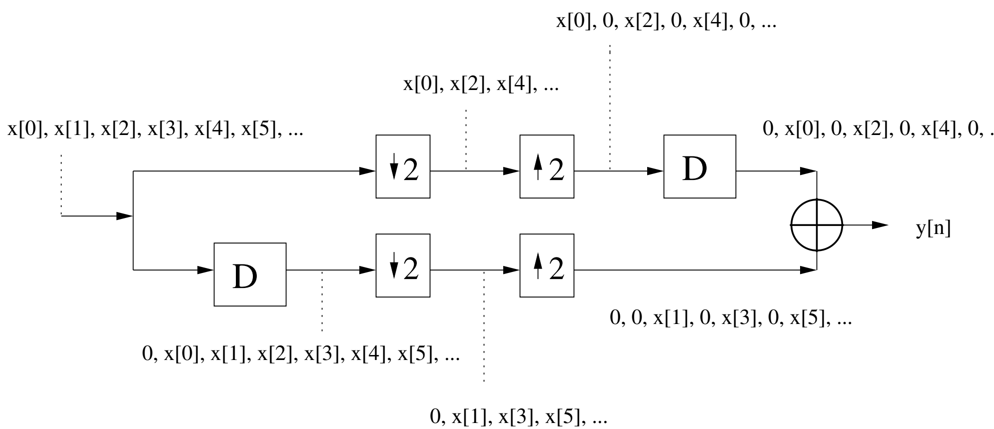



**Figure 9.10:** Tracing input to output of a simple filter bank.

Looking at the top channel of Figure 9.10, we see the input values go to the down-sampler, whose output is $x[0]$, $x[2]$, $x[4]$, etc., the even values of $x$. The forward transform ends here for this channel, but we will see what this channel contributes to the output. The up-sampler inserts zeros between values, to generate the sequence $x[0]$, 0, $x[2]$, 0, $x[4]$, 0, etc. Next comes the delay unit, which means that every input to it will appear at the output one time step later. We note this as a zero for its first output, followed by the signal that reaches it. The reason for the delay unit should be apparent when we consider the bottom channel. Thus, the top channel contributes a signal of 0, $x[0]$, 0, $x[2]$, 0, $x[4]$, 0, etc., the even values of $x$ with zeros between the values.

Now we turn our attention to the bottom channel of Figure 9.10. The delay unit shifts our inputs over one unit, so its first output is zero, followed by $x[0]$, $x[1]$, $x[2]$, etc. When this signal reaches the down-sampler, it has the effect of removing every other value. Since 0 is the first input, it passes through, while $x[0]$ is discarded. Thus, the down-sampler's output is 0, $x[1]$, $x[3]$, $x[5]$, etc. At the end of the forward transform for the bottom channel, we see that we have the odd values for $x$. The synthesis side of the QMF for the bottom channel simply up-samples the signal. This results in the pattern 0, 0, $x[1]$, 0, $x[3]$, 0, $x[5]$, etc.

Combining the top channel with the bottom channel with addition results in $0 + 0$, $x[0] + 0$, $0 + x[1]$, $x[2] + 0$, $0 + x[3]$, $x[4] + 0$, $0 + x[5]$, etc.

This demonstration shows that performing down and up-sampling on the channels of a QMF is not as crazy as it sounds. In effect, the QMF of Figure 9.8 breaks the input into its even values and its odd values, then adds them back together. This results in an output that matches the input, except for a delay.

### 9.5.2 Down-Sampling and Up-Sampling with 2 Coefficients

With wavelets, the filter bank structure will include down-samplers and up-samplers in the following configuration, Figure 9.11. In each channel, the signal undergoes several changes. To keep them distinct, they are each labeled. In the upper channel, we have $z[n]$, which is the output from the first or "analyzing" FIR filter, then $z_d[n]$ corresponds to $z[n]$ after it has been down-sampled, then $z_u[n]$ represents the signal after up-sampling, and finally $z_f[n]$ gives us the final values for that channel.

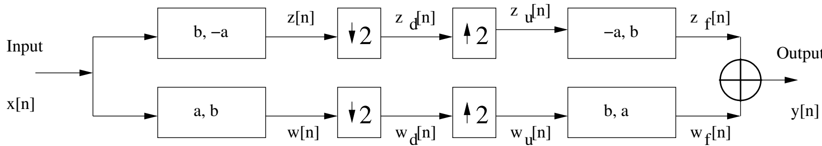



**Figure 9.11:** A two-channel filter bank with down/up-samplers.

Signals $w[n]$ and $z[n]$ are just as they were in section 9.1:

$$w[n] = ax[n] + bx[n-1]$$

$$z[n] = bx[n] - ax[n-1].$$

Signals $w[n-1]$ and $z[n-1]$, are again found by replacing $n$ above with $n-1$:

$$w[n-1] = ax[n-1] + bx[n-2]$$

$$z[n-1] = bx[n-1] - ax[n-2].$$

Looking at $w_d[n]$ and $z_d[n]$, we have to be a bit careful. We make the statements below for the even values of $n$. When $n$ is odd, there *is no value* for $w_d[n]$ or $z_d[n]$!

$$w_d[n] = w[n], \quad n \text{ is even}$$

$$z_d[n] = z[n], \quad n \text{ is even}$$

Signals $w_u[n]$ and $z_u[n]$ must be treated with equal caution.

$$w_u[n] = w_d[n] = w[n], \quad n \text{ is even}$$
$$w_u[n] = 0, \quad n \text{ is odd}$$
$$z_u[n] = z_d[n] = z[n], \quad n \text{ is even}$$
$$z_u[n] = 0, \quad n \text{ is odd}$$

We have values for $w_u[n]$ and $z_u[n]$ for even and odd values of $n$. So when using these signals for $n$ and $n-1$, one of the indices must be even, while the other is odd. The final signals of each channel are:

$$w_f[n] = bw_u[n] + aw_u[n-1]$$
$$w_f[n] = bw[n] + 0, \quad n \text{ is even}$$
$$w_f[n] = 0 + aw[n-1], \quad n \text{ is odd}$$
$$z_f[n] = -az_u[n] + bz_u[n-1]$$
$$z_f[n] = -az[n] + 0, \quad n \text{ is even}$$
$$z_f[n] = 0 + bz[n-1], \quad n \text{ is odd.}$$

When we combine $w_f[n]$ and $z_f[n]$ to form $y[n]$, we still have to concern ourselves with whether the index is even or odd.

$$y[n] = bw[n] - az[n], \quad n \text{ is even}$$
$$y[n] = aw[n-1] + bz[n-1], \quad n \text{ is odd}$$

These expressions can be expanded, to put the output $y[n]$ in terms of the original input $x[n]$.

$$y[n] = b(ax[n] + bx[n-1]) - a(bx[n] - ax[n-1]), \quad n \text{ is even}$$
$$y[n] = a(ax[n-1] + bx[n-2]) + b(bx[n-1] - ax[n-2]), \quad n \text{ is odd}$$

Simplifying:

$$y[n] = abx[n] + bbx[n-1] - abx[n] + aax[n-1], \quad n \text{ is even}$$
$$y[n] = aax[n-1] + abx[n-2] + bbx[n-1] - abx[n-2], \quad n \text{ is odd.}$$

Eliminating terms that cancel each other out, and simplifying further:

$$y[n] = (aa + bb)x[n-1], \quad n \text{ is even}$$
$$y[n] = (aa + bb)x[n-1], \quad n \text{ is odd.}$$

This last step is important, where we see that $y[n]$ is in terms of $x[n-1]$ regardless of whether $n$ is even or odd. If we compare this result to the result of section 9.1, also dealing with Haar coefficients, we see that the difference is a factor of 2. The Haar coefficients are usually given as $1/\sqrt{2}$, $1/\sqrt{2}$, so that $(aa + bb) = 1$.

### 9.5.3 Down-Sampling and Up-Sampling with Daubechies 4

This is why the coefficients from the Daubechies four-coefficient wavelet, when used without the down-sampler and up-sampler, ended up with $y[n] = 2x[n-3]$. If the down-samplers and up-samplers are included, then the 2 term drops out. This is shown later. First, we will start, as usual, with Figure 9.12, an updated filter bank for four coefficients in each filter.

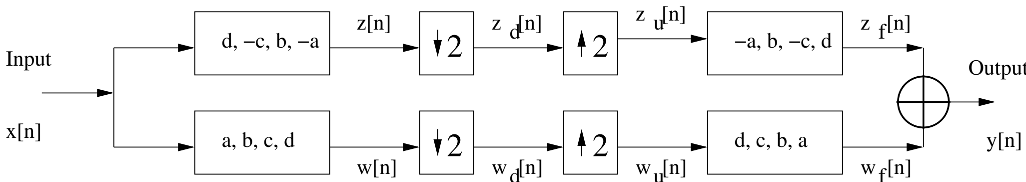



**Figure 9.12:** A filter bank with four taps per filter.

Signals $w[n]$ and $z[n]$ are just as they were earlier:

$$w[n] = ax[n] + bx[n-1] + cx[n-2] + dx[n-3]$$
$$z[n] = dx[n] - cx[n-1] + bx[n-2] - ax[n-3].$$

Signals $w[n-k]$ and $z[n-k]$, are again found by replacing $n$ with $n-k$:

$$w[n-k] = ax[n-k] + bx[n-k-1] + cx[n-k-2] + dx[n-k-3]$$
$$z[n-k] = dx[n-k] - cx[n-k-1] + bx[n-k-2] - ax[n-k-3].$$

Looking at $w_d[n]$ and $z_d[n]$, we again have to be careful about whether $n$ is even or odd. When $n$ is odd, there *is no value* for $w_d[n]$ or $z_d[n]$!

$$w_d[n] = w[n], \quad n \text{ is even}$$
$$z_d[n] = z[n], \quad n \text{ is even}$$

Signals $w_u[n]$ and $z_u[n]$ must be treated with equal caution:

$$w_u[n] = w_d[n] = w[n], \quad n \text{ is even}$$
$$w_u[n] = 0, \quad n \text{ is odd}$$
$$z_u[n] = z_d[n] = z[n], \quad n \text{ is even}$$
$$z_u[n] = 0, \quad n \text{ is odd.}$$

We have values for $w_u[n]$ and $z_u[n]$ for even and odd values of n. So when using these signals for $n$ and $n-1$, one of indices must be even, while the other is odd. Likewise, if $n$ is even, then so is $n-2$, $n-4$, etc. The final signals of each channel are:

$$w_f[n] = dw_u[n] + cw_u[n-1] + bw_u[n-2] + aw_u[n-3]$$
$$w_f[n] = dw[n] + 0 + bw[n-2] + 0, \quad n \text{ is even}$$
$$w_f[n] = 0 + cw[n-1] + 0 + aw[n-3], \quad n \text{ is odd}$$
$$z_f[n] = -az_u[n] + bz_u[n-1] - cz_u[n-2] + dz_u[n-3]$$
$$z_f[n] = -az[n] + 0 - cz[n-2] + 0, \quad n \text{ is even}$$
$$z_f[n] = 0 + bz[n-1] + 0 + dz[n-3], \quad n \text{ is odd.}$$

Keeping track of whether index $n$ is even or odd, we can find $y[n] = w_f[n] + z_f[n]$:

$$y[n] = dw[n] + bw[n-2] - az[n] - cz[n-2], \quad n \text{ is even}$$
$$y[n] = cw[n-1] + aw[n-3] + bz[n-1] + dz[n-3], \quad n \text{ is odd.}$$

These expressions can be expanded, to put the output $y[n]$ in terms of the original input $x[n]$:

$$\begin{aligned}
y[n] &= d(ax[n] + bx[n-1] + cx[n-2] + dx[n-3]) \\
&\quad +b(ax[n-2] + bx[n-3] + cx[n-4] + dx[n-5]) \\
&\quad -a(dx[n] - cx[n-1] + bx[n-2] - ax[n-3]) \\
&\quad -c(dx[n-2] - cx[n-3] + bx[n-4] - ax[n-5]), \quad n \text{ is even} \\
y[n] &= c(ax[n-1] + bx[n-2] + cx[n-3] + dx[n-4]) \\
&\quad +a(ax[n-3] + bx[n-4] + cx[n-5] + dx[n-6]) \\
&\quad +b(dx[n-1] - cx[n-2] + bx[n-3] - ax[n-4]) \\
&\quad +d(dx[n-3] - cx[n-4] + bx[n-5] - ax[n-6]), \quad n \text{ is odd.}
\end{aligned}$$

Simplifying:

$$\begin{aligned}
y[n] &= adx[n] + bdx[n-1] + cdx[n-2] + ddx[n-3] \\
&\quad +abx[n-2] + bbx[n-3] + bcx[n-4] + bdx[n-5] \\
&\quad -adx[n] + acx[n-1] - abx[n-2] + aax[n-3] \\
&\quad -cdx[n-2] + ccx[n-3] - bcx[n-4] + acx[n-5], \quad n \text{ is even} \\
y[n] &= acx[n-1] + bcx[n-2] + ccx[n-3] + cdx[n-4] \\
&\quad +aax[n-3] + abx[n-4] + acx[n-5] + adx[n-6] \\
&\quad +bdx[n-1] - bcx[n-2] + bbx[n-3] - abx[n-4] \\
&\quad +ddx[n-3] - cdx[n-4] + bdx[n-5] - adx[n-6], \quad n \text{ is odd.}
\end{aligned}$$

Eliminating terms that cancel each other out, and simplifying further:

$$\begin{aligned}
y[n] &= acx[n-1] + bdx[n-1] \\
&\quad +aax[n-3] + bbx[n-3] + ccx[n-3] + ddx[n-3] \\
&\quad +acx[n-5] + bdx[n-5], \quad n \text{ is even} \\
y[n] &= acx[n-1] + bdx[n-1] \\
&\quad +aax[n-3] + bbx[n-3] + ccx[n-3] + ddx[n-3] \\
&\quad +acx[n-5] + bdx[n-5], \quad n \text{ is odd.}
\end{aligned}$$

In this final step, we notice two things. First, the expressions are the same regardless of whether $n$ is even or odd, so now we can represent it as a single statement:

$$\begin{aligned}
y[n] &= aax[n-3] + bbx[n-3] + ccx[n-3] + ddx[n-3] \\
&\quad +acx[n-1] + bdx[n-1] + acx[n-5] + bdx[n-5].
\end{aligned}$$

Second, we see that we have $x[n-1]$ and $x[n-5]$ terms again, but that these can be eliminated if we require $ac = -bd$. Assuming this is the case, we have our final expression for $y[n]$:

$$y[n] = (aa + bb + cc + dd)x[n-3].$$

If we compare this result to the earlier case (without down-sampling) we see that the difference is a factor of 2. This explains why the sum of the squares of the Daubechies coefficients, $(aa + bb + cc + dd)$ equals 1. When that is the case, $y[n] = x[n-3]$, or $y[n]$ is simply a copy of $x[n]$, delayed by 3 time units.

## 9.6 Breaking a Signal Into Waves

Just as the discrete Fourier transform breaks a signal into sinusoids, the discrete wavelet transform generates "little waves" from a signal. These waves can then be added together to reform the signal. Figure 9.13 shows the wavelet analysis for three octaves on the left, with the standard synthesis (reconstruction) on the right. "LPF" stands for lowpass filter, while "HPF" means highpass filter. The "ILPF" and "IHPF" are the inverse low- and highpass filters, respectively. Although the down-sampling and up-sampling operations are not explicitly shown, they can be included in the filters without loss of generality. On the right, there is an implied addition when two lines meet.

Figure 9.14 shows an alternate way to do the reconstruction. While not as efficient as Figure 9.13, it serves to demonstrate the contribution of each channel.

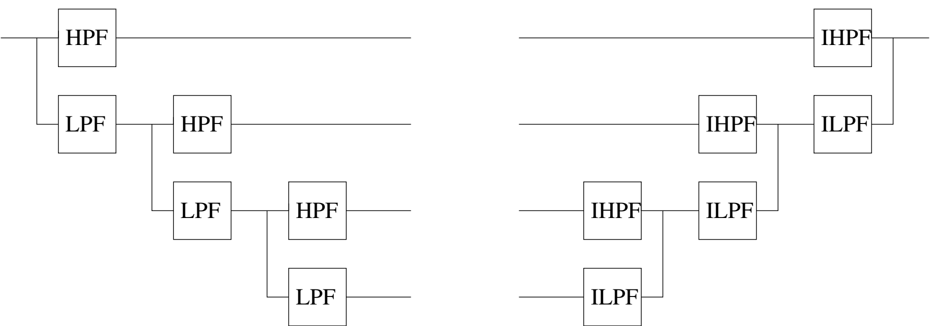



**Figure 9.13:** Wavelet analysis and reconstruction.

In the end, we have four signals: three detail signals, and one level-3 approximation.

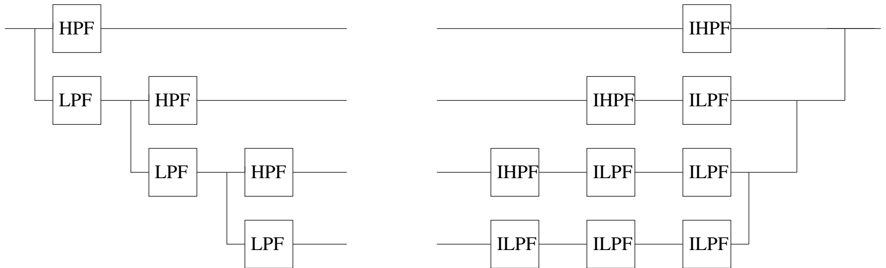



**Figure 9.14:** Alternate wavelet reconstruction.

This section rests on the argument that we can rearrange the operations in the reconstruction, i.e., that we can perform the up-sampling and filtering first, and then do the addition. We know that the ILPF and IHPF are made with FIR filters that have constant filter coefficients and are, therefore, linear. Thus, we know that it does not matter whether we add two signals together and filter their sum, or filter the two then add the results together. But what if we have down and up-samplers in the analysis and reconstruction? Specifically for our purposes here, does it matter if we up-sample two signals first, then add them together, as opposed to adding them together, then up-sampling their sum? For simplicity, let's call the two signals $x$ and $y$, and consider only the first four samples of each. If we add these signals, we get:

$$[x_0 + y_0, x_1 + y_1, x_2 + y_2, x_3 + y_3].$$

Next, we up-sample this result to get:

$$[x_0 + y_0, 0, x_1 + y_1, 0, x_2 + y_2, 0, x_3 + y_3, 0].$$

Now consider what happens if we up-sample each signal first, then add them. After up-sampling them, we have:

$$[x_0, 0, x_1, 0, x_2, 0, x_3, 0]$$

and

$$[y_0, 0, y_1, 0, y_2, 0, y_3, 0].$$

If we add them together now, we get:

$$[x_0 + y_0, 0, x_1 + y_1, 0, x_2 + y_2, 0, x_3 + y_3, 0].$$

We see that this exactly matches the result from adding first, then up-sampling. Also, the above argument works for arbitrary lengths of $x$ and $y$. Also note that we can recursively apply this reasoning and conclude that the reconstructions shown in Figures 9.13 and 9.14 are equivalent.

In Figure 9.15, we see the four signals that, when added together, reform the impulse function. Figure 9.16 shows another four waves that reform the impulse function when added. The difference between these two is the wavelet used; in Figure 9.15, we used the Haar wavelet, while we used the Daubechies-2 (`db2`) wavelet in Figure 9.16. Both of these figures show how the DWT represents the input signal as shifted and scaled versions of the wavelet and scaling functions. Since we use an impulse function, the wavelet function appears in the "High wave" signals, while the scaling function appears in the "Low wave" signals (or the "Low-Low-Low wave" signals in the case of these two figures).

It may be difficult to see from the figure that these four signals do cancel each other out. To make this easier, Table 9.1 shows the actual values corresponding to the plots shown in Figure 9.15. The impulse signal used has 100 values, all of which are zero except for a value of 1 at offset 50. Rather than give all values in the table, we leave out the zeros and just show the values in the range 49 to 56. As the reader can verify, all columns of this table add to zero except for the second column. Belonging to the offset 50, the sum of this column is 1 just as the original impulse signal has a 1 at offset 50.

Figure 9.17 shows the input (impulse function) versus the output. As expected, the output signal exactly matches the input signal.

A more typical example is seen in Figure 9.18, a signal chosen because it clearly shows the Haar wavelets in the first three subplots. (The example array {9, −2, 5, −2, 7, −6, 4, −4} was used here.) The top plot shows four repetitions of the Haar wavelet, each of a different scale. The second subplot shows two repetitions, but these are twice as long as the ones above it. In the third subplot, one Haar

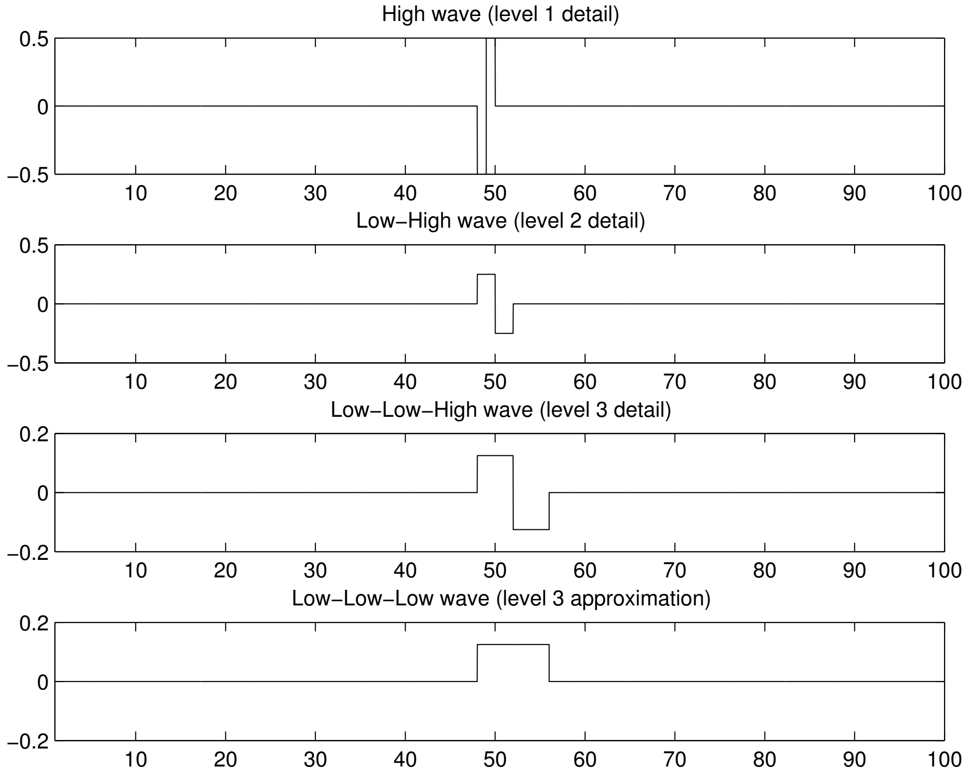



**Figure 9.15:** Impulse function analyzed with Haar.

**Table 9.1:** Impulse function analyzed with Haar.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| -0.5000 | 0.5000 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0.2500 | 0.2500 | -0.2500 | -0.2500 | 0 | 0 | 0 | 0 |
| 0.1250 | 0.1250 | 0.1250 | 0.1250 | -0.1250 | -0.1250 | -0.1250 | -0.1250 |
| 0.1250 | 0.1250 | 0.1250 | 0.1250 | 0.1250 | 0.1250 | 0.1250 | 0.1250 |

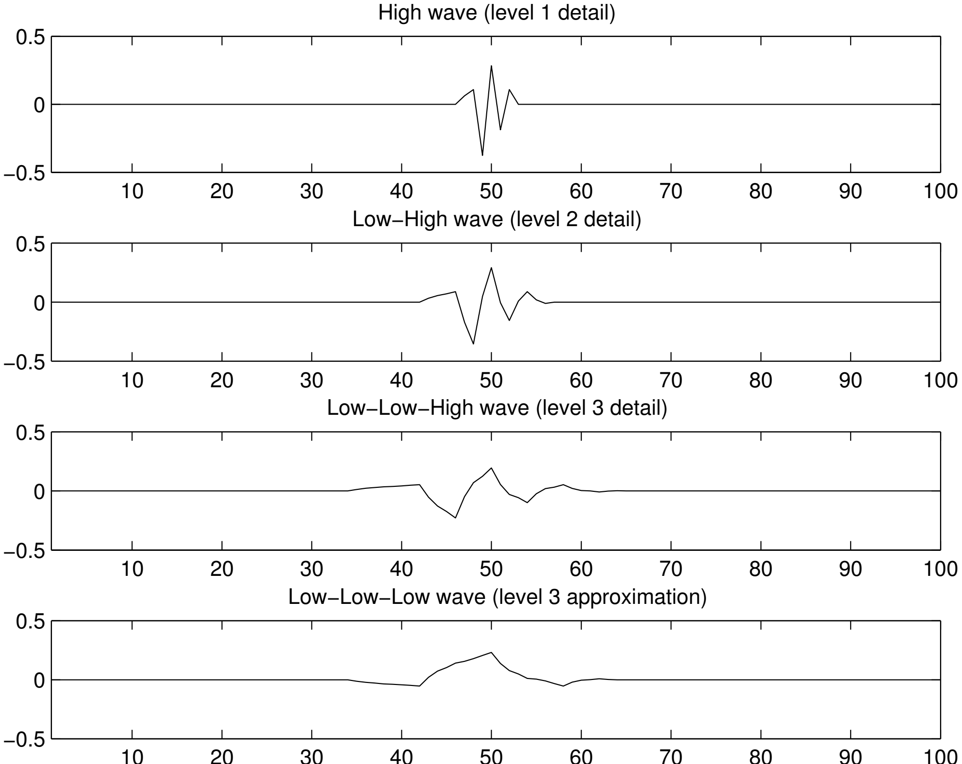



**Figure 9.16:** Impulse function analyzed with Daubechies-2.

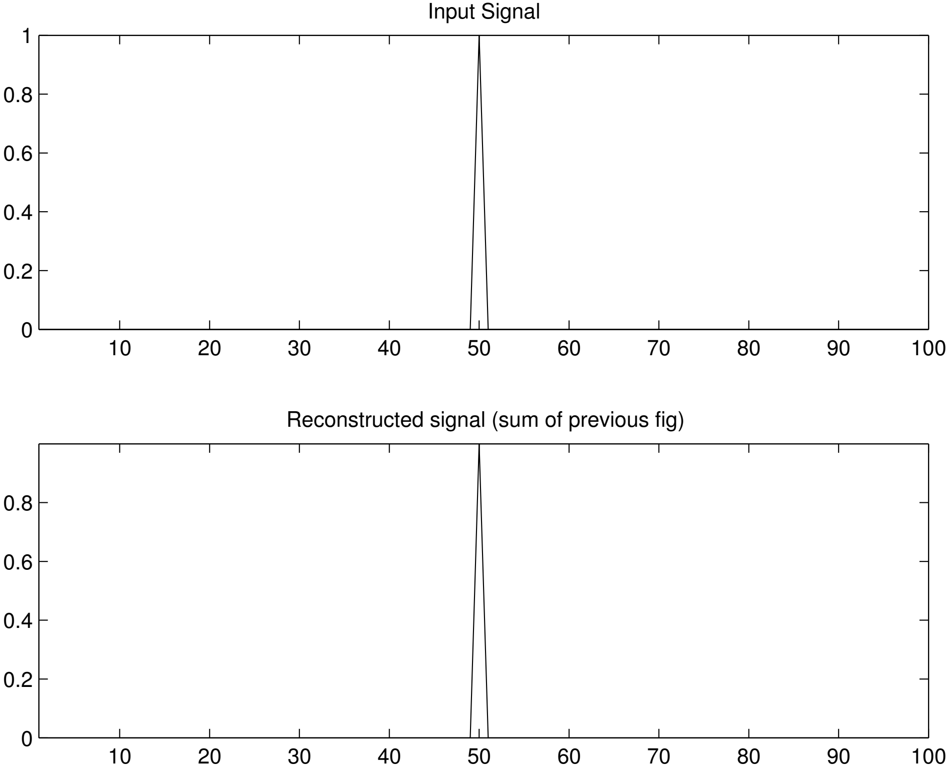



**Figure 9.17:** Impulse function (original) and its reconstruction.

wavelet is seen, twice the length of the wavelets above it. In these subplots, we can clearly see the wavelet shifted (along the x-axis) and scaled (subplots 2 and 3 are products of both the wavelet and the scaling function). The final subplot contains the approximation, clearly the result of several applications of the scaling function.

All of the wavelet-domain values for this particular signal are positive. This means that all of the wavelets are easy to spot. With a negative scaling value, the corresponding Haar wavelet would appear flipped around the x-axis.

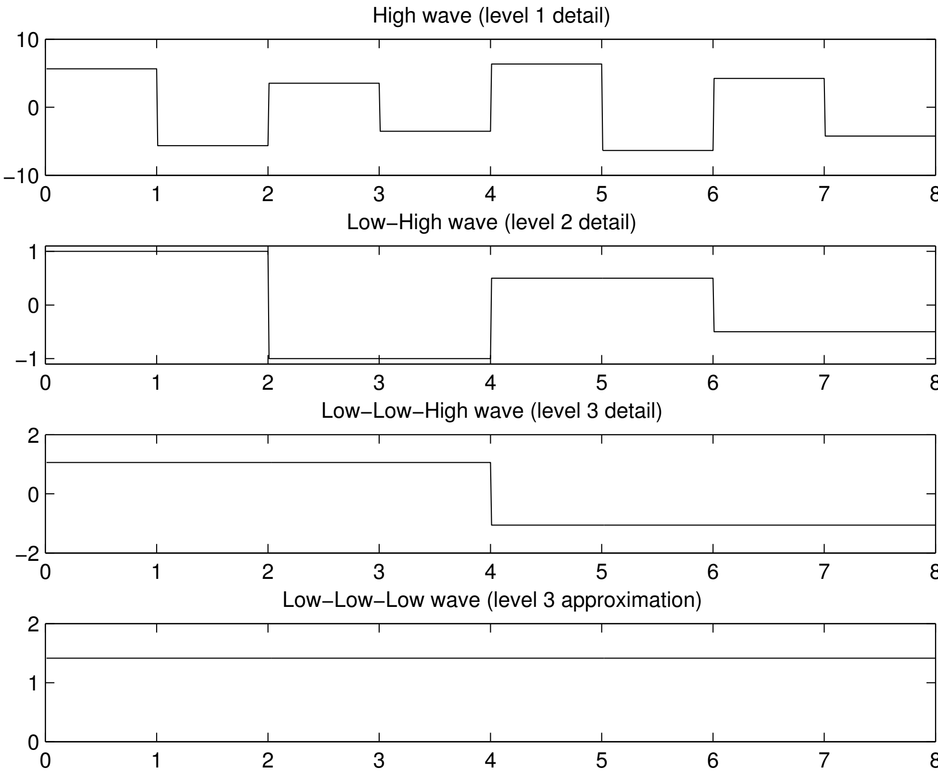



**Figure 9.18:** Example function broken down into 3 details and approximation.

An interesting comparison is between the DWT and DFT for the impulse function. Notice how the impulse function results in waves that are compact. Almost all of the wave values outside of the range 40:60 are zero. The Fourier transform, however, needs many more values to represent such a signal. Since the impulse function is well localized in time, it is spread out over the frequency-domain.

The following code shows an example run of the DWT waves program.

```
input = zeros(1,100);
input(50) = 1;
addwaves3(input,'db2')
```

If we use similar code to examine the Fourier transform, we would have something like the code below.

```
input = zeros(1,100);
input(50) = 1;
X = fft(input);
half = 1:ceil(length(X)/2);
subplot(2,1,1); plot(half, abs(X(half)), 'b');
title('Fourier Transform of impulse function (magnitudes)');
subplot(2,1,2); plot(half, angle(X(half)), 'r');
title('Fourier Transform of impulse function (phase)');
```

Figure 9.19 shows the results of this code. Notice how all of the magnitudes are 1, and all of the phase angles are different. This means that we have to use 50 different sinusoids with 50 different phase angles to encode the impulse function. We could store the 50 magnitudes compactly, since they are all the same value, but this is still a lot of information to remember. Note that the number 50 is specific to this example, and comes directly from the original signal being a real-valued impulse of 100 points. If we repeat this experiment with, say, 1000 points (all except one of which are 0), we would end up with 500 sinusoids to keep track of. To make this comparison fair, we will examine the DWT of the impulse function in wavelet space, that is, after the analysis (left half of Figure 9.13). Thus, we will use the following code.

```
input = zeros(1,100);
input(50) = 1;
[LPF, HPF, ILPF, IHPF] = myDB2;
% Do the wavelet transform
L = downsample(myconv2(input, LPF));
H = downsample(myconv2(input, HPF));
LL = downsample(myconv2(L, LPF));
LH = downsample(myconv2(L, HPF));
%[LLL, LLH] = myDWT(LL);
LLL = downsample(myconv2(LL, LPF));
```

```
LLH = downsample(myconv2(LL, HPF));
clear L LL
subplot(4,1,1); plot(1:length(H), H, 'b');
title('Impulse: DWT Highpass values');
subplot(4,1,2); plot(1:length(LH), LH, 'b');
title('Impulse: DWT Low-High values');
subplot(4,1,3); plot(1:length(LLH), LLH, 'b');
title('Impulse: DWT Low-Low-High values');
subplot(4,1,4); plot(1:length(LLL), LLL, 'b');
title('Impulse: DWT Low-Low-Low values');
length(H) + length(LH) + length(LLH) + length(LLL)

ans =

   108
```

As we see from Figure 9.20, most of wavelet-domain values are zero. From the addition of lengths, the total number of values that we have to keep is about the same for the DWT as for the FFT (108 versus 100). But since most of the wavelet-domain values are zero, we can store this information much more compactly. In fact, examining the wavelet-domain values reveals that only eight are nonzero! From this we can conclude that *some signals* can be stored more efficiently in the wavelet-domain than in the frequency-domain. Of course, it depends on the signal; those well localized in time will have a more efficient wavelet representation, and those well localized in frequency will have a more efficient frequency representation.

Another example of the DWT versus the FFT shows the opposite extreme. If we let the input function be a pure sinusoid, and repeat the above analysis, we see that it is easy to represent in the Fourier-domain.

```
t=0:0.01:.99; input = cos(2*pi*10*t + pi/4);
```

After performing the FFT on this signal, we see that there is only one nonzero magnitude. While the FFT will return phase angles that are nonzero, only the one that corresponds to the nonzero magnitude is significant. Thus, we have only two values to remember to reconstruct this signal from the frequency-domain. Looking at the same signal in the wavelet-domain, we find that all 108 values are nonzero. This confirms what we suspected above: that the optimal representation of a signal depends upon the signal's characteristics. Both the Fourier transform and the wavelet transform have a place in signal processing, but signals will likely be represented more efficiently by one transform than the other.

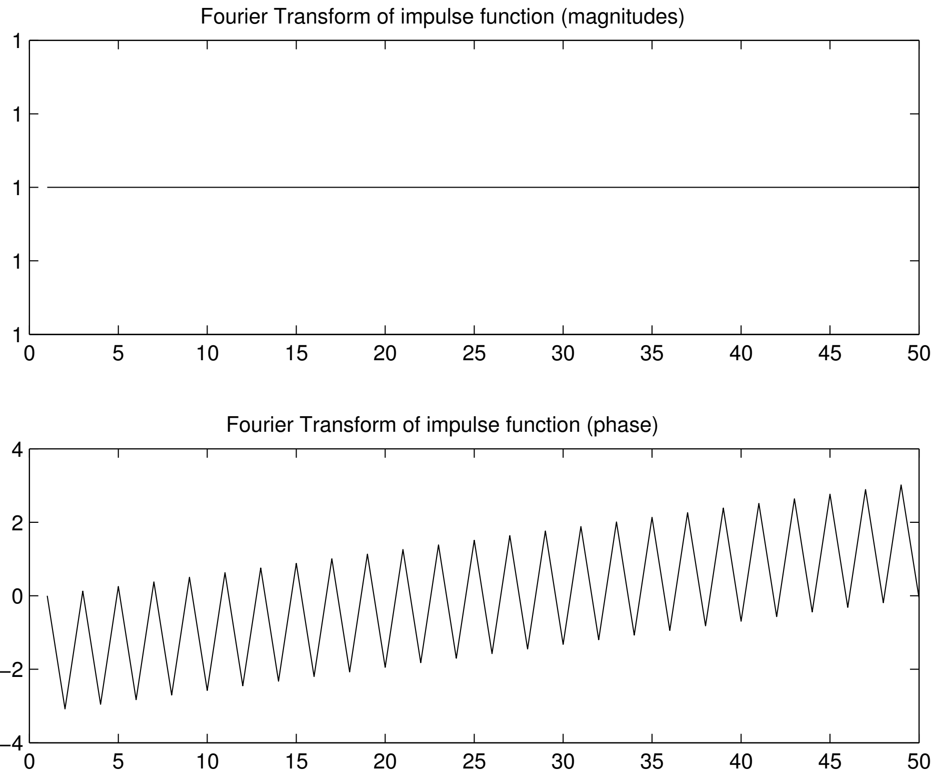



**Figure 9.19:** Impulse function in Fourier-domain.

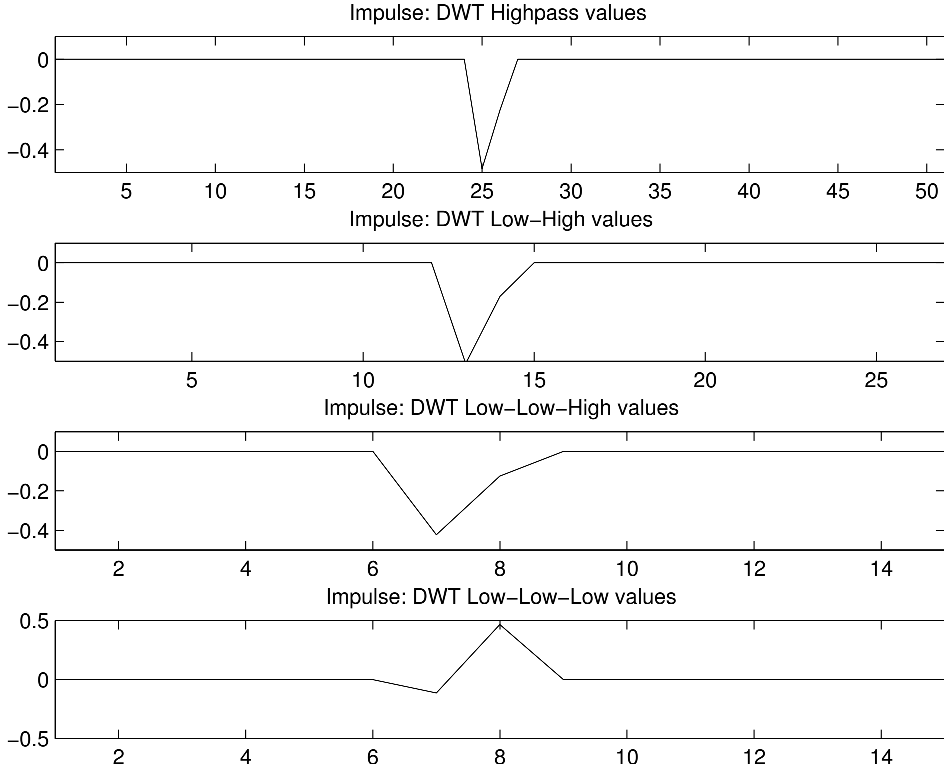



**Figure 9.20:** Impulse function in Wavelet-domain.

The following program created the graphs of waves above.

```
%
% Given an input signal, break it down with DWT.
% Show the contribution of several channels of the DWT.
%
%  example:  input = floor(rand(1,100)*256);
%            addwaves3(input, 'db2');
% Works for 'haar' and 'db2' (Haar is the default)
%
function addwaves3(input, wavelet)

% Get the Daubechies2 wavelet/scaling coeffs
if (strcmp(wavelet,'db2'))
   [LPF, HPF, ILPF, IHPF] = myDB2;
else
   [LPF, HPF, ILPF, IHPF] = myHaar;
end

% Do the wavelet transform
L = downsample(myconv2(input, LPF));
H = downsample(myconv2(input, HPF));
%[LL, LH] = myDWT(L);
LL = downsample(myconv2(L, LPF));
LH = downsample(myconv2(L, HPF));
%[LLL, LLH] = myDWT(LL);
LLL = downsample(myconv2(LL, LPF));
LLH = downsample(myconv2(LL, HPF));

% L is not needed anymore, nor is LL
clear L LL

% Get the waves for each subsignals contribution
% This effectively finds the reconstruction.
if (strcmp(wavelet,'db2'))
    % The ILow and IHigh functions are hard-coded for db2.
    wave1 = ILow(ILow(ILow(LLL)));
    wave2 = ILow(ILow(IHigh(LLH)));
    wave3 = ILow(IHigh(LH));
    wave4 = IHigh(H);
```

```
else
    % The ILowHaar, IHighHaar functions are hard-coded for haar.
    wave1 = ILowHaar(ILowHaar(ILowHaar(LLL)));
    wave2 = ILowHaar(ILowHaar(IHighHaar(LLH)));
    wave3 = ILowHaar(IHighHaar(LH));
    wave4 = IHighHaar(H);
end


% The signal above needs to be padded (made longer).
%  First, find the longest length.
max_len = max(max(length(wave1), length(wave2)), ...
              max(length(wave3), length(wave4)));
% Now adjust the signals, as needed
if (length(wave1) < max_len)
    wave1(length(wave1)+1:max_len) = 0;
end
if (length(wave2) < max_len)
    wave2(length(wave2)+1:max_len) = 0;
end
if (length(wave3) < max_len)
    wave3(length(wave3)+1:max_len) = 0;
end
if (length(wave4) < max_len)
    wave4(length(wave4)+1:max_len) = 0;
end

% Add all the waves together
waves = wave1 + wave2 + wave3 + wave4;

% Make the input the same length as the output
% do this by truncating the output
output = waves(1:length(input));

disp(sprintf('Total error between input and output: %10.9f.', ...
    sum(abs(input - output))));


% Now plot the results.
```

```
figure(1);
subplot(4,1,1);
plot(wave4);
%plot100to1(wave4);
title('High wave (level 1 detail)');
subplot(4,1,2);
plot(wave3);
%plot100to1(wave3);
title('Low-High wave (level 2 detail)');
subplot(4,1,3);
plot(wave2);
%plot100to1(wave2);
title('Low-Low-High wave (level 3 detail)');
subplot(4,1,4);
plot(wave1);
%plot100to1(wave1);
title('Low-Low-Low wave (level 3 approximation)');

figure(2);
subplot(2,1,1);
plot(input);
%plot100to1(input);
title('Input Signal');
subplot(2,1,2);
%plot(1:length(output),output,'b',1:length(input),input,'r');
plot(output);
%plot100to1(output);
title('Reconstructed signal (sum of previous fig)');
```

The program calls either the built-in `plot` function, or the special `plot100to1` program for the output figure, depending on which lines are commented out. The `plot100to1` program makes the Haar wavelets look their best, since it plots 100 points for every 1 original point, as the name implies. This results in a graph that has sharper transitions, such as shown in Figure 9.15.

## 9.7 Wavelet Filter Design—Filters with Four Coefficients

Here we will examine designing a general filter bank with four taps per filter, very similar to that in the last section. However, one difference is that we use the $z$-transform in this section, which actually makes the analysis a bit easier. Figure 9.21 shows what this would look like. We specify values $a, b, c, d$ and $d, c, b, a$ for coefficients in one channel, with the understanding that the coefficients for the other channel are a reversed and sign-changed version of these. That is, in the other channel we would have the coefficients $d, -c, b, -a$ and $-a, b, -c, d$, respectively.

We will use $H_0(z)$ and $H_1(z)$ for the lowpass and highpass filters' transfer functions, respectively, while $F_0(z)$ and $F_1(z)$ are for the corresponding inverse filters. $H_0(z)$ and $F_0(z)$ go with the bottom channel in Figure 9.21.

Examining the individual filter's responses, we see

$$\begin{aligned}
F_0(z) &= d + cz^{-1} + bz^{-2} + az^{-3} \\
H_0(z) &= a + bz^{-1} + cz^{-2} + dz^{-3} \\
F_1(z) &= -H_0(-z) = -a + bz^{-1} - cz^{-2} + dz^{-3} \\
H_1(z) &= F_0(-z) = d - cz^{-1} + bz^{-2} - az^{-3}.
\end{aligned}$$

Using $P_0(z)$ and $P_0(-z)$ notation to represent the product of two sequential filters [3], for the top channel and the bottom channel, respectively, we get the following equations for a CQF with four coefficients per filter.

$$\begin{aligned}
P_0(z) &= H_0(z)F_0(z) \\
P_0(z) &= (a + bz^{-1} + cz^{-2} + dz^{-3})(d + cz^{-1} + bz^{-2} + az^{-3}) \\
&= ad + acz^{-1} + abz^{-2} + aaz^{-3} \\
&\quad + bdz^{-1} + bcz^{-2} + bbz^{-3} + baz^{-4} \\
&\quad + cdz^{-2} + ccz^{-3} + cbz^{-4} + caz^{-5} \\
&\quad + ddz^{-3} + dcz^{-4} + dbz^{-5} + daz^{-6} \\
P_0(-z) &= ad - (ac + bd)z^{-1} + (ab + bc + cd)z^{-2} \\
&\quad - (aa + bb + cc + dd)z^{-3} + (ab + bc + cd)z^{-4} \\
&\quad - (ac + bd)z^{-5} + adz^{-6} \\
P_0(z) - P_0(-z) &= 2(ac + bd)z^{-1} + 2(aa + bb + cc + dd)z^{-3} + 2(ac + bd)z^{-5}
\end{aligned}$$

Note that $-P_0(-z) = H_1(z)F_1(z)$, but we do not have to calculate this explicitly. Instead, we just repeat the expression for $P_0$ with $-z$ as the argument. As we saw in

section 8.7, $-z$ raised to a negative power gives a negative value for an odd power, but returns a positive value for an even one.

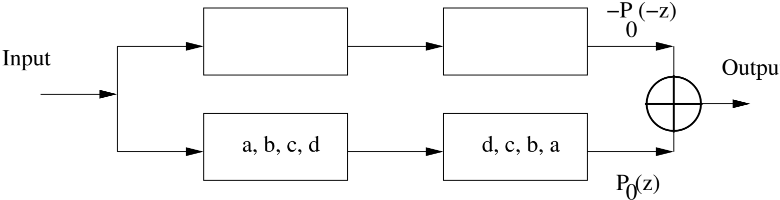



**Figure 9.21:** Designing a filter bank with four taps each.

That is, we can determine the filters' responses and see how the terms cancel each other out. Then we can divide by two, which is the same effect that down/up-samplers have.
Without down/up-samplers:

$$P_0(z) - P_0(-z) = 2(ac + bd)z^{-1} + 2(aa + bb + cc + dd)z^{-3} + (ac + bd)z^{-5} = 2z^{-L}.$$

With down/up-samplers:

$$P_0(z) - P_0(-z) = (ac + bd)z^{-1} + (aa + bb + cc + dd)z^{-3} + (ac + bd)z^{-5} = z^{-L}.$$

We are left with the output in terms of three delayed versions of the input: $z^{-1}$, $z^{-3}$, and $z^{-5}$. Two of these must be eliminated, which can be accomplished by setting $ac + bd = 0$. Also, we do not want the result to be a scaled version of the input, so $aa + bb + cc + dd = 1$. It is also desirable that the sum of the highpass filter coefficients be zero [3], i.e., $d - c + b - a = 0$, for this to be a wavelet transform.

The question now becomes, can we find values for $a, b, c$, and $d$ to satisfy these equations?

$$\begin{aligned} ac + bd &= 0 \\ a^2 + b^2 + c^2 + d^2 &= 1 \\ d - c + b - a &= 0 \end{aligned}$$

One possibility is to use the Daubechies wavelet filter coefficients, since they satisfy these criteria.

## 9.8 Orthonormal Bases

The wavelet transforms above form orthonormal bases, meaning that they are both orthogonal and normalized. We saw above that normalization means that the output of a CQF is a delayed version of the input, instead of a scaled version. We can show this as the inner product of the lowpass filter with itself (or with the highpass filter with itself), as in $< lpf, lpf >$. Given that the lowpass filter coefficients are $[a, b, c, d]$, we calculate the inner product as the multiplication of the first parameter with the transpose of the complex conjugate of the second one. Here, both parameters are the same and real-valued, and we get $a^2 + b^2 + c^2 + d^2$, which must be 1 for the transform to be normalized.

Orthogonality, in two dimensions, simply means that the components are at right angles. It goes without saying in the Cartesian coordinate system that a point $(x, y)$ can be plotted by moving $x$ units to the right (or left if $x$ is negative), and then moving $y$ units up or down (up/down being 90 degrees away from right/left) to find the point's location. The bases for the Cartesian point $(x, y)$ are $(1, 0)$ and $(0, 1)$. If we find the inner product, $< [1\ 0], [0\ 1] >$, we see that it results in $0 \times 1 + 1 \times 0 = 0$. We define *orthogonality* as the attribute that the inner product of the bases equals zero, which holds for higher dimensions as well.

Since the lowpass filter and highpass filter of the wavelet transform form an orthogonal basis, the signal can be broken up into separate parts. Even with down-sampling, the two channels contain enough information to reconstruct the whole signal, just as a filter bank that breaks the signal into odd and even components does.

The program below demonstrates, for several wavelets, that the filter coefficients are both orthogonal and normal. We assume that all coefficient values are real, thus there is no need to find the complex conjugates.

```
%
%  Show that scaling/wavelet coefficients form
%  a basis, i.e., lpf * hpf.' = 0, and lpf * lpf.' = 1
%

lpf = [1 1] * (1/sqrt(2));
hpf = [-1 1]* (1/sqrt(2));
disp(' finding the inner product of Haar wavelet/wavelet');
disp(' it should be one');
hpf * hpf.'
disp(' now find the inner product of Haar scaling/scaling');
disp(' it should be one, too.');
```

```
lpf * lpf.'
disp(' now find the inner product of Haar scaling/wavelet');
disp(' it should be zero.');
lpf * hpf.'
```

First, we can run this code to verify the Haar basis.

```
 finding the inner product of Haar wavelet/wavelet
 it should be one

ans =

    1.0000

 now find the inner product of Haar scaling/scaling
 it should be one, too.

ans =

    1.0000

 now find the inner product of Haar scaling/wavelet
 it should be zero.

ans =

     0
```

As expected, this works for the Haar basis.

Now we can repeat this for the Daubechies four-coefficient wavelet. We label the filters $lpf2$ and $hpf2$ to keep them separate from the lowpass filter and highpass filter coefficients that we used above.

```
a = (1-sqrt(3))/(4*sqrt(2));
b = (3-sqrt(3))/(4*sqrt(2));
c = (3+sqrt(3))/(4*sqrt(2));
d = (1+sqrt(3))/(4*sqrt(2));

lpf2 = [a b c d];
hpf2 = [-d c -b a];
```

```
ans2a = hpf2 * hpf2.';
disp(sprintf(' Daubechies wavelet/wavelet = %6.4f', ans2a));
ans2b = lpf2 * lpf2.';
disp(sprintf(' Daubechies scaling/scaling = %6.4f', ans2b));
ans2c = lpf2 * hpf2.';
disp(sprintf(' Daubechies scaling/wavelet = %6.4f', ans2c));
```

Running this code produces:

```
 Daubechies wavelet/wavelet = 1.0000
 Daubechies scaling/scaling = 1.0000
 Daubechies scaling/wavelet = 0.0000
```

Thus, the code above confirms the four-coefficient Daubechies wavelet.

Below is code to repeat this experiment for several other wavelets. However, instead of specifying the filter coefficients ourselves, we will rely on built-in wavelet functions.

```
try
   [lpf3, hpf3, lo_recon, hi_recon] = wfilters('db44');
catch
   disp('You must have the wavelets toolkit for the others');
   break;
end


ans3a = hpf3 * hpf3.';
disp(sprintf(' Daubechies44 wavelet/wavelet = %6.4f', ans3a));
ans3b = lpf3 * lpf3.';
disp(sprintf(' Daubechies44 scaling/scaling = %6.4f', ans3b));
ans3c = lpf3 * hpf3.';
disp(sprintf(' Daubechies44 scaling/wavelet = %6.4f', ans3c));

[lpf4, hpf4, lo_recon, hi_recon] = wfilters('coif1');
ans4a = hpf4 * hpf4.';
disp(sprintf(' Coiflets1 wavelet/wavelet = %6.4f', ans4a));
ans4b = lpf4 * lpf4.';
disp(sprintf(' Coiflets1 scaling/scaling = %6.4f', ans4b));
ans4c = lpf4 * hpf4.';
disp(sprintf(' Coiflets1 scaling/wavelet = %6.4f', ans4c));
```

```
[lpf5, hpf5, lo_recon, hi_recon] = wfilters('sym8');
ans5a = hpf5 * hpf5.';
disp(sprintf(' Symlets8 wavelet/wavelet = %6.4f', ans5a));
ans5b = lpf5 * lpf5.';
disp(sprintf(' Symlets8 scaling/scaling = %6.4f', ans5b));
ans5c = lpf5 * hpf5.';
disp(sprintf(' Symlets8 scaling/wavelet = %6.4f', ans5c));

[lpf6, hpf6, lo_recon, hi_recon] = wfilters('dmey');
ans6a = hpf6 * hpf6.';
disp(sprintf(' Discrete Meyers wavelet/wavelet = %6.4f', ans6a));
ans6b = lpf6 * lpf6.';
disp(sprintf(' Discrete Meyers scaling/scaling = %6.4f', ans6b));
ans6c = lpf6 * hpf6.';
disp(sprintf(' Discrete Meyers scaling/wavelet = %6.4f', ans6c));
```

The output of this program (using the student version of MATLAB, to show the `try` and `catch` commands) appears below.

```
EDU>> innerproducts
You must have the wavelets toolkit for the others
EDU>>
```

One final note about the inner products is that the inner product of a signal (vector) with itself gives the length of the vector, squared.

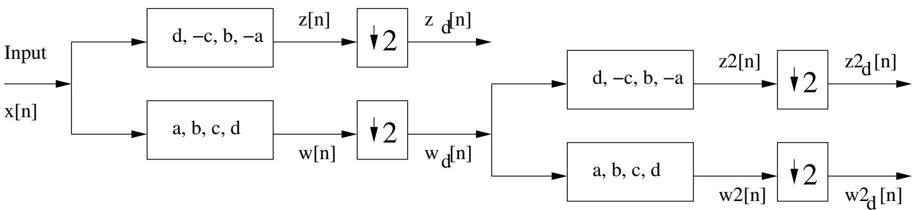
$$|x|^2 = < x, x >$$

If $x = \{x_0, x_1\}$, we know its inner product: $< x, x >= x_0^2 + x_1^2$. We also know its length is $|x| = \sqrt{x_0^2 + x_1^2}$. Using these two pieces of information, it is easy to verify that we can define the vector's length in terms of an inner product, as above. Signal $x$'s support does not matter, in that we define the length as the square root of the inner product of $x$ with itself, even when $x$ has more than two values. Some texts [36] also call this length the *norm*.

## 9.9 Multiresolution

The above discussion is for a single pair of analysis filters. *Multiresolution* is the process of taking the output from one channel and putting it through another (or

more) pair of analysis filters. For the wavelet transform, we do this with the lowpass filter's output. Wavelet packets, however, use an additional filter pair for each channel. We will examine multiresolution starting with the Daubechies four-coefficient wavelet transform, with down-sampling and up-sampling, for two levels of resolution (octaves), as shown in Figure 9.22.



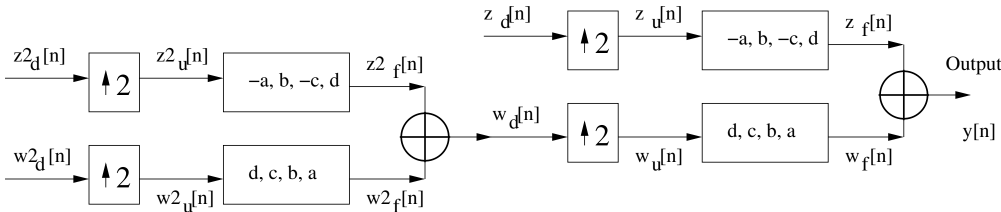




**Figure 9.22:** Two levels of resolution.

Signals $w[n]$, $w_d[n]$, $z[n]$, and $z_d[n]$ are the same as before.

$$w[n] = ax[n] + bx[n-1] + cx[n-2] + dx[n-3]$$

$$z[n] = dx[n] - cx[n-1] + bx[n-2] - ax[n-3]$$

$$w_d[n] = w[n], \quad n \text{ is even}, 0 \text{ otherwise}$$

$$z_d[n] = z[n], \quad n \text{ is even}, 0 \text{ otherwise}$$

To generate $w2[n]$, $w2_d[n]$, $z2[n]$, and $z2_d[n]$, we first notice that they have the same relationship to $w_d[n]$ as $w_d[n]$ and $z_d[n]$ have to $x[n]$. In other words, we can reuse the above equations, and replace $x[n]$ with $w_d[n]$. Also, since there is a down-sampler between $w2[n]$ and $w2_d[n]$, every other value of $w2[n]$ will be eliminated. Signal $w2[n]$, by definition, is based upon $w_d[n]$, a down-sampled version of $w[n]$. Using the original $n$, therefore, means that we must say that every other value from the even values of $n$ will be eliminated. In other words, the only values that will get

through the second down-sampler are the ones where $n$ is evenly divisible by 4.

$$w2[n] = aw_d[n] + bw_d[n-1] + cw_d[n-2] + dw_d[n-3]$$
$$z2[n] = dw_d[n] - cw_d[n-1] + bw_d[n-2] - aw_d[n-3]$$
$$w2_d[n] = w2[n], \quad n \text{ is divisible by } 4$$
$$z2_d[n] = z2[n], \quad n \text{ is divisible by } 4$$

Looking now at the reconstruction (synthesis) part, $w2_u[n]$ and $z2_u[n]$ are up-sampled versions of $w2_d[n]$ and $z2_d[n]$. We have to keep track of values of $n$ that are divisible by 4, and also values of $n$ that are even, but not divisible by 4, such as the number 6. Since we have taken a signal where all of the index terms are already even, the function of the down-sampler followed by the up-sampler is that the "even-even" (divisible by 4) indices are kept, but the "odd-even" indices (divisible by 2, but not divisible by 4, such as 2, 6, 10, etc.) are eliminated, then made to zero by the up-sampler.

$$w2_u[n] = w2_d[n] = w2[n], \quad n \text{ is even-even}$$
$$w2_u[n] = 0, \quad n \text{ is odd-even}$$
$$w2_u[n] \text{ is undefined, otherwise}$$
$$z2_u[n] = z2_d[n] = z2[n], \quad n \text{ is even-even}$$
$$z2_u[n] = 0, \quad n \text{ is odd-even}$$
$$z2_u[n] \text{ is undefined, otherwise}$$

Signals $w2_f[n]$ and $z2_f[n]$ are found as before:

$$w2_f[n] = dw2_u[n] + cw2_u[n-1] + bw2_u[n-2] + aw2_u[n-3]$$
$$w2_f[n] = dw2[n] + 0 + bw2[n-2] + 0, \quad n \text{ is even-even}$$
$$w2_f[n] = 0 + cw2[n-1] + 0 + aw2[n-3], \quad n \text{ is odd-even}$$
$$z2_f[n] = -az2_u[n] + bz2_u[n-1] - cz2_u[n-2] + dz2_u[n-3]$$
$$z2_f[n] = -az2[n] + 0 - cz2[n-2] + 0, \quad n \text{ is even-even}$$
$$z2_f[n] = 0 + bz2[n-1] + 0 + dz2[n-3], \quad n \text{ is odd-even.}$$

We can finish the reconstruction for the second octave by looking at the result when $w2_f[n]$ and $z2_f[n]$ are added together to recreate $w_d[n]$. We will call this recreated signal $w_d[n]'$, until we are certain that it is the same as $w_d[n]$.

$$w_d[n]' = w2_f[n] + z2_f[n]$$
$$w_d[n]' = dw2[n] + bw2[n-2] - az2[n] - cz2[n-2], \quad n \text{ is even-even}$$
$$w_d[n]' = cw2[n-1] + aw2[n-3] + bz2[n-1] + dz2[n-3], \quad n \text{ is odd-even}$$

The important thing to do here is to show that the two signals marked $w_d[n]$ in Figure 9.22 are exactly the same. This means finding the previous expression in terms of $w[n]$ only. First, we will take the expressions for $w2[n]$ and $z2[n]$, and find $w2[n-k]$ and $z2[n-k]$.

$$w2[n] = aw_d[n] + bw_d[n-1] + cw_d[n-2] + dw_d[n-3]$$
$$z2[n] = dw_d[n] - cw_d[n-1] + bw_d[n-2] - aw_d[n-3]$$
$$w2[n-k] = aw_d[n-k] + bw_d[n-k-1] + cw_d[n-k-2] + dw_d[n-k-3]$$
$$z2[n-k] = dw_d[n-k] - cw_d[n-k-1] + bw_d[n-k-2] - aw_d[n-k-3]$$

Now, we can plug these into the previous $w_d[n]$ expressions.

$$\begin{aligned}
w_d[n]' &= d(aw_d[n] + bw_d[n-1] + cw_d[n-2] + dw_d[n-3]) \\
&+ b(aw_d[n-2] + bw_d[n-3] + cw_d[n-4] + dw_d[n-5]) \\
&- a(dw_d[n] - cw_d[n-1] + bw_d[n-2] - aw_d[n-3]) \\
&- c(dw_d[n-2] - cw_d[n-3] + bw_d[n-4] - aw_d[n-5]), \quad n \text{ is even-even} \\
w_d[n]' &= c(aw_d[n-1] + bw_d[n-2] + cw_d[n-3] + dw_d[n-4]) \\
&+ a(aw_d[n-3] + bw_d[n-4] + cw_d[n-5] + dw_d[n-6]) \\
&+ b(dw_d[n-1] - cw_d[n-2] + bw_d[n-3] - aw_d[n-4]) \\
&+ d(dw_d[n-3] - cw_d[n-4] + bw_d[n-5] - aw_d[n-6]), \quad n \text{ is odd-even}
\end{aligned}$$

Rewriting these expressions and lining them up into columns gives us:

$n$ is even-even:

$$\begin{array}{llllll}
 & w_d[n]' = & & & & \\
 & adw_d[n] & + bdw_d[n-1] & + cdw_d[n-2] & + ddw_d[n-3] & \\
 & & & + abw_d[n-2] & + bbw_d[n-3] & + bcw_d[n-4] \\
+ bdw_d[n-5] & & & & & \\
 & -adw_d[n] & + acw_d[n-1] & - abw_d[n-2] & + aaw_d[n-3] & \\
 & & & - cdw_d[n-2] & + ccw_d[n-3] & - bcw_d[n-4] \\
+ acw_d[n-5]. & & & & &
\end{array}$$

$n$ is odd-even:

$$\begin{array}{llllll}
 & w_d[n]' = & & & & \\
 & acw_d[n-1] & + bcw_d[n-2] & + ccw_d[n-3] & + cdw_d[n-4] & \\
 & & & + aaw_d[n-3] & + abw_d[n-4] & + acw_d[n-5] \\
+ adw_d[n-6] & & & & & \\
+ bdw_d[n-1] & & - bcw_d[n-2] & + bbw_d[n-3] & - abw_d[n-4] & \\
 & & & + ddw_d[n-3] & - cdw_d[n-4] & + bdw_d[n-5] \\
- adw_d[n-6]. & & & & &
\end{array}$$

Canceling out terms gives us:
$n$ is even-even:

$$\begin{array}{llll}
w_d[n]' = & bdw_d[n-1] & + ddw_d[n-3] & \\
 & & + bbw_d[n-3] & + bdw_d[n-5] \\
 & + acw_d[n-1] & + aaw_d[n-3] & \\
 & & + ccw_d[n-3] & + acw_d[n-5].
\end{array}$$

n is odd-even:

$$\begin{array}{llll}
w_d[n]' = & acw_d[n-1] & + ccw_d[n-3] & \\
 & & + aaw_d[n-3] & + acw_d[n-5] \\
 & + bdw_d[n-1] & + bbw_d[n-3] & \\
 & & + ddw_d[n-3] & + bdw_d[n-5].
\end{array}$$

Examining the above equations, we see that we have the same expression for when $n$ is even-odd as when it is even-even. Therefore, we can rewrite the above expression, with the note that $n$ must be even. We know from the previous section that $ac = -bd$, so the terms with $w_d[n-1]$ and $w_d[n-5]$ will cancel out. This leaves us with $w_d[n]' = (aa+bb+cc+dd)w_d[n-3]$. As previously noted, $(aa+bb+cc+dd) = 1$, so we get the final expression for $w_d[n]' = w_d[n-3]$. The reconstructed $w_d[n]'$ is the same as a delayed version of the original $w_d[n]$. This is no surprise, since we have

simply inserted a copy of the 1-octave filter bank structure between the original and the reconstruction. When we looked at the filter bank structure for 1-octave, we saw that the result $y[n]$ was a delayed version of $x[n]$.

The next step is to take the reconstructed signal, $w_d[n]$, and recombine it with $z_d[n]$ to get $y[n]$. Note that $w_d[n]$ in the analysis (top part) of Figure 9.22 and the $w_d[n]'$ in the synthesis (bottom part) of this figure *must* be exactly the same. But we saw above that they are not exactly the same, since $w_d[n]'$ is a delayed version of $w_d[n]$. We can deal with this in a couple of ways. If we are designing hardware to perform the wavelet transform, we can simply add three (that is, *length of filter*$-1$) delay units to channel with the $z_d[n]$ signal. If we are writing software to do this, we can put three zeros before $z_d[n]$, or we could shift $w_d[n]'$ forward by three positions, getting rid of the first three values. The important thing is that $w_d[n]'$ and $z_d[n]$ are properly aligned. We will assume that this second method has been used to "correct" $w_d[n]'$ to be exactly equal to $w_d[n]$. If we were to use the first method of delaying $z_d[n]$, then the output would be reconstructed using $w_d[n-3]$ and $z_d[n-3]$, implying that the output would be delayed by an additional three units. That is, instead of having $y[n] = x[n-3]$ as the result, we would have $y[n] = x[n-3-3] = x[n-6]$.

Tracing the path of $w_d[n]$ and $z_d[n]$ is very similar to that of $w2_d[n]$ and $z2_d[n]$.

$$w_u[n] = w_d[n] = w[n], \quad n \text{ is even}$$
$$w_u[n] = 0, \quad n \text{ is odd}$$
$$z_u[n] = z_d[n] = z[n], \quad n \text{ is even}$$
$$z_u[n] = 0, \quad n \text{ is odd}$$

Notice that the "otherwise" case has been dropped, since $n$ is either odd or even. Finding $w_f[n]$ and $z_f[n]$ is just like finding $w2_f[n]$ and $z2_f[n]$:

$$w_f[n] = dw_u[n] + cw_u[n-1] + bw_u[n-2] + aw_u[n-3]$$
$$w_f[n] = dw[n] + 0 + bw[n-2] + 0, \quad n \text{ is even}$$
$$w_f[n] = 0 + cw[n-1] + 0 + aw[n-3], \quad n \text{ is odd}$$

$$z_f[n] = -az_u[n] + bz_u[n-1] - cz_u[n-2] + dz_u[n-3]$$
$$z_f[n] = -az[n] + 0 - cz[n-2] + 0, \quad n \text{ is even}$$
$$z_f[n] = 0 + bz[n-1] + 0 + dz[n-3], \quad n \text{ is odd.}$$

Adding $w_f[n]$ to $z_f[n]$ produces $y[n]$, just as we saw for the single-octave case. The important thing to notice is that the reconstruction is exactly the same as the single-octave case, provided that $w_d[n]$ is exactly the same in both the analysis and synthesis (top and bottom) parts of Figure 9.22.

Here we have seen that we can make the filter bank structure work recursively, and still get a perfectly reconstructed signal. While this section uses only two levels of recursion (octaves), it should be clear that we can use as many octaves as we want. The only limitation is that imposed by the length of the input signal, since every octave uses half the number of inputs as the octave above it. In other words, $w_d[n]$ exists only for even values of the index $n$, while the input signal $x[n]$ has values for even and odd values of $n$. If $x[n]$ has 16 samples, then $w_d[n]$ would have 8, and $w2_d[n]$ would have 4, etc.

## 9.10 Biorthogonal Wavelets

When talking about wavelets, the transform is classified as either orthogonal, or biorthogonal. The previously discussed wavelet transforms (Haar and Daubechies) were both orthogonal. These can be normalized as well, such as by using $1/\sqrt{2}$ and $-1/\sqrt{2}$ instead of $1/2$ and $-1/2$ for the Haar coefficients. If you assume that we just take the square root of $1/2$, you have missed a step. Instead, multiply by $\sqrt{2}$, i.e., $\sqrt{2}(1/2) = \sqrt{2}/2 = 1/\sqrt{2}$. When it is both orthogonal and normal, we call it *orthonormal*.

The following structure (Figure 9.23) shows an example biorthogonal wavelet transform, and its inverse, for a single octave.

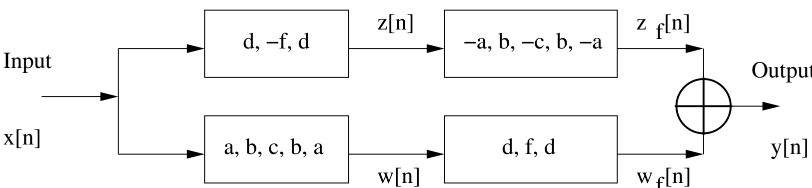



**Figure 9.23:** Biorthogonal wavelet transform.

Proceeding as before, we start with definitions for $w[n]$ and $z[n]$:

$$w[n] = ax[n] + bx[n-1] + cx[n-2] + bx[n-3] + ax[n-4]$$

$$w[n-k] = ax[n-k] + bx[n-k-1] + cx[n-k-2] + bx[n-k-3] + ax[n-k-4]$$

$$z[n] = dx[n] - fx[n-1] + dx[n-2]$$
$$z[n-k] = dx[n-k] - fx[n-k-1] + dx[n-k-2].$$

Now we can define $w_f[n]$ and $z_f[n]$ in terms of $w[n]$ and $z[n]$:

$$w_f[n] = dw[n] + fw[n-1] + dw[n-2]$$
$$z_f[n] = -az[n] + bz[n-1] - cz[n-2] + bz[n-3] - az[n-4].$$

Next, we replace the $z[n]$'s and $w[n]$'s with their definitions:

$$\begin{aligned}
w_f[n] &= adx[n] + bdx[n-1] + cdx[n-2] + bdx[n-3] + adx[n-4] \\
&+ afx[n-1] + bfx[n-2] + cfx[n-3] + bfx[n-4] + afx[n-5] \\
&+ adx[n-2] + bdx[n-3] + cdx[n-4] + bdx[n-5] + adx[n-6] \\
z_f[n] &= -adx[n] + afx[n-1] - adx[n-2] \\
&+ bdx[n-1] - bfx[n-2] + bdx[n-3] \\
&- cdx[n-2] + cfx[n-3] - cdx[n-4] \\
&+ bdx[n-3] - bfx[n-4] + bdx[n-5] \\
&- adx[n-4] + afx[n-5] - adx[n-6].
\end{aligned}$$

These are added together to give us $y[n]$, the output:

$$y[n] = w_f[n] + z_f[n].$$

Removing the terms that cancel, and simplifying, gives us:

$$y[n] = 2(bd + af)x[n-1] + (4bd + 2cf)x[n-3] + 2(af + bd)x[n-5].$$

To make $y[n]$ a delayed version of $x[n]$, we need the above expression to be in terms of only one index for $x$, that is, $x[n-3]$. This can be accomplished if we set $2(bd + af) = 0$, or $bd = -af$. Another condition is that we do not want to have to divide the output by a constant. In this case, that means that the $(4bd + 2cf)$ term should be equal to 1. However, we are likely to use this transform with down-samplers and up-samplers, which means that we would want the term $(2bd + cf)$ to

be equal to 1.

Here, we repeat the above analysis with the added complexity of down-samplers and up-samplers. Figure 9.24 shows this case.

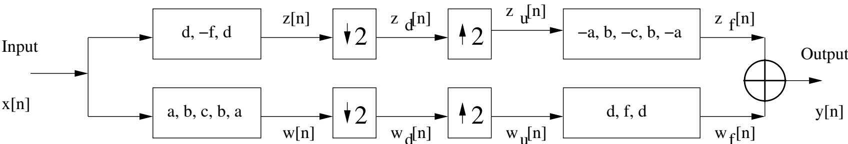



**Figure 9.24:** Biorthogonal wavelet transform.

The definitions for $w[n]$ and $z[n]$ are the same as above. The signals after the down-samplers are $w_d[n]$ and $z_d[n]$. After the up-samplers, the signals are labeled $w_u[n]$ and $z_u[n]$.

$$w_d[n] = w[n], \quad n \text{ is even}$$
$$w_d[n], \text{ is undefined}, \quad n \text{ is odd}$$
$$w_u[n] = w_d[n] = w[n], \quad n \text{ is even}$$
$$w_u[n] = 0, \quad n \text{ is odd}$$

$$z_d[n] = z[n], \quad n \text{ is even}$$
$$z_d[n], \text{ is undefined}, \quad n \text{ is odd}$$
$$z_u[n] = z_d[n] = z[n], \quad n \text{ is even}$$
$$z_u[n] = 0, \quad n \text{ is odd}$$

$$w_f[n] = dw_u[n] + fw_u[n-1] + dw_u[n-2]$$
$$z_f[n] = -az_u[n] + bz_u[n-1] - cz_u[n-2] + bz_u[n-3] - az_u[n-4]$$

$$y[n] = w_f[n] + z_f[n]$$

Next, we replace the $z_u[n]$'s and $w_u[n]$'s with their definitions.

$$w_f[n] = dw_u[n] + dw_u[n-2], \quad n \text{ is even}$$
$$z_f[n] = -az_u[n] - cz_u[n-2] - az_u[n-4], \quad n \text{ is even}$$
$$w_f[n] = fw_u[n-1], \quad n \text{ is odd}$$
$$z_f[n] = bz_u[n-1] + bz_u[n-3], \quad n \text{ is odd}$$
$$y[n] = dw[n] + dw[n-2] - az[n] - cz[n-2] - az[n-4], \quad n \text{ is even}$$
$$y[n] = fw[n-1] + bz[n-1] + bz[n-3], \quad n \text{ is odd}$$

Multiplying out:

$$\begin{aligned} y[n] = \quad & d(ax[n] && + bx[n-1] && + cx[n-2] && + bx[n-3] && + ax[n-4]) \\ & + d(ax[n-2] && + bx[n-3] && + cx[n-4] && + bx[n-5] && + ax[n-6]) \\ & - a(dx[n] && - fx[n-1] && + dx[n-2]) \\ & - c(dx[n-2] && - fx[n-3] && + dx[n-4]) \\ & - a(dx[n-4] && - fx[n-5] && + dx[n-6]), && n \text{ is even} \end{aligned}$$

$$\begin{aligned} y[n] = \quad & f(ax[n-1] && + bx[n-2] && + cx[n-3] && + bx[n-4] && + ax[n-5]) \\ & + b(dx[n-1] && - fx[n-2] && + dx[n-3]) \\ & + b(dx[n-3] && - fx[n-4] && + dx[n-5]), && n \text{ is odd.} \end{aligned}$$

Simplifying:

$$\begin{aligned} y[n] = \quad & adx[n] && + bdx[n-1] && + cdx[n-2] && + bdx[n-3] && + adx[n-4] \\ & + adx[n-2] && + bdx[n-3] && + cdx[n-4] && + bdx[n-5] && + adx[n-6] \\ & - adx[n] && + afx[n-1] && - adx[n-2] \\ & - cdx[n-2] && + cfx[n-3] && - cdx[n-4] \\ & - adx[n-4] && + afx[n-5] && - adx[n-6], && n \text{ is even} \end{aligned}$$

$$\begin{aligned} y[n] = \quad & afx[n-1] && + bfx[n-2] && + cfx[n-3] && + bfx[n-4] && + afx[n-5] \\ & + bdx[n-1] && - bfx[n-2] && + bdx[n-3] \\ & + bdx[n-3] && - bfx[n-4] && + bdx[n-5], && n \text{ is odd.} \end{aligned}$$

Combining them together, and removing terms that cancel, gives us:

$$\begin{aligned}
y[n] = \quad & bdx[n-1] && + bdx[n-3] && && \\
& && + bdx[n-3] && + bdx[n-5] && \\
& + afx[n-1] && + cfx[n-3] && + afx[n-5], && n \text{ is even}
\end{aligned}$$

$$\begin{aligned}
y[n] = \quad & afx[n-1] && + cfx[n-3] && + afx[n-5] && \\
& + bdx[n-1] && + bdx[n-3] && && \\
& && + bdx[n-3] && + bdx[n-5], && n \text{ is odd.}
\end{aligned}$$

Whether $n$ is even or odd,

$$y[n] = (bd + af)x[n-1] + (2bd + cf)x[n-3] + (bd + af)x[n-5].$$

We see that this is the same equation as the one for biorthogonal wavelets without down/up-samplers, except without a factor of 2 throughout. This again confirms the notion that the down/up-samplers remove redundant data, making the transform efficient.

## 9.11 Wavelet Transform Theory

The discrete wavelet transform convolves the input by the shifts (translation in time) and scales (dilations or contractions) of the wavelet. Below are variables commonly used in wavelet literature:

$g$ represents the highpass (wavelet) filter
$h$ represents the lowpass (scaling) filter
$J$ is the total number of octaves
$j$ is the current octave (used as an index. $1 <= j <= J$ )
$N$ is the total number of inputs
$n$ is the current input (used as an index. $1 <= n <= N$ )
$L$ is the width of the filter (number of taps)
$k$ is the current wavelet coefficient
$W_f(a, b)$ represents the continuous wavelet transform (CWT) of function $f$
$W_h[j, n]$ represents the discrete wavelet transform of function $f$
$W[j, n]$ represents the discrete scaling function of $f$, except:
$W[0, n]$ which is the input signal.

The continuous wavelet transform is represented by:

$$W_f(a,b) = \int f(t)\psi(at+b)dt$$

where $f(t)$ is the function to analyze, $\psi$ is the wavelet, and $\psi(at+b)$ is the shifted and scaled version of the wavelet at time $b$ and scale $a$ [7]. An alternate form of the equation is:

$$W_f(s,u) = \int_{-\infty}^{\infty} f(t)\sqrt{s}\psi(s(t-u))dt$$

again where $\psi$ is the wavelet, while the wavelet family is shown above as $\sqrt{s}\psi(s(t-u))$, shifted by $u$ and scaled by $s$. We can rewrite the wavelet transform as an inner product [1]:

$$W_f(s,u) = \left\langle f(t), \sqrt{s}\psi(s(t-u)) \right\rangle .$$

This inner product is essentially computed by the filters.

So far, this background focuses on how to get the wavelet transform given a wavelet, but how does one get the wavelet coefficients and implement the transform with filters? The relationship between wavelets and the filter banks that implement the wavelet transform is as follows. The scaling function, $\phi(t)$, is determined through recursively applying the filter coefficients, since multiresolution recursively convolutes the input vector after shifting and scaling. All the information about the scaling and wavelet functions is found by the coefficients of the scaling function and of the wavelet function, respectively [3]. The scaling function is given as:

$$\phi(t) = \sqrt{2}\sum_k h[k]\phi(2t-k).$$

The wavelet function is given as:

$$\psi(t) = \sqrt{2}\sum_k g[k]\phi(2t-k).$$

There is a finite set of coefficients $h[k]$. Once these coefficients are found, allowing us to design the lowpass filter, then the highpass filter coefficients are easy to find. Daubechies [2] chose the following coefficients, with many desirable (and required) properties, and solved the above equations with them.

Lowpass (scaling) coefficients: [3]

$$h[0], h[1], h[2], h[3] = \frac{1+\sqrt{3}}{4\sqrt{2}}, \frac{3+\sqrt{3}}{4\sqrt{2}}, \frac{3-\sqrt{3}}{4\sqrt{2}}, \frac{1-\sqrt{3}}{4\sqrt{2}}.$$

These produce a space, $V_j$, that has invariance to shift and scale, which is needed for multiresolution analysis. Now that we have coefficients for the lowpass filter, the corresponding highpass (wavelet) filter coefficients can be calculated, as follows:

$$g[0], g[1], g[2], g[3] = \frac{1-\sqrt{3}}{4\sqrt{2}}, \frac{-3+\sqrt{3}}{4\sqrt{2}}, \frac{3+\sqrt{3}}{4\sqrt{2}}, \frac{-1-\sqrt{3}}{4\sqrt{2}}.$$

Notice that $g[0] = h[3]$, $g[1] = -h[2]$, $g[2] = h[1]$, and $g[3] = -h[0]$.

This pattern appears in the filter bank shown in Figure 9.4. The significance of the above coefficients is that they are used in a filter bank to generate the wavelet transform [3].

Writing the wavelet equation with regard to the discretely sampled sequence $x[n]$ [37]:

$$W_f[a, b] = \frac{1}{\sqrt{a}} \sum_{n=b}^{aL+b-1} x[n] g\left(\frac{n-b}{a}\right)$$

where the wavelet is replaced by function $g$, which is obtained by sampling the continuous wavelet function. For the discrete case, we let $a = 2k$ and require that the parameters $(a, b)$ as well as $k$ be integers. To make this clearer, we replace $a$ and $b$ (which represent real numbers above) with $j$ and $n$ (which represent whole numbers). With the discrete case, redundancy is not required in the transformed signal. To reduce redundancy, the wavelet equation must satisfy certain conditions. First, we must introduce an orthogonal scaling function, which allows us to have an approximation at the last level of resolution (the initial level of resolution is simply the input signal). As we saw in previous sections, the coefficients for the scaling filter (lowpass) and the wavelet filter (highpass) are intimately related. The functions that produce these coefficients are also dependent on one another. Second, the representation of the signal at octave $j$ must have all the information of the signal at octave $j+1$, which makes sense: the input signal has more information than the first approximation of this signal. The function $x[n]$ is thus changed to $W[j-1, m]$, which is the scaling function's decomposition from the previous level of resolution, with $m$ as an index. Third, after $J$ levels of resolution, the result of the scaling function on the signal will be 0. After repeatedly viewing a signal at coarser and coarser approximations, eventually the scaling function will not produce any

useful information. Fourth, the scaling function allows us to approximate any given signal with a variable amount of precision [7]. The scaling function, $h$, gives us an approximation of the signal via the following equation. This is also known as the lowpass output:

$$W[j,n] = \sum_{m=0}^{N-1} W[j-1,m]h[2n-m].$$

The wavelet function gives us the detail signal, also called highpass output:

$$W_h[j,n] = \sum_{m=0}^{N-1} W[j-1,m]g[2n-m].$$

The $n$ term gives us the shift, the starting points for the wavelet calculations. The index $2n-m$ incorporates the scaling, resulting in half the outputs for octave $j$ compared to the previous octave $j-1$.

---

**Example:**
Suppose $g = \{\frac{1}{\sqrt{2}}, \frac{-1}{\sqrt{2}}\}$, and $x = \{34, 28, 76, 51, 19, 12, 25, 43\}$. Find the detail signal produced by the wavelet transform on this signal.

**Answer:**

$$W[0,n] = x[n]$$

$$\begin{aligned} W_h[1,n] = {} & W[0,0]g[2n-0] + W[0,1]g[2n-1] + W[0,2]g[2n-2] + W[0,3]g[2n-3] \\ & + W[0,4]g[2n-4] + W[0,5]g[2n-5] + W[0,6]g[2n-6] + W[0,7]g[2n-7] \end{aligned}$$

For this example, the value of $g$ with any index value outside of [0, 1] is zero. Also, the value of $W[j,n] = 0$ for any value of $n$ less than 0 or greater or equal to $N$. So we can find the individual values for $W_h$ as follows.
**Looking at octave 1:**

$$\begin{aligned} W_h[1,n] = {} & 34g[2n-0] + 28g[2n-1] + 76g[2n-2] + 51g[2n-3] + 19g[2n-4] \\ & + 12g[2n-5] + 25g[2n-6] + 43g[2n-7] \end{aligned}$$

$$W_h[1,0] = 34g[0]+28g[-1]+76g[-2]+51g[-3]+19g[-4]+12g[-5]+25g[-6]+43g[-7]$$

$$W_h[1,0] = 34\frac{1}{\sqrt{2}}$$

$$W_h[1,1] = 34g[2]+28g[1]+76g[0]+51g[-1]+19g[-2]+12g[-3]+25g[-4]+43g[-5]$$

$$W_h[1,1] = 28\frac{-1}{\sqrt{2}} + 76\frac{1}{\sqrt{2}}$$

$$W_h[1,2] = 34g[4] + 28g[3] + 76g[2] + 51g[1] + 19g[0] + 12g[-1] + 25g[-2] + 43g[-3]$$

$$W_h[1,2] = 51\frac{-1}{\sqrt{2}} + 19\frac{1}{\sqrt{2}}$$

$$W_h[1,3] = 34g[6] + 28g[5] + 76g[4] + 51g[3] + 19g[2] + 12g[1] + 25g[0] + 43g[-1]$$

$$W_h[1,3] = 12\frac{-1}{\sqrt{2}} + 25\frac{1}{\sqrt{2}}$$

$$W_h[1,4] = 34g[8] + 28g[7] + 76g[6] + 51g[5] + 19g[4] + 12g[3] + 25g[2] + 43g[1]$$

$$W_h[1,4] = 43\frac{-1}{\sqrt{2}}$$

$$W_h[1,5] = 34g[10] + 28g[9] + 76g[8] + 51g[7] + 19g[6] + 12g[5] + 25g[4] + 43g[3]$$

$$W_h[1,5] = 0.$$

It should be obvious that any value $W_h[1,n] = 0$ when $n \geq 5$. Comparing this to the filter bank approach, we look at Figure 9.25, which shows a channel from a filter bank. Keeping in mind that $g[0] = a$, and $g[1] = b$, we can trace the signal through as follows.

$$y = \frac{34-0}{\sqrt{2}}, \frac{28-34}{\sqrt{2}}, \frac{76-28}{\sqrt{2}}, \frac{51-76}{\sqrt{2}}, \frac{19-51}{\sqrt{2}}, \frac{12-19}{\sqrt{2}}, \frac{25-12}{\sqrt{2}}, \frac{43-25}{\sqrt{2}}, \frac{0-43}{\sqrt{2}}$$

The output is a down-sampled version of $y$, so when we eliminate every other value,

we get:

$$output = \frac{34}{\sqrt{2}}, \frac{76-28}{\sqrt{2}}, \frac{19-51}{\sqrt{2}}, \frac{25-12}{\sqrt{2}}, \frac{-43}{\sqrt{2}}.$$

We end up with the same result, as expected. Of course, the scaling equation is the same basic equation, with different coefficients.

x[n] → [a, b] → y[n] → [↓2] → output

**Figure 9.25:** One channel of the analysis side of a filter bank.

Before we can look at octave 2, we need to know the $W[1,n]$ values, which we present as the following. Realize that these can be found in the same fashion as above, except that both filter coefficients are positive.

$$W[1,0] = \frac{34}{\sqrt{2}}$$

$$W[1,1] = \frac{28+76}{\sqrt{2}} = \frac{104}{\sqrt{2}}$$

$$W[1,2] = \frac{51+19}{\sqrt{2}} = \frac{70}{\sqrt{2}}$$

$$W[1,3] = \frac{12+25}{\sqrt{2}} = \frac{37}{\sqrt{2}}$$

$$W[1,4] = \frac{43}{\sqrt{2}}$$

**Looking at octave 2:**

$$W_h[2,n] = W[1,0]g[2n-0] + W[1,1]g[2n-1] + W[1,2]g[2n-2] + W[1,3]g[2n-3]$$
$$+W[1,4]g[2n-4] + W[1,5]g[2n-5] + W[1,6]g[2n-6] + W[1,7]g[2n-7]$$
$$W_h[2,n] = \frac{34}{\sqrt{2}}g[2n-0] + \frac{104}{\sqrt{2}}g[2n-1] + \frac{70}{\sqrt{2}}g[2n-2] + \frac{37}{\sqrt{2}}g[2n-3]$$
$$+\frac{43}{\sqrt{2}}g[2n-4] + 0g[2n-5] + 0g[2n-6] + 0g[2n-7]$$

$$W_h[2,0] = \frac{34}{\sqrt{2}}g[0] + \frac{104}{\sqrt{2}}g[-1] + \frac{70}{\sqrt{2}}g[-2] + \frac{37}{\sqrt{2}}g[-3]$$

$$+\frac{43}{\sqrt{2}}g[-4] + 0g[-5] + 0g[-6] + 0g[-7]$$

$$W_h[2,0] = \left(\frac{34}{\sqrt{2}}\right)\left(\frac{1}{\sqrt{2}}\right)$$

$$W_h[2,0] = \frac{34}{2} = 17$$

$$W_h[2,1] = \frac{34}{\sqrt{2}}g[2] + \frac{104}{\sqrt{2}}g[1] + \frac{70}{\sqrt{2}}g[0] + \frac{37}{\sqrt{2}}g[-1]$$

$$+\frac{43}{\sqrt{2}}g[-2] + 0g[-3] + 0g[-4] + 0g[-5]$$

$$W_h[2,1] = \left(\frac{104}{\sqrt{2}}\right)\left(\frac{-1}{\sqrt{2}}\right) + \left(\frac{70}{\sqrt{2}}\right)\left(\frac{1}{\sqrt{2}}\right)$$

$$W_h[2,1] = \frac{70-104}{2} = \frac{-34}{2} = -17$$

$$W_h[2,2] = \frac{34}{\sqrt{2}}g[4] + \frac{104}{\sqrt{2}}g[3] + \frac{70}{\sqrt{2}}g[2] + \frac{37}{\sqrt{2}}g[1]$$

$$+\frac{43}{\sqrt{2}}g[0] + 0g[-1] + 0g[-2] + 0g[-3]$$

$$W_h[2,2] = \left(\frac{37}{\sqrt{2}}\right)\left(\frac{-1}{\sqrt{2}}\right) + \left(\frac{43}{\sqrt{2}}\right)\left(\frac{1}{\sqrt{2}}\right)$$

$$W_h[2,2] = \frac{43-37}{2} = \frac{6}{2} = 3$$

$$W_h[2,3] = \frac{34}{\sqrt{2}}g[6] + \frac{104}{\sqrt{2}}g[5] + \frac{70}{\sqrt{2}}g[4] + \frac{37}{\sqrt{2}}g[3]$$

$$+\frac{43}{\sqrt{2}}g[2] + 0g[1] + 0g[0] + 0g[-1]$$

$$W_h[2,3] = 0.$$

Again, we can verify this with the filter bank approach. Notice how the $g[2n-m]$ term takes care of down-sampling for us. We can continue on, with the third octave. First, we need the lowpass outputs of octave 2, found in the same way as above,

only with different coefficients.

$$W[2,0] = \frac{34}{2} = 17$$

$$W[2,1] = \frac{174}{2} = 87$$

$$W[2,2] = \frac{80}{2} = 40$$

$$W[2,3] = 0$$

**Looking at octave 3:**

$$W_h[3,n] = W[2,0]g[2n-0] + W[2,1]g[2n-1] + W[2,2]g[2n-2] + W[2,3]g[2n-3]$$
$$+W[2,4]g[2n-4] + W[2,5]g[2n-5] + W[2,6]g[2n-6] + W[2,7]g[2n-7]$$

$$W_h[3,n] = 17g[2n-0] + 87g[2n-1] + 40g[2n-2]$$

$$W_h[3,0] = 17g[0] + 87g[-1] + 40g[-2] = \frac{17}{\sqrt{2}}$$

$$W_h[3,1] = 17g[2] + 87g[1] + 40g[0] = \frac{40-87}{\sqrt{2}} = \frac{-47}{\sqrt{2}}$$

$$W_h[3,2] = 17g[4] + 87g[3] + 40g[2] = 0.$$

Finding the lowpass outputs for this octave:

$$W[3,0] = 17h[0] + 87h[-1] + 40h[-2] = \frac{17}{\sqrt{2}}$$

$$W[3,1] = 17h[2] + 87h[1] + 40h[0] = \frac{40-87}{\sqrt{2}} = \frac{-47}{\sqrt{2}}$$

$$W[3,2] = 17h[4] + 87h[3] + 40h[2] = 0.$$

We stop here, because we get no more useful information after this point. Why? Because our original signal has only 8 samples, and each octave uses only half the data of the previous octave. In other words, since we have $2^3$ values, we should not perform more than 3 octaves of resolution on the signal.

---

We end this chapter with a program to show the frequency magnitude response

of a given wavelet.

```
% show_wavelet.m
%
% See the frequency magnitude responses of wavelet filters
%
% Name your wavelet here:   Try "help waveinfo" for wavelet names
% examples: wavelet='haar'; wavelet='db2';
%   wavelet='db15'; wavelet='dmey';
%   wavelet='coif2';
% For an interesting one, try  wavelet = 'bior3.1';
wavelet = 'db4';

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Get wavelet coefficients
try
   [h1, h2, h3, h4] = wfilters(wavelet) ;
catch
   disp('Using the default wavelet, db2.');
   wavelet = 'db2';
   a = (1-sqrt(3))/(4*sqrt(2));
   b = (3-sqrt(3))/(4*sqrt(2));
   c = (3+sqrt(3))/(4*sqrt(2));
   d = (1+sqrt(3))/(4*sqrt(2));
   h1 = [a b c d];    % forward coeffs
   h2 = [-d c -b a];
   h3 = [d c b a];    % reverse coeffs
   h4 = [a -b c -d];
end

% pad filter coeffs with 0s
wlen = length(h1);
h1(length(h1)+1:128) = 0;    % Zero-pad until we have 128 values
h2(length(h2)+1:128) = 0;
h3(length(h3)+1:128) = 0;    % Zero-pad until we have 128 values
h4(length(h4)+1:128) = 0;

H1 = fft(h1);
H1mag = abs(H1);
H1phase = angle(H1);
```

```
H2 = fft(h2);
H2mag = abs(H2);
H2phase = angle(H2);
half_len = ceil(length(H1)/2);
range = 1:half_len;
plot(range,real(H1mag(range)),'r',range,real(H2mag(range)),'b');
mystr = 'Frequency magnitude response of lowpass (red),';
mystr = strcat(mystr, ' highpass (blue)');
title(mystr);
xlabel(sprintf('wavelet = %s, with %d coeffs',wavelet,wlen));

figure(2);

% Look at Reconstruction filters
H3 = fft(h3);
H3mag = abs(H3);
H3phase = angle(H3);
H4 = fft(h4);
H4mag = abs(H4);
H4phase = angle(H4);

plot(range,real(H3mag(range)),'r',range,real(H4mag(range)),'b');
mystr = sprintf('Frequency magnitude response of lowpass ');
mystr = strcat(mystr, 'inverse (red), highpass inverse (blue)');
title(mystr);
xlabel(sprintf('wavelet = %s, with %d coeffs',wavelet,wlen));
```

---

Using the default `db2` wavelet, we get the graphs shown in Figure 9.26 and Figure 9.27. In order to see the frequency magnitude responses of other wavelets, the wavelets toolbox must be installed. These figures show what we should expect: the lowpass filter (and inverse lowpass filter) keep low frequencies about the same, but attenuate high frequencies.

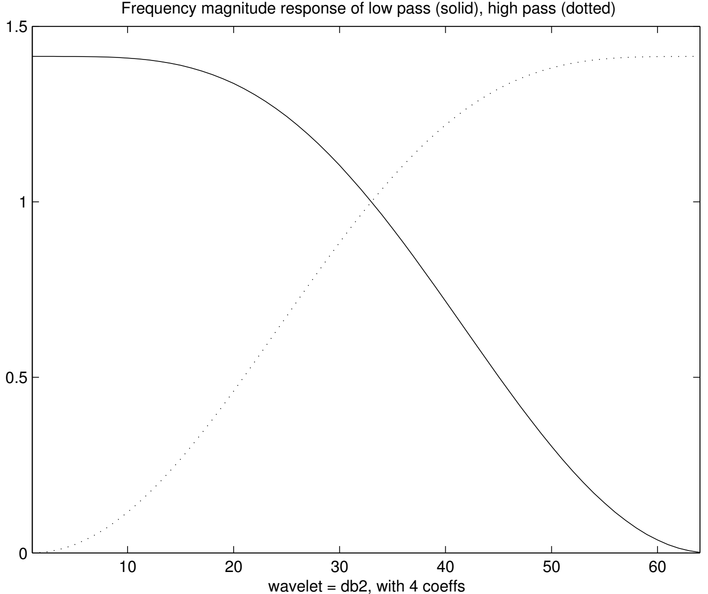



**Figure 9.26:** The first graph of the "show_wavelet" program.

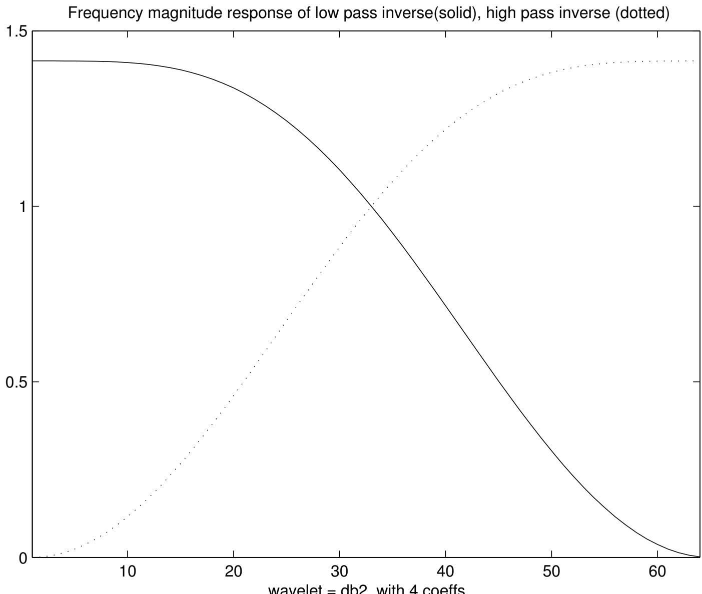



**Figure 9.27:** The second graph of the "show_wavelet" program.

## 9.12 Summary

This chapter presents the wavelet transform and shows how ordinary FIR filters can be used to implement this transform. We looked at the Haar transform, a simple wavelet transform of two filter coefficients, and then the Daubechies wavelets as an extension of the Haar transform. Biorthogonal wavelets were also presented.

The filter bank can be viewed as a way to perform this transform. Two common types of filter banks are quadrature mirror filters and conjugate quadrature filters. This chapter examined the down-samplers and up-samplers that go with a filter bank, as well as multiresolution, where we break a signal down at increasingly coarse scales. Multiresolution loosely corresponds to the frequency doubling seen in music, giving rise to the term "octaves" of resolution.

Functions to perform this transform include `dwt` and `idwt` for the forward and reverse discrete wavelet transforms, respectively. Anyone interested in image processing should consider the `dwt2` and `idwt2` commands.

## 9.13 Review Questions

1. Find (by hand) what the signals $z_d[n]$ and $w_d[n]$ would be for the filter bank in Figure 9.11. Let $x = \{8, 4, 0, 6, 3, 7, 2, 9\}$, $a = \frac{1}{2}$, and $b = \frac{1}{2}$. Be sure to show your work.

2. With a two-channel filter bank, such as in Figure 9.12, we found that the output
   $y[n] = (ac + bd)x[n-1] + (aa + bb + cc + dd)x[n-3] + (ac + bd)x[n-5]$.

   The Daubechies wavelet transform uses coefficients

   $a = \frac{1-\sqrt{3}}{4\sqrt{2}}$, $b = \frac{3-\sqrt{3}}{4\sqrt{2}}$, $c = \frac{3+\sqrt{3}}{4\sqrt{2}}$, and $d = \frac{1+\sqrt{3}}{4\sqrt{2}}$.

   a. What 2 constraints do the previous equation put on the coefficients? Show that the Daubechies coefficients satisfy these constraints.

   b. What effect do the down/up-samplers have on the output $y[n]$? How would removing them change the previous $y[n]$ equation?

3. What is multiresolution (i.e., a wavelet transform having more than one octave)? Demonstrate this idea with a figure.

4. For the filter coefficients below, use MATLAB to plot the frequency magnitude

response, and the phase response. Determine if the filters have linear phase, and determine if the filters are highpass, lowpass, bandpass, or bandstop.

a. $\{-0.0884, 0.0884, 0.7071, 0.7071, 0.0884, -0.0884\}$
b. $\{-0.1294, 0.2241, 0.8365, 0.4830\}$
c. $\{-0.4830, 0.8365, -0.2241, -0.1294\}$

5. Write a MATLAB program to perform the following transform (for one octave) and inverse transform. See Figure 9.28, where LPF = $\{1, 1\}$, HPF = $\{-1, 1\}$. (LPF stands for Low Pass Filter, and HPF stands for High Pass Filter).

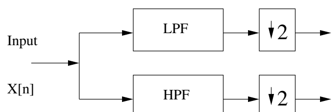



**Figure 9.28:** Analysis filters.

For the inverse transform, see Figure 9.29.

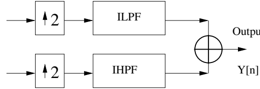



**Figure 9.29:** Synthesis filters.

Where ILPF = $\{-1, -1\}$, IHPF = $\{-1, 1\}$ (ILPF stands for Inverse Low Pass Filter, and IHPF stands for Inverse High Pass Filter). Is $y$ a scaled version of $x$, e.g., $y[n] \times (-1/4) = x[n]$?

6. Implement a Haar transform, with MATLAB commands, for three octaves.

7. We saw a pattern for a CQF, in that the coefficients of one filter are alternately negated and mirrored, so that if we know one filter's coefficients, we can find the coefficients for the three others. Does this work in general? Would this work for $a = 3$, $b = 1$ in Figure 9.4?

8. If the down-samplers keep only one of every four values, what effect would this have on the filter bank? What if the filter bank structure were modified to have four channels?

9. Write a function to return the one octave, Daubechies four-coefficient wavelet transform for a given signal. Include low- and highpass outputs.

10. Given the input signal $x[n] = 2\cos(2\pi 100n/300) + \cos(2\pi 110n/300 - \pi/8)$ for $n = 0..255$, write the commands to find the DWT for 3 octaves. Compare your results with those of the `dwt` function. Plot the original function, as well as the approximate signals.

11. For the Haar transform show in Figure 9.4, use values $a = b = \frac{1}{2}$ (no down-sampling), find signals $z$, $w$, and $y$, given an input of $x = \{6, 1, 3, 7, 2, 5, 8, 10\}$.

12. Suppose you have a 3-octave DWT. Draw the analysis structure in terms of filters and down-samplers.

13. For a four-octave DWT, suppose the input has 1024 samples. How long would the detail outputs be? How long would the approximate outputs be? What if down/up-sampling were not used? For simplicity, you can assume that the filter's outputs are the same lengths as their inputs.

14. For an input of 1024 samples, how many octaves of the DWT could we have before the approximation becomes a single number? For simplicity, you can assume that the filter's outputs are the same lengths as their inputs.

15. We stated above that the quadrature mirror filter will not work for four coefficients, i.e., $h_0 = g_0 = \{a, b, c, d\}$, $h_1 = \{a, -b, c, -d\}$, $g_1 = -h_1$. Show analytically that this will not work. That is, what constraints are there on the values $a$, $b$, $c$, and $d$, in order to get perfect reconstruction?

# Chapter 10

# Applications

This chapter presents several applications in MATLAB. First, it covers working with sound and images, including how to show sound as a frequency-domain graph. Other applications focus on the discrete wavelet transform, and how to design an FIR filter. This chapter also includes a set of programs to recursively solve a *Su Doku* puzzle. Finally, it ends with an example on compression.

## 10.1 Examples Working with Sound

MATLAB can read and write sound files in the *.wav* format. Of course, the computer must have a microphone attached. Unfortunately, at the time of this writing, some of the *.wav* features are only available on the Microsoft Windows® version of MATLAB.

The example below records 16,000 samples at 8000 samples/second, as *double* values. A little math reveals that this records for two seconds. After recording, the code plays the sound back.

```
% record from the microphone
x = wavrecord(16000, 8000, 'double');
% play it back
wavplay(x)
```

Running the above code produces an odd-sounding result. Since we did not specify the sampling rate, MATLAB chose one for us: 11,025 Hz (according to the `help` function). We next specify the correct sampling rate.

```
% play it back at the right sampling rate
wavplay(x, 8000)
```

What do you observe when the playback sampling rate (given as the second parameter) is less than the sampling rate used in the recording? What if you play it back with a higher sampling rate? Something else to consider is the `double` specification in the `wavrecord` command. What if we used `uint8` instead?

When we try this code on a Macintosh® running OS X, we get the following response.

```
>> x = wavrecord(16000, 8000, 'double');
??? Error using ==> wavrecord
WAVRECORD is only for use with Windows 95/98/NT machines.
```

One easy solution is to use software external to MATLAB to record data. Once saved as a file, we can use MATLAB to read it. Another possibility exists, that we will discuss shortly.

Suppose we overcome the recording issue, and have our sound data in $x$. Next, we might want to play it back.

```
>> wavplay(x)
??? Error using ==> wavplay
WAVPLAY is only for use with Windows 95/98/NT machines.
```

This can be overcome with the `sound` command. It works just like the `wavplay` command, but MATLAB supports it on OS X, as well as under Unix/Linux. Since it also works on computers running Microsoft operating systems, use `sound` instead of `wavplay` for compatibility.

```
>> sound(x)
>> sound(x, 8000);
```

As the example shows, it also takes a sampling rate as the second parameter.

What if you were not able to run the examples above? We can create an example sound with a sinusoid, and then play it. The next example does just that. This should work on any computer that has MATLAB and a working sound card.

```
% create an example sound
t = 0:0.0001:3;
x = 0.9*cos(2*pi*440*t);
% play it back
sound(x, 8000);
```

Assuming that we still have $x$, let's write it to a file. You will recognize 8000 as the sampling frequency, and `example_sound.wav` is the name of the file to create.

The other parameter, 16, gives the number of bits per sample. Though the `wavwrite` function uses 16 as the default, there are other possibilities.

```
wavwrite(x, 8000, 16, 'example_sound.wav');
```

Once we have created the example sound file, we will want to read it back sooner or later. Just as the sampling rate and number of bits are written as part of the file, these are returned when we read it as the second and third values, respectively.

```
[x2, fs, b] = wavread('example_sound.wav');
```

Inspecting the values for *fs* and *b* reveal that these are the same as the parameters that we gave to the `wavwrite` command.

Now let's create a new sound from the one we just read from disk. What does *mysound* sound like? Why? This code demonstrates random access, an advantage of digital media over analog.

```
mysound = x2(length(x2):-1:1);
% play it back
sound(mysound, fs);
```

We have seen some examples that work with the *.wav* audio format, though other possibilities exist. A program can read and write sound files in Sun's *.au* audio format, with `auread` and `auwrite`, respectively. Other sound file formats can be read, with a little help from an Internet search. You may find code available to read the file format you want, or with enough determination and documentation, you can write your own.

Another way to record uses the `audiorecorder` command, a recent addition to MATLAB (version 7 includes it, but version 6 does not). It uses special audio recorder objects, so our functional implementation will need a bit more explanation. Let's see what the code for our first record/playback example might look like. First, the code below sets up an "audiorecorder" object, that we label *x_object*. Next, we call the `record` function, which records data to an object. Here we use it with the microphone, but it could be used for other communications, such as with a device connected through a serial port. The second parameter in the `record` command specifies the number of seconds to record. We follow this with a `pause` command, to give the computer time to make the recording (2 seconds to record plus a safety margin .2 seconds). If we had something else to do, we could put that code here. Also, there is a similar function `recordblocking` that does not return until the recording finishes. Finally, the code below uses the `play` command, which works with

audiorecorder objects like sound or wavplay did with the data in previous examples. Leaving the semicolon off the end of the play command shows information about the object.

```
x_object = audiorecorder(44100, 8, 1);
record(x_object, 2);
pause(2.2);
play(x_object)
```

Though we may desire to record at a lower sampling rate, it may not be supported. The author found that calling the record function for an audioplayer object of 8000 samples/second resulted in a warning message about "unsupported audio format," while a similar call to function recordblocking failed to return! The difference between the two calls should simply be when the control returns to the user; record returns right away (continuing work in the background), while recordblocking returns after the recording time elapses.

Can we use the sound function with the audioplayer object? Yes, though we must first get the data in a format that the sound function can use. In other words, we want the raw sound data, not the nice object encapsulation. The getaudiodata function provides such a facility, as seen below.

```
% convert the data to an array of doubles
x = getaudiodata(x_object, 'double');
% play it back
sound(x, 44100);
```

In summary, of the commands used in this section, we saw that wavrecord and wavplay are not supported on all platforms. To get around this incompatibility, other software (besides MATLAB) can be used to record sound, and the sound command can be used in place of wavplay. The audiorecorder command, and related functions, also allow the user to work with sound. These commands are worth exploring, though they are not implemented in older MATLAB versions.

## 10.2 Examples Working with Images

Working with images can be rewarding. The results are readily apparent, in a concrete way. MATLAB provides functions to make image processing very easy. The examples in this section are meant to give the reader a quick understanding of what MATLAB can do.

The code below draws a small, white rectangle on a black background. The imwrite command below saves the image to disk, in the TIFF format.

```
% Create a 128x128 matrix of zero values
x = zeros(128, 128);
% Draw a solid white rectangle
x(20:30, 20:40) = 255;
% Save it as the file "whiterect.tiff"
imwrite(x, 'whiterect.tiff', 'TIFF');
% Show it on the screen
imshow(x);
```

Next, we will read the file from disk, and switch the rows and columns of the image. Of course, the `imread` command is not necessary if the image is still in memory.

```
% first, read the image
x = imread('whiterect.tiff');
% switch rows with columns
y = x.';
imshow(y);
```

If the image does not appear when the above code runs, the window for the image is probably just hidden. A call to `figure(1)` will bring the window to the front, assuming that it is the first figure.

We can also switch the rows and columns of a more complex image. The "dog on porch" image is available on the accompanying CD-ROM, and it will be used for the following examples. We will repeat the above example, only a bit more concisely. First, let's simply load the image and display it.

```
dog = imread('dog256x256.gif');
imshow(dog);
```

Now let's show it with the rows and columns switched, as a second figure.

```
figure(2);
imshow(dog.');
```

As one can see from the two images, transposing an image is not the same as rotating the image. For a clockwise rotation, we need to show the first column, from the bottom to the top, as the first row. Then repeat this for the next column, and the next column, etc. The following code does this, and shows the rotated image.

```
for r=1:256
    clockwise_dog(r, 1:256) = dog(256:-1:1, r);
end
figure(3);
imshow(clockwise_dog);
```

This works because we happen to know the image's dimensions in advance. What if we do not know this? Let's see what happens when we generalize this code. First, we will make a new image as a subset of the original one. While we are at it, we will close all of the open figures, then display the new image.

```
dog2 = dog(10:100, 20:200);
close all
imshow(dog2);
```

We see that only the top part of the image is copied to the new image. Now we will rotate it, as if we did not know the dimensions.

```
[MAX_ROWS, MAX_COLS] = size(dog2);
for r=1:MAX_COLS
    clockwise_dog2(r, 1:MAX_ROWS) = dog2(MAX_ROWS:-1:1, r);
end
figure(2);
imshow(clockwise_dog2);
```

Running the above code shows the new image, rotated clockwise, as expected.

In this section, we saw a few examples of working with images in the MATLAB programming environment.

## 10.3 Performing the 2D Discrete Wavelet Transform on an Image

Most texts on the discrete wavelet transform include an image such as that shown in Figure 10.1. It shows that the approximation is just that, and that the three details include horizontal, vertical, and diagonal edges. How difficult is it to generate an image like this?

First, we will load the image, and show it. Then we will perform the 2D Discrete Wavelet Transform (DWT) on that image. Next, we will make the results presentable. The `make_image.m` program takes an input matrix and returns something we can display. Once we do the DWT, the results are floating-point, and could

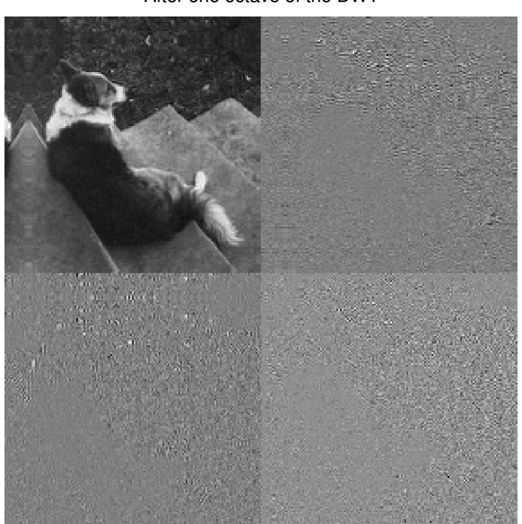



**Figure 10.1:** Two-dimensional DWT on an image.

be negative, and certainly can be greater than 255 (which makes the pixel value outside the grayscale range). Therefore, we need to map the values to grayscale.

Here is the `make_image.m` program that we need to form an image. It finds the range of values within a matrix, scales them to the range 0 to 255, and then converts them to `uint8`, unsigned, 8-bit integer values.

```
% make_image.m
%   Take a 2D matrix, and return it as an image
%   (uint8 values, from 0 to 255).
%
function imX = make_image(x);

minX = min(min(x));
maxX = max(max(x));

scale = maxX - minX;
imX = uint8(floor((x - minX)*255/scale));
```

Notice that we used `min` twice, since `min(x)` returns an entire row with the

minimum values from the columns. Similarly, the `max` function returns an array when given a two-dimensional matrix.

We can verify the minimum and maximum values returned by the `make_image` program by giving it small example matrices and checking the results. We will start with an easy one, and make them a little more complex each time.

```
>> make_image([0, 1; 6, 512])

ans =

     0     0
     2   255

>> make_image([0, 1; -6, 512])

ans =

     2     3
     0   255


>> make_image([ -255, 1; -6, 255])

ans =

     0   128
   124   255

>> make_image([ -255, 1; -6, 255])

ans =

     0   128
   124   255

>> make_image([ 100, 101; 106, 512])

ans =

     0     0
```

```
3   255
```

As an exercise, generate a random array and verify that the `make_image` program always returns results between 0 and 255. Also, try it with floating-point values. What if complex values were passed to this program? How could it be fixed to return real integers only?

### 10.3.1 2D DWT of a Grayscale Image

Now we have our program to show the effects of the 2D DWT on an image. First, we will load the image and show it to the screen.

```
% load the image
x = imread('dog256x256.gif');
% show the original
figure(1);
imshow(x); title('original image');
```

Next, we will perform the 2D discrete wavelet transform on the image. Here we use the Daubechies' `db16` wavelet, though we could use any discrete wavelet that we like.

```
[A, B, C, D] = dwt2(double(x), 'db16');
```

To make the different subimages viewable, we need to map their values to grayscale pixel values. We use the `make_image` program from the previous section.

```
imA = make_image(A);
imB = make_image(B);
imC = make_image(C);
imD = make_image(D);
```

We need to know how large the subimages are. We can assume that they are the same size, and create a large image matrix (*superx*) based on the size of subimage *imA*. The `zeros` function simply returns a matrix filled with zeros, with the given dimensions. In this case, the matrix values are all of type `uint8`.

```
[MAXA_row, MAXA_col] = size(imA);
% The dimensions should all be the same.
%[MAXB_row, MAXB_col] = size(imB);
%[MAXC_row, MAXC_col] = size(imC);
%[MAXD_row, MAXD_col] = size(imD);
superx = zeros(MAXA_row*2, MAXA_col*2, 'uint8');
```

Now we can place the subimages in the four corners of the *superx* matrix.

```
% put A in top left
superx(1:MAXA_row, 1:MAXA_col) = imA;
% put B in top right
superx(1:MAXA_row, MAXA_col+1:MAXA_col*2) = imB;
% put C in bottom left
superx(MAXA_row+1:MAXA_row*2, 1:MAXA_col) = imC;
% put D in bottom right
superx(MAXA_row+1:MAXA_row*2, MAXA_col+1:MAXA_col*2) = imD;
```

Finally, we show the *superx* matrix as an image.

```
figure(2);
imshow(superx);
title('After one octave of the DWT');
```

When we execute this code, we will see an image just like Figure 10.1. This works fine for grayscale images, and we expand this program for color images in the next section.

### 10.3.2 2D DWT of a Color Image

We can also perform the 2D DWT of a color image. One way to handle this is to treat the three color components (red, green, blue) as separate signals. We take this approach with the code below.

```
% Perform the DWT of a color image

% first, load and show the image
x = imread('colorimage.jpg');
figure(1);
imshow(x); title('original image');

% Do the 2D DWT on the red component
[Ar, Br, Cr, Dr] = dwt2(double(x(:,:,1)), 'db16');
% Do the 2D DWT on the green component
[Ag, Bg, Cg, Dg] = dwt2(double(x(:,:,2)), 'db16');
% Do the 2D DWT on the blue component
[Ab, Bb, Cb, Db] = dwt2(double(x(:,:,3)), 'db16');
```

In order to show the results, we must first map the values to the range 0 to 255. We do this for each color component separately.

```
% Make images from the red DWT results
imAr = make_image(Ar);
imBr = make_image(Br);
imCr = make_image(Cr);
imDr = make_image(Dr);

% Make images from the green DWT results
imAg = make_image(Ag);
imBg = make_image(Bg);
imCg = make_image(Cg);
imDg = make_image(Dg);

% Make images from the blue DWT results
imAb = make_image(Ab);
imBb = make_image(Bb);
imCb = make_image(Cb);
imDb = make_image(Db);
```

Next, we make a matrix large enough to fit all four subimages. The dimensions for the subimages should all be the same, so we base our large matrix's size on the approximation. We do not use the original image's dimensions for this, since the `dwt` command may return extra values according to the boundary. This matrix must be three-dimensional, with the third dimension allowing us to differentiate between red, green, and blue color components.

```
% Find the dimensions (which are the same for all subimages).
[MAXA_row, MAXA_col] = size(imAr);
% create a large matrix to put the subimages
superx = zeros(MAXA_row*2, MAXA_col*2, 3, 'uint8');
```

Now we combine the three color components for the approximation $A$, and store them in the large matrix *superx*.

```
% put A in top left
superx(1:MAXA_row, 1:MAXA_col,1) = imAr;
superx(1:MAXA_row, 1:MAXA_col,2) = imAg;
superx(1:MAXA_row, 1:MAXA_col,3) = imAb;
```

Likewise, we combine the color components for each of the details, and put them in their respective corners.

```
% put B in top right
superx(1:MAXA_row, MAXA_col+1:MAXA_col*2,1) = imBr;
superx(1:MAXA_row, MAXA_col+1:MAXA_col*2,2) = imBg;
superx(1:MAXA_row, MAXA_col+1:MAXA_col*2,3) = imBb;
% put C in bottom left
superx(MAXA_row+1:MAXA_row*2, 1:MAXA_col,1) = imCr;
superx(MAXA_row+1:MAXA_row*2, 1:MAXA_col,2) = imCg;
superx(MAXA_row+1:MAXA_row*2, 1:MAXA_col,3) = imCb;
% put D in bottom right
superx(MAXA_row+1:MAXA_row*2, MAXA_col+1:MAXA_col*2,1) = imDr;
superx(MAXA_row+1:MAXA_row*2, MAXA_col+1:MAXA_col*2,2) = imDg;
superx(MAXA_row+1:MAXA_row*2, MAXA_col+1:MAXA_col*2,3) = imDb;
```

Now we show the resulting image.

```
imshow(superx);
```

You can find an example image, *colorimage.jpg*, on the accompanying CD-ROM. From running this program, we see that the approximation retains colors, but the three details are shades of gray.

## 10.4 The Plus/Minus Transform

The plus/minus transform is a simple one, used to explain key concepts of the wavelet transform.

Suppose we have the following example signal: {4, 7, 1, 8}. We will take every pair of numbers, add them together, and subtract the second one from the first. For the additions/pluses, we have 4+7, and 1+8, resulting in the numbers {11, 9}. For the subtractions/minuses, we find $4 - 7$ and $1 - 8$. The order is important; we always subtract the second number from the first of every pair. The numbers $\{-3, -7\}$ result from the subtractions. Thus, our transformation gives us two signals: 11, 9, and $-3$, $-7$.

How can we undo the transform? Suppose we forgot the original numbers, but remember how we got them. We will label the original sequence $\{x_1, x_2, x_3, x_4\}$. From the transform, we know:

$$x_1 + x_2 = 11$$

$$x_1 - x_2 = -3.$$

Adding these two equations together gives us:

$$2(x_1) + (x_2 - x_2) = 11 + (-3) \quad \text{thus,}$$

$$x_1 = (11 - 3)/2 = 4.$$

We can plug this number into either equation above that has both $x_1$ and $x_2$. Let's use the first one:

$$4 + x_2 = 11$$

$$x_2 = 11 - 4 = 7.$$

So we conclude that the first two original numbers are {4, 7}. To find the rest of the original numbers, we repeat the above logic on the next numbers from our signals. This is left as an exercise for the reader.

We saw how the example signal {4, 7, 1, 8} maps to the two signals {11, 9} and {−3, −7}. Suppose we find the plus/minus transform on each of these resulting signals:

$$11 + 9 = 20$$

$$11 - 9 = 2 \quad \text{and}$$

$$-3 + (-7) = -10$$

$$-3 - (-7) = 4.$$

This results in four signals, {20}, {2}, {−10}, and {4}. We cannot go further, or at least there is no point in going further due to our data size.

In our plus/minus transform, we add the inputs in pairs then skip to the next input pairs. Think of this as shifting the transform over the data. First we find 4+7, then shift over to find 1+8, etc. We could have a transform that involves a larger group of numbers, but still shift over by two for each calculation. Each level affects more of the signal (scaling). For example, level-1 computes 4+7 = 11, involving the first two numbers. Level-2 computes (4+7)+(1+8) = 20, where we used the whole signal. Even if we had a longer signal, we would eventually arrive at a level where we use the whole signal. This simple transform shows the shifting and scaling (of the index) that exists in the wavelet transform.

To summarize, we have an approximation of 20, and an average of 20/4. We have 2 details, level-1: {−3, −7} and level-2: {2}. Note the similarity to the discrete wavelet transform; it has 1 approximation signal, and $J$ detail signals, where $J$ is based on the input length. If we gave the DWT a signal of four values, we would expect only two possible octaves of resolution, since $2^2 = 4$. This leads us to

a natural question, "Why should we care about the plus/minus transform?" To answer this, first notice that we could multiply through by $1/\sqrt{2}$, to eliminate the need to divide at the end (i.e., 20/4). If we do this, our plus/minus transform becomes the Haar transform (from 1910). The plus operation corresponds to the scaling function $\phi$, while the minus operation performs the wavelet function $\psi$. With a slight change to our plus/minus transform, we see that it uses coefficients $\{1/\sqrt{2}, 1/\sqrt{2}\}$ and $\{1/\sqrt{2}, -1/\sqrt{2}\}$. A wavelet is a small wave, shifted and scaled, and used to represent a signal as large, medium, and small waves. The plus/minus transform also does this.

## 10.5 Doing and Undoing the Discrete Wavelet Transform

For signal analysis using the wavelet transform, it makes sense to examine the results of the DWT after the transform, but before the signal is completely reconstructed. That is, we can view the signal as a set of subsignals which, when added together, combine to reconstitute the original data. These subsignals are what the wavelet transform removed from the original signal at different levels of resolution.

There are two programs (included on the CD-ROM) specifically for this sort of analysis: "DWT_undo.m" and "DWT_undo2.m," for the 1D and 2D cases, respectively. Both allow the input signal to be analyzed for up to 8 octaves. Keep in mind that the input size limits the number of octaves. If $J$ represents the number of octaves, then the input signal must be at least $2^J$ samples long (or in the case of a 2D signal, it must be at least $2^J \times 2^J$ samples). For both of these programs, the details and approximation subsignals are the same size as the original. Compare these to the normal DWT functions `dwt` and `dwt2`, where the size of the subsignals depends directly on the octave number. These two programs give the subsignals as the same size, so that adding all subsignals together recreates the original. This would not be appropriate for a compression application, but extra data is fine for analysis.

The code below shows how these programs work. First, we will examine the 1D case.

```
x = floor(rand(1,100)*100);
[A, D] = DWT_undo(x, 'db4', 3);
input_recon = A + D(1,:) + D(2,:) + D(3,:);
error = sum(abs(x - input_recon))
```

The code generates a random signal for $x$, then finds the DWT and Inverse Discrete Wavelet Transform (IDWT) on $x$ using the `db4` wavelet, for three octaves.

The results are returned in a one-dimensional, third-octave approximation $A$, and three detail signals $D$ (`D(1,:)`, `D(2,:)`, and `D(3,:)`). The variable *input_recon* will be a one-dimensional reconstruction of the original signal $x$. It should be an exact copy of $x$, which we verify in the final statement. Aside from some precision error, the difference between the two signals is zero. The actual output follows.

```
error =

   4.3289e-09
```

The random signal used above works well to test out the code, but we will run it again with a real signal for $x$. As the code below shows, we make $x$ a sum of sinusoids, then apply the "DWT_undo" function. When we do, we see that we get the results as shown in Figures 10.2 and 10.3.

```
t = 0:0.001:0.1;
x = cos(2*pi*100*t) + 1.5*cos(2*pi*110*t + pi/6);
[A, D] = DWT_undo(x, 'db4', 3);
```

Figure 10.2 plots the original signal in the top, and the "undone" third-level approximation in the bottom. We see that the signals are the same size, which comes from the "DWT_undo" function. The approximation signal is not the compact, one-eighth size approximation, but instead the contribution from the approximation channel, ready to be added to the detail contributions sample by sample to reconstruct the original. Keep in mind that we are not after compactness here, but we want to see how the original signal breaks down into subsignals. Looking at the three "undone" detail signals in Figure 10.3, we may notice the scale of the details grows according to the octave. The first octave's detail signal contains many small variations, while the second octave's detail signal has larger variations but fewer of them. The second octave's details would be half the number of the first octave's, resulting in fewer variations. The larger magnitudes show that the approximation at octave 1 becomes the detail at octave 2. In other words, our idea of "detail" varies (gets larger) with each succeeding octave.

As the details get larger in magnitude from octave 1 to octave 2, so they do between octaves 2 and 3. The details at octave 3 contain an interesting feature; we see the magnitudes get smaller around sample 40 and larger afterwards, just like in the original signal. In fact, examining the third-octave approximation (bottom plot of Figure 10.2) reveals that before sample 85, the approximation could very easily be 0. The three detail signals actually contain the salient signal information, at least before the last few samples.

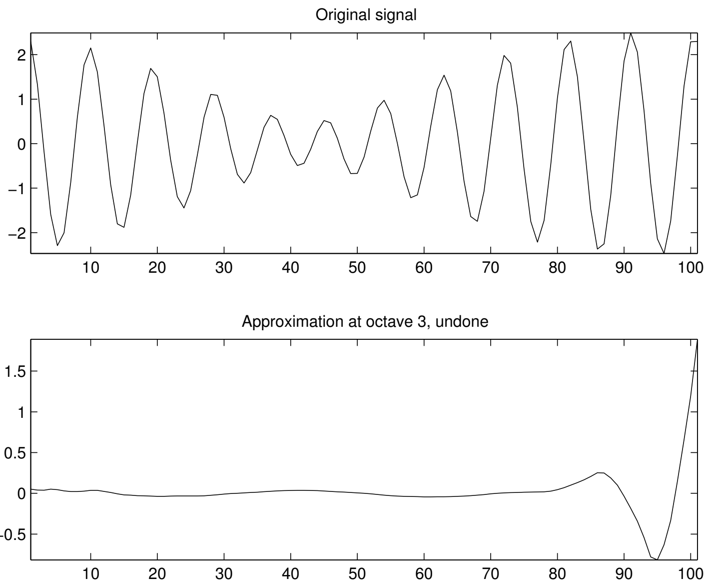



**Figure 10.2:** Results of "DWT undo" for a 1D signal: original and approximation.

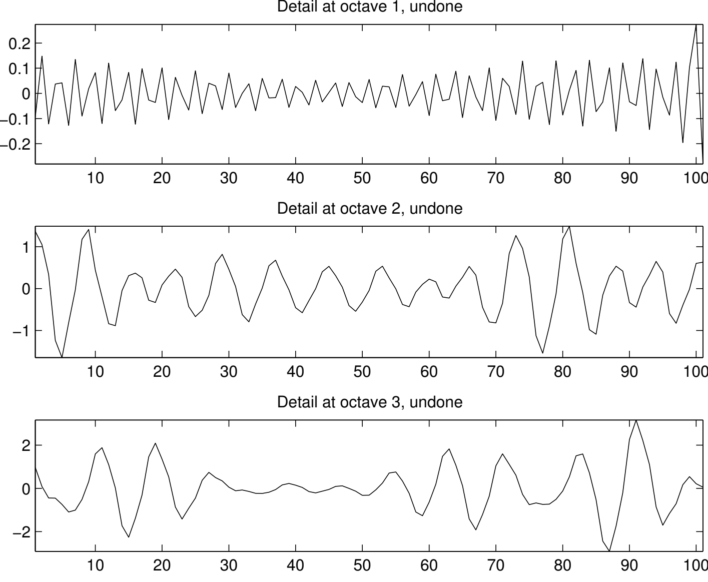



**Figure 10.3:** Results of "DWT undo" for a 1D signal: details for octaves 1–3.

Again, we reconstruct the input signal, and compute the error only to find it to be very small.

```
>> input_recon = A + D(1,:) + D(2,:) + D(3,:);
>> error = sum(abs(x - input_recon))

error =

   3.3799e-10
```

For the 2D case, let's start with an image. Any relatively large image will do. We will store it (using *double* precision) in variable $x$. Next, we call "DWT_undo2" to perform the 2D DWT and IDWT using wavelet `db4`, for eight octaves. After this, we reconstruct the signal and find the error.

```
tic
x = double(imread('dog256x256.gif'));
[A, B, C, D] = DWT_undo2(x, 'db4', 8);
y = A;
for k=1:8
    tempB(:,:) = B(:,:,k);
    tempC(:,:) = C(:,:,k);
    tempD(:,:) = D(:,:,k);
    y = y + tempB + tempC + tempD;
end
error= sum(sum(abs(x-round(y))))
toc
```

The output appears below. The commands `tic` and `toc` allow us to specify the start and stop of a virtual stopwatch to know how long the code takes. Note that this includes not only the time for "DWT_undo2," but also the time taken by the reconstruction. The total time is just under 17 seconds.

```
error =

     0

Elapsed time is 4.191529 seconds.
```

Exactly zero error? This comes from the `round` function, essentially converting the reconstructed signal to integer values. As a grayscale image, the original signal's values are all integers between 0 and 255, so it makes sense to round the result.

The matrices $B$, $C$, and $D$ above store the image details for the number of octaves given. That is, $B$ not only contains the detail matrix corresponding to octave 1, it also stores the detail matrix for octaves 2, 3, etc. The differences between $B$, $C$, and $D$ are the filters we use to generate each (low-high, high-low, or high-high). These matrices hold the details, and can be used to see edges within the image. In fact, $B$, $C$, and $D$ are commonly referred to as the horizontal, vertical, and diagonal components of an image. This is why MATLAB's documentation (`help dwt2`) lists the outputs as "CA, CH, CV, and CD," for approximate, horizontal, and so forth.

## 10.6 Wavelet Transform with Matrices

This section will show how the DWT can be computed with matrices. We will consider only one channel, since the second channel will work the same way and matrix addition is easy to do. For the second channel, we could either create a separate matrix or reuse the same one by interlacing the channels or appending one to the bottom of the other.

First, consider the channel without filter banks. We have two filters along the channel, an analysis filter $h$ followed by a synthesis filter $f$. Our transformed data $w$ will be the input $x$ convolved with $h$.

$$w = h * x$$

Then, we convolve $w$ with $f$ to create our channel output.

$$output = f * w$$

So the whole idea here boils down to performing convolution with a matrix operation. Consider a single value, $w_n$ from the first convolution. To make things simple, we will assume that $h$ has four values. Thus,

$$w_n = h_3 x_{n-3} + h_2 x_{n-2} + h_1 x_{n-1} + h_0 x_n.$$

Notice that this could be accomplished with a $1 \times 4$ matrix multiplied by a $4 \times 1$ matrix.

$$w_n = [h_3 \; h_2 \; h_1 \; h_0] \begin{bmatrix} x_{n-3} \\ x_{n-2} \\ x_{n-1} \\ x_n \end{bmatrix}$$

All we need to do is generalize the $h$ matrix for all $n$.

$$\begin{bmatrix} w_n \\ w_{n+1} \\ w_{n+2} \\ w_{n+3} \\ \vdots \end{bmatrix} = \begin{bmatrix} h_3 & h_2 & h_1 & h_0 & 0 & 0 & \cdots & 0 \\ 0 & h_3 & h_2 & h_1 & h_0 & 0 & \cdots & 0 \\ 0 & 0 & h_3 & h_2 & h_1 & h_0 & 0 & \vdots \\ \vdots & & & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 0 & h_3 & h_2 & h_1 & h_0 \end{bmatrix} \begin{bmatrix} x_{n-3} \\ x_{n-2} \\ x_{n-1} \\ x_n \\ x_{n+1} \\ x_{n+2} \\ x_{n+3} \\ \vdots \end{bmatrix}$$

We can create a similar matrix with $f$ to convolve with $w$ for the other filter.

But how do we deal with down-sampling? Surely we do not want to throw away half of our calculations! The key here is to notice that we only have $w$ values for even indices $n$. (Remember that we are considering only the analysis filter for the moment.) In other words, assuming $n$ is even, we want to generate $w_n$, $w_{n+2}$, $w_{n+4}$, etc. We can accomplish this by eliminating every other row in the $h$ matrix above. For the matrix as a whole, this appears that we are shifting the $h$ values over by two positions every time we go down a row.

$$\begin{bmatrix} w_n \\ w_{n+2} \\ w_{n+4} \\ w_{n+6} \\ \vdots \end{bmatrix} = \begin{bmatrix} h_3 & h_2 & h_1 & h_0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & h_3 & h_2 & h_1 & h_0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & h_3 & h_2 & h_1 & h_0 & 0 & \vdots \\ \vdots & & & & & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 0 & 0 & 0 & h_3 & h_2 & h_1 & h_0 \end{bmatrix} \begin{bmatrix} x_{n-3} \\ x_{n-2} \\ x_{n-1} \\ x_n \\ x_{n+1} \\ x_{n+2} \\ x_{n+3} \\ \vdots \end{bmatrix}$$

Now for the up-sampling and reconstruction. Every other input ($w_n$, when $n$ is odd) should be zero.

$$output_n = f_3 0 + f_2 w_{n-2} + f_1 0 + f_0 w_n, \quad n \text{ even}$$

$$output_n = f_3 w_{n-3} + f_2 0 + f_1 w_{n-1} + f_0 0, \quad n \text{ odd}$$

Instead of putting zeros in between $w$ values, we prefer to alter the matrix $f$. This leads us to the following $f$ matrix. Rather than shifting the $f$ values over twice between rows, we write them shifted down by two rows between columns. Looking at the patterns for $f$ and $h$ values, we see the similarity between them and a knight's

move in a chess game.

$$\begin{bmatrix} output_n \\ output_{n+1} \\ output_{n+2} \\ output_{n+3} \\ \vdots \end{bmatrix} = \begin{bmatrix} f_3 & f_1 & 0 & 0 & \cdots & 0 \\ 0 & f_2 & f_0 & 0 & \cdots & 0 \\ 0 & f_3 & f_1 & 0 & & \vdots \\ 0 & 0 & f_2 & f_0 & 0 & 0 \\ 0 & 0 & f_3 & f_1 & 0 & 0 \\ \vdots & & 0 & f_2 & f_0 & 0 \\ 0 & \cdots & 0 & f_3 & f_1 & \ddots \end{bmatrix} \begin{bmatrix} w_{n-6} \\ w_{n-4} \\ w_{n-2} \\ w_n \\ w_{n+2} \\ w_{n+4} \\ w_{n+6} \\ \vdots \end{bmatrix}$$

Taking the two matrices, $H$ and $G$, some interesting properties of these matrices are [3]:

```
% Daubechies_coeffs.m
%
% Daubechies wavelet coeffs.
%

% Highpass (wavelet) coefficients
d0 = (1-sqrt(3))/(4*sqrt(2));
d1 = -(3-sqrt(3))/(4*sqrt(2));
d2 = (3+sqrt(3))/(4*sqrt(2));
d3 = -(1+sqrt(3))/(4*sqrt(2));
% Lowpass (scaling) coefficients
c0 = -d3;
c1 = d2;
c2 = -d1;
c3 = d0;

H = [c3 c2 c1 c0  0  0  0  0  0  0; ...
      0  0 c3 c2 c1 c0  0  0  0  0; ...
      0  0  0  0 c3 c2 c1 c0  0  0; ...
      0  0  0  0  0  0 c3 c2 c1 c0];

G = [d3 d2 d1 d0  0  0  0  0  0  0; ...
      0  0 d3 d2 d1 d0  0  0  0  0; ...
      0  0  0  0 d3 d2 d1 d0  0  0; ...
      0  0  0  0  0  0 d3 d2 d1 d0];
```

```
disp('multiplying Scaling coeffs by their transpose = I');
H*H.'

disp('multiplying Wavelet coeffs by their transpose = I');
G*G.'

disp('multiplying Scaling coeffs by Wavelet coeffs = 0 matrix');
H*G.'
```

When we run the above program, we get the following output:

```
multiplying Scaling coeffs by their transpose = I

ans =

    1.0000    0.0000         0         0
    0.0000    1.0000    0.0000         0
         0    0.0000    1.0000    0.0000
         0         0    0.0000    1.0000

multiplying Wavelet coeffs by their transpose = I

ans =

    1.0000    0.0000         0         0
    0.0000    1.0000    0.0000         0
         0    0.0000    1.0000    0.0000
         0         0    0.0000    1.0000

multiplying Scaling coeffs by Wavelet coeffs = 0 matrix

ans =

   1.0e-17 *

    0.3648   -0.1637         0         0
   -0.1306    0.3648   -0.1637         0
         0   -0.1306    0.3648   -0.1637
         0         0   -0.1306    0.3648
```

From the above run, we see that multiplying a matrix of coefficients by its transpose results in the identity matrix, i.e., $HH^T = I$, and $GG^T = I$. Also, when we multiply the two matrices together, we get a zero matrix: $HG^T = 0$. (Do not forget that each value from the final output above is multiplied by a very tiny exponent. These nonzero values can be attributed to precision error.)

This covers only one octave of the wavelet transform. We can extend this to several octaves, as well as several dimensions. Above, we used the coefficient matrices to define the transform. We can use them recursively to compute further octaves. That is, $w = Hx$ (with $H$ representing the matrix of $h$ values). The other channel's analysis would be $v = Gx$ (where $G$ is just like $H$, except generated from a different filter). We will let $G$ represent the matrix made from highpass (wavelet) coefficients and $H$ represent a similar matrix made from the lowpass (scaling) coefficients. Refer to the top of Figure 10.4 for the analysis of $x$ for three octaves.

First, we will find the first level approximation. We will not keep this, but we use it to compute the other octaves.

$$approximation_1 = Hx$$

To get the next octave's approximation, we use the previous result.

$$approximation_2 = H approximation_1$$

$$approximation_2 = H(Hx)$$

So the outputs of three octaves of the DWT would be:

$$detail_1 = Gx$$

$$detail_2 = G(Hx)$$

$$detail_3 = G(H(Hx))$$

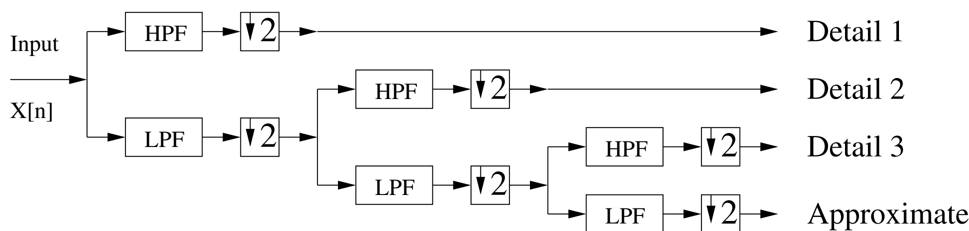
$$approximation_3 = H(H(Hx)).$$

We can calculate these values recursively, as above, or we can create transform matrices that have the same result.

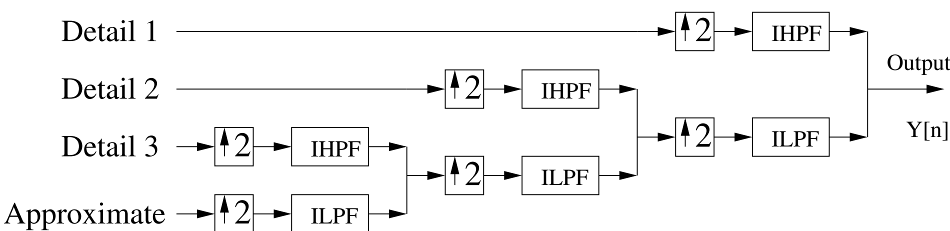
## 10.7 Recursively Solving a *Su Doku* Puzzle

*Su Doku* is a puzzle where digits 1 through 9 appear exactly once in every row, column, and $3 \times 3$ submatrix. The puzzle comes with some of the numbers already filled in, and the object is to fill in the rest. For example, imagine a row with the

The wavelet transform shown for 3 octaves (resolution levels)




Analysis (Forward Transform)




Synthesis (Inverse Transform)

**Figure 10.4:** Three octaves of the one-dimensional discrete wavelet transform.

**Table 10.1:** An example *Su Doku* puzzle.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 3 | 4 | 0 | 6 | 7 | 8 | 0 |
| 4 | 0 | 6 | 7 | 0 | 9 | 1 | 0 | 3 |
| 7 | 8 | 0 | 1 | 0 | 3 | 0 | 5 | 6 |
| 3 | 1 | 2 | 0 | 0 | 0 | 9 | 7 | 8 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6 | 9 | 8 | 0 | 0 | 0 | 5 | 1 | 4 |
| 2 | 7 | 0 | 3 | 0 | 8 | 0 | 9 | 1 |
| 9 | 0 | 1 | 6 | 0 | 2 | 8 | 0 | 5 |
| 0 | 6 | 4 | 9 | 0 | 5 | 2 | 3 | 0 |

first eight numbers given as {3, 1, 5, 9, 7, 6, 8, 4} followed by a blank. We know that the number that goes in the blank cannot be a repeat of any of the numbers already in the row, and by the process of elimination we arrive at the conclusion that 2 must go in the blank spot. With similar logic, we should be able to complete the puzzle. A well-constructed *Su Doku* puzzle has only one solution. But how do we know if this is the case?

We can treat the puzzle as a matrix, and use MATLAB to solve it for us. Sure, this takes the fun out of it, but we will go one step further: we will have MATLAB find *all possible* solutions.

To solve this problem, we will use recursion. We start with the puzzle in an incomplete form, with 0s for the blanks. Next, we will find the first blank (zero), and put a guess in its place. We then check to see if our guess causes a problem; if this number already exists in the row, column, or $3 \times 3$ submatrix. If a conflict exists, we repeat the process with a new guess. To keep it simple, our guesses will always start with 1, then increment to 2, then increment to 3, and so forth until we reach 9.

Table 10.1 shows an example *Su Doku* puzzle with 0s in the place of blanks. Table 10.2 gives the answer. If you have not seen this puzzle before, see if you can solve it on your own.

To solve the problem, we will use three MATLAB files: a program to give the incomplete puzzle and start things off, a function to check to see if the puzzle is valid, and a function to look for blanks and fill them in with guesses. We will call the starting program `sudoku.m`, then the function that checks the puzzle `sudoku_check.m`, and the search and guess function `sudoku_search.m`.

First, let's look at the starting program. We define the puzzle as a matrix, with

**Table 10.2:** The example *Su Doku* puzzle's solution.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 4 | 5 | 6 | 7 | 8 | 9 | 1 | 2 | 3 |
| 7 | 8 | 9 | 1 | 2 | 3 | 4 | 5 | 6 |
| 3 | 1 | 2 | 5 | 6 | 4 | 9 | 7 | 8 |
| 5 | 4 | 7 | 8 | 9 | 1 | 3 | 6 | 2 |
| 6 | 9 | 8 | 2 | 3 | 7 | 5 | 1 | 4 |
| 2 | 7 | 5 | 3 | 4 | 8 | 6 | 9 | 1 |
| 9 | 3 | 1 | 6 | 7 | 2 | 8 | 4 | 5 |
| 8 | 6 | 4 | 9 | 1 | 5 | 2 | 3 | 7 |

zeros in place of the blanks. We next check out the puzzle, to make sure it is valid, using the `sudoku_check` function. If it is not, the search function will never find a solution, so we look for this up front. Then we call our `sudoku_search` function.

```
puzzle = [ 0, 2, 3, 4, 0, 6, 7, 8, 0;
           4, 0, 6, 7, 0, 9, 1, 0, 3;
           7, 8, 0, 1, 0, 3, 0, 5, 6;
           3, 1, 2, 0, 0, 0, 9, 7, 8;
           0, 0, 0, 0, 0, 0, 0, 0, 0;
           6, 9, 8, 0, 0, 0, 5, 1, 4;
           2, 7, 0, 3, 0, 8, 0, 9, 1;
           9, 0, 1, 6, 0, 2, 8, 0, 5;
           0, 6, 4, 9, 0, 5, 2, 3, 0]

if (sudoku_check(puzzle))
    sudoku_search(puzzle);
end
```

The function that checks the puzzle appears below. It works simply by isolating each row, and counting the occurrence of each of the numbers 1 through 9. If it finds more than one occurrence of a number, then it sets the *OK* flag to `false` to indicate that the puzzle has an invalid entry. Next, we check each column in a similar way. Finally, we check every submatrix. Once the function completes, it returns the *OK* flag. The calling program hands `sudoku_check.m` a puzzle, and it gives back a simple `true` or `false` value. Rather than show the whole function, we

will show just the first part that checks each row. See the function listing on the CD-ROM for the checks on the columns and submatrices.

```
% sudoku_check.m
%
% Check a Su Doku puzzle -
%   make sure each number appears in only one row,
%   column, and 3x3 submatrix.
%
function OK = sudoku_check(puzzle)

% assume the puzzle is OK, until we find otherwise
OK = true;

% Check to see if the puzzle matrix is good
% First, check along the rows. r = row number
for r = 1:9
    % Isolate one row of the puzzle
    row = puzzle(r, 1:9);
    % let n be the number (1..9)
    for n = 1:9
        % initialize count for this number
        num_count = 0;
        % examine each column
        for c = 1:9
            % if the puzzle entry matches the number
            % we are looking for, add one to the count.
            if (row(c) == n)
                num_count = num_count + 1;
            end
        end
        % make sure every number 1..9 appears <= 1 time
        if (num_count > 1)
            % we found a duplicate in this row
            OK = false;
        end
    end
end
% See sudoku_check.m for the column check
% and the 3x3 submatrix check
```

Our goal is to show all possible solutions. Since we do not need to keep the solutions that we find, we will simply print them to the screen. The `sudoku_search` function finds the blanks, and if it does not find a blank, it prints the puzzle.

This search function receives a puzzle matrix which may or may not be complete. It sets up a variable called *blank* that we use to indicate the presence of one or more blanks in the puzzle. Actually, we do not care how many blanks there are, we just want to know if there is one. So we assume there are no blanks, then look through the matrix. If we find one, we update our *blank* variable to indicate this, and go on to the next part. If we find a blank in the first spot that we look, why should we bother looking through the rest of the puzzle? This explains why we use `while` loops instead of `for` loops; we might want to exit the loops early.

In the next part, there are two possibilities. Either we found a blank and exited the loops early, or we looked through the whole puzzle and did not find one. We assume that all puzzles passed to the search function are valid, so if there are no blanks, then we must have a solution. In this case, we print the matrix and exit the function. If we found a blank, then we want to try a guess in that position. Once we have a guess in place, we check to make sure the guess is valid. Just because it is valid does not mean that it will be part of the solution; we may find that it leads to a contradiction later. For example, looking at the puzzle in Table 10.1, we see the upper-left corner has a blank, and that both 1 and 5 are valid possibilities here. But only one of these possibilities is the correct answer.

When we have a valid guess in our puzzle, we need a way to search for more blanks, and fill them in. (Doesn't this sound familiar?) This is where recursion comes in; the search function *calls itself* with the updated puzzle! The computer keeps track of the functions that we call, and allows us to call a function from within itself. Of course, we have to be careful here—done carelessly, the function could call itself over and over again indefinitely, possibly crashing the computer as a result. Our solution will work because we take a step in the right direction every time we call our search function. There are many possible calls we could make, but they are finite. We can even start with a matrix of zeros, and the computer will print every possible *Su Doku* solution.

Our solution also rests on the fact that the search function has a `function` keyword in its definition. The computer will keep a copy of its data separate from other functions. It allocates variables as needed, then gets rid of these variables when the function quits. Instead of there being one matrix called *puzzle*, there are actually many of them depending on how many calls we have made to `sudoku_search`. All these different versions of the puzzle work to our advantage, since they allow back-tracking by the computer. That is, if the search function inserts a bad (but valid) guess, it will call itself with that guess in place. If the next call of the search

function cannot make the puzzle work, control will come back to the previous call of the function, and it can try a new guess. The code for the search function appears below.

```
%
% sudoku_search
%
%  Try to recursively fill out a Su Doku puzzle.
%
function blank = sudoku_search(puzzle)

% assume that there are no blanks,
% until we find one
blank = false;

% Part I:
% look through the puzzle, until we find a blank
% or get to the last puzzle position
row = 1;
% examine the rows
while ((row <= 9) && (~blank))
    % examine every column
    col = 1;
    while ((col <= 9) && (~blank))
        % Does the puzzle have a blank?
        if (puzzle(row,col) == 0)
            blank = true;
        else
            % Go to next column
            col = col + 1;
        end
    end
    % Go to next row, unless we are done.
    if (~blank)
        row = row + 1;
    end
end

% Part II:
% Did we find a blank?
```

```
if (blank)
    % found a blank to fill in,
    % so try all values 1..9
    for value=1:9
        % Try the current value there
        puzzle(row,col) = value;
        % maybe this value is already in this row,
        % or in this column, or 3x3 submatrix ?
        if (sudoku_check(puzzle))
            % the value does not cause a problem
            % so try to fill in any remaining blanks
            sudoku_search(puzzle);
        end
    end
else
    % Show the puzzle if no blanks were found.
    % That is, we know the puzzle checks out
    % (or this function would not have been called),
    % so if no blanks are left, then we have a
    % solution!
    puzzle
end
```

Recursion allows the programmer to solve a complex problem by writing a relatively small function. First, a recursive function checks to see if it is done. If not, it takes the problem one step closer to the solution and calls itself. We could have written the search function in a way that does not use recursion, but then we would have to keep track of what changes we made to the puzzle and implement our own mechanism to undo changes as needed.

Now that we have seen how the puzzle should work, let's see an example that does not work so well. In the example puzzle above, we have 4 as the first entry on the second row. But what if we replace this with a blank? We will test it out with the program.

```
>> puzzle = [ 0, 2, 3, 4, 0, 6, 7, 8, 0;
              0, 0, 6, 7, 0, 9, 1, 0, 3;
              7, 8, 0, 1, 0, 3, 0, 5, 6;
              3, 1, 2, 0, 0, 0, 9, 7, 8;
              0, 0, 0, 0, 0, 0, 0, 0, 0;
              6, 9, 8, 0, 0, 0, 5, 1, 4;
```

```
                2, 7, 0, 3, 0, 8, 0, 9, 1;
                9, 0, 1, 6, 0, 2, 8, 0, 5;
                0, 6, 4, 9, 0, 5, 2, 3, 0]

>> sudoku_search(puzzle);
```

Shortly afterwards, MATLAB responds with the following output.

```
puzzle =

     1     2     3     4     5     6     7     8     9
     4     5     6     7     8     9     1     2     3
     7     8     9     1     2     3     4     5     6
     3     1     2     5     6     4     9     7     8
     5     4     7     8     9     1     3     6     2
     6     9     8     2     3     7     5     1     4
     2     7     5     3     4     8     6     9     1
     9     3     1     6     7     2     8     4     5
     8     6     4     9     1     5     2     3     7


puzzle =

     1     2     3     4     5     6     7     8     9
     5     4     6     7     8     9     1     2     3
     7     8     9     1     2     3     4     5     6
     3     1     2     5     6     4     9     7     8
     4     5     7     8     9     1     3     6     2
     6     9     8     2     3     7     5     1     4
     2     7     5     3     4     8     6     9     1
     9     3     1     6     7     2     8     4     5
     8     6     4     9     1     5     2     3     7
```

As we see from the output, the puzzle now has two solutions. Examining the two solutions shows that the columns 1 and 2 in rows 2 and 5 are the only differences.

## 10.8 Converting Decimal to Binary

To convert a decimal number to binary, most people are familiar with a divide-by-two algorithm. We start with the decimal number and divide by 2, keeping track of

**Table 10.3:** Converting from decimal to binary.

| **number** | **number / 2** | **remainder** |
|---|---|---|
| 38 | 19 | 0 |
| 19 | 9 | 1 |
| 9 | 4 | 1 |
| 4 | 2 | 0 |
| 2 | 1 | 0 |
| 1 | 0 | 1 |

the remainders (which must be either 0 or 1). Repeat this until we get zero for the result. Then we write these remainders from the bottom to the top.

For example, suppose we want to convert the number 38 to binary. As we can see from Table 10.3, the remainders (in reverse order) are 100110. Since we have not established how many bits we use in storing the fixed-point binary number, and because the left-most bit typically indicates sign, we append a zero to the left to avoid confusion. This gives us 0100110 as our answer. We can check this by converting it back to decimal: $32 + 4 + 2 = 38$.

This assumes that the decimal number is positive. When it is negative, use the absolute value instead. At the end, convert all 0s to 1s and all 1s to 0s, then add 1. Keep in mind that $1 + 1 = 10$ in binary. This final step (called two's complement) returns the correct bit pattern for negative numbers. We will not worry about negative numbers below, since the floating-point standard uses a sign bit instead of using two's complement.

But what if we wanted to convert 38.1 to a fixed-point binary representation? We already know the bits to the left of the radix point, and we use a similar algorithm to find the fractional part.

We will start with the fractional number, multiply by 2 (or, if you prefer to think of it this way, divide by $2^{-1}$), and keep track of the whole part. We repeat this with the fractional part of the previous number until we get a zero for the fractional part.

Notice that in Table 10.4 the last fractional number, 0.4, was already seen in the table. This fractional part does not have a compact binary representation. Instead, like the number 1/3 in decimal, it repeats forever. The best we can do is store an approximation of $.0\overline{0011}$.

Using these two decimal-to-binary techniques together, we conclude that 38.1

**Table 10.4:** Converting from a decimal fraction to fixed-point binary.

| **fractional number** | **number $\times$ 2** | **whole part** |
|---|---|---|
| 0.1 | 0.2 | 0 |
| 0.2 | 0.4 | 0 |
| 0.4 | 0.8 | 0 |
| 0.8 | 1.6 | 1 |
| 0.6 | 1.2 | 1 |
| 0.2 | 0.4 | 0 |

converted to fixed-point binary is:

$$0100110.0\overline{0011}.$$

What if we want to store this value as a floating-point number? The Institute of Electrical and Electronics Engineers has a standard for floating-point storage in 32 or 64 bits, commonly called the *IEEE 754* standard [38]. We break the binary representation into three fields: the sign bit, the significand, and the exponent. (The *significand* may also be called by the older name of *mantissa*.) In other words, the number appears as $\pm significand \times 2^{\pm exponent}$. Of course, the size of the storage word directly affects precision; in the IEEE 754 standard, we label the *precision* the number of bits used to represent the significand. As the example above shows, the significand that we store depends upon the precision. With 32 bits for the entire floating-point representation (called single precision), we use 23 bits for the significand. When we have a 64-bit word (double precision), we use 52 bits for it. One bit stores the sign, and the remaining bits store the exponent. Therefore, the exponent will consist of 8 or 11 bits, according to the precision.

You may also see the precision given as 24 bits and 53 bits, respectively. The one bit difference comes from the "hidden" bit, which is not explicitly stored [38]. Remember that the significand consists only of 0s and 1s, and the magnitude of each bit depends on the exponent. For example, $0.1 \times 2^0$ has the same value as $1.0 \times 2^{-1}$. We can always make the first bit a 1 by altering the exponent, so why not require it? This way, we do not need to store that bit. The IEEE 754 standard uses this strategy.

One final note about the IEEE 754 standard is that the exponent ranges from $-126$ to $+127$ (for an 8-bit exponent), or $-1022$ to $+1023$ (for 11-bit exponents).

The computer stores these exponents with a *bias* of 127 and 1023, respectively. To find the actual exponent value to use, we subtract the *bias* value from the value stored. For example, suppose the stored exponent value for a 32-bit floating-point value is 10000011 in binary, or 131 in decimal. The actual exponent would be $131 - 127 = 4$, so the number in the significand is multiplied by $2^4$.

With 126 possible negative exponent numbers, 127 positive numbers for the exponent, and 1 number for a zero exponent, we have 254 possible values. The other two possibilities allow for special meanings, such as a zero significand when all exponent bits are 0. Otherwise, the bits stored in the significand would be interpreted as starting with a hidden 1.

Let's create a `double` value and print it to the screen in hexadecimal. The formatting string in the `sprintf` command may be a bit daunting. The percent sign indicates that a variable should be printed, as in the programming language C. The `x` and `\n` are standard C-like formatting, meaning hexadecimal and new-line, respectively. We could do without the `\n`, but it gives us a nice space before the next MATLAB prompt. The final formatting parameter `b` tells MATLAB to use a double type. Thus, we see the `double` values as hexadecimal strings that can easily be converted to binary with a lookup chart.

```
>> disp(sprintf('%bx \n',0.0))
0000000000000000

>> disp(sprintf('%bx \n',-0.0))
8000000000000000

>> disp(sprintf('%bx',38.1))
40430ccccccccccd
```

We see from the above MATLAB output that zero and negative zero are actually stored in two different ways! MATLAB takes care of this for us, as the following code shows.

```
if (-0.0 == 0.0)
   disp('equal');
else
   disp('not equal');
end
```

When we run it, we see that MATLAB behaves as we expect.

```
equal
```

In other words, though zero and negative zero have different internal representations, MATLAB will treat them as equal.

Let's do one more example of a number stored in the IEEE 754 standard. Notice the `t` parameter that says to treat the number as a 32-bit floating-point value.

```
>> disp(sprintf('%tx',16.875))
41870000
```

We can use the chart in Table 10.5. (As a side note, you may see hexadecimal values in either uppercase or lowercase. For example, 1a3f is the same as 1A3F.) Writing out the binary values, we have:

| hex: | 4 | 1 | 8 | 7 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|
| binary: | 0100 | 0001 | 1000 | 0111 | 0000 | 0000 | 0000 | 0000 |

Now we rewrite the binary string, according to the floating-point standard.

| sign | exponent | significand |
|---|---|---|
| 0 | 10000011 | 00001110000000000000000 |

The interpretation is positive for the sign. For the exponent we have $10000011_2 = 131_{10}$, subtracting the bias we get $131-127 = 4$. With the hidden bit, the significand becomes 1.0000111. The result is $+1.0000111 \times 2^4$, or simply 10000.111. If we convert the left half to decimal, we find it is $2^4 = 16$. The right half converts to $2^{-1} + 2^{-2} + 2^{-3} = .875$, so the original number must be 16.875, just like in the previous `sprintf` statement.

## 10.9 Frequency Magnitude Response of Sound

The following program plots the frequency magnitude response of sound. Since it repeats itself in a loop, the graph appears to change in (almost) real time.

The first few lines set up the variables to save time later. This includes an array that we will use to show only the first half of the frequency information, as well as another array for the x-axis, to show it in terms of actual frequencies instead of sample number.

```
% show_sound.m
%
% Get the FMR of a sound
%
```

**Table 10.5:** Hexadecimal to binary chart.

| **hexadecimal digit** | **binary** | **decimal equivalent** |
|---|---|---|
| 0 | 0000 | 0 |
| 1 | 0001 | 1 |
| 2 | 0010 | 2 |
| 3 | 0011 | 3 |
| 4 | 0100 | 4 |
| 5 | 0101 | 5 |
| 6 | 0110 | 6 |
| 7 | 0111 | 7 |
| 8 | 1000 | 8 |
| 9 | 1001 | 9 |
| a | 1010 | 10 |
| b | 1011 | 11 |
| c | 1100 | 12 |
| d | 1101 | 13 |
| e | 1110 | 14 |
| f | 1111 | 15 |

```
figure(1);  % bring figure 1 to the front
fs = 44100;
N = 4410*1;
half = 1:ceil(N/2);
MaxFreq = ceil(max(half)*fs/N);
xaxis = half*fs/N;
```

Next, we try to use the `audiorecorder` feature of MATLAB. Either it will be supported, or the program will later try to use the `wavrecord` instead. The code is not perfect, though, since some computers with an older version of MATLAB will not be able to use the `wavrecord` function either.

```
% detect if we should use audiorecorder
try
    audiorec = 1;
    x_object = audiorecorder(44100, 16, 1);
catch
    audiorec = 0;
    disp('No audiorecorder support.');
end
```

The heart of the program is a `for` loop that records sound then displays it on screen. The `pause` command must be in place to give the system a chance to update the screen. The other commands simply make the plot look nice and consistent.

```
for i=1:10
    % record some sound
    if (audiorec)
        recordblocking(x_object, 0.5);
        % find FFT's magnitudes
        X = abs(fft(getaudiodata(x_object, 'double')));
    else
        x = wavrecord(N, fs, 1);
        % find FFT's magnitudes
        X = abs(fft(x));
    end

    % plot the FMR
    plot(xaxis, X(half));
    % make the plot look good
```

```
        axis([0 MaxFreq 0 100]);
        xlabel('Frequency (Hz)');
        ylabel('Amplitude');
        title('Frequency Magnitude Response for sound');
        % Give the system time to show the graph
        pause(0.01);
    end
```

When running, the program records sound and shows its frequency magnitude response (FMR) as quickly as possible. To make it run longer, simply adjust the numbers in the `for` line. Also, users with both `wavrecord` support and a later version of MATLAB may want to try both recording functions. The `wavrecord` function allows a smaller interval of recording time to make the program seem to run in real time.

Figure 10.5 contains an example of the program's output, where the microphone's input is a whistle sound. As the program runs, the graph changes according to the input signal. With this program, we can see how simple sounds (like the whistle) appear in the frequency-domain, or look at more complex signals, such as speech. If a musical instrument is played nearby, the harmonics from the notes will show up as spikes in the frequency-domain.

## 10.10 Filter Design

We saw how various filters affect a signal. But how does one determine the filter coefficients to use, to achieve a certain effect? We examine FIR filter design in this section.

### 10.10.1 Windowing Methods

MATLAB provides the `fir1` function to design FIR filters in the Signal Processing toolbox. As its documentation points out, this function uses the *windowing method*. This means that the filter coefficients have a gradual rise based on the window given, to minimize the ill-effects cause by sudden transitions. For example, consider the code below. We have an impulse function $x$, filtered by two different filters. The first step is to make the impulse function.

```
imp = zeros(1, 1000);
imp(50) = 1;
```

Now we will filter this with filter coefficients $\{1, 1, 1, 1, 1\}$. Once filtered, we show the frequency magnitude response.

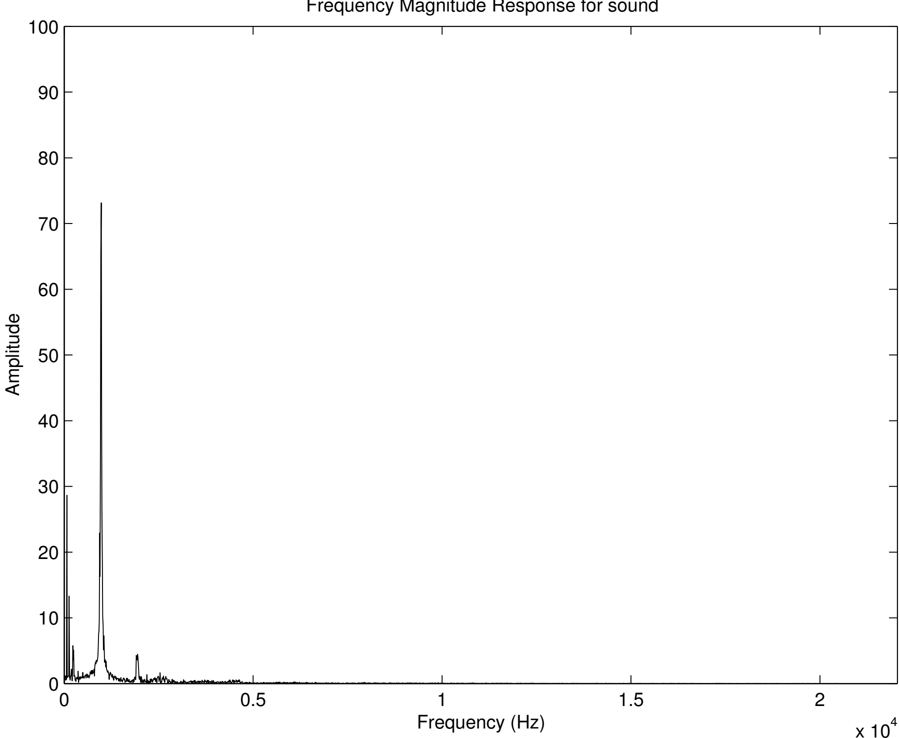



**Figure 10.5:** Example of the "show_sound" program.

```
myfilt = [1, 1, 1, 1, 1];
y = conv(imp, myfilt);
Y = fft(y);
maxY = max(abs(Y));
half = 1:ceil(length(y)/2);
subplot(2,1,1);
plot(half, abs(Y(half))/maxY, 'b');
title('FMR for filter coeffs [1, 1, 1, 1, 1]');
axis tight;
```

We can compare this to the effect of another filter, one where the filter coefficients more gradually approach 1, then gradually fall.

```
myfilt2 = [1/3, 2/3, 1, 2/3, 1/3];
y2 = conv(imp, myfilt2);
Y2 = fft(y2);
maxY2 = max(abs(Y2));
subplot(2,1,2);
plot(half, abs(Y2(half))/maxY2, 'b');
title('FMR for filter coeffs [1/3, 2/3, 1, 2/3, 1/3]');
axis tight;
```

From the graph, shown in Figure 10.6, we see that both sets of filter coefficients produce a lowpass filter. However, the top graph shows a sizeable *side lobe* where the frequency response becomes larger and then smaller. While the frequency response is very small around index 200, we see that it becomes much larger after this point. On the other graph, we see the frequency response approach zero a bit more slowly (we say it has a wider *main lobe*). We also see that it has a much smaller side lobe. These graphs demonstrate the effect of a window; in fact, the second filter's coefficients were chosen from the simple "triangle" window. In other words, the first set of filter coefficients were "windowed" to make the second set.

The command below returns the 5-value "triangle" window function. Note that the `window` function is part of the optional Signal Processing toolbox in MATLAB.

```
>> w = window(@triang, 5)

w =

    0.3333
    0.6667
```

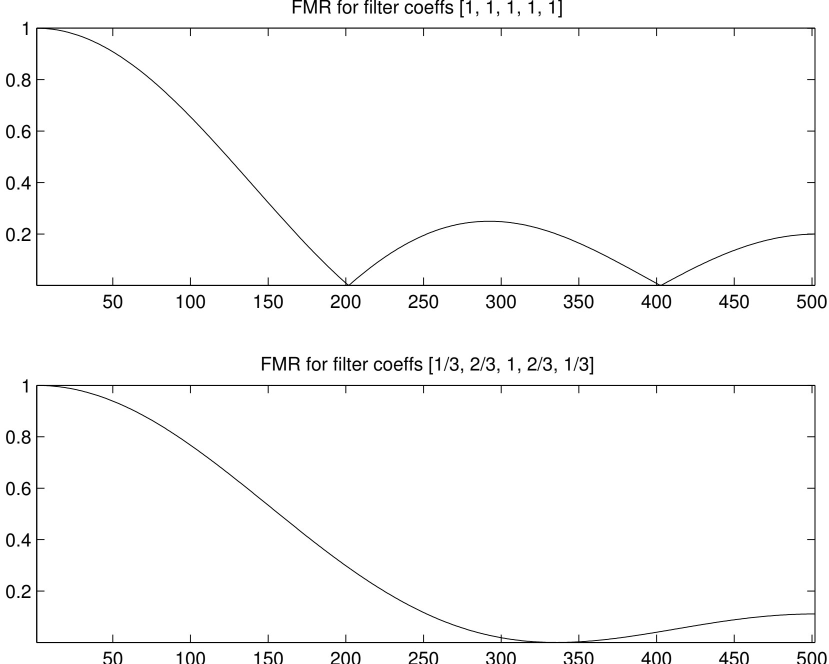



**Figure 10.6:** Two similar filters, with and without a gradual transition.

```
1.0000
0.6667
0.3333
```

Now that we have an idea about how windows apply to filter coefficients, we return to the topic of FIR filter design. If we have an impulse function in the time-domain, we need an infinite number of frequency-domain values to represent it. The reverse is also true; a frequency-domain impulse requires an infinite number of time-domain points. To make a time-domain filter, we decide how its frequency-domain response should look. We can control the cutoff frequency, that frequency above which a lowpass filter starts zeroing out frequency content. Ideally, this would be a binary function where all frequency content below the cutoff would remain the same while content above the cutoff would be zero. In practice, the best we can hope for is a steep slope. We can also control how many filter coefficients we have, called the taps or order. (The order equals number of taps minus 1.) So, to make a reasonable compromise, we window the filter coefficients to the number of taps given. More coefficients give a frequency response that better approximates the ideal, but at the expense of more processing (be it more software processing or more hardware area needed to implement the filters).

### 10.10.2 Designing an FIR Filter

Let's try to design an FIR filter. We will start with the frequency response that we want, and work from there. For simplicity, we will have a lowpass filter with a cutoff frequency at 30%. We need an impulse function to work with, so we define and use *imp*. We will use the length *MAXLEN* a few more times, so we might as well define it now.

```
% create an impulse function
MAXLEN = 1000;
imp = zeros(1, MAXLEN);
imp(50) = 1;
```

Next, we need to define our cutoff frequency in terms of how many frequency-domain values to use; 150 out of 500 corresponds to 30%, so we will use this. (The 500 comes from half the spectrum size of *MAXLEN*.) Also, we will define the number of coefficients to use.

```
low_pass_end = 150;
taps = 51;
```

We simulate the response we want in the frequency-domain as an ideal function, *fx*. Most of the response should be zero, with the lower end including ones. That is, the filter should allow the bottom 30% of frequencies to pass through. Also, we set the top half of the ideal function *fx* as a mirror image of the bottom half, as we would expect for a frequency magnitude response of a real-valued function. (This corresponds to the top half of the frequency magnitude response that we do not plot.)

```
fx = zeros(1, MAXLEN);
fx(1:low_pass_end) = 1;  % lowpass
% include mirror image of lowpass
fx((MAXLEN-low_pass_end):MAXLEN) = 1;
```

Once we have our ideal frequency response specified, we convert it to the time-domain. We shift it to move the center around the midpoint, instead of around zero. We can do this since the magnitudes of the frequency-domain data would remain the same, only the phases would change (see the material on DFT shifting theory, section 6.5).

```
% convert to time-domain
x = ifft(fx);
% shift frequency data
mid_point = floor(MAXLEN/2);
x2 = [x(mid_point+1:MAXLEN), x(1:mid_point)];
```

We get the requested number of filter coefficients by taking an equal number from before the midpoint and after the midpoint. Including the midpoint, we have an odd number of filter coefficients (which explains why we selected 51 for the number of taps above).

```
half_taps = floor((taps-1)/2);
b2 = x2(mid_point - half_taps : mid_point + half_taps);
```

The variable $b2$ stores the filter coefficients that we want to use. However, we cannot use them directly, since they are complex values. We may be tempted to use the `abs` function to convert them to real values, but then they would all be positive. To preserve the signs, we can examine each value and store its sign as well as magnitude. In the process, we create a new variable $c2$ to hold these values.

```
for k=1:length(b2)
```

```
        if (sign(b2(k)) < 0)
            c2(k) = -abs(b2(k));
        else
            c2(k) = abs(b2(k));
        end
    end
```

Finally, we have filter coefficients that we can use. Here we omit the details for plotting these coefficients and the resulting frequency magnitude response, but the program `filter_test2.m` contains the commands. Figure 10.7 shows the results.

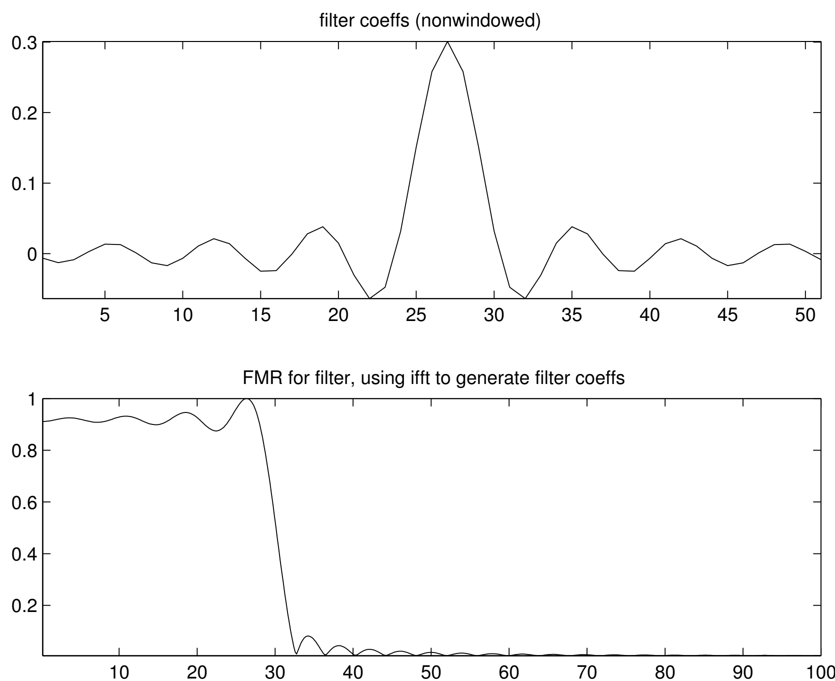



**Figure 10.7:** Using nonwindowed filter coefficients.

As we see from Figure 10.7, the frequency magnitude response contains ripples. We saw that windowing can reduce these ripples, so we will next improve our filter coefficients with a window. Here we use a Hamming window, one of many possible choices. With the window $w$, we multiply each value in $c2$ to create a new set of filter coefficients, $d2$. You may notice the transpose on the window $w$; this simply

changes it from a column vector to a row vector so that the `.*` operation does not have a problem with a mismatch of matrix dimensions.

```
w = window(@hamming, taps);
d2 = w.' .* c2;
```

Figure 10.8 shows the new filter coefficients and an improved frequency magnitude response. At first glance, the filter coefficients shown in this figure look just like the ones in Figure 10.7. The differences are best seen in the first 10 and last 10 filter coefficients. The frequency magnitude response looks much better in Figure 10.8 since the ripples have been smoothed out. (The ripples are still there, just greatly reduced, as a logarithmic plot would show.)

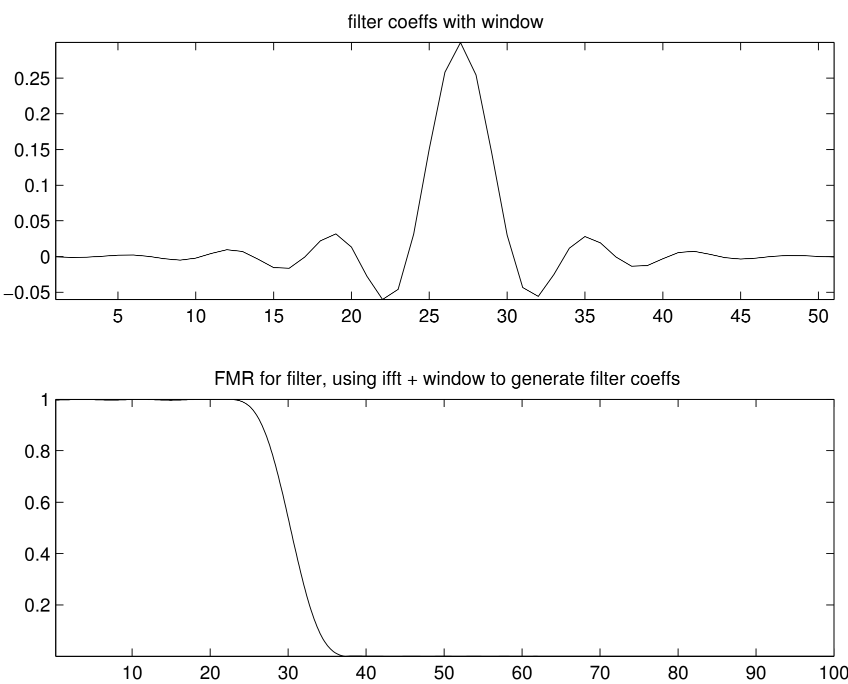



**Figure 10.8:** Using windowed filter coefficients.

Now let's see what the `fir1` command will do for us. Here we ask for the same number of filter coefficients, with the same cutoff frequency (30%). We request $taps - 1$ coefficients, since the function expects the order. That is, starting the

count from 0, the order is the highest index value. So the `fir1` will return one more coefficient than we specify.

```
b1 = fir1(taps-1, 0.3);
```

Figure 10.9 shows the two sets of filter coefficients and their respective frequency magnitude responses. The filter coefficients look very similar (and are very similar in value), even though they are not lined up. For the frequency magnitude responses, we see the plots are on top of one another. To really see a difference, we would need a logarithmic plot.

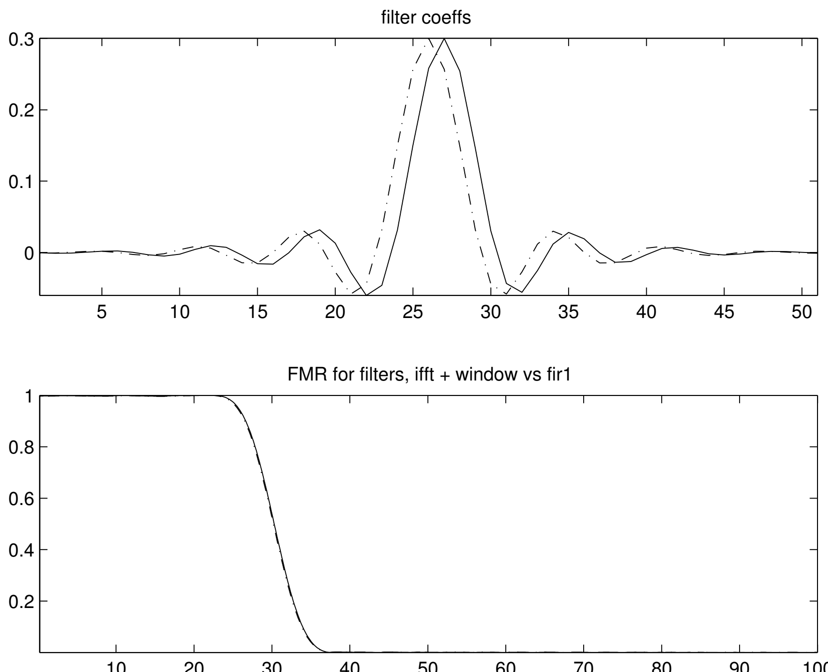



**Figure 10.9:** Windowed filter coefficients and those generated by `fir1`.

Our example FIR filter coefficients closely match the ones generated by the `fir1` function in MATLAB. From this discussion, you should be able to generate coefficients for any lowpass filters you want simply by changing the parameters (number of taps and cutoff frequency). But what if you do not want a lowpass filter? Multiplying the filter coefficients by a sinusoid has the effect of moving the frequency magnitude response in the frequency-domain.

A high-frequency sinusoid would jump from $-1$ to $+1$, which we simulate. A final modification of the filter coefficients will be to apply an alternating flip to the sign of the values.

```
f2 = d2;
flip = 1;
% flip every other sign
for k=1:length(f2)
    if (flip == 1)
        f2(k) = -f2(k);
        flip = 0;
    else
        flip = 1;
    end
end
```

When we run this code, our filter coefficients will appear as in Figure 10.10. As the frequency magnitude response shows, the filter now attenuates low frequencies and passes the high frequencies. Thus, we have created a highpass filter. We can also have MATLAB give us a highpass filter, again with the `fir1` command. Notice how we use 0.7 instead of 0.3 for the cutoff. What would be the result if we left it as 0.3?

```
b3 = fir1(taps-1, 0.7, 'high');
```

This section gives an overview of windowing functions, and shows why they are useful. Also, we explored making our own FIR filters both with a MATLAB Signal Processing toolkit command, as well as our own program. Though this program uses the `window` command, also part of the Signal Processing toolkit, it should be clear to the reader how this could be done without that toolkit.

## 10.11 Compression

Compression programs take the redundancy out of data to store it in a compact form. There are *lossy* compression algorithms, that tolerate some loss of information. For example, the popular JPEG and MP3 compression standards (files with extensions of `.jpg` and `.mp3`, respectively) do not return an exact copy of the original to the user, but a version close enough to the original that the user should not notice the difference after uncompressing it. Another possibility is a *lossless* compression algorithm. Programs such as *gzip* and *winzip* store data compactly, but do not lose

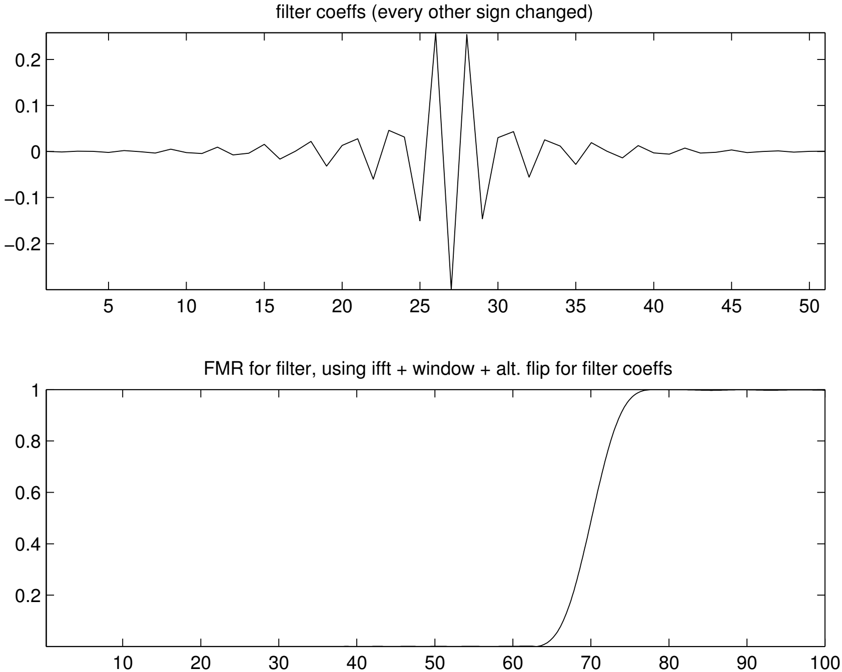



**Figure 10.10:** Alternating flip for the windowed filter coefficients.

any information in the process. Obviously, there are times when a lossy compression works well, such as with images for a Web page, and other times when a lossless compression program should be used instead, like when storing a program or word-processing document.

JPEG stands for Joint Photographic Experts Group. This group has set a couple of standards for images, namely the "basic" JPEG standard and JPEG 2000 (with extension `.jp2`).

There are three steps to compression algorithms, namely transformation, quantization, and entropy coding. Applying a transform should remove some of the redundancy in the data. Quantization maps the data to a set number of values, such as rounding the transformed data to the nearest integer. Of course, data may be lost in this step. The final compression step, entropy coding, represents the quantized data in a compact way. This is much like the way people communicate with instant messenger applications; why spell out "how are you doing today" when "hw RU 2day" gets the idea across in less than half the number of characters?

### 10.11.1 Experimenting with Compression

Let's experiment with an image to see how JPEG compression affects an image. First, we need a suitable noncompressed image. The code below uses a file called *PIA02187.tif*, an image from the planet Mars. (This image was obtained from NASA's Jet Propulsion Laboratory, *http://photojournal.jpl.nasa.gov/targetFamily/Mars*, posted on February 17, 2006.) The `.tif` extension means that the image follows the TIFF (Tagged Image File Format) standard. TIFF files may be compressed, or may be stored as "raw" data. The command below reads the image and stores it in the `uint8` array *original*. The filename comes first, followed by the file type.

```
original = imread('PIA02187.tif', 'TIFF');
```

If you are not able to locate this image, any suitable, uncompressed, grayscale image should work.

Next, we save the image as several different files. The `imwrite` command takes the variable storing the image data as the first parameter, followed by the name of the file to create, then the type of image format to use. The last two parameters specify that the optional parameter "Quality" should be set to the number given, up to 100. We will use the term "image of quality 25" to refer to the image stored in the last line below. The number does *not* really indicate quality, but refers only to the parameter passed to the `imwrite` function.

```
imwrite(original, 'PIA_100.jpg', 'JPEG', 'Quality', 100);
```

```
imwrite(original, 'PIA_75.jpg', 'JPEG', 'Quality', 75);
imwrite(original, 'PIA_50.jpg', 'JPEG', 'Quality', 50);
imwrite(original, 'PIA_25.jpg', 'JPEG', 'Quality', 25);
```

We will momentarily use the file sizes to tell us about the compression. But we should not compare the JPEG images to the TIFF image, since these two different formats vary. Our purpose is not to compare the two different formats, but to investigate the compression and quality of one image. Therefore, we need to store the image in a lossless fashion. Note that the image may still be compressed (stored in a compact way), but without losing information. Fortunately, storing the image as a lossless JPEG is easy, as the command below demonstrates.

```
imwrite(original, 'PIA_noloss.jpg', 'JPEG', 'Mode', 'lossless');
```

We can check on the files from the MATLAB prompt. The `ls` command (and `dir` command) work like they do from a command-line interface (such as a terminal window under Linux, Unix, and OS X, or the *command.exe* program under Microsoft DOS or Windows). Related commands like `cd` (change directory), `pwd` (print working directory), and `mkdir` (make directory) are also supported.

```
>> ls -l PIA*
-rw-r--r--   1 mweeks  staff  1446095 Feb 17 18:41 PIA02187.tif
-rw-r--r--   1 mweeks  staff  1074473 Feb 17 18:45 PIA_100.jpg
-rw-r--r--   1 mweeks  staff   142951 Feb 17 18:45 PIA_25.jpg
-rw-r--r--   1 mweeks  staff   209001 Feb 17 18:45 PIA_50.jpg
-rw-r--r--   1 mweeks  staff   298521 Feb 17 18:45 PIA_75.jpg
-rw-r--r--   1 mweeks  staff   978460 Feb 17 18:45 PIA_noloss.jpg
```

As we see from the output, the file sizes are 1.4 megabytes for the original, 140 kilobytes for the compressed one with a quality of 25, 204 kilobytes and 291 kilobytes for the compressed files with qualities 50 and 75, respectively. It should be obvious that the quality numbers do not indicate percentage of the file size, otherwise we would expect the file *PIA_25.jpg* to be half the size of *PIA_50.jpg*.

The lossless version occupies 956 kilobytes of space. Oddly, we see that the compressed image with quality of 100 is actually 94 kilobytes *larger* than the lossless image! To properly explain this requires a detailed examination of the underlying JPEG algorithm, which is outside the scope of this text. Suffice it to say that there are times when the extra steps in a compression algorithm produce more overhead than savings.

Now that we have the same image stored as several different compressed files, we compare them. To do this, we need to know the files' contents, so we read them into separate matrices.

```
compress_no_loss = imread('PIA_noloss.jpg', 'JPEG');
compress100 = imread('PIA_100.jpg', 'JPEG');
compress75 = imread('PIA_75.jpg', 'JPEG');
compress50 = imread('PIA_50.jpg', 'JPEG');
compress25 = imread('PIA_25.jpg', 'JPEG');
```

Plotting the images with the `imshow` command will show that they all look the same. A careful observer may be able to pick out a few differences, but a quick glance cannot distinguish between the compressed versions and the original. To find the differences, we will use the computer.

```
diff25 = original - compress25;
disp(sprintf('difference 25 = %d', ...
    sum(sum(abs(diff25)))));
```

The code segment shows how we compute the difference between the original and the compressed version of quality 25. We simply subtract the *compress25* matrix from the original, find the absolute value of the result, then sum along the columns, and finally sum along the rows. We can (and should) easily find the sum of differences between the original and each of the other matrices that store the image. The raw differences follow, as reported by MATLAB. As expected, there is zero difference between the *compress_no_loss* image and the original. If there were a difference, then it would not be lossless!

```
difference no_loss = 0
difference 100 = 60011
difference 75 = 2174508
difference 50 = 2944901
difference 25 = 3962384
```

One way to define the compression ratio is to divide the size of the original signal by the size of the compressed signal. Using this definition, we can find the compression ratio by comparing the file sizes to the original file size.

```
>> disp(sprintf('compress ratio q25 = %6.4f:1', 978460/142951 ));
compress ratio q25 = 6.8447:1
```

We see that the compression ratio is 6.8:1 for the image of quality 25, that is, the compressed file has only 1 byte for every 6.8 bytes of the original. We have 4.7:1 for quality 50, and 3.3:1 for quality 75. As expected, the compression ratios get smaller as the quality goes up.

How good are the resulting images? We turn to two common measures of image quality. By quality, we mean here the faithfulness of the image compared to the original. One measure is the root mean square error, (RMSE). To compute it, we take the sum of the square differences and divide by the number of pixels. Here we will compute this for the image compressed at quality 25.

```
[rows, cols] = size(original);
num_pixels = rows * cols;

diff25 = original - compress25;
mse = sum(sum(diff25.*diff25)) / num_pixels;

rmse25 = sqrt(mse)
```

We find *num_pixels* to be the number of rows times the number of columns. For this image, *num_pixels* = 1444256. Since each matrix location stores the grayscale value for one pixel, this variable stores the total number of pixels. Next, we find the difference between the original and the compressed version. Actually, we have already done this earlier, but want to emphasize how we arrived at *diff*25. The next line finds the square of the differences, then calculates the sum, and finally divides by the number of pixels. We call it *mse* since it is the Mean Square Error. The final step takes the square root of the mean square error to generate the RMSE. Below we have the RMSE values output by MATLAB.

```
RMSE
  with compress_no_loss = 0.0000
  with compress100 = 0.2038
  with compress75 = 2.9732
  with compress50 = 3.9321
  with compress25 = 4.9551
```

We expect the lossless image to have a zero root mean square error (RMSE). As the compressions save more and more space, we see the trend for the RMSE to rise.

A related quality measure is the Peak Signal to Noise Ratio (PSNR). Once we know the RMSE, we can easily find the PSNR.

```
psnr25 = 20 * log10(255/rmse25);
```

The constant 255 comes from the maximum possible value of our signal (in this case, a pixel). Since we have a grayscale image, the pixels range in value from 0 to 255. The output from MATLAB as we calculate the PSNR values appears below.

```
PSNR
  with compress_no_loss =    Inf
  with compress100 = 61.9449
  with compress75 = 38.6664
  with compress50 = 36.2383
  with compress25 = 34.2297
```

The computer was nice enough to print "Inf" instead of an error message for the lossless PSNR. Of course, since the corresponding RMSE equals zero, we divided by zero to get the answer of infinity. For the other peak signal to noise ratios, we have more typical values ranging from 34 to 39. The PSNR of 62 is quite high, but then again the amount of error in the *compress*100 signal is very low. To put this in perspective, let's examine how many pixels were actually changed in the *compress*100 image.

```
>> sum(sum(round(diff100 ./ (diff100 + 1))))

ans =

       60011
```

The above code needs a bit of explanation. The `./` division works on each element individually. So each element of *diff*100 is divided with its value plus 1. You may be thinking, "won't this always give a result of 1?", especially since we round the result. But one notable exception occurs when the value equals zero. Therefore, we generate a matrix of zeros and ones, where each one represents a number that does not match the original image. So the summation of this matrix reveals to us how many pixels were different between the original image and the one stored with 100 quality. Earlier, we saw that the sum of differences for that image was the same number as above (60011). This means that the largest difference was only 1. Next, we verify this statement.

```
>> max(max(diff100))

ans =

    1
```

We know that many pixels (60011) in the reconstruction are different from the original. How significant is this number? We can find out by comparing it to the total number of pixels.

```
>> sum(sum(round(diff100 ./ (diff100 + 1)))) / num_pixels

ans =

    0.0416
```

Only 4.16% of the pixels have a different value compared to the original, and then the maximum difference is only 1. Compare this to the image stored at quality 25, where 43% of pixels have a different value than the original, with a large maximum difference. For grayscale, a difference of 81 in a pixel value is quite significant. Next we use MATLAB to check out the image at quality 25. Of the pixels changed from the original, the average change is 6.

```
>> % number of changed pixels
>> sum(sum(round(diff25 ./ (diff25 + 1))))

ans =

      619209

>> % average difference
>> sum(sum(diff25)) / 619209

ans =

    6.3991

>> % maximum difference
>> max(max(diff25))

ans =

   81

>> % number of changed pixels, percent
```

```
>> 100 * sum(sum(round(diff25 ./ (diff25 + 1)))) / num_pixels

ans =

   42.8739
```

But how do these images look? Are they "good enough"? Of course, this is subjective and will vary from person to person. Often, we must strike a balance between image size and perceived quality. For more information about compression, see S.W. Wu's book, *Additive Vector Decoding of Transform Coded Images* [39]. Another source, especially for compressing degraded images, is *Lossy Compression of Noisy Images* by O. K. Al-Shaykh and R. M. Mersereau [40].

### 10.11.2 Compressing an Image Ourselves

Let's try to compress an image ourselves. As stated earlier, we have three steps: transform, quantization, and entropy coding. First, we need a test image. We read it from the file, and call it *original*.

```
original = imread('dog256x256.gif', 'GIF');
```

We will use the DWT for our transform, specifically the 2D DWT with the `db4` coefficients, for three octaves. In the code below, we find the three octaves one at a time. The capital letters mark the DWT output, while the number denotes the octave. For example, $B2$ contains the data from detail $B$ on octave 2. Since we do not need $A1$ and $A2$ when we are done, we will get rid of these variables.

```
wavelet = 'db4';
[A1, B1, C1, D1] = dwt2(double(original), wavelet);
% octave 2
[A2, B2, C2, D2] = dwt2(A1, wavelet);
% octave 3
[A3, B3, C3, D3] = dwt2(A2, wavelet);
clear A1 A2
```

Now we proceed to the quantization step, where we will use a simple quantization function. In fact, the function has only one line! It takes the input matrix, divides each element by a factor, then rounds the result and multiplies this by the factor. This may appear to do nothing at first glance, but it actually reduces the data's precision. For example, suppose we pass a scalar value of 31.3, with a factor of 2.

Dividing the value by the factor results in 15.65, which we then round to 16. Finally, we multiply 16 by the factor, to find the number 32. Not only does this eliminate the fractional part, it also always returns an even number for the factor 2. Higher factor values mean that the function finds a rougher approximation to the input values.

```
function out = simple_quantize(in, factor)

out = factor*round(in/factor);
```

For each of the subsignals $B1$, $B2$, and so forth, we use this simple quantization function to find approximations. Here we will use a rather conservative factor value of 8, which will result in low compression but a high PSNR. Next we have an example showing the simple quantization function used on $B1$ to generate $B1_q$, the quantized version of $B1$.

```
factor = 8;
B1_q = simple_quantize(B1, factor);
```

The third step is entropy coding. To keep things simple, we will convert the data from 2D matrices to 1D arrays. The "flatten" function does this by putting the number of rows and columns in the first two array positions, and appending the matrix row by row. We will later use the corresponding "unflatten" function to reverse it (both are included on the CD-ROM). As the code below shows, we use $_flat$ to keep track of this version of $B1$. Of course, we repeat this for each of the flattened subsignals.

```
b1_flat = flatten(B1_q);
```

Now to do the actual entropy coding. Included on the CD-ROM are functions "huffman" and "undo_huffman" that perform the entropy coding and unencoding, respectively. *Huffman coding* is a famous entropy coding algorithm named for its inventor. From input data, it creates a list of unique values, then counts how many times these values appear in the data. From this popularity list, it makes a binary tree with the unique values as its leaves. The ones that appear frequently get a spot close to the root. Encoding a value is then simply a matter of finding it in the tree. Suppose the value that we want to encode occupies a leaf node two left branches from the root, then a right branch. We encode 0s for left branches and 1s for right ones, so the encoding would be 001. The more frequently a number appears in the data, the fewer bits will be used to represent it. Numbers that appear infrequently are stored with longer bit strings.

Again, we will use the modified $B1$ subsignal in a code example:

```
b1_huff = huffman(b1_flat);
```

Notice that the entropy coding needs some overhead to work; if we use information to create the codes, we need to store that information so that we can recreate the codes later. The array *b1_huff* contains this overhead data as well as the encoded data. We expect that this entropy encoded result (*b1_huff*) will be smaller than the nonentropy coded version (*b1_flat*), otherwise there would be no point in doing it. However, a signal with no repeating values would generate a *larger* entropy-coded result. We show an extreme example of this next, where we need nine times as much space:

```
>> r = 1:1000;
>> q = huffman(r);
>> length(q)

ans =

        9254
```

This explains how a "compressed" signal may take up more space than the uncompressed version. Realistically, our signals will have much redundancy that we can use to our advantage. Now the quantization step should be clearer; we use that to reduce the number of values that we need to store. Next, we see how the entropy-encoded version of $B1$ (after) needs only one-quarter the space compared to the nonencoded version (before). This is a much more typical example.

```
>> length(b1_flat)  % before

ans =

       17163

>> length(b1_huff)  % after

ans =

        4334
```

All that we need to do now is store our subsignals in a file. The function "writeAfile" (included on the CD-ROM) was made for this purpose. The compressed file only takes 23,687 bytes. Compare that to the original size of 65,536 bytes, or

one byte for $256^2$ pixels. This is only 36% of the original size. Note that we do not consider the size of the original image's file, since it is stored as a `GIF` encoded image. The RMSE for the compressed image is 1.6, with a PSNR of 44 dB. Visually, the resulting image compares favorably to the original.

In this section, we have seen the fundamentals of compression, and how the transform, quantization, and entropy encoding steps allow us to store data in an approximate, but efficient, manner. See the final project files "compress_test.m" and "uncompress_test.m," available on the CD-ROM. The former program encompasses what we have covered above, while the latter reverses the operations at each step. Together, they show that the lossy compression algorithm that we explored here does a good job of storing the image efficiently. The reader is encouraged to try these programs and alter the wavelet and factor parameters to see their effects on the compression.

## 10.12 Summary

This chapter presents several diverse applications of digital signal processing and MATLAB. We started with the basics of sound manipulation, including recording, reading and writing to disk, and playing back. Next, we saw how images can be created, saved, and loaded. After that, we took images as input to the DWT, and displayed the results of this transform as an image. Continuing with the wavelet transform, the plus/minus transform presents a new way of thinking about it. We then looked at a couple of programs from the CD-ROM that perform the DWT, then undo the transform but preserve each channels' contributions. These programs are good for analysis, such as locating edges within an image. We also saw how the wavelet transform can be performed with matrix operations. We demonstrated recursive programming with the *Su Doku* puzzle, and found that a poorly constructed puzzle could have multiple solutions. Precision and data storage are important aspects of digital signal processing, and we looked at conversion to and from the IEEE 754 floating-point standard. Next, we had a sound-sampling program that responds with a frequency plot to any nearby sound. We looked into filter design, and showed how we could specify the filter coefficients ourselves. Finally, this chapter includes a brief introduction to compression.

## 10.13 Review Questions

1. Suppose variable $x$ contains sound information, stored as `double` values. What effect does the command `sound(x*.7)` have? What effect does the command

`sound(x*1.1)` have?

2. Given a short sound file, show how you can play it back 10 times with a different sampling rate each time.

3. How can you concatenate two sound arrays to play back as one after the other? What if they have two channels each?

4. Given a sound file, write MATLAB code to play it backwards.

5. Convert a sound signal to the frequency-domain, shift the signal by copying the frequency-domain data to new indices, and convert back. Can you accomplish this by changing the sampling frequency? What if you reverse the order of the indices before converting back?

6. Create a program to reverse a grayscale image. For example, reversing an image of a white box on a black background would result in a black box on a white background. What would happen if the image were color?

7. Earlier we saw code that rotates an image clockwise. Modify it to rotate the image counterclockwise.

8. When the *Su Doku* solver runs, the puzzle has different states as guesses replace blanks. These states are stored on the "stack," an internal computer data structure. What is the value of *puzzle* after the program runs? How could we store the puzzle states in a way that we could access them later?

9. What would happen if the *Su Doku* solver's search function did not include the `function` keyword in its definition?

10. Examine the *Su Doku* solver code to find the number of comparison operations involved. If we had a hexadecimal version of the *Su Doku* puzzle, how would the number of operations change?

11. We saw how to convert the IEEE 754 floating-point standard's internal representation of 4187000 (hexadecimal) back to the original value of 16.875. What value does 41118000 (hexadecimal) represent? What value does 411570a4 represent?

12. Write a program to convert a double number to its binary representation. (Hint, if you show the number in hexadecimal as well as binary, it will be easy to check it.)

13. When using the `window` function to design filter coefficients, many windows are supported. Experimenting with the windows `gausswin`, `hamming`, `rectwin`, and `triang`, determine which window gives the best results for a lowpass FIR filter.

14. We saw that flipping every other sign for a lowpass filter's coefficients has the effect of making it a highpass filter. The alternating flip mimics multiplying each value with a high-frequency sinusoid. Write code to change a set of filter coefficients by multiplying them with a slower sinusoid. What effect do you observe?

15. Finding the number of distinct values is basically counting. One way to do it as an algorithm is to sort the values, then eliminate any duplicates, and the number of values left is the number of distinct values. For example, in the series {7, 3, 5, 1, 3, 3, 5, 4, 1, 7, 4, 5}, there are only 5 distinct values {1, 3, 4, 5, 7}. The nearest power of 2 greater than (or equal to) this is 8, or $2^3$. Thus we can assign each value a different code of 3 bits. What we want to do is store a signal in less space. That is, we could represent these values with 3 bits each. This is not the most efficient way to store these values, but it is meant to give you an idea of how compression works. (Huffman encoding is more efficient.)

    Suppose we have the following example signal:

    $$x[n] = \{0, -1, 9, -1, 9, 9, 0, 3, -1, 3, 0, 0\}.$$

    Storing these in binary as integers, we would probably use 16 bits to store each value, for a total of 192 bits, but this is wasteful in terms of the amount of space.

    a. How many distinct values are there in the example signal $x[n]$ above?
    b. Decide what bit patterns should represent each distinct value (forming a lookup table).
    c. Show what the entire example signal above ($x[n]$) would look like encoded. Remember that we want to be able to undo this as well—given the bit pattern and lookup table, we should be able to get $x[n]$ back.
    d. How many bits does the encoded signal take up? What if you included the encoding table?

# Appendix A

# Constants and Variables Used in This Book

## A.1 Constants

The Greek letter pi is $\pi \approx 3.14159$

Euler's constant
$e \approx 2.71828$

$j = \sqrt{-1}$

## A.2 Variables

$\phi$ is a variable used to represent some angle. Often, we use it to represent the phase angle in a time-varying sinusoid. This is typically given in radians, e.g., $\phi = \pi/6$.

$\theta$ is a variable used to represent some angle. It is often used as the argument of a sine or cosine function, e.g., $\cos(\theta)$.

$\sigma$ is used with statistics, for example, $\sigma^2$ is the variance.

$\omega$ is another variable used to represent some angle. Also, it is used as a frequency, or more precisely, a radian frequency. In this sense, $\omega = 2\pi f$. It has units of radi-

ans/seconds.

$a$ stands for amplitude, used with a sinusoid.

$f$ represents a frequency, given in Hertz.

$g$ represents an FIR filter coefficients, appearing with wavelets since $h$ is already used. Typically, $g$ corresponds to the highpass filter of a filter pair.

$h$ represents the unit impulse response. For FIR filters, $h$ will be the same thing as the filter coefficients. With a filter pair, $h$ corresponds to the lowpass filter.

$j$ typically means $\sqrt{-1}$, except when it comes to wavelets, where it is used as an index.

$J$ is the upper limit for $j$ when talking about wavelets.

$k$ is an integer.

$m$ is used as an index. It is always an integer. Usually, $m$ is used when $n$ is already being used for something else.

$n$ is used as an index. It is always an integer, and sometimes it can even be negative.

$K$ is used as a limit, with the index $k$.

$N$ is used as a limit, usually for the size of an array. It commonly is used with $n$, such as $n = 1$ to $N$, or $n = 0..N - 1$.

$t$ represents time. It is a continuous variable.

$w$ usually refers to a signal, often used when $x$ and $y$ are already defined. Do not confuse this with $\omega$ (the lowercase Greek letter omega) since they have different meanings.

$W$ and $W_h$ refer to the lowpass and highpass outputs of the wavelet transform, respectively.

$x$ is used throughout to mean a signal, often an input. It may be continuous,

such as $x(t)$, or it may be discrete as $x[n]$.

$X$ means either the $z$-transform or Fourier transform of $x$.

$y$ is used throughout to mean a signal, usually an output. It may be continuous, such as $y(t)$, or it may be discrete as $y[n]$.

$z$ usually stands for a complex variable, such as in the $z$-transform. It is also used as the name for a signal, if $x$ and $y$ are already used.

Anytime there is a lowercase letter, the same letter in uppercase means the upper limit, or the frequency-domain information, depending on whether the lowercase letter is an index or a signal. For example, $Y[m]$ would be considered the Fourier transform (or $z$-transform) of signal $y[n]$, where $m$ and $n$ are integer indices, likely ranging from 0 to $N - 1$.

## A.3 Symbols Common in DSP Literature

$\in$ is used to indicate "element of." For example, "$b \in \{0, 1\}$" means that $b$'s value must be in the set, that $b$ is either 0 or 1.

$\mathbb{N}$ is the set of all natural numbers, starting with 1. That is, {1, 2, 3, ...}.

$\mathbb{Z}$ is the set of all natural numbers, with 0 also included. Saying $i \in \mathbb{Z}$ is like defining "`unsigned int i;`" in a C/C++ program (except that some integers are too big for the range defined by the programming language).

$\mathbb{R}$ is the set of all real numbers. For example, $a \in \mathbb{R}$ means that $a$ is a real number. This is like saying `float a;` in C/C++, except that floats have some limitations that abstract mathematical variables do not, such as limits on precision.

$\mathbb{R}^n$ is the set of all vectors of real numbers of length $n$. This is similar to all possible arrays of real values. A way to think about "$c \in \mathbb{R}^n$" is to think of $c$ as if it were defined as a real array, e.g., `float c[n];`. (Note the limits mentioned above.)

**Table A.1:** Greek alphabet.

| **Name** | **capital** | **lowercase** | **equivalent** |
|---|---|---|---|
| alpha | A | $\alpha$ | a |
| beta | B | $\beta$ | b |
| gamma | $\Gamma$ | $\gamma$ | g |
| delta | $\Delta$ | $\delta$ | d |
| epsilon | E | $\epsilon, \varepsilon$ | e |
| zeta | Z | $\zeta$ | z |
| eta | H | $\eta$ | e |
| theta | $\Theta$ | $\theta, \vartheta$ | th |
| iota | I | $\iota$ | i |
| kappa | K | $\kappa$ | k |
| lambda | $\Lambda$ | $\lambda$ | l |
| mu | M | $\mu$ | m |
| nu | N | $\nu$ | n |
| xi | $\Xi$ | $\xi$ | x |
| omicron | O | o | o |
| pi | $\Pi$ | $\pi, \varpi$ | p |
| rho | P | $\rho, \varrho$ | r |
| sigma | $\Sigma$ | $\sigma$ | s |
| tau | T | $\tau$ | t |
| upsilon | Y, $\Upsilon$ | $\upsilon$ | u |
| phi | $\Phi$ | $\phi, \varphi$ | ph |
| chi | X | $\chi$ | ch |
| psi | $\Psi$ | $\psi$ | ps |
| omega | $\Omega$ | $\omega$ | o |

# Appendix B

# Equations

## B.1 Euler's Formula

Euler's equation

$$e^{-j\phi} = \cos(\phi) - j\sin(\phi)$$

Euler's equation also works for a positive exponent, i.e.,

$$e^{j\phi} = \cos(\phi) + j\sin(\phi)$$

When the angle in question happens to be $\pi$,

$$e^{j\pi} = \cos(\pi) + j\sin(\pi) = -1 + j0$$

$$e^{j\pi} = -1$$

Euler's inverse formula

$$\cos(\phi) = \frac{e^{j\phi} + e^{-j\phi}}{2}$$

## B.2 Trigonometric Identities and Other Math Notes

$$\sin(-\phi) \quad = \quad -\sin(\phi)$$

$$\cos(-\phi) = \cos(\phi)$$

$$\sin(\theta) = \cos(\theta - \pi/2)$$

if $k$ is an integer, $\cos(\theta + 2\pi k) = \cos(\theta)$.

$$cos^2(\theta) + sin^2(\theta) = 1$$

From *Elements of Calculus with Analytic Geometry* by E. W. Swokowski [26] :

$$\cos(\phi + \theta) = \cos(\phi)\cos(\theta) - \sin(\phi)\sin(\theta)$$

$$\sin(\phi + \theta) = \cos(\phi)\sin(\theta) + \sin(\phi)\cos(\theta)$$

$$\cos(\phi - \theta) = \cos(\phi)\cos(\theta) + \sin(\phi)\sin(\theta)$$

$$\sin(\phi - \theta) = \cos(\phi)\sin(\theta) - \sin(\phi)\cos(\theta)$$

From *Exploring Numerical Methods: An Introduction to Scientific Computing Using MATLAB* by P. Linz and R. L. C. Wang [10] :

$$\cos(mx)\cos(nx) = \frac{1}{2}[\cos((m+n)x) + \cos((m-n)x)]$$

$$\sin(mx)\cos(nx) = \frac{1}{2}[\sin((m+n)x) + \sin((m-n)x)]$$

$$\sin(mx)\sin(nx) = \frac{1}{2}[\cos((m-n)x) - \cos((m+n)x)]$$

Cartesian to polar:

$$r = \sqrt{x^2 + y^2}$$

$$\theta = \arctan(y/x)$$

If $x$ and $y$ are both negative, then add $\pi$ to $\theta$.
If $x$ is negative but $y$ is positive, subtract $\pi$ from $\theta$.
If $x$ is zero, and $y$ is negative, then $\theta = -\pi/2$.
If $x$ is zero, and $y$ is positive, then $\theta = \pi/2$.

Polar to Cartesian:

$$x = r\cos(\theta)$$

$$y = r\sin(\theta)$$

Quadratic formula:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Note: $\sqrt{\alpha\beta} = \sqrt{\alpha}\sqrt{\beta}$, but ONLY when both $\alpha$ and $\beta$ are positive integers. This is a common mistake that comes up in some DSP problems.

---

## B.3 Sampling

Nyquist criterion:

$$f_s \geq 2B$$

$$t_s = 1/f_s$$

sampling: $x(t)$ becomes $x(nT_s) = x[n]$

Bandpass sampling from *Understanding Digital Signal Processing* by R. Lyons [11]:

$$\frac{2f_c - B}{m} \geq f_s \geq \frac{2f_c + B}{m+1}$$

---

## B.4 Fourier Transform (FT)

Continuous Fourier transform:

$$F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-j\omega t}dt$$

Inverse continuous Fourier transform:

$$f(t) = \int_{-\infty}^{\infty} F(\omega)e^{j\omega t}dt$$

Discrete Fourier Transform (DFT)

$$X[m] = \sum_{n=0}^{N-1} x[n] e^{-j2\pi nm/N}$$

Alternate form:

$$X[m] = \sum_{n=0}^{N-1} x[n](\cos(2\pi nm/N) - j\sin(2\pi nm/N))$$

where $m = 0..N-1$.

Inverse Discrete Fourier Transform (IDFT)

$$x[n] = \frac{1}{N} \sum_{m=0}^{N-1} X[m] e^{j2\pi nm/N}$$

Alternate form:

$$x[n] = \frac{1}{N} \sum_{m=0}^{N-1} X[m](\cos(2\pi nm/N) + j\sin(2\pi nm/N))$$

where $n = 0..N-1$.

Magnitude of DFT outputs:

$$X_{magnitude}[m] = \sqrt{X_{real}[m]^2 + X_{imaginary}[m]^2}.$$

Phase of DFT outputs:

If $X_{real}[m] > 0$,

$$X_{phase}[m] = tan^{-1}(X_{imaginary}[m]/X_{real}[m])$$

else

$$X_{phase}[m] = tan^{-1}(X_{imaginary}[m]/X_{real}[m]) + \pi.$$

Analysis frequencies are given in *Understanding Digital Signal Processing* by R. Lyons [11]:

$$f_{analysis}[m] = \frac{mf_{sampling}}{N}.$$

---

## B.5 Convolution

Convolution:

$$y[n] = \sum_{k=0}^{M-1} h[k]x[n-k].$$

Convolution has the following properties [12], it is:
commutative: $a * b = b * a$;
associative: $a * (b * c) = (a * b) * c$;
distributive: $a * (b + c) = a * b + a * c$.

---

## B.6 Statistics

$$\sigma^2 = \sum_{k=1}^{N}(input[k] - result[k])^2$$

Variance of a random variable from R. E. Walpole and R. H. Myers, *Probability and Statistics for Engineers and Scientists, Third Edition* [13]:

$$\sigma^2 = \sum_{k=1}^{N}(x-\mu)^2 p(x)$$

where $\mu$ is the mean.
Sample variance (also from Walpole and Myers) [13]:

$$S^2 = \frac{\sum_{k=1}^{N}(input[k] - result[k])^2}{N-1}$$

The Mean Square Error (MSE) between an image and its reconstruction (perhaps after compression) is:

$$\text{MSE} = \frac{\sum_{m=1}^{M}\sum_{n=1}^{N}(original[m,n] - reconstruction[m,n])^2}{MN}.$$

The Root Mean Square Error (RMSE) is given by the following formulas.

$$\text{RMSE} = \sqrt{\frac{\sum_{k=1}^{N}(input[k] - result[k])^2}{length(input)}} = \sqrt{\frac{\sigma^2}{N}}$$

For 2D data such as images:

$$\text{RMSE} = \sqrt{\frac{\sum_{m=1}^{M}\sum_{n=1}^{N}(original[m,n] - reconstruction[m,n])^2}{MN}} = \sqrt{\frac{MSE}{M \times N}}$$

where MSE stands for Mean Square Error.

The Signal to Noise Ratio (SNR) equation follows.

$$\text{SNR} = 10 \; log_{10}\left(\frac{{\sigma_x}^2}{\sigma^2}\right)$$

where $\sigma_x$ is the mean square of the input signal [41]. A related formula commonly used in image processing is the Peak Signal to Noise Ratio (PSNR).

$$\text{PSNR} = 20 \; log_{10}\left(\frac{MaxPossibleValue}{RMSE}\right)$$

where *Max Possible Value* comes from the encoding, e.g., 255 for grayscale [42].

## B.7 Wavelet Transform

DWT Scaling function

$$\phi(t) = \sqrt{2}\sum_{k} h[k]\phi(2t-k)$$

or

$$W[j,n] = \sum_{m=0}^{N-1} W[j-1,m]h[2n-m]$$

DWT Wavelet function

$$\psi(t) = \sqrt{2}\sum_{k} g[k]\phi(2t-k)$$

or

$$W_h[j,n] = \sum_{m=0}^{N-1} W[j-1,m]g[2n-m]$$

---

## B.8 *z*-Transform

Laplace transform

$$F(s) = \int_{0}^{\infty} f(t)e^{-st}dt,$$

where s = $\sigma$ + j$\omega$

Notice how this integral starts at 0, instead of $-$*infinity*. Here we assume that the signal starts at $t = 0$.

*z*-Transform of a discrete signal $x[n]$ is

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

$$Y(z) = X(z) \sum_{k=0}^{N} b[k] z^{-k} + Y(z) \sum_{k=1}^{M} a[k] z^{-k},$$

where

$$z = re^{j\omega}.$$

Transfer function:

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

# Appendix C

# DSP Project Ideas

Below are some ideas for semester projects in digital signal processing. Some of the projects are more complex than others and will take more time. Also, these projects can be made more interesting by applying advanced techniques such as neural networks, hidden Markov modeling, adaptive filtering, and other tools beyond the scope of this book.

Over the span of a couple of months, projects are not going to be industrial-strength. Shortcuts should be included in the original project proposal, and acknowledged in a final report. For example, the number of test signals may be very small, and the effect on the project's results should be discussed.

1. **Image watermarker**
   Devise a way to put a simple watermark in an image that is invisible to a human observer.

2. **Noise filtering**
   Create a program to filter a noisy signal, such as cleaning up a speech signal recorded in a room full of other conversations.

3. **Compression**
   Given a music signal, store the information of both channels as efficiently as possible. What if your program subtracts one channel from the other? What if you use the previous sample as the expected value for the next sample, and only store the difference?

4. **Music distinguisher**
   Study songs representative of different music types, and write a program that distinguishes between them. For example, one might expect rap music to have more energy in the low frequency range (bass) compared to country music.

5. **Color changer**
   Using a color image, write a program to automatically map all the colors to different values. For example, the program could make all the colors like pastels. Or, the program could change all colors to gray except for one, e.g., highlighting green.

6. **Red-eye removal**
   Often pictures show people with red eyes, due to the flash. An interesting project would be to automatically detect and remove red-eye. To make the problem easier, assume that the eye areas are given by the user.

7. **Tank finder**
   Devise a way to find a specific object in an image, such as a military tank. Your program should return a flag indicating whether or not a tank is present, and if so, the position of the tank in the image. This can be a very complex project, so you may want to simplify things by assuming that the tank is a black outline on a white background. Your program should be able to find the tank regardless of its place in the image, called translation. Variations on this idea (each of which would be a project on its own) include: detecting the object regardless of rotation, detecting the object regardless of obstruction (such as being partially obscured by another object), and detecting the object regardless of scale (smaller or larger).

8. **Cat identification system**
   Many cat owners can recognize their cat from its meow. Develop a system that attempts to correctly identify a particular cat from many different cat recordings.

9. **Dog translator**
   Dog owners often know what their dog wants by the way it barks and whines. Develop a system to discern between the noises your dog makes when it sees another dog, it is hungry, someone knocks at the door, it wants to go out, etc.

10. **Object separation**
    Make a program to separate an image into a foreground and background image. For example, take a picture of a person's face and store it as a grayscale image. Your program should be able to create a new image from this with only the person's face on a white background.

11. **Music finder**
    Using MATLAB and a microphone, record a fraction of a second of sound,

and analyze it to determine if the signal is music, speech, or silence. You can play a radio near the microphone and see how well it functions.

12. **Whistling as a form of remote control**
    Using MATLAB and a microphone, have the computer record a sound signal and analyze it, looking for a whistling sound, then repeat in a loop. If a whistle is detected, give an output indicating so. An advanced system should be able to detect different whistles by the same user and respond accordingly. For example, one whistle could cause the computer to announce the time, while another would trigger a day announcement.

13. **Weather forecaster**
    Using the Fourier transform of the past month's temperatures, try to predict what the temperature will be for the next few days. Can you use the error of the past predictions to do a better job?

    Besides temperature, a more complex task is to use barometric pressure readings, temperature, and wind speed/direction to predict other weather patterns, like storms.

14. **Basketball tracker**
    Given a video clip of a basketball game, try to automatically track the ball. This could work with any type of game.

15. **Image compression**
    Develop your own lossy image compression program and evaluate how well it works.

# Appendix D

# About the CD-ROM

The CD-ROM should be readable by any operating system. It contains the MATLAB examples found in this text.

Included on the CD-ROM are also figures from the text.

For each chapter, there is an example project. See the “README” files for a quick introduction to each, i.e., “chapter1.README” for the first chapter.

# Appendix E

# Answers to Selected Review Questions

## Chapter 1

1.1 A measurable, variable phenomenon, often a physical quantity that varies with time.

1.3 The operation that the system performs.

1.5 Signal "$x$" is indexed by $t$. Signal "$x$" is continuous. Index "$t$" is also continuous. $x(t)$ represents an analog signal.

1.7 01001011.101 in binary. Since we did not specify how many bits this number occupies, we will leave it as is, without appending 0s (or truncating it).

1.9 $r = 5.3852$
Since this point is in the first quadrant, no correction is needed.

$$\theta = 1.1903 \text{ radians } (68.1986 \text{ degrees})$$

1.11 When a continuous (analog) signal is sampled, or converted to a discrete time-domain signal, we have limited precision on both its time and its amplitude. It is quantized in time in the sense that we will have samples corresponding to measurements at various times, but we do not have samples in between. We then have samples of the signal at $x[n]$, or $x[0]$, $x[1]$, $x[2]$, etc. But we do NOT have data at $x[1.5]$.

The signal is quantized in amplitude in that we impose our precision on the samples that we collect. Each sample will be subjected to the limited range that we allow. Any number that is "too large" or has a part that is "too small" will not be stored correctly.

1.13 The argument to the arctan function will either be positive or negative. If both $a$ and $b$ are negative, the negative signs cancel and the argument is positive. Thus, the conversion gives the same result as when both $a$ and $b$ are positive, though the angle is clearly in a different quadrant when the point is plotted. Similarly, positive $a$ and negative $b$ give the same result as negative $a$ and positive $b$. Therefore, we need to correct the angle based on the quadrant. (See the chapter for more detail.)

## Chapter 2

2.1
```
% mysort.m
%
% Function to sort array of numbers in descending order
% (highest to lowest)
%
% example:
%    x = round(100*(0.5 - rand(1,100)));
%    y = mysort(x);
%    plot(y)

function sorted = mysort(in)

% Yes, we could use the built in "sort" function.
% We could also use the "min" and "max" functions.
% But instead, we will just use comparisons and loops.
% This is based on bubble-sort.
% There are more efficient ways, but this is simple.

% copy our data
sorted = in;
% make sure we do the loop at least once
swap = true;
while (swap)
    swap = false; % default
```

```
    % If we look through the whole array,
    % and do not swap values,
    % then we know we are done.
    for k=2:length(sorted)
        % Compare current value to its neighbor
        if (sorted(k-1) < sorted(k))
            % swap values
            temp = sorted(k-1);
            sorted(k-1) = sorted(k);
            sorted(k) = temp;
            swap = true;
        end
    end
end
```

2.3 No, $sum(3x-4)$ is not the same value as $3sum(x)-4$. In the first computation, we take each value of $x$ and multiply each one by 3, then subtract 4 before we find the sum. In the second computation, we find the sum first, then do the multiplication and subtraction. As a result, the $-4$ is effectively applied once per value of $x$ in the first computation, while it is applied only once in the second one.

```
>> x = [121, 145, 167, 192, 206];
>> y = 3*x - 4

y =

   359   431   497   572   614

>> sum(y)

ans =

        2473

>> 3*sum(x) -4

ans =

        2489
```

2.5
```
>> t = 0:0.00001:0.002;
>> x = 2 * cos(2*pi*1500*t + pi /2);
>> plot(t, x, 'k')
```

2.7 a. The following MATLAB code will find the smallest number we can represent.

```
n=0;
my_value = 1/(2^n);
while (my_value > 0)
   n = n + 1;
   my_value = 1/(2^n);
end
n = n - 1;
sprintf('n = %d, 1/(2^n) = %f', n, 1/(2^n) )
```

$n = 1023$.

b.

```
n=0;
my_value = 2^n;
while (my_value < Inf)
   n = n + 1;
   my_value = 2^n;
end
n = n - 1;
sprintf('n = %d, 2^n = %f', n, 2^n )
```

$n = 1023$. When the value is too large, it is stored as "Inf," or infinite. It is not surprising that the answer here is the same as in part *a*. In both of these problems, we are storing the number $+1 \times 2^n$, where $n$ is either positive (as in part *b*) or negative (as in part *a*). That is, the exponent changes as $n$ gets larger or smaller. The difference between parts *a* and *b* is the sign of the exponent.

c.

```
n=0;
my_value = 1+1/(2^n);
while (my_value ~= 1)
```

```
    n = n + 1;
    my_value = 1+1/(2^n);
end
n = n - 1;
sprintf('n = %d, 1+1/(2^n) = %f', n, 1+1/(2^n) )
```

Here, $n = 52$. As $n$ gets larger, the number $1 + 1/(2^n)$ gets closer and closer to 1. When it is stored as 1, we know we have reached our limit. This answer is different from part $a$, since the number to be stored is 1.0[0..0]1, meaning that the exponent stays the same. In part $a$, the significand may remain the same as the exponent becomes smaller.

## Chapter 3

3.1 Finite impulse response filter. It means that if you give it some nonzero input, then zeros, you will eventually get zeros at the output.

3.3 Linear is a property of some systems (like an FIR or IIR filter) where the output relation has a sum of inputs multiplied by constants.

3.5 An easy (and lazy) answer is to use MATLAB's built-in `conv` function, as follows.

```
b = [ 0.1, 0.2, 0.3, 0.2, 0.1];
x = 1:100;
y = conv(x, b);
```

A better solution would use loops to mimic the operations carried out in convolution, as in the function below.

```
%
%  myconv2.m  A solution to the question,
%             how do we implement the difference
%             equation in Matlab.
%   This should be equivalent to Matlab's "conv" function
%   This function is NOT the most efficient way to do this,
%   but it is the direct way.
%  Usage:
%    Y = myconv2(A, X);
%
```

```
function [Y]= myconv2(A, X)

% number of inputs - 1  (because we count from 0..N)
N = length(X)-1;
% number of filter coefficients - 1
M = length(A)-1;

% number of outputs = N + M
% since we would normally count 0..M+N,
% in Matlab we have to add 1 to this
% when used as an array index, e.g., count from 1..M+N+1
for n=0:M+N,
    % initialize our output to zero
    Y(n+1)=0;
    % perform the summation
    for k=0:M,
        x_index = n-k;
        % X(n) = 0  whenever n < 0 or n > length(X)
        if (and(x_index >= 0, x_index <= N))
            Y(n+1) = A(k+1)*X(x_index+1) + Y(n+1);
        end
    end
end
```

Using it, we get the same results as before.

```
>> y2 = myconv2(x, b);
>> sum(abs(y2-y))

ans =

   1.8385e-13
```

3.7 Convolution

3.9 We call $h[n]$ the impulse response. It shows the way a filter behaves when it gets a single nonzero input. For FIR filters, this shows us what effect the filter has on an input signal's frequencies.

3.11 $System_1$ is linear, causal, and time-invariant. $System_2$ is not linear, but it is causal and time-invariant.

3.13 a. Examining the indices of the inputs versus the output, we see that the output's index is as large (or larger) than all the indices used for the inputs. Therefore, we conclude that this system is causal.

b. This system is linear.

c. This system is time-invariant.

## Chapter 4

4.1 The cos function has a maximum of +1 and a minimum of −1. Therefore, the minimum and maximum values for $x(t)$ are 3 and −3, respectively.

4.3 a.

```
>> t = 0:0.000001:0.001;
>> x1 = 3*cos(2*pi*2000*t + pi/4);
>> x2 = 2*cos(2*pi*5000*t);
>> x3 = cos(2*pi*11000*t - pi/7);
>> subplot(3,1,1);  plot(t, x1);
>> title('3cos(2\pi 2000 t + \pi/4)')
>> subplot(3,1,2);  plot(t, x2);
>> title('2cos(2\pi 5000 t)')
>> subplot(3,1,3);  plot(t, x3);
>> title('cos(2\pi 11000 t - \pi/7)')
```

b.

```
>> t = 0:0.000001:0.001;
>> x = 3*cos(2*pi*2000*t + pi/4) + 2*cos(2*pi*5000*t) ...
        + cos(2*pi*11000*t - pi/7);
>> plot(t, x)
```

c. The greatest common value between 2000, 5000, and 11,000 is 1000, thus $f_0 = 1000$ Hz. So we can represent $x$ as:

$$x(t) = 3\cos(2\pi 2(f_0)t + \pi/4) + 2\cos(2\pi 5(f_0)t) + \cos(2\pi 11(f_0)t - \pi/7)$$

For amplitudes, we know that $a_2 = 3$, $a_5 = 2$, and $a_{11} = 1$. All other amplitudes are 0. For the phase angles, we see that $\phi_2 = \pi/4$, $\phi_5 = 0$, and $\phi_{11} = -\pi/7$. All other phase angles are 0.

4.5 Amplitude is a quantity without units, could be positive or negative. Magnitude is a quantity without units, always positive. When we have a function between vertical bars, it means to find the magnitude. It is implemented in MATLAB as the `abs` function; the absolute value.

4.7 Spectrum

4.9 Technically, amplitudes can be positive or negative. Practically, a negative amplitude does not change the signal, except to shift it in time. We can change the phase angle to get the same effect. Thus, we do not need negative amplitudes.

## Chapter 5

5.1 1/0.008 = 125 Hz

5.3 Undersampling occurs when we do not take samples often enough. We do not want to do this, since we will not be able to faithfully recreate the sampled signal.

5.5 Converting a digital signal to an analog one, so named because the signal was typically analog to begin with.

5.7 Ideally, yes, the output should exactly match the input. But there are practical things to consider that would make the two signals different. The circuitry making up the ADC and DAC may not be fast enough to handle all possible analog signal values.

5.9 The frequency 800 Hz is clearly greater than $f_s$ (500 samples/sec), so it will have an alias at (800 - $f_s$) = 300 Hz.

Note that the 300 Hz frequency component will have a replica at $-300$ Hz, which has a replica at $f_s + (-300) = 200$ Hz.

5.11 $B$ (bandwidth) is between 18 and 24 kHz. $B$ = 24-18 = 6 kHz.
$f_c$ (center frequency) = (18+24)/2 = 21 kHz.
$m$ = number of replicas = 2.
$2B$ = 12 kHz (Nyquist), so $f_s$ in the range of 16.67 to 18 kHz is OK.

5.13 $2B$ = 10 MHz (Nyquist), so $f_s$ in the range of 15 to 20 MHz is OK. Also, $f_s$ = 10 MHz is OK. Anything beyond $m = 2$ will not work.

5.15 a.

$$x[n] = 3\cos(2\pi n 0.2 + \pi/4) + 2\cos(2\pi n 0.5) + \cos(2\pi n 1.1 - \pi/7)$$

b. This can be done in MATLAB.

```
n = 1:100;
x = 3*cos(2*pi*n*0.2 + pi/4) + 2*cos(2*pi*n*0.5) ...
    + cos(2*pi*n*1.1 - pi/7);
plot(n, x)
```

c. The 5 kHz frequency has aliases every ±10 kHz, i.e., at -20 kHz, -10 kHz, -5 kHz, 15 kHz, 25 kHz, etc. Each of these also appears mirrored around 0 Hz, i.e., +20 kHz, +10 kHz, +5 kHz, -15 kHz, -25 kHz, etc.

d. This is 2 times the bandwidth. Bandwidth $= 11,000 - 2000 = 9000$ Hz, so the critical Nyquist frequency is 18,000 Hz. The sampling rate must be at least 18 kHz.

e. Bandpass sampling uses a sampling frequency at least $2B$, and locates a replica between the original frequency content and 0 Hz. The problem here is that a replica cannot fit; there is no way a 9 kHz band will fit between 0 and 2 kHz. Since there will be overlap between the original frequency content and the replica, we should not use bandpass sampling here.

5.17 a.

```
t = 0:0.00001:0.001;
x1 = 6*cos(2*pi*7000*t + pi/2);
x2 = 4*cos(2*pi*8000*t);
x3 = 2*cos(2*pi*6000*t);
x1_n = 6*cos(-2*pi*7000*t - pi/2);
x2_n = 4*cos(-2*pi*8000*t);
x3_n = 2*cos(-2*pi*6000*t);
subplot(3,1,1);
plot(t, x1, 'b', t, x1_n, 'r')
subplot(3,1,2);
plot(t, x2, 'b', t, x2_n, 'r')
subplot(3,1,3);
plot(t, x3, 'b', t, x3_n, 'r')
```

b. The signals x1 and x1_n are the same, as are x2 and x2_n, and x3 and x3_n. Therefore, the cosine function returns the same value whether or not the argument is negated.

c. No, the sine function does not return the same value for a negated argument. Actually, it returns a negated value for a negated argument, i.e., $\sin(-\theta) = -\sin(\theta)$. This comes from the definitions of sin and cos, i.e., on a graph, $x$ is positive whether the angle is $0..\pi/2$ radians or $-\pi/2..0$ radians.

5.19
```
Xsize = 10;
Ts = 1/8000;
% Make our X signal
n=0:Xsize-1;
x(n+1) = cos(2*pi*1000*n*Ts)/2 + cos(2*pi*2000*n*Ts + pi/4);
sprintf('The first %d samples are : ',Xsize)
x
```

5.21 a. Yes, it does. The 100 kHz noise will have a replica at $100 - 44 - 44 = 22$ kHz. The 102 kHz noise will have a replica at $102 - 44 - 44 - 44 = -30$ kHz, which will also appear at $-30 + 44 = 14$ kHz.

b. The sampling rate must be at least twice the Bandwidth. This sampling rate violates the Nyquist criterion.

# Chapter 6

6.1 We have a DC component (0 Hz) of 10 units, and at analysis frequencies:
$f(3)$ = 3*5000/250 = 60 Hz
$f(5)$ = 5*5000/250 = 100 Hz
$f(9)$ = 9*5000/250 = 180 Hz
we have sinusoids of magnitudes 4.47, 1, and 4.47, respectively (there are no units for these numbers). These sinusoids have phases of 1.107 radians (63.4 degrees), 0, and −1.107 radians (−63.4 degrees), respectively.

6.3 a. There are 10 samples total. We have a DC component (0 Hz) of 3 units.

b. At analysis frequencies:

| | |
|---|---|
| $f(1)$ = 1*100/10 = 10 Hz | Amount: sqrt(20) units = 4.5 units |
| $f(2)$ = 2*100/10 = 20 Hz | Amount: 1 units |
| $f(3)$ = 3*100/10 = 30 Hz | Amount: sqrt(34) units = 5.8 units |

$f(4) = 4*100/10 = 40$ Hz Amount: 0 units
$f(5) = 5*100/10 = 50$ Hz Amount: 0 units
(We will stop here.)
The frequency components rank in this order (largest to smallest):
30 Hz, 10 Hz, and 20 Hz.

c.

```
>> X = [3, 2+4j, 1, 5-3j, 0, 0, 0, 5+3j, 1, 2-4j];
>> x = ifft(X);
>> disp(x(1:5))
    1.9000    0.4768   -1.6607   -0.2899    0.4476

>> disp(x(6:10))
   -0.9000    0.2468    1.9371    0.5663    0.2760
```

6.5

```
x = round(rand(1, 20)*100);
h1 = [0.5, 0.5, 0.5, 0.5, 0.5];
h2 = [0.1, 0.3, 0.5, 0.3, 0.1];
h3 = [0.9, 0.7, 0.5, 0.7, 0.9];
out1 = conv(x, h1);
out2 = conv(x, h2);
out3 = conv(x, h3);
subplot(3,1,1); plot(out1);
title('output 1');
subplot(3,1,2); plot(out2);
title('output 2');
subplot(3,1,3); plot(out3);
xlabel('filter outputs');
title('output 3');

figure(2);
Out1 = fft(out1);
Out2 = fft(out2);
Out3 = fft(out3);
subplot(3,1,1); plot(abs(Out1));
title('FMR of output 1');
subplot(3,1,2); plot(abs(Out2));
title('FMR of output 2');
subplot(3,1,3); plot(abs(Out3));
```

```
title('FMR of output 3');
```

The second filter produces a smoother version of the original, while still retaining the original's characteristics. The coefficients $h_2[k]$ have the smoothest transition (with 0.1 at each end). The filter coefficients in $h_3[k]$ have the most transition (first and last values), and are visibly the worst of the three.

6.7

$$3e^{-j2\pi 0.2} = 0.9270 - j2.8533$$

# Chapter 7

7.1 3.6055 $e^{0.9828j}$

7.3

$$6e^{j\phi} = 6(\cos(2\pi) + j\sin(2\pi))$$

$$6e^{j\phi} = 6(1 + j0) = 6$$

Since the complex part is zero, this vector reduces to a real number.

7.5

$$2\cos(2\pi 100t + 3\pi/8) + j2\sin(2\pi 100t + 3\pi/8)$$

7.7 $\omega$ tells us the speed of rotation, and direction. If $\omega$ is positive, it rotates counterclockwise. If is it negative, then it rotates clockwise. Yes, we CAN have a positive frequency or a negative frequency.

$\phi$ gives the initial position, i.e., where the vector is at time 0.

7.9 $e^{-j\phi}$

7.11 We observe that the vector is rotated counterclockwise by 90 degrees each time we multiply by $j$. The four vectors are:

$$4 + 5j$$

$$4j + 5(-1) = -5 + 4j$$

$$-5j + 4(-1) = -4 - 5j$$

$$-4j - 5(-1) = 5 - 4j$$

The angles (in degrees) are:

```
>> v = [4 + 5j, -5 + 4j, -4 - 5j, 5 - 4j];
>> disp(angle(v)*360/(2*pi))
   51.3402  141.3402 -128.6598  -38.6598
```

If we add 360 degrees to the last two angles, we get 231.3402 and 321.3402. Clearly, each angle is 90 degrees plus the previous angle.

7.13 We can do this in MATLAB, to check the hand-drawn plot. We put the point (0,0) between the $z_n$ values so there is a line going to the origin.

```
z_1 = exp(-j*pi/5);
z_2 = 2*exp(j*pi/6);
z_3 = z_1*z_2;
x = real([z_1, 0, z_2, 0, z_3]);
y = imag([z_1, 0, z_2, 0, z_3]);
plot(x,y,'b')
```

7.15 There are a couple of different ways that we could plot these rotating vectors. Below, we plot the real values on the x-axis, and the imaginary ones on the y-axis.

```
t = 0:0.01:1;
z_1 = 4*exp(j*(2*pi*25*t - 5*pi/4));
z_2 = 5*exp(j*(2*pi*33*t + pi/6));
z_3 = z_1 + z_2;

% put 0 between each value
z0_1(1:length(z_1)*2) = 0;
z0_2(1:length(z_2)*2) = 0;
z0_3(1:length(z_3)*2) = 0;
for k = 1:length(z_1)
    z0_1(k*2 -1) = z_1(k);
    z0_2(k*2 -1) = z_2(k);
    z0_3(k*2 -1) = z_3(k);
end
% do the plots
figure(1);
plot(real(z0_1), imag(z0_1), 'b');
```

```
figure(2);
plot(real(z0_2), imag(z0_2), 'b');
figure(3);
plot(real(z0_3), imag(z0_3), 'b');

% Now plot them as frequencies
figure(4);
plot(1:length(z_1), abs(fft(z_1)), 'r', ...
     1:length(z_2), abs(fft(z_2)), 'g', ...
     1:length(z_3), abs(fft(z_3)), 'b');
```

We see from the plots that adding the two together produces a new signal that contains both frequencies, just like when we add 2 sinusoids (but without the mirror image on the frequency plot).

## Chapter 8

8.1

$$X(z) = 3z^0 + 2z^{-1} + 4z^{-2}$$

8.3 First we find the frequency response.

```
h = [1, 1];
% simulate impulse function
imp = zeros(1,1000);
imp(10) = 1;
% get frequency response
H = fft(conv(h, imp));
half = 1:ceil(length(H)/2);
plot(abs(H(half)))
```

We see that it is a lowpass filter. Now we find the zeros.

$$H(z) = 1 + z^{-1}$$

$$z(1 + z^{-1}) = z + 1$$

$$\text{Set} \quad z + 1 = 0$$

We see that it has a zero at $z = -1$. Finally, we simulate a sampled signal for the input to this filter.

```
n = 1:1000;
x = cos(2*pi*100*n/200);
% Now put through filter
y = conv(x, h);
plot(y)
```

We see that y is zero, except for the beginning and end. The frequency of 100 Hz is very high ($f_s/2$) when the sampling rate is 200 samples/second. Since this is lowpass filtered, we expect the signal to be attenuated.

8.5 a.

$$H(z) = -17 + 29z^{-1} + 107z^{-2} + 62z^{-3}$$

b.

$$H(z) = \frac{-17 + 29z^{-1} + 107z^{-2} + 62z^{-3}}{1 - (4z^{-1} - 7z^{-2} - 26z^{-3} - 15z^{-4})}$$

8.7

$$H(2e^{j\pi/6}) = 1 - (2e^{j\pi/6})^{-1} + (2e^{j\pi/6})^{-2}$$

$$= 1 - \frac{(1.7321 - j)}{(1.7321 + j)(1.7321 - j)} + \frac{(2 - 3.4641j)}{(2 + 3.4641j)(2 - 3.4641j)}$$

$$= 0.6920 + 0.0335j$$

```
>> z = 2*exp(j*pi/6);
>> H = 1 - z^(-1) + z^(-2)

H =

   0.6920 + 0.0335i
```

## Chapter 9

9.1 Down-sampling removes every other value, to give $z_d[n]$ = {4, -2, -1.5, -2, -4.5}. Down-sampling removes every other value, to give $w_d[n]$ = {4, 2, 4.5, 4, 4.5}.

9.3 Multiresolution is the recursive repetition of a transform (and later repeating the inverse transform accordingly). See the multiresolution figures in the text for an example.

9.5 First, we will use the `conv` function to do the convolution. Then we simulate down-sampling. Next, we simulate up-sampling, and do the inverse transform. Finally, we add the results of the two channels to find $y$. Notice how we make sure that $x$ and $y$ have the same length (since filtering makes the output longer).

```
% find forward transform
channel1 = conv(x, [1, 1]);
channel2 = conv(x, [-1, 1]);
% down-sample
ch1_out = channel1(2:2:length(channel1));
ch2_out = channel2(2:2:length(channel2));
clear channel1 channel2
% now undo the transform
% up-sample
channel1(1:length(ch1_out)*2) = 0;
channel2(1:length(ch2_out)*2) = 0;
for k = 1:length(ch1_out)
    channel1(k*2 -1) = ch1_out(k);
    channel2(k*2 -1) = ch2_out(k);
end
% find inverse transform
y1 = conv(channel1, [-1, -1]);
y2 = conv(channel2, [-1, 1]);
% Add result. Trim extra size.
y = y1(1:length(x)) + y2(1:length(x));
plot(1:length(x), x, 'b', 1:length(y), y, 'g')
```

When we examine the relationship of $y$ to $x$, we find that $y[n] = -2x[n]$.

9.7 No, this does not work in general. Not only is the pattern important, but the filter values are carefully chosen. We cannot use just any values for the filter coefficients.

9.9 We need to convolve the Daubechies coefficients with the input, then down-sample the results. In the code below, `ch1_out` is the lowpass output, while `ch2_out` is the highpass output.

```
% get coefficients
d0 = (1-sqrt(3))/(4*sqrt(2));
```

```
d1 = -(3-sqrt(3))/(4*sqrt(2));
d2 = (3+sqrt(3))/(4*sqrt(2));
d3 = -(1+sqrt(3))/(4*sqrt(2));
LPF = [ d0, -d1, d2, -d3 ];
HPF = [ d3, d2, d1, d0 ];
% find forward transform
channel1 = conv(x, LPF);
channel2 = conv(x, HPF);
% down-sample
ch1_out = channel1(2:2:length(channel1));
ch2_out = channel2(2:2:length(channel2));
```

9.11

```
x = [6, 1, 3, 7, 2, 5, 8, 10];
z = conv(x, [0.5, -0.5]);
w = conv(x, [0.5, 0.5]);
y1 = conv(z, [-0.5, 0.5]);
y2 = conv(w, [0.5, 0.5]);
y = y1 + y2;
```

We see that $y$ is the same signal as $x$, except that it has an extra zero at each end.

9.13 The actual length of the outputs depends on the filter size, that is, the `db4` filter adds a few more outputs than the `db2` does. We will instead answer in terms of the general pattern. With the down-samplers, each octave has (approximately) half the detail outputs as the octave before it. Therefore, we would have:

$$\text{octave 1} \approx 512 \text{ outputs}$$

$$\text{octave 2} \approx 256 \text{ outputs}$$

$$\text{octave 3} \approx 128 \text{ outputs}$$

$$\text{octave 4} \approx 64 \text{ outputs.}$$

The approximate outputs would be the same size as the details. But we do not need to keep the approximations from the first 3 octaves. Therefore, the total number of outputs is $\approx 64 + 64 + 128 + 256 + 512$, or 1024 outputs.

If up/down-sampling is not used, then each channel would output the same number of inputs. The total number of outputs would be $\approx 1024 + 1024 + 1024 + 1024 + 1024$, or 5120 outputs.

9.15 We will not consider up/down-samplers as part of this answer, since their

effect will only scale our results. To find the top channel's effect on the input, we convolve $h_0$ with $g_0$, and find the $z$-transform of the result. Our result is $aa$, $2ab$, $2ac + bb$, $2(ad + bc)$, $2bd + cc$, $2cd$, $dd$. The $z$-transform is:

$$aa + 2abz^{-1} + (2ac + bb)z^{-2} + 2(ad + bc)z^{-3} + (2bd + cc)z^{-4} + 2cdz^{-5} + ddz^{-6}.$$

Now we find the bottom channel's effect on the input, we convolve $h_1$ with $g_1$, and find the $z$-transform of the result. The $z$-transform of the result is:

$$-aa + 2abz^{-1} - (2ac + bb)z^{-2} + 2(ad + bc)z^{-3} - (2bd + cc)z^{-4} + 2cdz^{-5} - ddz^{-6}.$$

When we add the two $z$-transforms together, to get $Y(z)$, we get:

$$2(2ab)z^{-1} + 2(2(ad + bc))z^{-3} + 2(2cd)z^{-5}.$$

Every other term cancelled out. Our goal is to have $Y(z)$ with exactly one term. To satisfy this requirement, two of the three terms must be zero, i.e., $ab = 0$, and $cd = 0$. The only way we can have $ab = 0$ is for $a$ or $b$ to be zero. Thus, we conclude that this will not work, except for the trivial case where two of the four coefficients are zero. That would not be different from a filter with 2 taps.

## Chapter 10

10.1 Multiplying the vector `x` with a fraction will reduce every value in it. The resulting sound will be of a lower volume. Using 1.1 as a multiplying factor boosts the sound. It will sound louder, but the sound values could exceed +1, and therefore be clipped.

10.3 Assume that we have 2 sound arrays already in memory, `x1` and `x2`. The code below will find the concatenation of two sound arrays, assuming they both have 2 channels.

```
[r1, c1] = size(x1);
[r2, c2] = size(x2);
y = x1;
y(r1+1:r1+r2, 1:c2) = x2;
sound(y, fs)
```

The sound should be stored as column vectors. If they are stored as row vectors, then we need to alter the line before the `sound` command.

10.5 We will read a sound file, and store it in variable `x`. Assuming that `x` has 2 channels, with sound in column vectors, we will only use the first channel, and make the result a row vector.

```
[x, fs, b] = wavread('test_file.wav');
x = x(:,1).';
X = fft(x);
% Shift indices by 50
Xshifted = [X(50:length(X)), X(1:49)];
% convert to time-domain
xshifted = ifft(Xshifted);
% play result
sound(real(xshifted), fs)
```

The code above plays the sound with a "beating"-like noise. This shifts the whole spectrum, instead of just the first half. Since the halves would be symmetrical, it makes sense to work on the first half only, then mirror the results.

```
% Shift indices in first half only
Xhalf = X(1:ceil(length(X)/2));
firstHalf = [Xhalf(50:length(Xhalf)), Xhalf(1:49)];
% now mirror this half onto the second half
secondHalf = firstHalf(length(Xhalf):-1:1);
Xshifted = [firstHalf, secondHalf];
xshifted = ifft(Xshifted);
sound(real(xshifted), fs)
```

This still sounds strange. The beating noise is gone, but the sound has an odd timing issue. No, we will not be able to replicate this by simply changing the sampling frequency. (Though this might work if the signal is a single frequency.) Now let's reverse the spectrum:

```
% Shift indices in first half only
midpoint = ceil(length(X)/2);
firstHalf = X(midpoint:-1:1);
% now mirror this half onto the second half
secondHalf = firstHalf(midpoint:-1:1);
Xreversed = [firstHalf, secondHalf];
xreversed = ifft(Xreversed);
sound(real(xreversed), fs)
```

The sounds are barely audible, and at very high frequencies.

10.7 To find the answer to a problem like this, first try it yourself with a small matrix (say 4 by 4). Once you see the desired outcome, try to isolate a row (or column) of the solution, then replicate it for the other rows (or columns).

```
x = imread('dog256x256.gif');
% Rotate the image and show it
% This should work for any image, regardless of dimensions
[MAX_ROWS, MAX_COLS] = size(x);
for r=1:MAX_COLS
    cclockwise(:,r) = x(r, MAX_ROWS:-1:1);
end
imshow(cclockwise);
```

10.9 The program would not work. We get an error message from MATLAB, since we must change each call to only the name, without parameters, i.e., `sudoku_search` instead of `sudoku_search(puzzle)`. Assuming that we take care of this, the program *still* will not work.

The `function` keyword means that the computer will create its own copies of the parameters, and get rid of them after the function finishes. Removing this keyword means that there will be only one copy of the puzzle. Once the puzzle has a guess added to it, the programs will not be able to "undo" the guess later.

10.11

```
>> disp(sprintf('%tx',9.0938))
41118034


>> disp(sprintf('%tx',9.34))
411570a4
```

10.13 We can modify the `filter_test2.m` program from the CD-ROM, to get rid of the things we do not need, and to add in several windows.

```
% get different window values
w1 = window(@gausswin, taps);
w2 = window(@hamming, taps);
w3 = window(@rectwin, taps);
```

```
w4 = window(@triang, taps);
% now combine window with filter coeffs
coeffs1 = w1.' .* c2;
coeffs2 = w2.' .* c2;
coeffs3 = w3.' .* c2;
coeffs4 = w4.' .* c2;
```

From plotting the frequency magnitude responses, we can tell right away that the plots corresponding to the rectangle and triangle windows are not very good. The other two are both good, and we see a classic filter design trade-off. While the Gaussian window has smaller side-lobes, the Hamming window has a narrower main lobe. The Gaussian window's frequency response appears to be just a little bit better than the Hamming window's response.

10.15 a. There are 4 ($2^2$) distinct values $\{0, -1, 3, 9\}$, so we would need 2 bits to store each. Thus, we need $12 \times 2 = 24$ bits to store the above sequence, plus a table to let us know how to decode the bits.

b. Let $0 = 00$, $-1 = 01$, 3=10, 9=11 (this is *not* the same as storing them in binary!). Also, there are several other ways of making this table; the mapping here is somewhat arbitrary.

c. The above signal would be stored as: 00 01 11 01 11 11 00 10 01 10 00 00

d. The encoded signal takes 24 bits. We would also need to store the decoding table. We need a table length, say 16 bits for this. We could use 16 bits per value, then a byte to encode the bit pattern and count (i.e., 0010 0000 for the first value). Total: 16 bits + 4 (16 + 8) + 24 = 136 bits. This is a small example where the encoding table takes up a large amount of space, but we are able to save it in 70% of the space.

# Appendix F

# Glossary

*ADC* : analog-to-digital converter, a device that samples continuous values and outputs them as discrete values.

*Aliasing* : replications, caused by sampling, that interfere with signal.

*Amplitude* : a quantity without units, could be positive or negative.

*Array* : a one-dimensional list of numbers, such as $\{1, 2, 3\}$. This is also called a *vector*.

*Attenuate* : greatly reduce certain frequencies.

*Bandpass sampling* : sampling at less than twice the maximum frequency, but at least twice the bandwidth.

*Bandwidth* : range of signal's frequencies.

*Causal* : a system that uses only current or previous inputs.

*Continuous Fourier Transform (CFT)* : a version of the Fourier transform for continuous signals. Here we use it as a mathematical abstraction, useful for understanding theory.

*Complex conjugate* : a number identical to a complex one, except with a different sign for the complex part. For example, $1+2j$ and $1-2j$ are complex conjugates.

*Complex number* : an extension of a number, a complex number has a real and an imaginary part, e.g., $3 - 4j$.

*Critical sampling* : sampling at exactly twice the bandwidth, just fast enough.

*DAC* : digital-to-analog converter, a device that "fills in the blanks" between digital samples and outputs them as continuous values.

*DC component* : from Direct Current, this term means the amplitude at 0 Hz in a frequency-domain representation of a signal.

*DFT leakage* : a resolution problem of the DFT where frequency information is spread out over several frequencies.

*Discrete Fourier Transform (DFT)* : a way to convert discrete time-domain data to discrete frequency-domain data. The Fast Fourier Transform (FFT) is more efficient, and gives the same results.

*Fast Fourier Transform (FFT)* : an efficient way to convert discrete time-domain data to discrete frequency-domain data. It gives the same results as the Discrete Fourier Transform. MATLAB provides this function.

*Filter bank* : a pair of filters used with the discrete wavelet transform, one lowpass, one highpass.

*Finite Impulse Response (FIR)* : this type of filter gives a finite number of nonzero outputs (response) to an impulse function input. It does not use feed-back.

*Fixed point* : a binary representation where a set number of bits hold the whole and fractional parts, with an understood radix point.

*Folding* : a type of aliasing where the sampling frequency is between the bandwidth and twice the bandwidth.

*Fourier Transform (FT)* : a way to convert time-domain data to the frequency-domain.

*Frequency component* : one (of many) sinusoids that make up a signal.

*Frequency Magnitude Response (FMR)* : the relative magnitudes of frequencies in a frequency-domain plot of a signal.

*Harmonic signal* : several sinusoids added together, each with an integer multiple of a fundamental frequency.

*Highpass* : A highpass filter lets the high frequencies through.

*Impulse response* also $h[n]$ : shows the way a filter behaves when it gets a single nonzero input. For FIR filters, this shows us what effect the filter has on an input signal's frequencies.

*Infinite Impulse Response (IIR)* : this type of filter uses feed-back, so it could have an infinite number of nonzero outputs (response) to an impulse function input.

*Linear* : A property of some systems (like an FIR or IIR filter) where the output relation has a sum of inputs multiplied by constants.

*Lowpass* : A lowpass filter lets the low frequencies through.

*Lowpass sampling* : assumes the bandwidth starts at 0 and goes to the maximum frequency.

*LTI* : Linear and Time-Invariant.

*MAC* : Multiply Accumulate Cell. It is a regular structure that can be used to implement filters in hardware.

*Magnitude* : a quantity without units, always positive.

*Matrix* (plural *Matrices*) : a multidimensional group of values. An image is a 2D group of pixel values, and can be considered a matrix.

*Multiresolution* : If the transform/inverse transform works once, we can do the transform again and again. Later, we do the inverse transform once for every time we did the forward transform.

*Noise* : any unwanted frequency content.

*Normalized* : scaling done on filter coefficients, so no scaling is needed at end. That is, a filter and inverse filter should output the same values that were input. If the filter coefficients are not normalized, then the output would be the input multiplied by a constant.

*Octave* : a level of resolution in the discrete wavelet transform. Details at octave $n$ are coarser than those at octave $n-1$.

*Orthogonal* : at right angles (in 2D). More formally (and for higher dimensions), orthogonality is the property that the inner product of the coordinate bases equals zero.

*Orthonormal* : orthogonal and normalized.

*Oversampling* : taking samples more frequently than needed.

*Period* : the length of time before a (periodic) signal repeats itself.

*Pixel* : Short for *picture element*, this refers to the tiniest dots that make up a display, such as in a television. This term also means the color (or number corresponding to the color) for a pixel, as in *pixel value*.

*Phasor* : A vector, typically one that rotates. Imagine a directed arrow from the origin.

*Pole* : where the denominator becomes 0 in a transfer function (the frequency response of a filter).

*Reconstruction* : converts a digital signal back to analog (also used to mean application of an inverse transform).

*Region of Convergence (RoC)* : area where the $z$-transform converges, typically inside the unit circle.

*Sampling* : the process that converts analog to a digital representation.

*Scalar* : a single number by itself, such as 3.1. Compare to *vector*.

*Signal* : a measurable, variable phenomenon, often a physical quantity that varies

with time.

*Sinusoid* : a general term to mean a sine or cosine function.

*Spectrum* : frequency plot of a signal.

*Subband coding* : the type of operation produced by a filter bank in the wavelet transform, where complementary filters divide the signal into a subsignal of low-frequency components, and one of high-frequency components.

*System* : something that performs an operation on a signal.

*Time-Invariant* : A property of some systems where a shift in input produces a corresponding shift in the output.

*Transform* : the operation that the system performs.

*Transfer function (H(z))* : an output/input function that describes the behavior of a filter. It is based on the $z$-transform of the filter coefficients.

*Two-channel filter bank* : General term for combination of analysis and synthesis filters.

*Undersampling* : occurs when we do not take samples often enough.

*Vector* : a one-dimensional list of numbers, such as {1, 2, 3}. This is also called an *array*. *Vector* is also used to mean a directed arrow in 2D (or greater) space.

*Word size* : the number of bits that a microprocessor accesses in a single operation.

*Zero* : where the numerator becomes 0 in a transfer function (the frequency response of a filter).

*Zero padding* : appending zeros to time-domain signal.

*z-transform* : transforms data from time-domain to frequency-domain. Under certain conditions, it reduces to the Fourier transform.

# Bibliography


[1] S. Mallat, "A Theory for Multiresolution Signal Decomposition: The Wavelet Representation," *IEEE Pattern Analysis and Machine Intelligence*, vol. 11, no. 7, pp. 674–693, 1989.

[2] I. Daubechies, *Ten Lectures on Wavelets.* Montpelier, Vermont: Capital City Press, 1992.

[3] G. Strang and T. Nguyen, *Wavelets and Filter Banks.* Wellesley-Cambridge Press, 1997.

[4] R. R. Coifman and M. V. Wickerhauser, "Wavelets and Adapted Waveform Analysis," in *Proceedings of Symposia in Applied Mathematics* (I. Daubechies, ed.), pp. 119–153, Providence, Rhode Island: American Mathematics Society, November 6–9, 1993. Volume 47.

[5] The Learning Company, Inc., *Compton's Interactive Encyclopedia.* Cambridge, Massachusetts: Simon and Schuster, 1995.

[6] J. H. McClellan, R. W. Schafer, and M. A. Yoder, *DSP First: A Multimedia Approach.* Prentice Hall, 1998.

[7] B. B. Hubbard, *The World According to Wavelets.* Wellesley, Massachusetts: A. K. Peters, 1996.

[8] R. Nave, *http://hyperphysics.phy-astr.gsu.edu/HBASE/hph.html, Hyper-Physics.* Department of Physics and Astronomy, Georgia State University, 2005.

[9] T. Bose, *Digital Signal and Image Processing.* John Wiley & Sons, 2004.

[10] P. Linz and R. L. C. Wang, *Exploring Numerical Methods: An Introduction to Scientific Computing Using MATLAB.* Jones and Bartlett Mathematics, 2003.

[11] R. Lyons, *Understanding Digital Signal Processing.* Addison-Wesley, 1997.

[12] S. W. Smith, *Digital Signal Processing A Practical Guide for Engineers and Scientists.* Newnes, 2003.

[13] R. E. Walpole and R. H. Myers, *Probability and Statistics for Engineers and Scientists, Third Edition.* New York: MacMillan Publishing Company, 1985.

[14] E. C. Ifeachor and B. W. Jervis, *Digital Signal Processing A Practical Approach, Second Edition.* Harlow, England: Prentice-Hall, 2002.

[15] C. B. Boyer, *The History of the Calculus and Its Conceptual Development.* New York: Dover Publications, Inc., 1959.

[16] C. J. Richard, *Twelve Greeks and Romans Who Changed The World.* Rowman & Littlefield, 2003.

[17] L. Gonick, *The Cartoon History of the World, Volumes 1–7.* New York: Doubleday, 1990.

[18] M. N. Geselowitz, "Hall of Fame: Heinrich Hertz," *IEEE-USA News & Views*, November 2002.

[19] D. D. Andrew Bruce and H.-Y. Gao, "Wavelet Analysis," *IEEE Spectrum*, pp. 26–35, October 1996.

[20] E. A. Lee and P. Varaiya, *Structure and Interpretation of Signals and Systems.* Addison-Wesley, 2003.

[21] R. W. Hamming, *Digital Filters, Third Edition.* Mineola, New York: Dover Publications, 1998.

[22] D. Sheffield, "Equalizers," *Stereo Review*, pp. 72–77, April 1980.

[23] G. Kaiser, *A Friendly Guide to Wavelets.* Boston: Birkhauser, 1994.

[24] P. J. Nahin, *An Imaginary Tale: The Story of* $\sqrt{-1}$. Princeton University Press, 1998.

[25] S. P. Thompson and M. Gardner, *Calculus Made Easy.* New York: St. Martin's Press, 1998.

[26] E. W. Swokowski, *Elements of Calculus with Analytic Geometry.* Boston, Massachusetts: Prindle, Weber and Schmidt, 1980.

[27] W. K. Chen, ed., *The Electrical Engineering Handbook.* Burlington, MA: Elsevier Academic Press, 2005.

[28] J. Ozer, "New Compression Codec Promises Rates Close to MPEG," *CD-ROM Professional*, September 1995.

[29] A. Hickman, J. Morris, C. L. S. Rupley, and D. Willmott, "Web Acceleration," *PC Magazine*, June 10, 1997.

[30] W. W. Boles and Q. M. Tieng, "Recognition of 2D Objects from the Wavelet Transform Zero-crossing Representations," in *Proceedings SPIE, Mathematical Imaging*, (San Diego, California), pp. 104–114, July 11–16, 1993. Volume 2034.

[31] M. Vishwanath and C. Chakrabarti, "A VLSI Architecture for Real-time Hierarchical Encoding/decoding of Video Using the Wavelet Transform," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP '94)*, (Adelaide, Australia), pp. 401–404, April 19–22, 1994. Volume 2.

[32] M. Weeks and M. Bayoumi, "Discrete Wavelet Transform: Architectures, Design and Performance Issues," *Journal of VLSI Signal Processing*, vol. 35, pp. 155–178, September 2003.

[33] A. Haar, "Zur theorie der orthogonalen funktionensysteme," *Mathematische Annalen*, vol. 69, pp. 331–371, 1910.

[34] M. J. T. Smith and T. P. Barnwell III, "Exact Reconstruction Techniques for Tree-structured Subband Coders," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, pp. 434–441, June 1986.

[35] S. Jaffard, Y. Meyer, and R. D. Ryan, *Wavelets Tools for Science & Technology.* Philadelphia: Society for Industrial and Applied Mathematics (SIAM), 2001.

[36] H. Anton, *Elementary Linear Algebra, 6th Edition.* New York: John Wiley & Sons, Inc., 1991.

[37] C. Chakrabarti and M. Vishwanath, "Efficient Realizations of the Discrete and Continuous Wavelet Transforms: From Single Chip Implementations to Mappings on SIMD Array Computers," *IEEE Transactions on Signal Processing*, vol. 43, pp. 759–771, March 1995.

[38] D. Goldberg, "What Every Computer Scientist Should Know About Floating-point Arithmetic," *Computing Surveys*, pp. 171–264, March 1991.

[39] S.-W. Wu, “Additive Vector Decoding of Transform Coded Images,” *IEEE Transactions on Image Processing*, vol. 7, pp. 794–803, June 1998.

[40] O. K. Al-Shaykh and R. M. Mersereau, “Lossy Compression of Noisy Images,” *IEEE Transactions on Image Processing*, vol. 7, pp. 1641–1652, December 1998.

[41] E.-B. Fgee, W. J. Phillips, and W. Robertson, “Comparing Audio Compression Using Wavelets with Other Audio Compression Schemes,” *IEEE Canadian Conference on Electrical and Computer Engineering*, vol. 2, pp. 698–701, 1999.

[42] T. Rabie, “Robust Estimation Approach for Blind Denoising,” *IEEE Transactions on Image Processing*, vol. 14, pp. 1755–1765, November 2005.

# Index

.jpg format, 385
.mp3 format, 385
.tif format, 387
&& operator, 38
$z$-transform, 254, 409

abs, 66, 68
aliasing, 164
amplitude, 135
analysis, 277
associative, 103
attenuated, 106
autocorrelation, 117

Bach, 187
bandpass, 104
bandpass sampling, 176
bandstop, 104, 108
bandwidth, 160
basis, 280, 281
bias, 372

catch, 313
causal, 99
cd, 388
center frequency, 180
clc, 32
clear, 32, 216
commutative, 103
complex conjugate, 204, 236
complex number, 6
complex numbers, 5
conditionally stable, 113
conjugate quadrature filter, 279
converting binary to decimal, 4
convolution, 96, 407
    associative property, 103, 407
    commutative property, 103, 407
    distributive property, 103, 407
correlation coefficient, 115
CQF, 279
critical sampling, 160
cross-correlation, 117
cyclic frequency, 135

DC component, 141
DFT, 109, 191
DFT leakage, 212
DFT shifting theory, 203
discrete Fourier transform, 191
discrete wavelet transform, 275
disp, 36
distributive, 103
down-sampling, 288
DTFT, 189
DWT, 275
dwt command, 336

e, 399
edge effects, 144
else, 38
else if, 38
elseif, 38
end, 38, 39

Euler, 399
Euler's equation, 403
Euler's formula, 403
Euler's inverse formula, 403

feed-forward, 91
filter coefficients, 91
finite impulse response, 87, 88
FIR, 87, 88
floating-point representation, 371
folding, 168
folding frequency, 168
for, 39
forward transform, 277
Fourier series, 189
Fourier transform, 5, 109, 405
frequency, 135
frequency response
    pole, 264
    zero, 264
fundamental frequency, 140

Gibbs' phenomenon, 144

h[n], 96
Haar transform, 279
harmonic, 140
harmonics, 214
Heisenberg's uncertainty principle, 219
help command, 31
Hertz, 135
highpass, 104
Huffman coding, 394

IDFT, 204
idwt command, 336
IEEE 754 standard, 371
if, 36
IIR, 111, 112
imaginary numbers, 5
impulse function, 88
impulse response, 96
infinite impulse response, 111, 112
inner product, 311
interpolation, 162
inverse DFT, 204
inverse discrete Fourier transform, 204
inverse transform, 21, 277

j, 5
JPEG, 387

Laplace transform, 409
linear, 99
lowpass, 104
lowpass sampling, 176

MAC, 103
magnitude, 135
main lobe, 378
mantissa, 371
MATLAB
    assignment statement (=), 33
MATLAB commands
    ..., 39
    abs, 63, 66, 135, 192
    and, 37, 38
    angle, 192
    assignment, 37
    atan, 282
    audiorecorder, 341, 375
    auread, 341
    auwrite, 341
    axis, 56
    catch, 71, 314
    cd, 388
    ceil, 69, 199
    clc, 32
    clear, 32, 216
    dir, 388

disp, 36
dwt, 336, 349, 352
dwt2, 276, 352
else, 38
elseif, 38
end, 29, 38
fft, 190
figure, 343
fir1, 110, 376, 384
fix, 69
floor, 69
for, 39, 366
getaudiodata, 342
help, 31, 32, 52
idwt, 336
idwt2, 276
if, 29, 36
ifft, 190
imread, 343, 387
imshow, 389
imwrite, 342, 387
inv, 51
length, 24
ls, 388
max, 73, 346
mean, 73
median, 73
min, 73, 345
mkdir, 388
pause, 60, 216, 341
play, 341
plot, 23, 55, 308
pwd, 388
rand, 76
record, 341
recordblocking, 341
round, 69, 199, 356
size, 41
sort, 73
sound, 340
sprintf, 35, 372
sum, 43, 118
tic, 356
toc, 356
transpose, 42
try, 71, 314
wavplay, 340
wavrecord, 340, 375
wavwrite, 341
wfilters, 71
while, 29, 39, 366
who, 33
whos, 33
window, 378
zeros, 347
zplane, 266
MATLAB keywords
db16, 347
db2, 286, 297
db4, 352, 356, 393
double, 340
false, 364
function, 29, 366
true, 364
uint8, 340, 345, 347
mean square error, 408
mkdir, 388
MSE, 408
multiply accumulate cell, 103
multiresolution, 314

norm, 314
normal, 311
notch, 104
Nyquist rate, 175

octave, 275, 320, 336
octaves, 315
order, 93

orthogonal, 311, 320
orthogonality, 311
orthonormal, 311, 320
oversampling, 160

PE, 104
peak signal to noise ratio, 390, 408
perfect reconstruction, 278
period, 60, 160
periodic, 144
phase shift
    principal value of, 138
phasor, 39
plot, 216
pole, 264
processing elements, 104
PSNR, 390, 408
pwd, 388

QMF, 279
quadrature mirror filter, 279
quantization, 16

radian frequency, 135
radians, 136
Reconstruction, 162
recursion, 366
region of convergence, 254
RMSE, 64, 390, 408
RoC, 254
root mean square error, 64, 390, 408

sampling, 14, 159
sampling frequency, 160
sampling period, 160
separability, 276
side lobe, 378
signal, 10
signal to noise ratio, 408
significand, 371
sinc, 85
SNR, 408
spectrum, 152
sprintf, 35, 36
stable, 113
subband coder, 279
subband coding, 277
support, 92
synthesis, 277
system, 19

taps, 93
TIFF standard, 387
time-invariant, 101
transfer function, 253, 264, 271
transform, 20
trigonometric identities, 404
try, 313
two-channel filter bank, 277

undersampling, 160
unit impulse, 96
unit impulse response, 96
unstable, 113
up-sampling, 288

vector, 39

wavelet packets, 315
wavelets, 275
wfilters, 313
while, 39
who, 33
whos, 33
window, 213
windowing, 376
windows, 85
word size, 3

zero-padding, 191
zeros, 264

# DIGITAL SIGNAL PROCESSING

## *Using MATLAB® and Wavelets*

### *Michael Weeks*

Although DSP has long been considered an EE topic, recent developments have also generated significant interest from the computer science community. DSP applications in the consumer market, such as bioinformatics, the MP3 audio format, and MPEG-based cable/satellite television have fueled a desire to understand this technology outside of hardware circles.

Designed for upper division engineering and computer science students as well as practicing engineers, *Digital Signal Processing Using MATLAB and Wavelets* emphasizes the practical applications of signal processing. Over 100 MATLAB examples and wavelet techniques provide the latest applications of DSP, including image processing, games, filters, transforms, networking, parallel processing, and sound.

The book also provides the mathematical processes and techniques needed to ensure an understanding of DSP theory. Designed to be incremental in difficulty, the book will benefit readers who are unfamiliar with complex mathematical topics or those limited in programming experience. Beginning with an introduction to MATLAB programming, it moves through filters, sinusoids, sampling, the Fourier transform, the z-transform and other key topics. An entire chapter is dedicated to the discussion of wavelets and their applications. A CD-ROM (platform independent) accompanies the book and contains source code, projects for each chapter, and the figures contained in the book.

**FEATURES:**

- Contains over 100 short examples in MATLAB used throughout the book
- Includes an entire chapter on the wavelet transform
- Designed for the reader who does not have extensive math and programming experience
- Accompanied by a CD-ROM containing MATLAB examples, source code, projects, and figures from the book
- Contains modern applications of DSP and MATLAB project ideas

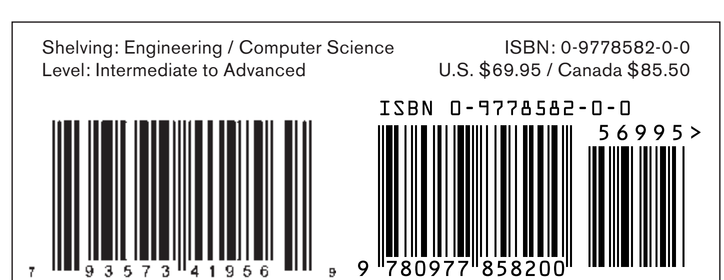
**BRIEF TABLE OF CONTENTS:**

**1. Introduction. 2. MATLAB. 3. Filters. 4. Sinusoids. 5. Sampling. 6. The Fourier Transform. 7. The Number *e*. 8. The *z*-Transform. 9. The Wavelet Transform 10. Applications. Appendix A. Constants and Variables. B. Equations. C. DSP Project Ideas. D. About the CD. Answers. Glossary. Index.**

**ABOUT THE AUTHOR:**

**Michael Weeks** is an associate professor at Georgia State University where he teaches courses in Digital Signal Processing. He holds a PhD in computer engineering from the University of Louisiana at Lafayette and has authored or co-authored numerous journal and conference papers.

Shelving: Engineering / Computer Science
Level: Intermediate to Advanced
ISBN: 0-9778582-0-0
U.S. $69.95 / Canada $85.50



INFINITY SCIENCE PRESS
11 Leavitt Street
Hingham, MA 02043
(781) 740-4487
(781) 740-1677 FAX
info@infinitysciencepress.com
www.infinitysciencepress.com

All trademarks and service marks are the property of their respective owners.
Cover design: Tyler Creative